Master Thesis

Design and Development of a Dymola/Modelica Library for Discrete Event-oriented Systems using DEVS Methodology

ETH Zürich
Department of Computer Science
Institute of Computational Science

Tamara Beltrame

Adviser: Prof. François E. Cellier
Responsible: Prof. Walter Gander

11th March 2006

Abstract

Continuous-time systems can be converted to discrete-event descriptions using the Quantized State Systems (QSS) formalism. Hence it is possible to simulate continuous-time systems using a discrete-event simulation tool, such as a simulation engine based on the DEVS formalism.
A new Dymola library, ModelicaDEVS, was developed that implements the DEVS formalism. DEVS has been shown to be efficient for the simulation of systems exhibiting frequent switching operations, such as flyback converters. ModelicaDEVS contains a number of basic components that can be used to carry out DEVS simulations of physical systems. Furthermore, it is also possible - with some restrictions - to combine the two simulation types of ModelicaDEVS and Dymola (discrete-event and discrete-time simulation) and create hybrid models that contain ModelicaDEVS as well as standard Dymola components.

Contents

1 Introduction **1**
1.1 Goal . . . 1
1.2 Motivation . . . 1
1.3 Structure of the Thesis . . . 3

2 Continuous Systems Simulation with DEVS **5**
2.1 Discrete-Event Simulation . . . 5
2.2 The DEVS Formalism . . . 7
2.2.1 Atomic Models . . . 8
2.2.2 Coupled Models . . . 12
2.3 DEVS Integration with QSS . . . 14
2.3.1 Discrete Simulation of a Continuous System . . . 14
2.3.2 Quantised State Systems (QSS) . . . 15

3 PowerDEVS **19**
3.1 The Modelling Environment . . . 19
3.2 The Simulator . . . 21
3.2.1 Simulator Theory . . . 22
3.2.2 Simulator Architecture . . . 29

4 ModelicaDEVS – the Simulator **39**
4.1 Dymola Programming/Simulation Environment . . . 39
4.1.1 Simultaneous Equations . . . 39
4.1.2 Simulation Time . . . 40
4.1.3 State and Time Events . . . 40
4.2 Atomic Models . . . 41
4.2.1 Events/Ports . . . 41
4.2.2 Transitions . . . 42
4.2.3 Miscellaneous . . . 44
4.2.4 Basic Model Structure . . . 45
4.3 Coupled Models . . . 45
4.3.1 Time-advance Mechanism . . . 46
4.3.2 Direct Block Interaction . . . 46
4.3.3 Concurrent Events . . . 47
4.3.4 Loops . . . 47
4.3.5 Priorities . . . 48

4.4 Hierarchic Models . . . 50
4.5 Time/State Events . . . 51

5 ModelicaDEVS – the Library **53**
5.1 WorldModel . . . 53
5.2 Source Blocks . . . 55
5.2.1 Constant . . . 57
5.2.2 Pulse . . . 57
5.2.3 PWM Signal . . . 58
5.2.4 Ramp . . . 59
5.2.5 Saw . . . 60
5.2.6 Sine . . . 61
5.2.7 Square . . . 63
5.2.8 Step . . . 64
5.2.9 Trapezoid . . . 64
5.2.10 Triangular . . . 66
5.2.11 SamplerLevel . . . 66
5.2.12 SamplerTime . . . 67
5.2.13 SamplerTrigger . . . 68
5.3 Function Blocks . . . 69
5.3.1 Unary and Binary Operators . . . 69
5.3.1.1 Unary Operators . . . 69
5.3.1.2 Binary Operators . . . 70
5.3.2 CommandedSampler . . . 72
5.3.3 Comparator . . . 73
5.3.4 CrossDetect . . . 74
5.3.5 Delay . . . 75
5.3.6 Hold . . . 77
5.3.7 Hysteresis . . . 77
5.3.8 Integrator . . . 78
5.3.9 Quantiser . . . 80
5.3.10 Saturation . . . 80
5.3.11 Switch . . . 81
5.4 Sink Blocks . . . 83
5.4.1 The Interpolator as a simple Interface . . . 83
5.4.2 Function Smoothening . . . 84
5.4.3 Interpolator Specific Ports . . . 87
5.5 Templates . . . 88

6 Testing and Efficiency **91**
6.1 Testing . . . 91
6.1.1 Block Testing . . . 91
6.1.2 Testing of Mixed Systems . . . 91
6.1.3 Testing of Hybrid Systems . . . 95
6.2 Efficiency . . . 95
6.2.1 Flyback Converter Scheme (Dymola) . . . 96
6.2.2 Flyback Converter Block Diagram (PowerDEVS/ModelicaDEVS) . . . 98

6.2.3 Performance Comparison 103
6.2.3.1 General Remarks 103
6.2.3.2 Comparison 104

7 Conclusion **107**
7.1 Summary 107
7.1.1 Modelling 107
7.1.2 Mixed Systems 107
7.1.3 Performance 107
7.1.4 Substitution of Dymola components 108
7.2 Outlook and Open Problems 108
7.2.1 Extension of ModelicaDEVS 108
7.2.2 Index Reduction Problem 109

Bibliography **111**

A Official Task Description **113**

Chapter 1

Introduction

1.1 Goal

The goal of this thesis is to implement a new Dymola/Modelica library, consisting of a number of blocks that enable simulation according to the DEVS formalism.
ModelicaDEVS – which is the new library's name – should allow for the same modelling/simulation possibilities as they are provided by PowerDEVS [13], one of the currently existing DEVS implementations.

1.2 Motivation

Dymola/Modelica primarily deals with continuous physical problems, usually described by differential equations. Integration is therefore an important subject and justifies the attempt to improve/enhance Dymola's current possibilities regarding integration methods.

It may be a bit surprising at a first glance why a methodology such as DEVS, originally designed for discrete systems, should be useful for the simulation of continuous systems.
Toward the end of the nineties however, a new approach for numerical integration has been developed by Zeigler et al. [20]: given the fact that all computer-based simulations have to undergo a discretisation in one way or another – as digital machines are not able to process raw continuous signals – the basic idea of the new integration approach was to replace the discretisation of time by a discretisation of state, such that the system is not supposed to advance from time step to time step anymore, but rather from state to state. While conventionally the time axis was subdivided into intervals of equal length, the time instants when the system executes the next simulation step are now distributed irregularly because of their dependence on the time instants when the system enters a new state which is not necessarily at equidistant time instants.
The DEVS formalism turned out to be particularly suited to implement such a state quantisation approach given that it is not limited to a finite number of system states, in contrast to other discrete-event simulation techniques.
The Quantised State Systems introduced by Kofman [11] in 2001 improved the original quantised state approach of Zeigler, and hence gave rise to efficient DEVS simulation of large and complex systems.

The simulation of a continuous system by a (discrete) DEVS model comes with several benefits:

- Usually, the classic methods use a discretisation of time, which however has the drawback that the variables have to be updated contemporarily[1]. Thus, the time steps have to be chosen according to the variable that changes the fastest, otherwise a change in that variable could be missed. In a large system where probably very slow but also very fast variables are present (many slow and a few fast variables, in the worst case), this is critical to computation time, since the slow variables have to be updated way too often. The DEVS formalism however allows for asynchronous variable updates, whereby the computational costs can be reduced significantly: every variable updates at its own speed; there is no need anymore for an adaptation to the fastest one in order not to miss important developments between time steps.
  This property could be extremely useful in stiff systems that exhibit widely spread eigenvalues, i.e., that feature mixed slow and fast variables.

- The DEVS formalism is very well suited for problems with frequent switching operations such as electrical power systems. Given that the problem of iteration at discontinuities does not apply anymore, it even allows for real-time simulation.

- For hybrid systems with continuous-time, discrete-time and discrete-events parts, a discrete-event method provides a "unified simulation framework": discrete-time methods can be seen as a particular case of discrete-events methods [11] and continuous-time parts can be transformed straightforwardly to discrete-time/discrete-events systems.

- When using the Quantised State Systems approach of Kofman in order to transform a continuous system into a discrete system, there exists a closed formula for the global error bound [3], which allows a mathematical analysis of the simulation.

Since the mid seventies, when Zeigler introduced the DEVS formalism [18], there have emerged several DEVS implementations, most of them designed to simulate **discrete** systems. However, one simulation/modelling software systems that is aimed at simulating **continuous** systems, is PowerDEVS: it provides the components for modelling the block diagram representation of any system described by DAE's.
Given the fact that the implementation of the PowerDEVS functionality in Modelica perfectly matches our goal of providing Dymola with an additional simulation method for continuous systems, ModelicaDEVS is developed as closely as possible to PowerDEVS.

Although ModelicaDEVS is primarily an attempt at enhancing Dymola's integration competences by the benefits of DEVS integration using the synchronous information flow principle of the programming language Modelica, the new library can of course also be used for common discrete-event simulations that do not need to integrate anything.

[1] Note that this is not true for methods with dense output. However, the above statement holds for the majority of today's integration methods, since they rarely make use of dense output.

1.3 Structure of the Thesis

Chapter 2 gives an introduction to the DEVS formalism, in order to provide for sufficient knowledge about DEVS and related topics on which the subsequent chapters are based.
Chapter 3 examines the PowerDEVS software: how does it work and how has it been implemented. The implementation part is rather detailed and not necessarily needed in order to understand the rest of the thesis. It is only meant for a reader who is already familiar with object-oriented concepts and C++, but who would like to be given a "point of entrance" into the simulator framework code of PowerDEVS.
After these introductory sections, Chapters 4 and 5 discuss the ModelicaDEVS simulator and the library. More precisely, they present the transfer of the PowerDEVS functionality to Dymola, emphasizing the problems encountered during the implementation process, and give an individual description of each component in the ModelicaDEVS library.
In Chapter 6, the performance of ModelicaDEVS in comparison with Dymola and PowerDEVS is examined. Further testing regarding mixed and hybrid systems is presented, too.
Finally, Chapter 7 summarizes this thesis, discussing the achievements, but also pointing out encountered difficulties and possible improvements (open problems).

Chapter 2

Continuous Systems Simulation with DEVS

This chapter provides an introduction into discrete-event simulation in general, the DEVS formalism in particular, and reveals how discrete DEVS models can represent continuous systems.

2.1 Discrete-Event Simulation

In simulation theory, systems can be classified by the following criteria:

- Static/dynamic systems [1]: *"We define a static system to be one where the output $y(t)$ is independent of past values of the input $u(\tau)$, $\tau < t$ for all t. A dynamic system is one where the output generally depends on past values of the input. Thus, determining the output of a static system requires no "memory" of the input history, which is not the case for a dynamic system."* ... *"Differential or difference equations are generally required to describe the behaviour of dynamic systems."*

- Time-invariant/time-varying dynamic systems [1]: *"A system is said to be time-invariant if it has the following property: If an input $u(t)$ results in an output $y(t)$, then the input $u(t-\tau)$ results in the output $y(t-\tau)$ for any τ. In other words, if the input function is applied to the system τ units of time later than t, the resulting output function is identical to that obtained at t, translated by τ. This is illustrated in Figure 2.1, where the input $u(t)$ applied at time $t = 0$ results in the output $y(t)$. When a replica of the function $u(t)$ is applied as input at time $t = \tau > 0$, the resulting output is an exact replica of the function $y(t)$."* ... *"This property, also called stationarity, implies that we can apply a specific input to a system and expect it to always respond in the same way."*

- Deterministic/stochastic systems[1]: *"We define a system to be stochastic if at least one of its output variables is a random variable. Otherwise, the system is said to be deterministic. In general, the state of a stochastic dynamic system defines a random process, whose behaviour can be described only probabilistically. Thus, in a deterministic system with the input $u(t)$ given for all $t \geq t_0$, the state $x(t)$ can be evaluated. In a*

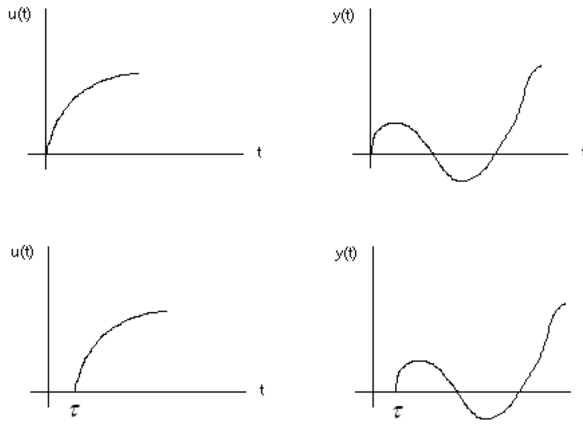


Figure 2.1: The time-invariant property [1].

stochastic system, the state at time t is a random vector, and it is only its probability distribution function that can be evaluated."

- Continuous/discrete systems [9]: *"A discrete system is one for which the state variables change instantaneously at separated points in time." ... "A continuous system is one for which the state variables change continuously with respect to time." ... "It should be mentioned that a discrete model is not always used to model a discrete system and vice versa."*

- Continuous-state/discrete-state systems [1]: *In continuous-state models the state space X is a continuum consisting of all n-dimensional vectors of real (or sometimes complex) numbers." ... "This normally leads to differential equations as in* (2.1) *and associated techniques for analysis.*

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t) \tag{2.1}$$

  In discrete-state models the state space is a discrete set. In this case, a typical sample path is a piecewise constant function, since state variables are only permitted to jump at discrete points in time from one discrete state value to another."

- Linear/nonlinear models [22]: *"A precise mathematical definition of a linear system is that superposition holds, where superposition for a system with any two inputs $x_1(t)$ and $x_2(t)$ is defined as*

$$\begin{aligned} \mathcal{H}\left[\alpha_1 x_1(t) + \alpha_2 x_2(t)\right] &= \alpha_1 \mathcal{H}\left[x_1(t)\right] + \alpha_2 \mathcal{H}\left[x_2(t)\right] \\ &= \alpha_1 y_1(t) + \alpha_2 y_2(t) \end{aligned}$$

  where $y_1(t)$ is the response of the system when input $x_1(t)$ is applied alone, and $y_2(t)$ is the response of the system when input $x_2(t)$ is applied alone; α_1 and α_2 are arbitrary constants."

Discrete-event simulation models belong to the group of discrete-dynamic models. The remaining classification properties mentioned above can occur in almost any combination (e.g. a continuous-state stochastic time-invariant nonlinear discrete-event system). The term "event"

stands for the instantaneous (asynchronous) changes of the state variables that are typical for discrete-dynamic systems. Thus, a change in the set of state variables is said to be caused by the occurrence of an event, and events can be defined as occurrences that take the system from the current state into the next one.
Cassandras and Lafortune [1] give the following definition of discrete-event systems: *"When the state space of a system is naturally described by a discrete set like 0,1,2,..., and state transitions are only observed at discrete points in time, we associate these state transitions with "events" and talk about a "discrete event system"."*

A very important issue in discrete-event simulations is the time-advance mechanism: dynamical systems usually include a variable, the simulation clock, that keeps track of the simulation time. The time-advance mechanism defines the way the simulation clock is advanced. In principle, there are two different approaches: next-event time advance and fixed-increment[1] time advance. Given that the latter approach is a special case of the former, only the next-event time-advance mechanism needs to be discussed in more detail.
At the beginning of the simulation, the simulation clock is set to zero. In order to start the simulation, the time instance of the first event is evaluated and the simulation clock is then advanced to the time when this particular event will occur. As stated before, the event takes the system to a new state. Again, as soon as the system is updated, the time instance of the imminent event is determined, now based on the new state the system has adopted meanwhile. The system clock is advanced in this manner until a termination condition is fulfilled. Note that generally, there is no direct relation between the elapsed simulation time at the end of a simulation and the actual time needed to run the simulation. In particular, such a relationship is inhibited by a time-advance mechanism as described above which may advance the simulation clock by arbitrarily long steps.
Due to the event-dependent time advance, only important simulation points regarding the dynamics of the system are simulated, and idle periods of the system (intervals where no events occur) are simply skipped. This is perfectly legal given that these periods do not contribute new information to the evolution of the system, anyway.
Fixed-increment time advance on the other hand would also simulate inactive periods and is therefore more time consuming (in terms of real computer time as opposed to simulation time).

Discrete-event systems may be modelled using (among other possibilities) Petri nets (introduced in the early sixties by Petri [17]), finite state machines [6], Markov chains [16], state charts (introduced in the late eighties by Harel [8]), or the DEVS formalism (introduced in the mid seventies by Zeigler [18]) which will be discussed in greater detail in the next section.

2.2 The DEVS Formalism

The origins of the DEVS formalism, ascribable to Bernard Zeigler, are rooted back in the mid seventies (1976). Because of Zeigler's interest in the continuous simulation of systems with frequent discontinuities [3] one of this new formalism's main purposes were to provide a common framework for continuous as well as for discrete-event models.

[1] Fixed-increment time advance corresponds to the traditional way of using time steps in order to simulate a continuous system on a digital computer

The DEVS formalism was among the first discrete systems theories with a mathematical background: while before the advent of the computer, continuous systems were mostly described by differential equations, with the growing computational power of computers discrete-event simulations were made possible but usually lacked a profound mathematical theory. Only roughly since the early seventies, it was recognized that deeper, i.e. mathematical, understanding of complex systems can help their efficient simulation, and research in this area began to be carried out. The DEVS formalism was one of the resulting mathematically well-founded methodologies in the area of discrete-event systems simulation.

During the past years, several implementations of the theoretical concept defined by the DEVS formalism have emerged: DEVSJava [21], DEVSim++ [10], PowerDEVS [13] and many others[2].

The next two sections will discuss the theory of the DEVS formalism as it has been defined by Zeigler. The first section covers the topic of atomic models which actually reveals the fundamental principles of the formalism. The second section discusses an "extension" of these ideas, namely a) the possibility of linking models to each others such that they form a coupled (multi-component) system and b) the hierarchic use of a coupled model as a component of another coupled model.

2.2.1 Atomic Models

A DEVS model has the following structure (see also [3], Chapter 11):

$$M = (X, Y, S, \delta_{int}(s), \delta_{ext}(s, e, x), \lambda(s), ta(s))$$

where the variables have the following meaning:

- X represents all possible inputs, Y represents the outputs and S is the set of states.
- The variable e (used within the dext function) indicates the amount of time the system has already been in the current state.
- $\delta_{ext}(s, e, x)$ is the external transition which is executed after having received an external event.
  As an example, assume the system adopted state 6 at time $t = 7.5$ and it receives an external event with the value 3.33 at time $t = 9$. Then, the new state is computed by $s_{new} = \delta_{ext}(6, 1.5, 3.33)$.
- $\delta_{int}(s)$ is the internal transition which is executed as soon as the system has elapsed the time indicated by the time-advance function.
  Example: If the system is in state 3 at a given time instant, the new state is computed by $s_{new} = \delta_{int}(3)$.
- ta(s) is the so called time-advance function which indicates how much time is left until the system undergoes the next internal transition. Note that the ta function is always dependent on the current state.

[2] See also the introductory part of [13] for additional examples

If for example at time $t = 2$ the system is in state 4, and the value of $ta(s = 4)$ is 3, the system will change its state autonomously at time $t = 5$. If on the other hand, an event arrives in the meantime (let us say at time $t = 4.5$) which puts the system into state 9, the new time when the system will change by itself is computed by $t_{new} = 4.5 + ta(s = 9)$. Note that the time-advance function manipulates the simulation time of a model "by hand". There is no global clock anymore that gives equidistant ticks. When the system's time-advance function adopts a certain value, let us say three, it means that the system will be idle for the next three units of time (provided the absence of external inputs), so it does not make any sense to simulate this period of inactivity, and the system clock is allowed to directly jump three time units forward.
The time-advance function is often represented by the variable σ which holds the value for the amount of time that the system still has to remain in the current state (in the absence of external events). If σ is present, the time-advance function merely returns σ.

- The λ-function is the output function. It is only active before internal transitions. Thus, external transitions do not produce any output.

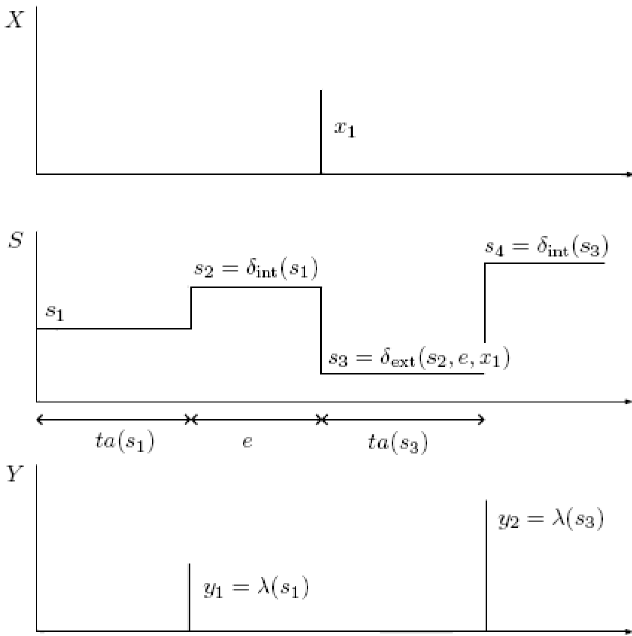


Figure 2.2: Trajectories in a DEVS model.

Figure 2.2 illustrates the mechanism of a DEVS model: the system receives inputs (see the uppermost graph) at certain time instants, changes its states according to the internal and external transitions, and gives outputs (see the lowermost graph). The graph in the middle shows the state trajectory of the system. Apparently, the system starts in state s_1 and changes to state s_2 after the time advance $ta(s_1)$ has been elapsed. Right before executing the internal transition, it generates the output y_1. When it receives the input x_1, it again changes its state, this time though undergoing the external transition. Note that no output is produced. When the time advance of state s_3 has been elapsed, it produces an output (y_2),

executes the internal transition again and thereby changes to state s_4.

Whereas figure 2.2 illustrates the DEVS formalism in a still rather abstract way, a short example of a DEVS model (taken from [3]) shall give further insight into the interplay of internal and external transitions.
Assume the following scenario: our model receives positive numbers at given time instants. Provided that it has received a number with value x at time t_{cur}, it gives an output with value $x/2$ at time $t_{cur} + 3 \cdot x$. Between the arrival of a number and the time instant the system gives an output, it shall not accept other inputs.
From these requirements we can deduce the need of the following model parts:

- A variable σ that represents the time until the next internal transition. Due to the presence of this variable, the time-advance function simply returns σ. This trick allows us to have a time-advance function that is not only dependent on the current state but also on the previous one. This is critical to the model described above since we have to adapt sigma between the acceptance of a new number x and the output that will follow $3 \cdot x$ time units later: right at the occurrence of this external event, σ is set to $3 \cdot x$. At every further external event during the period of $3 \cdot x$ time units, sigma is set to $sigma_{new} = sigma_{old} - e$ where e is the time that has elapsed since the last occurrence of an external event.

- An internal transition that sets σ to infinity, given that the system is driven by the numbers arriving from the outside.

- An external transition that checks whether the system is in its accepting phase (i.e. has not received a number the according output of which is still pending) and accepts or refuses the current input.

- A variable z that represents the phase of the system (∞ if the system is allowed to accept a new input and any other value if the system still has to give the output triggered by a previous occurrence of an external event). If z is not infinite, it carries the value of the last number accepted.
  Note that a state is defined by the two variable z and σ.

- An output function that returns the value of $x/2$ (or $z/2$ since z represents the last accepted number) at appropriate time instants.

Assembling all these parts leads to the following model definition:

$$\begin{aligned}
&X = Y = \mathbb{R}^+ \\
&S = \mathbb{R}^+ \times \mathbb{R}_0^+ \\
&\delta_{int}(s) = \delta_{int}(z, \sigma) = (\infty, \infty) \\
&\delta_{ext}(s, e, x) = \delta_{ext}(z, \sigma, e, x) = \tilde{s} \\
&\lambda(s) = \lambda(z, \sigma) = z/2 \\
&ta(s) = ta(z, \sigma) = \sigma
\end{aligned}$$

where

$$\tilde{s} = \begin{cases} (x, 3 \cdot x) & \text{if } z = \infty \\ (z, \sigma - e) & \text{otherwise} \end{cases}$$

Let us assume the following inputs: at time t= 1: x=2,
at time t= 3: x=1,
at time t=10: x=5.

Then, the dynamics[3] and the outputs of our model are given below:

time t=0 : $s = (\infty, \infty)$
$e = 0$
$ta(s) = ta(\infty, \infty) = \infty$

time t=1^- : $s = (\infty, \infty)$
$e = 1$
time t=1 (ext) : $s = \delta_{ext}(s, e, x) = \delta_{ext}(\infty, \infty, 1, \mathbf{2}) = (\mathbf{2}, 3 \cdot \mathbf{2})$
time t=1^+ : $s = (2, 6)$
$e = 0$
$ta(s) = ta(2, 6) = 6$

time t=3^- : $s = (2, 6)$
$e = 2$ *The system has been in this state since t=1.*
time t=3 (ext) : $s = \delta_{ext}(s, e, x) = \delta_{ext}(2, 6, 2, \mathbf{1}) = (2, 6 - 2)$
time t=3^+ : $s = (2, 4)$
$e = 0$
$ta(s) = ta(2, 4) = 4$

time t=7^- : $s = (2, 4)$
$e = 4$ *The system has been in this state since t=3.*
time t=7 (int) : output with value $\lambda(s) = \lambda(2, 4) = 2/2 = 1$
$s = \delta_{int}(s) = \delta_{int}(\infty, \infty) = (\infty, \infty)$
time t=7^+ : $s = (\infty, \infty)$
$e = 0$
$ta(s) = (\infty, \infty) = \infty$

time t=10^- : $s = (\infty, \infty)$
$e = 3$ *The system has been in this state since t=7.*
time t=10 (ext) : $s = \delta_{ext}(s, e, x) = \delta_{ext}(\infty, \infty, 2, \mathbf{5}) = (\mathbf{5}, 3 \cdot \mathbf{5})$
time t=10^+ : $s = (5, 15)$
$e = 0$
$ta(s) = (5, 15) = 15$
...

[3] The declarations "(ext)" and "(int)" indicate the reason why the system changes its state in that particular time step. According to the DEVS formalism, this is possible either because of an external input or because the system has elapsed the time-advance period, and an internal transition has to be invoked.
The time steps are divided in three parts each: a part that represents the state of the system right before the execution of the transition function (indicated by "−"), a part that shows the actual transition of the system, and a third part that gives the variables right after the transition has taken place (indicated by "+").

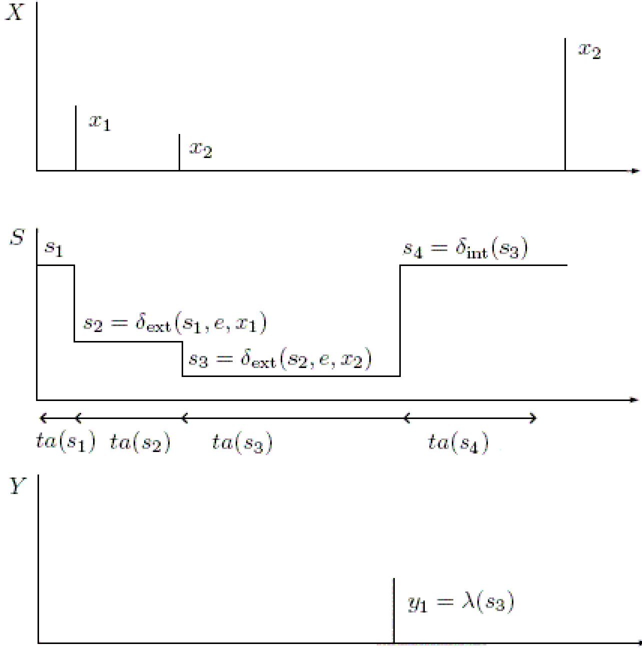


Figure 2.3: Trajectories in the example DEVS model.

The fact that there is no evaluation during time $t = 2$ until $t = 5$ and $t = 6$ until $t = 9$ is one of the benefits of DEVS: as long as the system remains in the same state, no evaluations take place and thus computational costs can be saved.

Figure 2.3 illustrates the dynamics described above.

2.2.2 Coupled Models

DEVS models can describe arbitrarily complex systems. The only drawback is that the more complex the system is, the more difficult it is to set up the correct functions (δ_{int}, δ_{ext}, time-advance and λ-function) that describe the system. Fortunately, most complex systems can be broken down into simpler submodels that are easier to handle. It is therefore convenient not to have just one model that tries to simulate the whole system, but an interconnection of a number of smaller models that, together, represent the system. Even though this approach seems to be very intuitive, it is only possible because DEVS models are closed under coupling [3], which means that a coupled model can can be described by the same functions as an atomic model (internal, external, time-advance and lambda function). Due to this fact, coupled models can be seen as atomic models, too, which allows for hierarchical modelling.
Figure 2.4 illustrates these ideas: the model N consists of two coupled atomic models M_a and M_b. N can be seen as an atomic model itself and it is therefore possible to connect it to further models (coupled or atomic ones).
In such a network where submodels interact with each other through ports, so that output events of one submodel are transformed automatically into input events for other submodels, events do not only carry the output value but also a number indicating the port where the

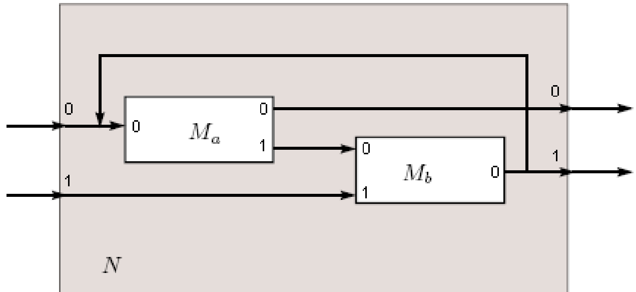


Figure 2.4: Coupled DEVS models [3].

event is appearing. Hence, inputs and outputs take the following form:

$$X = Y = \mathbb{R} \times \mathbb{N}_0$$

Figure 2.4 suggests this circumstance by numbering the input and output ports of the models. Besides "normal" connections between submodels (M_a and M_b in our case), there are also connections that lead to or come in from the outside: as already mentioned, model N can be seen as an atomic model with two input and two output ports. Events coming in through port number 0 are immediately forwarded to submodel M_a, those from port number 1 to M_b. This kind of connection is called external input connection. Analogous, the connections leading from the output ports of the submodels to the output ports of the surrounding model N are called external output connections.

The behaviour of model N is determined by the behaviour of its submodels M_a and M_b. The actual task of N is to wrap M_a and M_b in order to make them look like as if they were one single model. The dynamics of N (or any multi-component/coupled model) can be defined by the following loop [13]:

1. Evaluate the atomic model that is the next one to execute an internal transition. Let this model be called d^* and let tn be the time when the transition has to take place. In case there are more than one candidates ready to execute their internal transitions, choose the one that is designed by the so called tie-breaking function[4].

2. Advance the simulation time (of the coupled model) to $t = tn$ and let d^* execute the internal transition. Note that $tn = t_{cur} + ta(s_{cur})$.

3. Forward the output event produced by d^* to all atomic models that are connected to d^* and let them execute their external transitions.
   Go to step 1.

While until now we have justified coupled models by the need of splitting up too complex models into smaller, simpler submodels, it has of course also the reverse effect of allowing the predefinition of a number of atomic models (a "model pool" of frequently used models such as switches, comparators, etc.), which then can be used to build more complex models.

[4]The tie-breaking function imposes an ordering onto the components of the model, such that in case of two components ready to execute their internal transition, the simulation knows which one to process first.

In Chapter 3 we shall see a concrete realisation of this possibility.

2.3 DEVS Integration with QSS

This section – the core section of the current chapter – explains a) why the DEVS formalism is suited to simulate a continuous system and b) how this is done in a correct way preserving the DEVS formalism's rule of legitimacy.

2.3.1 Discrete Simulation of a Continuous System

For a system to be representable by a DEVS model, the only condition is to show an input/output behaviour that is describable by a sequence of events. In other words, the DEVS formalism is able to model any system with piecewise constant input/output trajectories, since piecewise constant trajectories can be described by events [3].
Hence, if we want to simulate a continuous system by a DEVS model, we "just" have to transform the continuous system into a system with piecewise constant input/output trajectories. This was the basic idea of a quantisation based method developed in the late nineties by Zeigler et al. [20] to approximate a continuous system by a discrete-event simulation: a quantisation function[5] was used to transform the continuous state variables into quantised discrete valued variables.
Let us see now what it needs to quantise a continuous system. Consider the following system given in its state-space representation (remember that usually, continuous systems are described by a set of differential equations):

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t)$$

where $\mathbf{x}(t)$ is the state vector and $\mathbf{u}(t)$ is the input vector, i.e. a piecewise constant function. The corresponding quantised state system has the following form:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t), \mathbf{u}(t), t)$$

where $\mathbf{q}(t)$ is the (componentwise) quantised version of the original input vector $\mathbf{x}(t)$, whereas a very simple quantisation function could be

$$\mathbf{q}(t) = floor(\mathbf{x}(t)).$$

Figure 2.5 shows the respective block diagrams for the systems defined above. Note that the representation by block diagrams will be useful as soon as we will want to map the system onto a (coupled) DEVS model.

Unfortunately the subject is not as simple as it may seem: the transformation of a continuous system into a discrete one by applying an arbitrarily chosen quantisation function can yield an illegitimate system[6], performing an infinite number of transitions in a finite time interval.

[5] A quantisation function maps all real numbers into a discrete set of real values.

[6] Definition [3]: *"A DEVS model is said to be legitimate if it cannot perform an infinite number of transitions in a finite interval of time."* Illustrative examples of cycling (illegitimate) systems can be found in [11] and [3].

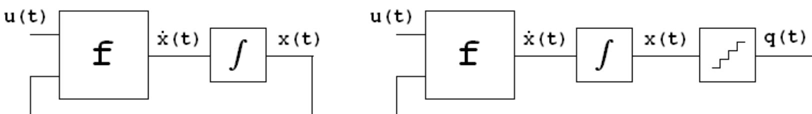


Figure 2.5: Block diagrams for a continuous-time and a quantised continuous-time system.

As a matter of fact, even Zeigler's quantisation based method featured this problem, as the original approach of using a piecewise constant quantisation function did not preclude illegitimate systems automatically [11]. Thus, the quantisation function has to be chosen very carefully, such that it prevents the system from switching states an infinite number of times in a finite time interval. This property can be achieved by adding hysteresis to the quantisation function (proven in [11]), which leads straightforward to the notion of Quantised State Systems that have been introduced by Kofman in 2001 in order to circumvent the issue of illegitimate system transformations.

The subsequent paragraphs shall give a brief introduction into the theory and use of Quantised State Systems as a (sufficient) means to approximate continuous systems by discrete events. Special attention is paid to the role of hysteresis in the quantisation function.

2.3.2 Quantised State Systems (QSS)

Quantised State Systems are defined as follows [11]: *"QSS are continuous time systems where the input trajectories are piecewise constant functions and the state variable trajectories – being themselves piecewise linear functions – are converted into piecewise constant functions via a quantisation function equipped with hysteresis."*

The goal of Quantised State Systems is to provide a legitimate system that can be simulated by the DEVS formalism. These two properties are achieved by a) the fact that the input/output trajectories are piecewise constant functions, which allows the simulation by the DEVS formalism (see Section 2.3.1), and b) the addition of hysteresis to the quantisation function by which the continuous system is transformed into the discrete representation.

A hysteretic quantisation function is defined as follows [3]: Let $Q = Q_0, Q_1, ..., Q_r$ be a set of real numbers where $Q_{k-1} < Q_k$ with $1 \leq k \leq r$. Let Ω be the set of piecewise continuous trajectories and let $x \in \Omega$ be a continuous trajectory. The mapping $b : \Omega \rightarrow \Omega$ is a hysteretic quantisation function if the trajectory $q = b(x)$ satisfies:

$$q(t) = \begin{cases} Q_m & \text{if } t = t_0 \\ Q_{k+1} & \text{if } x(t) = Q_{k+1} \wedge q(t^-) = Q_k \wedge k < r \\ Q_{k-1} & \text{if } x(t) = Q_k - \epsilon \wedge q(t^-) = Q_k \wedge k < 0 \\ q(t^-) & \text{otherwise} \end{cases}$$

and

$$m = \begin{cases} 0 & \text{if } x(t_0) < Q_0 \\ r & \text{if } x(t_0) \geq Q_r \\ j & \text{if } Q_j \leq x(t_0) < Q_{j+1} \end{cases}$$

The discrete values Q_i and the distance $Q_{k+1} - Q_k$ (usually constant) are called the quantisation levels and the quantum respectively. The boundary values Q_0 and Q_r are the upper and the lower saturation values, and ϵ is the width of the hysteresis window. Figure 2.6 shows a quantisation function with uniform quantisation intervals.

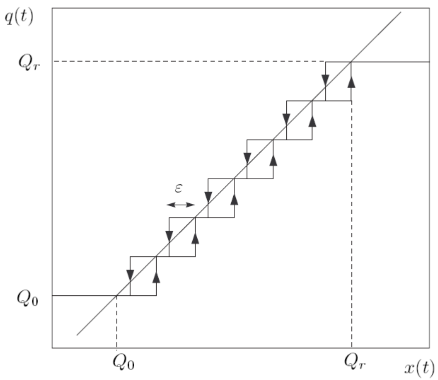


Figure 2.6: Quantisation function with hysteresis [3].

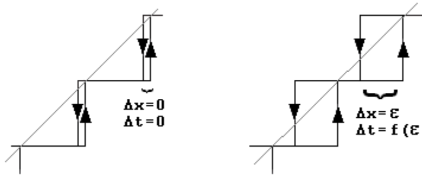


Figure 2.7: Difference between a quantisation step without and with hysteresis.

Figure 2.7 reveals the difference between a quantisation step without and with hysteresis: the left part in Figure 2.7 represents a quantisation function without hysteresis that may change the value of $q(t)$ due to an infinitesimal variation of the state variable $x(t)$ ($\Delta x = 0$), which entails of course a potential change of $q(t)$ within the same time step ($\Delta t = 0$). The right part embodies a hysteresis window of width ϵ and thus the change of $q(t)$ is delayed, i.e. only performed when $x(t)$ has changed to a sufficient extent (defined by ϵ: $\Delta x = \epsilon$) which introduces a time delay depending on how long it takes $x(t)$ to change for the given extent ($\Delta t = f(\epsilon)$).
A formal prove why adding hysteresis to a quantisation function guarantees a legitimate system transformation is given in [11].

The Quantised State System described above is a first-order approximation of the real system trajectory. Kofman however has also introduced second- and third-order approximations that – without the application of smaller "step" sizes (i.e. a smaller quantum value) – may reduce the error made by the approximation. These systems are referred to as QSS2 and QSS3. The higher-order Quantised State Systems are based on the Taylor series up to first-

or second-order (for QSS2 and QSS3, respectively).
Since the possibility of having different types of QSS has only been mentioned in view of Chapter 4, they are not discussed any further here. However, a detailed description of the QSS2 and the QSS3 can be found in [12] and [14] respectively.

Chapter 3

PowerDEVS

PowerDEVS offers a comfortable way for building and simulating models specified according to the DEVS formalism. A short but accurate description of the software's main achievements is given in the abstract of "PowerDEVS: A DEVS-Based Environment for Hybrid System Modeling and Simulation" [13]: *"PowerDEVS allows defining atomic DEVS models in C++ language which can be then graphically coupled in hierarchical block diagrams to create more complex systems. Both, atomic an coupled models, can be organized in libraries which facilitate the reusability features. The environment automatically translates the graphically coupled models into a C++ code which executes the simulation."*

This chapter illustrates the simulation mechanism of PowerDEVS, in order to be able to compare it to the mechanism of ModelicaDEVS that will be presented in the next chapter.

3.1 The Modelling Environment

Like many other simulation software systems, PowerDEVS provides a graphical editor (see Figure 3.1) where models can be built graphically. Note that it would be possible to declare them directly in C++ code. This however would be rather cumbersome and error-prone, and using the automatic transformation of PowerDEVS is recommended.
Since the handling of the modelling environment is fairly intuitive and anyway not critical to the understanding of the simulator which will be discussed in the next section, it is not explained in greater detail here.

An interesting subject to dwell some more on, however, is the way the graphical representation of a model is transformed into C++ code, in order to be utilisable by the simulator framework. Assume the sample model in Figure 3.1. When the "run" button in the modelling environment is pressed, PowerDEVS creates a file called `model.h` that holds a C++ description of the model. All information that will be needed for the simulation later on is stored in this file. The following code corresponds to the model in Figure 3.1 (note that for simplicity, the output components "ToDisk1", "ToDisk2", "ToDisk3" and "QuickScope1" have been removed):

```
1 #include "textonly/simulator.h"
2 #include "textonly/root_simulator.h"
3 #include "textonly/connection.h"
4 #include "textonly/coupling.h"
```

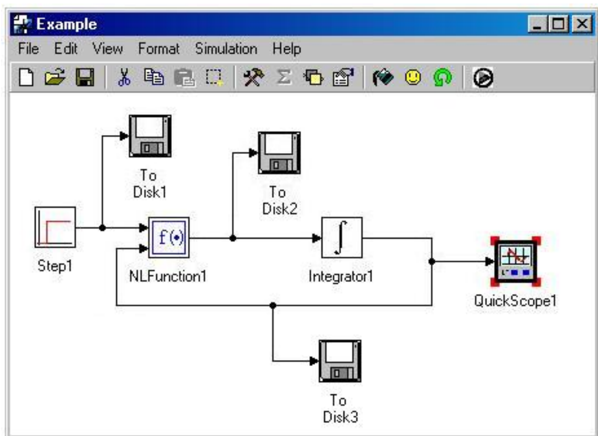


Figure 3.1: The graphical editor of PowerDEVS with an example model.

```
 5 #include "textonly/root_coupling.h"
 6 #include <Source\step13.h>          //include the Step block
 7 #include <Continuous\nlfunction.h>  //include the NLfunction block
 8 #include <Continuous\integrador.h>  //include the integrator
 9
10 class model:public root_simulator {
11
12  root_coupling* Coupling0001;
13  simulator* child00010001;   //step
14  simulator* child00010002;   //NLfunction
15  simulator* child00010003;   //integrator
16  connection* EIC0001[1];     //external input connections
17  connection* EOC0001[1];     //external output connections
18  connection* conn00010001;
19  connection* conn00010002;
21  connection* conn00010003;
21  connection* IC0001[3];      //array containing all connections between blocks
22  simulator* D0001[3];        //array containing all blocks
23
24  public:
25
26  model(bool* r):root_simulator(r){}    //the model allocated in main() is a root_simulator
27
28  void configure(){    //this function will be called from the main() function in model.cpp
29
30    Coupling0001=new root_coupling();   //the "global" coupling
31
32    child00010001= new step13();        //allocate a new Step block
33    D0001[0]=child00010001;             //insert the Step block into block array
34    child00010002= new nlfunction();
35    D0001[1]=child00010002;
36    child00010003= new integrador();
37    D0001[2]=child00010003;
38
39    conn00010001= new connection();       //allocate a new connection
40    conn00010001->setup(0 , 0 , 1 , 0);   //connection from step to NLfunction, port 0
41    IC0001[0]=conn00010001;               //insert into array of connections
42    conn00010002= new connection();
```

```
43    conn00010002->setup(1 , 0 , 2 , 0);   //connection from NLfunction to integrator
44    IC0001[1]=conn00010002;
45    conn00010003= new connection();
46    conn00010003->setup(2 , 0 , 1 , 1);   //connection from Integrator to NLfunction, port 1
47    IC0001[2]=conn00010003;
48
49    //setup coupling, passing the block and connection arrays (and their sizes)
50    Coupling0001->setup(&D0001[0], 3, &IC0001[0], 3);
51
52    MainCoupling = Coupling0001;       //set MainCoupling
53    Coupling0001->rsim=this;
54    ti = 0;                            //simulation start time
55    tf = 10;                           //simulation final time
56
57    //initialize blocks with the appropriate values for their parameters
58    child00010001->init(0, 0.0,1.76,10.0);
59    child00010002->init(0, ‘‘-u1+u0’’, 2.0);
60    child00010003->init(0, ‘‘QSS’’,1.0,10.0,’’integrator.csv’’);
61    Coupling0001->init(0);
62  }
63 };
```

Figure 3.2: model.h, created according to the current simulation model

As a last point worth mentioning, there is the incorporation of the tie-breaking function (see Section 2.2) by the so called Priority Window[1], which allows the modeller to specify a particular priority order among the components (see Figure 3.3).

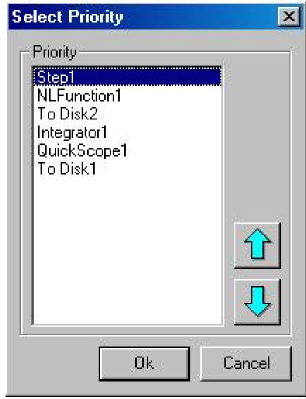


Figure 3.3: The priority window for the model in Figure 3.1.

3.2 The Simulator

The PowerDEVS simulator framework enables the user to simulate a system that is available in the C++ format shown in Figure 3.2. Normally, this is the result of modelling it graphically in the PowerDEVS graphical editor followed by an automatic transformation into the

[1]Accessible through the menu Edit, Priorities

aforementioned C++ data structures.
The subsequent sections will give insight into the simulator architecture both on a more theoretical level as well as on the level of its actual implementation in C++.

3.2.1 Simulator Theory

The PowerDEVS simulator is based on the abstract simulator concept developed by Zeigler [19]. To make allowance for the possibility of coupled/hierarchical DEVS systems, PowerDEVS models embody the structure shown in Figure 3.4, whereas simulators represent atomic models and coordinators represent coupled models.

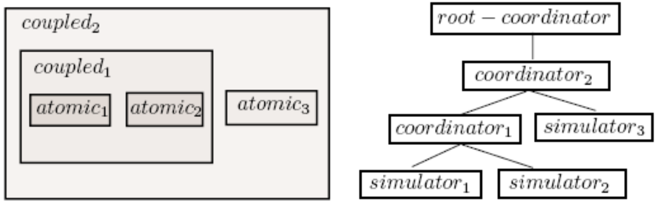


Figure 3.4: Hierarchical model and according simulation scheme [13].

As soon as there are two or more components constituting the model, it is necessary to make them able to communicate with each other (send and receive output/input signals). The most intuitive approach would probably be the following: each block keeps a list of the blocks that are connected to its output port, and whenever the block produces an output event, it sends a message to the connected blocks in order to make them undergo their external transition. It is however not the approach chosen in Zeigler's abstract simulator/PowerDEVS. PowerDEVS rather leaves the control of the interaction within a set of blocks to the coordinator that is declared to be responsible for these particular blocks. If for example block A has to generate an output event that has to be sent to block B, it is the coordinator that a) sends a message[2]) to block A in order to trigger the λ-function and b) passes the generated output event to the block B. Of course, both approaches (the intuitive one as well as the one applied in PowerDEVS) lead to the same result. Thus, a modeller does not have to think of the complicated coordinator-simulator concept but may consider the blocks to act autonomously ("block A decides to produce an event and sends it to block B").

Let us study the PowerDEVS solution to the component interaction issue in more detail. Figure 3.4 depicts the possible hierarchical structure and the corresponding simulation scheme of a DEVS model. The messages between coordinators and simulators up and down the tree consist of the following two types:

- Down-messages: coordinators send messages to their children, triggering the execution of the different functions (δ_{int}, δ_{ext}, λ-function).

[2]Called "lambdamessage" in PowerDEVS

- Up-messages: when the λ-function of a simulator has been called by the coordinator, it returns the output value to its parent coordinator (the caller).
  Note that coordinators look like simple simulators to their parent coordinators. Hence, if a coordinator (e.g. $coupled_1$ in Figure 3.4) receives an output value that has to be propagated outside the corresponding coupled model, it sends this value to its own parent coordinator ($coupled_2$).

We can see that simulators do not interact directly with other simulators on the same layer, but only pass their outputs to the parent coordinator which then is responsible for the propagation of the signal to the appropriate simulators (i.e. those which are connected to the "firing" simulator). The communication between simulators can thus be said to be monitored by the associated coordinator.

Since both coordinators and simulators have to be valid DEVS models, they have to feature the typical DEVS functions: δ_{int}, δ_{ext}, the time-advance and the λ-function. In simulators, these functions simply define the inherent behaviour of the block in question. In the case of a coordinator however, they have a slightly different meaning/effect:

- External transition – the purpose of the external transition of a coordinator is to invoke external transitions in the appropriate simulators: When a coordinator receives a signal, it simply forwards it to those among its children that have their input ports connected to the input ports of the coordinator (see Figure 3.5), and thereby triggers these simulators to execute their external function.

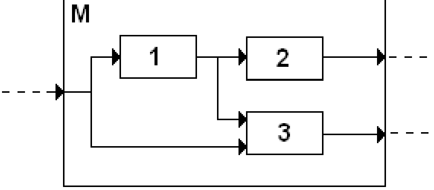


Figure 3.5: The signal from the outside world to the model M will be forwarded to submodel 1 and 3 but not to 2.

- Internal transition – the purpose of the internal transition of a coordinator is to invoke internal transitions in the appropriate simulators: All simulators and coordinators have a local variable *tn* which indicates the time when their next internal transition will take place. In a simulator, this variable corresponds simply to the sum of the current time and the value of the time-advance function (or σ). The value of *tn* in a coordinator on the other hand is the minimum of the *tn* values of its associated components, thus indicating the next time instant when a simulator undergoes its internal transition. Besides the time **when** the transition is executed, the coordinator also stores the information **where** (at which simulator) it takes place. For this reason, it features a second variable *dast* (corresponding to d^*, see Section 2.2.2) where it stores the ID of the respective simulator. Having this information at its disposal, the internal transition of a coordi-

nator just consists of sending an internal-transition-message[3] to the simulator stored in *dast*, which then will execute its own internal transition.
Note that in order to always have an updated pair *dast*/*tn* available, a coordinator scans its children every time it receives the instruction to execute an internal or external transition, and stores the block with the smallest time-advance into *dast* and the corresponding time-advance value into *tn*.

Let us investigate now, how a signal moves through the tree of coordinators and simulators. Consider a hierarchy like the one in Figure 3.6 (same topology as the example in Figure 3.4 but with directed connections).
Suppose that $coupled_2$ has just noticed that $atomic_3$ is ready to undergo its internal transition (apparently, it has the smallest *tn* value). For this reason, $coupled_2$ sends a message to $atomic_3$, which then executes its internal transition. Instead of sending the output directly to $coupled_1$ (which looks like another simulator to $atomic_3$), it returns it to its father coordinator, $coupled_2$. $coupled_2$ checks now what components are attached to the output port of $atomic_3$ and finds $coupled_1$, to which it propagates the output of $atomic_3$. $coupled_1$ has now just received an external event that makes it execute an external transition. Since $coupled_1$ is a coordinator, this transition just consists of forwarding the signal to the simulators connected to its input ports. In our example, this is only $atomic_1$, which now executes its external transition and thereby sets its local variable *tn* to a new value. As mentioned before, when a coordinator ($coupled_1$ in this case) forwards an external event, it scans the *tn* values of its children in order to update *dast* and its own *tn* value. Note that this update already involves the new *tn* value of $atomic_1$. The next step depends now on whether $atomic_1$ or $atomic_2$ has the smaller *tn* value and therefore is the imminent component to execute its internal transition.

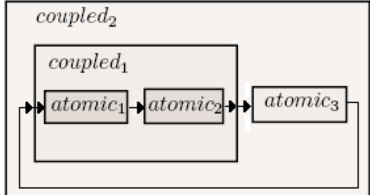


Figure 3.6: Coupled model with explicit input/output directions.

So much for the theoretical background of the PowerDEVS model simulator. A sample simulation run shall illustrate the event flow between the components of a model that is simulated according to the theory presented above.

Assume again the system in Figure 3.1 (without the output blocks for the first section). Note that this is a flat simulation model without multi-component submodels.
Besides the graphical presentation of simulation results (as shown in Figure 3.7), PowerDEVS also provides ASCII files that contain a detailed listing of the output events of a certain component: for every ToDisk output block that is attached to a connection between two blocks,

[3]Called "dintmessage" in PowerDEVS.

such an ASCII files is created and lists the events that are sent over the respective connection. While of course the graph allows an overall understanding of the simulation results, the ASCII file is more precise since it lists the full content of events (both the output value and the event creation time). Thus, if we “snoop” the connections between blocks by inserting a ToDisk output component, we can learn a lot about the event flow between the different components.

Figure 3.7: The result of the model in Figure 3.1. Once plotted as a stair function, once with the output points linearly connected.

As can be seen (Figure 3.1), there are three blocks that participate in the actual computation of the final result (Step, NLfunction and the Integrator) plus four blocks used to produce output (ToDisk1, ToDisk2, ToDisk3 and QuickScope1). The output blocks do not contribute to the simulation result and therefore are not treated here, but only used as a means to analise the interaction of the other three components.
Let us start our investigation with the simplest one, the Step block. Its parameters are set in such a way that its value is 0 in the beginning and changes to 10 at the step time t=1.76. Its output events look like this (stored by ToDisk1):

0, 0
1.76, 10

Which means that it sends a signal (event) through its output port at time instants $t = 0$ and $t = 1.76$. At $t = 0$, the signal value is 0, whereas at $t = 1.76$ it is 10. These events will be seen at the input port of the NLfunction block.
Note that since the Step block is a source component, it is completely autonomous and does not depend on other blocks. Therefore it does not have an external transition because it does not receive external events.

As a second and third component, there are the Integrator and the NLfunction block which produce the following output, respectively (stored by ToDisk2 and ToDisk3):

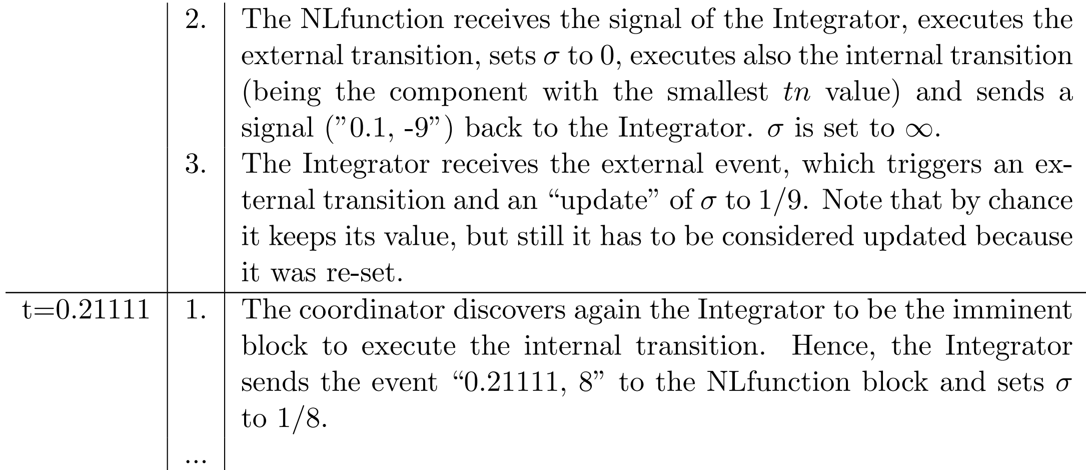
Integrator		NLfunction	
0,	10	0,	0
0.1,	9	0,	-10
0.211111,	8	0.1,	-9
0.336111,	7	0.211111,	-8
0.478968,	6	0.336111,	-7
0.645635,	5	0.478968,	-6
0.845635,	4	0.645635,	-5
1.09563,	3	0.845635,	-4
1.42897,	2	1.09563,	-3
1.96776,	3	1.42897,	-2
2.11062,	4	1.76,	8
2.27728,	5	1.96776,	7
2.47728,	6	2.11062,	6
2.72728,	7	2.27728,	5
3.06062,	8	2.47728,	4
3.56062,	9	2.72728,	3
4.56062,	10	3.06062,	2
		3.56062,	1
		4.56062,	0

The Step block and the Integrator are self-starting blocks, which means that the time-advance function (usually represented by the variable σ) is initialised to 0 at the beginning of the simulation, such that they execute an internal transition right away, even before they receive an external signal.
The NLfunction block on the other hand is not self-starting (σ is set to infinity in the beginning). It has to wait for an external event to arrive, which will trigger an external transition accompanied by a first update of the local variables.
The reason why the NLfunction block generates two outputs at time $t = 0$ is that it receives two signals, one coming from the Step block and one coming from the Integrator. To both signals it responds by first executing the external and right afterwards its internal transition, producing an output each time. Which of the signals – the one from the Integrator or the one from the Step block – is processed first, is determined by the priority settings of the model (modifiable by means of the Priority Window of the graphical editor, see Figure 3.3). In our case the ordering was set to Step - NLfunction - Integrator. Had we set it to Step - Integrator - NLfunction, the NLfunction would have generated just one event at $t = 0$ since the signal from the Step block would have been overwritten by the signal of the Integrator.

For a better understanding of the communication mechanism, let us see how the above listed events have been generated. To this end, we track the first few steps of the simulation, turning our attention to the interaction between the three blocks in the model. Note that up to this point, we do not know anything yet about the actual internal and external transitions, which means that we cannot retrace by what algorithms the output events or σ have been assigned their values. This is however not the subject of the current section (see Chapter 4 or the ModelicaDEVS/PowerDEVS implementations for details) and the reader is requested to assume subsequent indications regarding variable values to be correct (as a matter of fact,

they have been taken from a real simulation run).

Time		**Event**
t=0	1.	The Step block and the Integrator are ready to execute their internal transition. Because of the priority setting (Step - NLfunction - Integrator), the coordinator of our three blocks allows only the Step block to proceed, while the Integrator has to wait. Hence, the Step block sends the event “0, 0 “ to the NLfunction block and thereafter sets σ to 1.76, the time instance when the change of its output value has to take place.
	2.	The NLfunction receives the signal, executes the external transition and sets σ to 0. The NLfunction and the Integrator are now ready for their internal transition. Again, the priority setting makes the Integrator wait once more and elects the NLfunction to be the next block to execute its internal transition. Therefore, the NLfunction block generates an event “0, 0”, which is sent to the Integrator. During the internal transition, NLfunction sets σ to infinity.
	3.	The Integrator receives the signal from the NLfunction and has now an internal **and** an external transition to execute. In such a case – due to the way PowerDEVS has been implemented (see Section 3.2.2) – the external transition gets executed first. Within this transition, the Integrator does not modify σ, which therefore keeps its initial value of 0. When the tn values of the three blocks are compared once more, the Integrator finally turns out to be the component with the smallest tn value (compared to the Step and the NLfunction block with a tn of 1.76 and ∞ respectively). Hence, the Integrator is allowed to execute the internal transition, sending an output (”0, 10”) to the NLfunction block and setting σ to infinity.
	4.	The NLfunction receives the signal from the Integrator, executes again first the external transition, sets σ to 0, thereby becomes the block with the smallest tn value, is therefore allowed to execute the internal transition and sends the event “0, -10” to the Integrator. σ is set to infinity.
	5.	The Integrator receives the external event from the NLfunction, executes the internal transition, but this time sets the value of σ to 0.1. Given that none of the three blocks are currently ready for an internal transition, the simulation clock has to be advanced to the time when the next transition is scheduled to take place, which is at $t = 0.1$ (the current tn values of the three blocks are: $tn_{NLfunction} = \infty$, $tn_{Step} = 1.76$ and $tn_{Integrator} = 0.1$).
t=0.1	1.	The coordinator discovers the Integrator to be the imminent block to execute the internal transition. Thus, an event “0.1, 9 “ is sent from the Integrator to the NLfunction block. σ is set to 1/9.



	2.	The NLfunction receives the signal of the Integrator, executes the external transition, sets σ to 0, executes also the internal transition (being the component with the smallest *tn* value) and sends a signal ("0.1, -9") back to the Integrator. σ is set to ∞.
	3.	The Integrator receives the external event, which triggers an external transition and an "update" of σ to 1/9. Note that by chance it keeps its value, but still it has to be considered updated because it was re-set.
t=0.21111	1.	The coordinator discovers again the Integrator to be the imminent block to execute the internal transition. Hence, the Integrator sends the event "0.21111, 8" to the NLfunction block and sets σ to 1/8.
	...	

Starting from here, the procedure described for the time step $t = 0.1$ is repeated until the time step $t = 1.76$ when the Step block switches its output from the defined start value (0) to the final value (10), thereby sending an event to the NLfunction block. This breaks the cycle of the Integrator and the NLfunction block for a moment – but only to be resumed as soon as the signal coming from the Step block is processed:

t=1.76	1.	The Step block sends a signal carrying the values "1.76, 10" to the NLfunction and sets σ to infinity.
	2.	The NLfunction executes the external transition and immediately afterwards the internal transition, which produces an output signal ("1.76, 8") targeted at the Integrator.
	3.	The Integrator notices the signal, goes through its external transition and sets σ to a new value, 0.20776.
t=1.96776	1.	Analogous to the time step $t = 0.1$, the coordinator spots the Integrator as the block with the smallest *tn* value. Hence, the Integrator sends the event "1.96776, 3" to the NLfunction block and sets σ to 1/8.
	2.	The NLfunction is triggered to execute its external transition, which leads to an internal transition and an output event "1.96776, 7" right afterwards.
	3.	The signal reaches the Integrator that executes the external transition and sets the variable sigma to the value 0.142857.

Continuing from here, it will be again a cycle between the NLfunction and the Integrator.

An important point to mention explicitly is that due to the topological loop in our model (NLfunction-Integrator) there may arrive several events at the same block within one time step. Nevertheless, the amount of such event occurrences is limited as a system is not permitted to produce an infinite number of events within a finite time interval (DEVS rule of legitimacy (cf. [3] and Section 2.3.2)).
The situation of a system looping for a certain number of times is caused by two or more blocks connected in a cycle that update their local variable σ to the value of 0 during their

external transition. The loop is broken by any block that sets σ to a non-zero value.

So far we have seen the theoretical principles behind the PowerDEVS model simulator and how a simulation is carried out. Let us now switch to the more technical part, revealing what entities (in terms of C++ classes) constitute the simulator framework and how they collaborate with each other during a simulation run.

3.2.2 Simulator Architecture

Remark: this section is not necessarily needed to understand the functioning of the simulator. It is however recommended to the reader who a) is interested in a more technical description of the simulator framework and b) is preferably already fairly familiar with the concept of polymorphism and object-oriented languages such as C++ in general.

The core classes of the PowerDEVS simulator are shown in Figure 3.8, which illustrates their dependencies and relationships, pointing out the variables that are most important regarding the understanding of the polymorphic, hierarchical nature of the simulator framework. The following paragraphs will present these classes in greater detail.

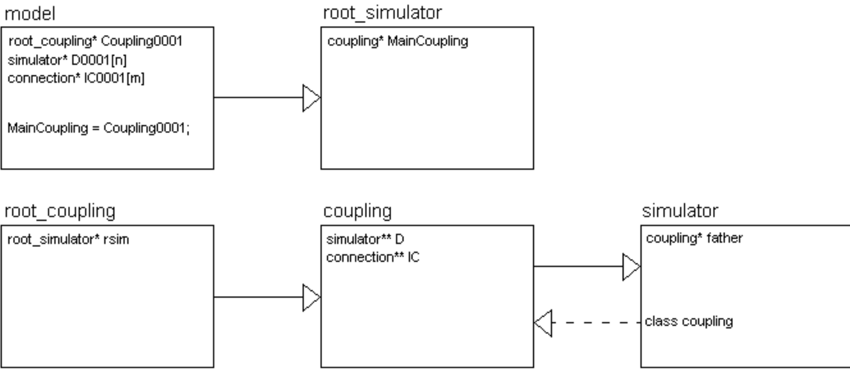


Figure 3.8: Class inheritance and relations.

coupling.cpp: Couplings represent coupled models (analogous to the coordinators in Figure 3.4). They are meant to coordinate a number of atomic models: a coupling is composed of simulators (blocks) or – for the sake of recursivity – couplings again.
In order to be able to fulfill the task of coordinating their children, couplings need a) to have a list of all associated simulators (their children) and b) to know about the connections linking their children to each other, or leading inside and outside the coupling (external input and output connections). To this end, the C++ coupling class features a pointer to an array of the type `simulator` where all simulators or couplings are stored, and another pointer to an array of the type connection where all connections are stored (see lines 4+5 in the following code snippet).

```
 1 class Coupling:public simulator {   //a coupling is a subclass of the simulator
 2
 3 public:
 4   simulator ** D;
 5   connection** IC, EIC, EOC;
 6   int Dsize, ICsize, EICsize, EOCsize, dast;
 7
 8 public:
 9   Coupling();
10   virtual ˜Coupling();
11   void init(float t, ...);
12   void exit();
13   void dint(float);
14   void propagate(Event, float);
15   Event lambda(float);
16   void dext(Event, float);
17   float ta(float);
18   void setup(simulator**, int, connection**, int, connection**, int, connection**, int);
19   void setup(simulator**, int, connection**, int);
20   virtual void changemytn(float, int);
21 };
```

Note that according to its definition, a coupling is also a simulator.

simulator.cpp: Simulators represent atomic models (or coupled models that however can be used as atomic submodels in larger models). They feature the following variables and functions (full listing of `simulator.h`):

```
 1 #ifndef simulator_h
 2 #define simulator_h
 3
 4 #include ‘‘event.h’’
 5
 6 class Coupling;
 7
 8 class simulator{
 9 public:
10   Coupling* father;
11   int myself;
12   float e, tl, tn;
13   Event output;
14 public:
15   simulator();
16   virtual ˜simulator();
17
18   virtual void init(float, ...)=0;
19   virtual Event lambda(float)=0;
20   virtual void dint(float)=0;
21   virtual void dext(Event, float)=0;
22   virtual float ta(float)=0;
23   virtual void exit();
24
25   void imessage(Coupling*, int, float);
26   Event lambdamessage(float);
27   void dintmessage (float);
28   void dextmessage(Event, float);
```

```
29   void externalinput(Event);
30 };
31 #endif
```

The virtual functions (lines 18-23) are defined in separate classes: following the idea of constructing complex models by simple submodels, PowerDEVS provides a number of predefined atomic models (blocks) from which more complex models can be built. For each of these blocks a single class file exists, consisting of exactly five methods (`init()`, `dint()`, `dext()`, `ta()` and `lambda()`) that define the mechanism of the component in question. For instance, the behaviour of the integrator is described in `integrator.cpp`, the behaviour of the weighted sum block in `wsum.cpp`.
Such classes inherit the `simulator` class: `class [componentName]:public simulator`. Thus, the integrator for example is both of the type `integrator` and of the basis type `simulator`. As will be seen in the sample simulation run that is carried out after the discussion of the different classes, the "message"-functions can be thought as wrapper functions for the corresponding functions (`lambdamessage()` for `lambda()`, `dintmessage()` for `dint()`, etc.) of either a real component or a coupling, depending on what the respective simulator represents.

root_coupling.cpp: There is always an instance of a coupling that holds all atomic models and submodels (analogous to the root coordinator at the top of the hierarchic tree in Figure 3.4. The type of this instance is `root_coupling`.
The `root_coupling` class is defined as `class root_coupling:public Coupling;`
Note that although the `MainCoupling` defined in the `root_simulator` class has been declared as a variable of the type `coupling`, in order to have all functions of a coupling at hand, it has been instantiated with a variable of the type `root_coupling` which in its turn has been assigned all components appearing in the model, in their full hierarchical structure (see Figure 3.2).
The root coupling (`MainCoupling`) can be thought of as the father of all other components and the "spokesman" of the root simulator as we shall see shortly.

root_simulator.cpp: This class manages only the global simulation time and does not feature any of the DEVS-related methods (δ_{int}, δ_{ext}, etc.) such as the normal `simulator` does. It contains however a pointer to the main coupling that enables it to control the whole simulation (by calling functions of the coupling):

```
Coupling* MainCoupling;

bool root_simulator::step(){
  t=MainCoupling->tn;
  if (t<=tf) {
    MainCoupling->lambdamessage(t);
    MainCoupling->dintmessage(t);
    return false;
  }else {
    MainCoupling->exit();
    return true;
  };
}
```

Only the main model is of the type `root_simulator`. The root simulator can be seen as the

director of the simulation, passing its orders to the `MainCoupling`, which then spreads them among its children.

Figure 3.9 shows again the simulation scheme tree of Figure 3.4 – this time not only with abstract constructs such as the coordinators and the simulators, but also with the real corresponding C++ classes of the PowerDEVS simulator framework.

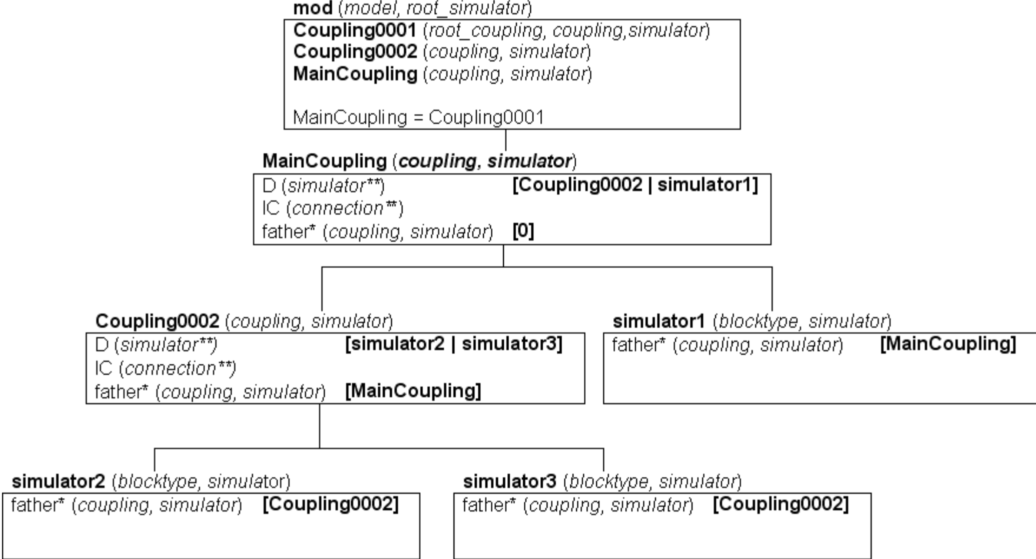


Figure 3.9: The hierarchical tree in Figure 3.4 represented by means of the C++ framework. The squares symbolize instants of the various classes, carrying their name on the upper left. The type(s) of the instances/variables are indicated in curly braces whereas the first type is the type of allocation and the subsequent types are those that can be adopted due to a class-subclass relation. The contents of the variables are put into square brackets and are printed in bold. Note that the variable `father` of the MainCoupling contains 0, thus the MainCoupling has no father coupling.

Let us now start a small simulation and see how the simulator constituted by the classes presented above works in practice. Assume again the example in Figure 3.1 without the output blocks.

In the beginning of every simulation, PowerDEVS creates the file `model.h` (see Figure 3.2). Thereafter, the simulation is ready to run.

```
1 #include "../model.h"
2 int main(int argc,char **argv){
3   model *mod=new model(false);
4   mod->configure();
5   mod->init();
6
7   while (*c!='q') {
```

```
 8     ...[other possible simulation modes (run(), timed()]...
 9     if (*c=='s') {
10       if (mod->step()) {
11         delete mod;
12         printf("SimulationEnd\n");
13         return 0;
14       }else{
15         printf("%f\n",mod->t/mod->tf*100);
16         fflush(stdout);
17     }
18   }
19   if (*c=='q') {
20     mod->MainCoupling->exit();
21     printf(''SimulationEnd\n'');
22   }
23 }
```

In lines 3+4, the `main()` function in `model.cpp` allocates an instance `mod` of the type `model` and calls `mod->configure()`.
The method `configure()` is defined in `model.h` and is responsible for the allocation and initialisation of blocks and connections and their assignment to the appropriate couplings (consider Figure 3.2 for the subsequent line references):

- Lines 32-47: the simulator and connection arrays `D0001` and `IC0001` are filled with instances of simulators and connections, such that they can be passed to the appropriate coupling later on.
- Line 50: function `setup()` passes the simulator and connection arrays to `Coupling0001`. This is how a coupling is provided with all required information about its children.
- Line 52: the class variable `MainCoupling` of the class `root_simulator` is assigned the variable `Coupling0001`.
  Note that `Coupling0001` is the coupling that contains all other simulators and couplings (i.e. it is the analogon to the root coordinator at the top of the hierarchic tree in Figure 3.4), and it is therefore justified to assign it to the variable `MainCoupling`, which will be used to conduct the simulation as we shall see later.
- Lines 58-61: the children of the `MainCoupling`, namely the Step block, the Integrator and the NLfunction block are initialised. The initialisation can be split in two parts:
  Lines 58-60, part 1: internalisation of the parameters defined in the graphical model. The instruction `child00010001->init(0, 0.0,1.76,10.0);` for example initializes the Step block with parameters $startingvalue = 0.0$, $steptime = 1.76$ and $stephight = 10.0$ as defined by the user.
  Line 61, part 2: the instruction `Coupling0001->init(0);` calls the `init()` function of the `coupling` class:

```
void Coupling::init(float t, ...){
  for(int i=0;i<Dsize;i++){          //Dsize = # of components in the coupling
    (*(D+i))->imessage(this, i, t);  //D = array of pointers to the components
```

```
  };
}
```

`init()` in its turn calls the method `imessage()` of each child of the coupling:

```
void simulator::imessage(Coupling* f, int my, float t){
  myself=my;    //number (ID) of the simulator
  father=f;     //to which coupling does this simulator belong
  e=0;          //epsilon
  tl=t;         //time of the last event
  tn=t+ta(t);   //time of the next event
}
```

This function sets some other important class variables (additionally to the parameter values from the first initialisation part) and thereby completes the second part of the block initialisation.

After the termination of the `configure()` function, the program falls back to line 5 in the `main()` function on page 32 and executes the instruction `mod->init()` which is actually a call to the `init()` function of the root simulator (remember that a model is also a root simulator, see Figure 3.8). This step completes the initialisation of the `MainCoupling`.

```
void root_simulator::init(){
  t=ti;
  MainCoupling->imessage(0,0,t);
  MainCoupling->ta(t);
}
```

Note that the coupling class does not feature a method `imessage()` itself. It inherits it though from the simulator class (remember that every coupling is also a simulator).
Calling `MainCoupling->ta(t)`[4] triggers the coupling to update its class variable `tn` which has to be equal to the smallest `tn` value of its children.

```
float Coupling::ta(float t){
  tn=4e10;
  for(int i=0;i<Dsize;i++){
    if ((*(D+i))->tn<tn){
      tn=(*(D+i))->tn;
      dast=i;
    };
  };
  float tal=tn-t;
  return tal;
};
```

In order to evaluate the appropriate value of `tn`, the coupling goes through the array of its children and compares their `tn` values to each other, always keeping the smallest one found so far. Note that the coupling scans its children in the order they were inserted in the simulator array (lines 32-37 in Figure 3.2), thereby introducing a certain priority order between them.

[4] Note that in the original DEVS definition, the time-advance function depends on the state s, whereas in PowerDEVS it depends on the time t. However, this is no contradiction, since from knowing both the exact simulation time and the block the time-advance function of which is called, the state s can be deduced. Hence, s and t are interchangeable in this case.

In our example, there are currently (at the beginning of the simulation) two components that have a `tn` of zero: the Integrator and the Step block (the NLfunction block has a `tn` value of infinity). The coupling though stores only the ID of the Step block in `dast` because it is the first component it finds that has a `tn` value smaller than 4e10 – the subsequent comparison with the Integrator will only result in equality, `dast` is therefore not updated and will still contain the ID of the Step block after the `ta()` function has terminated.

The steps up to here can be considered the initialisation of the model. The next step depends on what "type" of simulation the user wishes to carry out: a full simulation run, a step-by-step simulation or a simulation up to a certain point in time. The `main()` function will then execute the code segment corresponding to his choice: `mod->run()`, `mod->step()` or `mod->timed()`. For our example, we suppose that the user chooses the step-by-step simulation (line 10 in the `main()` function on page 32) provided that the `run()` and `timed()` methods depend on the `step()` method anyway.

```
 1 bool root_simulator::step(){
 2   t=MainCoupling->tn;   //the simulation runs until the value of tn ("time next")
 3   if (t<=tf){           //is bigger than the final simulation time tf.
 4     MainCoupling->lambdamessage(t);
 5     MainCoupling->dintmessage(t);
 6     return false;
 7   }else{
 8     MainCoupling->exit();
 9     return true;
10   };
11 }
```

As can bee seen above, this function first invokes the `lambdamessage()` function of the `MainCoupling` and subsequently its internal transition. Note that these functions actually belong to the simulator class, but again, a coupling is also a simulator.
Since the `lambdamessage()` entails a number of function calls before the `dintmessage()` method is finally called, let us first follow only the "lambdamessage"-branch and return to this point later in order to track the "dintmessage"-branch, too:

```
Event simulator::lambdamessage(float t){
  return lambda(t);
}
```

Function `simulator::lambdamessage()` always returns an event generated by the `lambda()` method. Depending on whether the component the `lambdamessage()` of which has been called is a simulator or a coupling, a different type of `lambda()` will be called (either the one defined in the simulator or in the coupling class): in the case of a coupling, `lambda()` returns a null-value, in the case of a pure simulator, it returns a real value.

```
 1 Event Coupling::lambda(float t){
 2   output=(*(D+dast))->lambdamessage(t);
 3   int p=output.port;
 4   if (output.IsNotNull()){
 5     propagate(output,t);
 6   };
 7   Event x;
 8   x.SetNullEvent();
```

```
 9   return(x);
10 };
```

In our case, MainCoupling is not a "real" simulator (like the Integrator, NLfunction or the Step block would be) but a coupling, and therefore the call of lambda() leads to a call of the respective function defined in the **coupling class** instead of the class of the particular block. The return value of lambda() is thus the aforementioned null-value (lines 8+9 in the above code snippet) .
Since the main task of a coupling is to control its children, the lambda() function is only a forwarding of the λ-function-command to the simulator or the coupling stored in dast, which in our case is the Step block that has been inserted when the ta() function of the coupling was called. So, the lambdamessage() method is called again (line 2), but this time, the lambdamessage() function belongs to the simulator class, not to the coupling class as it did before. Thus, the call of lambda() inside the simulator::lambdamessage() function is a call of the Step block's lambda() function:

```
Event step::lambda(float t){
  return Event(&S[0],0);  //the result has been stored in S[0] before
}
```

The lambda() function only returns an event to its caller. Let us track it back: the lambda() function of the Step block has been called by the lambdamessage() function of the simulator class, which in its turn has been called by the lambda() function of the coupling (line 2 in the function above). Thus, the output of the lambda() function of the Step block is stored in a variable output in the lambda() function of the coupling from where it will be sent to the blocks that are connected to the Step block (line 5).
Two things are important to note here: a) the output of the Step block is propagated to other blocks even before the Step block's internal transition is invoked and b) the function propagate() is called by the coupling, not by the simulator itself. This is why the communication between blocks can be said to be monitored by the coupling the blocks belong to.

```
 1 void Coupling::propagate(Event x, float t){
 2   output=x;
 3   int p=output.port;
 4   if(x.IsNotNull()){
 5     for(int i=0;i<ICsize;i++){          //search for blocks connected to child1
 6       if((*(IC+i))->child1==dast && (*(IC+i))->port1==p){
 7         output.port=(*(IC+i))->port2;   //input port of connected block
 8         int mod=(*(IC+i))->child2;      //ID of connected block
 9         (*(D+mod))->dextmessage(output, t);
10       };
11     };
12     ...[eventually propagate further through external output connections]...
13   };
14 }
```

The propagate() function checks the connections leading from the Step block (still stored in dast) to other components (line 5-6 in the above code snippet). In our example it finds the NLfunction block (line 8). In order to inform it of the arriving event, its dextmessage() method is called (line 9).

```
void simulator::dextmessage(Event x, float t){
  e=t-tl;
```

```
  dext(x, t);
  tl=t;
  tn=t+ta(t);
}
```

This in turn leads to a call of the `dext()` function in the class file of the block in question (NLfunction).

```
void nlfunction::dext(Event x, float t){
  ..[content rather complex -> not listed}..
}
```

The chain of function calls provoked by the instruction `MainCoupling->lambdamessage(t);` (line 4 in the `root_simulator::step()` function on page 35) is completed at this point, and the program continues with the next instruction, namely `MainCoupling->dintmessage(t);`

```
1 void simulator::dintmessage(float t){
2   e=t-tl;
3   dint(t);
4   tl=t;
5   tn=t+ta(t);
6 }
```

Given that the `MainCoupling` is an instance of the type `coupling`, the instruction `dint(t);` leads to the `dint()` function in the coupling class.

```
void Coupling::dint(float t){
  (*(D+dast))->dintmessage(t);
};
```

From here, the `dintmessage()` function of the simulator class is called, given that the component stored in `dast` is a simulator (still the Step block). This leads directly to a call of the `dint()` and the `ta()` function of the Step block.

```
void simulator::dintmessage(float t){
  e=t-tl;
  dint(t);    //calls dint() of the Step block
  tl=t;
  tn=t+ta(t); //calls ta() of the Step block
}

void step::dint(float t){
  if (Aux==false) {
    Sigma=Delay;
    Aux=true;
    S[0]=Vfin;
  }else {
    Sigma=4e10;
  };
}

float step::ta(float t){
  return Sigma;
}
```

Now, the program falls back to line 3 in the function simulator::dintmessage() on page 37, and the next instruction to be processed is tn=t+ta(t); (line 5). Remember that the call of the function simulator::dintmessage() was caused by MainCoupling->dintmessage(t); and hence the dintmessage() function belonged to the coupling class. For this reason, the ta() function that is called afterwards is the one of the coupling class (as opposed to the one of the simulator class).

```
float Coupling::ta(float t){
  tn=4e10;
  for(int i=0;i<Dsize;i++){
    if ((*(D+i))->tn<tn){
      tn=(*(D+i))->tn;
      dast=i;
    };
  };
  float tal=tn-t;
  return tal;
};
```

The ta() method scans all children of the coupling in order to find the smallest tn value. Once found, the coupling assigns this value to its own tn.
After the termination of ta(), the program falls back to the root_simulator::step() function, line 5 (on page 35). One simulation step of the simulation has now come to completion and the user has to trigger the next step manually[5] since he had chosen the step-by-step simulation. This will call again the function root_simulator::step() on page 35 and another simulation step starts.

In the light of Chapter 4, where differences between PowerDEVS and ModelicaDEVS will be discussed, it is worthwhile mentioning explicitly that due to the sequence of the called functions during the simulation, the generation of an output event of a particular block leads first to the execution of the external transitions of connected components before the internal transition of the currently active (”firing”) block is invoked. Hence, the pair lambda-function/internal transition is separated.

[5]PowerDEVS provides a graphical way of choosing the simulation type: right after model compilation, a window pops up, containing four buttons “run”, “step”, “timed”, “stop” and “close”. By clicking the “step”-button again, a second simulation step is executed.

Chapter 4

ModelicaDEVS – the Simulator

ModelicaDEVS is a Dymola/Modelica library that provides the tools for discrete-event simulation incorporating the DEVS formalism, where "the tools" are simply a number of predefined DEVS blocks from which large models can be built (a detailed description of these components is given in Chapter 5).

As pointed out in Section 1.2, the scope of ModelicaDEVS is to provide a similar functionality as PowerDEVS does. However, because of the different working paradigms of Modelica and C++ (see Section 4.1), the re-implementation of PowerDEVS in Dymola is not a mere translation from C++ to Modelica. Although Dymola is, due to some of its features, very well suited for an implementation of the DEVS formalism in Modelica, not all of its properties are advantageous. Some cause difficulties when trying to re-implement a Modelica version of PowerDEVS.
The most important/interesting implementation issues regarding this subject – preceded by a brief description of the Dymola environment – shall be presented in the subsequent sections. The section about atomic models may also serve as a HOWTO for the translation of further DEVS models from C++ code (e.g. in the case of future developments within PowerDEVS) into Modelica.

4.1 Dymola Programming/Simulation Environment

This section presents some properties of Dymola/Modelica that are necessary for a thorough understanding of the subsequent sections in this chapter.

4.1.1 Simultaneous Equations

While C++ is an imperative programming language and lives on the principle of functions calling other functions, Modelica models are described mathematically by equations, and the simulation makes use of the simultaneous evaluation of these model equations. This means that in every simulation step, Dymola collects **all** equations of a model, even if they are spread over several blocks, and then tries to evaluate them. After it has found a value for

each variable such that all equations are fulfilled, a new simulation step is started.
A direct consequence of this approach is the parallel update (in terms of simulation time) of the state variables. This can be exploited for the implementation of DEVS in Modelica, but at the same time causes some problems that do not have to be dealt with when using a programming language like C++.

4.1.2 Simulation Time

The simulation environment of Dymola comes with a global clock that is accessible through the variable `time`. Note that it is not possible to set the clock by assigning a certain value to `time`. The purpose of `time` is only to give the programmer the opportunity to query the current simulation time.

4.1.3 State and Time Events

Another particularity of Dymola is the time or state events that are triggered by equations containing inequalities.

The scope of state events is to inhibit discontinuities during integration. For an example, consider the following Modelica model (see also [4]) and the corresponding graph:

```
model EventTest
  input Real u;
  output Real y;
equation
  y = if u>0 then 1 else -1;
end EventTest;
```

At the point $u = 0$, the equation for y is discontinuous. Such discontinuities caused by a relation like $u > 0$ raise a problem because the branch of the if-statement can be switched at a certain point in time, and all of a sudden y can take a completely new value, "artificially" assigned, that has nothing to do with the previous trajectory of y.
The solution of Dymola to this problem[1] is to stop the integration at point $u = 0$, select the appropriate branch of the if-statement and restart the simulation with new initial conditions.

Inequalities that only depend on `time` as the only continuous unknown variable may also trigger time events instead of state events. The following table illustrates which relations still cause state events – even if `time` is their only continuous variable – and which entail only time events:

State Events	**Time Events**
time <= [discrete expression]	time >= [discrete expression]
time > [discrete expression]	time < [discrete expression]

[1] There are situations when no state events have to or even must be triggered. This can be achieved by using the `noEvent()` operator. For more details see [4] or [15].

Whereby the discrete expression may itself be a function of constants, prameters, and discrete variables, but not continuous variables.

The main difference between time and state events is that Dymola does not have to iterate on time events but "jumps" directly to the point in time when the time event will arise. Thus, they may save computation time, and it is recommended to use time events instead of state events wherever it is possible.
ModelicaDEVS has been implemented using time events as much as possible, limiting the use of state events to those few situations only, where they could not be avoided[2].

4.2 Atomic Models

The mapping of an atomic model from PowerDEVS to Modelica does not yet cause many difficulties but is rather a translation of the C++ code into Modelica – which however is not as straightforward as would be a translation to Java or Pascal, as we shall see.
Of course, the design of an atomic model has to provide the features requested by the simulation of a coupled model and therefore already involves the solutions to the issues that will be discussed in the subsequent sections. It is hence possible that the general block structure developed in this section is not yet fully understandable. However, the basic concepts should be clear and allow for a better understanding of Section 4.3.

4.2.1 Events/Ports

An event in PowerDEVS consists of four values: the event creation time and the coefficients to the first three terms (constant, linear and quadratic) of the function's Taylor series expansion. In other words, an event is described by the current function value, the first derivative of the function at the current time instant and its second derivative devided by two.
The higher-order approximated output of QSS2 and QSS3 models is then equal to the first-order or second-order Taylor series expansion: the output is given by

$$y[0] = y[0] + y[1] \cdot (t - t_{last}) + y[2] \cdot (t - t_{last})^2$$

which, with the entries of the y-vector holding the coefficients to the first three terms of the Taylor series expansion, corresponds to the Taylor series up to second-order for QSS3, or first-order for QSS2 where $y[2]$ is always 0.
Note that the coefficients to the linear and the quadratic terms of the Taylor series are only used for QSS2 and QSS3.
Since these values sufficiently describe an event, ModelicaDEVS simply adapts this structure

[2]Note that until three weeks before the deadline for this thesis, a slightly different approach to implementing the DEVS formalism in Dymola/Modelica was used. The main drawback of that version was, however, that it generated state events instead of time events which made the simulation become very slow. It was obvious that a more efficient solution should contain only (or at least mostly) time events, but no sound description of the models without using state events could be found. Thanks to the presentation of another approach of implementing DEVS in Modelica (see "Principles of Object-Oriented Modeling and Simulation with Modelica 2.1" of P. Fritzson [5]), the previous version of ModelicaDEVS could be adapted to this more efficient solution that uses indeed only time events.

to a large extent. The only difference is the replacement of the event creation time by a boolean variable that indicates whether a certain block is currently sending an event. Hence, whereas PowerDEVS port are represented by single vectors, ModelicaDEVS ports consist of a vector of size three (for the three input/output values) and the aforementioned boolean variable. The code snippet below shows the declaration of a ModelicaDEVS input port. Output ports are declared analogously.

```
connector DiscreteInPort "Discrete Connector"
  input Real signal[3];
  input Boolean event;
end DiscreteInPort;
```

Within the `equation` section of the Modelica model the parts of the input/output ports are referred to as `uVal[1]`, `uVal[2]`, `uVal[3]` and `uEvent` for input ports, and `yVal[1]`, `yVal[2]`, `yVal[3]` and `yEvent` for output ports. The name conversions from a vector named `signal` to a vector named `uVal` and a boolean variable `event` to a variable `uEvent` takes place in the block templates from which all ModelicaDEVS models are derived (see Section 5.5 for a description of block templates).

Remark: for simplicity reasons, we shall pretend that ModelicaDEVS input/output ports are represented by single vectors of size four, whereby the fourth entry is not of type real but of type boolean, instead of a vector of size three plus a boolean variable. In reality, it is of course not possible to have mixed arrays, but this assumption facilitates the writing and understanding of sections dealing with event-passing issues: it is no longer necessary to make a distinction between the vector part of the port and the boolean variable, but we can simply think of the boolean variable to be the fourth entry in the vectors that represent the input/output ports of ModelicaDEVS models.

4.2.2 Transitions

A DEVS model consists of the following subroutines (cf. Section 2.2): internal transition, external transition, time-advance, and lambda function. PowerDEVS components additionally contain an initialisation function. For ModelicaDEVS, this means that basically all these parts have to be explicitly or implicitly present (including the initialisation function).
As already mentioned, Modelica and C++ operate on different paradigms, (simultaneous equations vs. sequential processing) which impedes a direct translation without major modifications. With the help of some gimmicks however, respecting certain rules and restrictions, the C++ code can be transformed into Modelica rather easily. Note though that the code obtained by such a “mechanical” translation is only more or less a correct copy of the PowerDEVS model and has to be examined for possible intrinsic particularities that may require further block specific programming (see Section 5 for details/examples).
The following text discusses the way that has been chosen to create Modelica models with equal functionality to the PowerDEVS blocks.

The time-advance function is represented by a variable `sigma`. There is hence no reason for an explicit time-advance function anymore (PowerDEVS still implements it because it is needed for the hierarchic simulator framework, see Section 3.2.1). Note that even in the theoretical definition of a DEVS model it is a popular trick to represent the time-advance

function by a variable called sigma (see Section 2.2).

A variable `lastTime` holds the time of the last execution of a transition (internal or external). This variable is used for the computation of `e` (ϵ in the original DEVS definition)

```
e=time-pre(lastTime);
```

where `e` is the time the system has already spent in the current state.

Two boolean variables, `dint` and `dext`, are used to determine if there is currently any internal or external event to be processed.
An internal transition depends only on `sigma` (the time-advance function) and `e` (ϵ). Hence, the value of `dint` can be calculated as:

```
dint= time >= pre(lastTime)+pre(sigma);
```

The external transition of a block B attached to the output port of a block A is executed at the very moment when A generates an output event. This is where the fourth entry in the port vector comes in (explained in greater detail in Section 4.3). The variable `dext` of a block B is defined as follows:

```
dext = uEvent;
```

where `uEvent` is the fourth entry of the input vector of B (A.yEvent = B.uEvent).
The lambda function and the internal/external transitions are represented by when-statements. Most blocks feature two when-statements:

- A when-statement that is active when `dint` becomes true. It can be seen as the analogon to the lambda function, which always proceeds an internal transition.
- A when-statement that is active when either `dint` or `dext` become true. It holds the code for both the external and internal transitions.

What is the reason for this perhaps not very intuitive structure? Why not just have two when-statements, one representing the internal transition and the lambda function, the other one representing the external transition? The reason for packing the internal and external transitions into a single when-statement is due to a rule of the Modelica language specification that states that equations in different when-statements may be evaluated simultaneously (Modelica cannot/does not check for mutually exclusive when-statements). Hence, if there are two when-statements each containing an expression for a variable X, X is considered overdetermined. This circumstance would cause a syntactical problem with variables that have to be updated both during the internal and the external transition and thus would have to appear in both when-statements. For this reason, we need to have a when-statement that is active if either `dint` or `dext` becomes true. An additional discrimination is done **within** the when-statement, determining whether it was an internal (`dint` is true) or an external transition (`dext` is true) that made the when-statement become active, and as the case may be, updating the variables with the appropriate value (note that most variables take different values during internal transitions compared to external ones).
A typical "dint-dext" when-statement looks like one of the following (assume three variables `X`, `Y`, `Za` and `Zb` where `X` is updated during the internal transition, `Y` during the external transition, `Za` during both transitions but to different values and `Zb` also during both transitions but to the same value):

```
when {dint, dext} then
  if dint then
    X = 4;
    Y = pre(Y);
    Za= 6;
  else
    X = pre(X);
    Y = 9;
    Za= 7;
  end if;
  Zb= 2;
end when;
```

```
when {dint, dext} then
  X = if dint then 4 else pre(X);
  Y = if dint then pre(Y) else 9;
  Za= if dint then 6 else 7;
  Zb= 2;
end when;
```

Having now a place where equations for variables that are modified during the internal transition can go, what is the purpose of the second when-statement, which is also – though exclusively – active if `dint` becomes true? Its main justification is to separate the internal/external transitions from the output part. Furthermore, the output variables are only updated before an internal transition, thus, they do not necessarily need to appear within a dint-dext when-statement.
The typical "lambda-function" part (containing a when-statement and a separated instruction for the `yEvent` variable) looks as follows:

```
when dint then
  yVal[1]= ...;
  yVal[2]= ...;
  yVal[3]= ...;
end when;
  yEvent = edge(dint);
```

Note that the right hand side of the equations in the lambda function normally depends on `pre()` values of the used variables. This is due to the fact that the lambda function has to be executed **prior to** the internal transition. In ModelicaDEVS however, the two corresponding parts in the model are activated simultaneously (namely, when `dint` becomes true), so the temporal distinction between the lambda function and the internal transition has to be done via the `pre()` operator of Modelica.
The variable `yEvent` has to be true in the exact instant when an internal transition is executed and false otherwise. This behaviour is obtained by using the `edge()` operator[3] of Modelica.

4.2.3 Miscellaneous

For variables that are modified more than once during one simulation step, or that are updated by means of a complicated combination of if-then-else-statements, it is recommended to declare a function that contains an almost exact copy of the C++ code segment in question (an example where this method was applied is the Integrator block of ModelicaDEVS described in 5.3.8).

A very important advice in this context is to always carefully check for occurrences of the `sqrt()` function and make sure that it is only called with a positive argument. In case the

[3] The `edge()` operator returns true at the time instant when the value of its argument switches from false to true and false at any other time instant.

argument becomes negative, assign a value of infinity[4] to the variables the values of which would have been computed using the `sqrt()` function. Otherwise, the Modelica models show an erroneous behaviour, which is often fairly illogical and thus hard to find. It is therefore more than just advisable to pay attention to this detail since from the beginning.

4.2.4 Basic Model Structure

The average block of the ModelicaDEVS library exhibits the following basic structure:

```
block SampleBlock
  extends ModelicaDEVS.Interfaces. ... ;
  parameter Real ... ;

protected
  discrete Real lastTime(start=0);
  discrete Real sigma(start=...);
  Real e;     //epsilon - how long has the system been in this state yet?
  Boolean dext;
  Boolean dint;
  [...other variable declarations...]

equation
  dext = uEvent;
  dint = time>=pre(lastTime)+pre(sigma);

  when {dint} then
    yVal[1]= ...;
    yVal[2]= ...;
    yVal[3]= ...;
  end when;
    yEvent = edge(dint);

  when {dext, dint} then
   e=time-pre(lastTime);
    if dint then
      [...internal transition behaviour...]
    else
      [...external transition behaviour...]
    end if;
    lastTime=time;
  end when;

end SampleBlock;
```

4.3 Coupled Models

Recall the hierarchic model tree structure used by the the PowerDEVS simulator (Chapter 3), where consecutive layers communicate with each other through message passing, and a global simulator-component advances the simulation clock. How can this simulation framework be

[4]This is what a C++ compiler does. Therefore, the current version of PowerDEVS which does not yet check for negative arguments to `sqrt()`, works adequately. This is however a poor programming style, and the `sqrt()` issue is subject to be corrected in future versions of PowerDEVS.

mapped to the fairly different Dymola/Modelica environment where there exists no simple possibility to construct a global simulator similar to the one in PowerDEVS, and where there is already a simulation clock present?
Considering these differences, the communication mechanism of PowerDEVS cannot/should not be copied blindly. Instead, a slightly modified approach has to be found, preferably exploiting the benefits of the built-in Dymola/Modelica features, in order to compensate for possible drawbacks compared to imperative programming languages like C++.

4.3.1 Time-advance Mechanism

In PowerDEVS, the responsibility for the advancement of the simulation clock is assumed by the root simulator. In Dymola on the other hand, there is already a clock present, and it is recommended to use it as the actual simulation clock, since the graphical output (e.g. the values of state variables) will be plotted against time (on the x-axis), and if the simulation is not synchronized with the Modelica simulation clock, the graphical output may not be very illustrative anymore.

4.3.2 Direct Block Interaction

As a consequence of the absence of generic simulator/coordinator models, the "view" of each block is restricted to its own input and output ports, and there are no couplings that could handle the interaction between blocks as it is done in PowerDEVS.
The solution that was found for ModelicaDEVS to enable the required block interaction for coupled models is to pass a boolean signal along with the compulsory event values. The boolean signal is used to inform a block B about the occurrence of an output event at block A (given that block B is connected to the output port of block A).

Let us consider a small example. Assume a two-block system consisting of block A and block B, where block B is connected to the output port of block A. Every block features a variable `dext` accompanied by an equation

```
dext = uEvent;
```

where `uEvent` is the boolean component of the input vector that represents events. Suppose now that block A produces an event at time $t = 3$. As already pointed out, an event consists of three state-related values plus a boolean variable. At the precise instant when A generates an event, it updates its output vector with the appropriate values and sets `yEvent` to true:

```
when dint then
  yVal[1]= ...; //new output value 1
  yVal[2]= ...; //new output value 2
  yVal[3]= ...; //new output value 3
end when;
  yEvent = edge(dint);
```

Still at time $t = 3$, block B notices that now `uEvent` has become true (note that `B.uEvent = A.yEvent`) and therefore `dext` has become true, too. Block B is then going to execute its external transition.

4.3.3 Concurrent Events

Two events that are generated at one and the same time are called concurrent events. There are four types of concurrent events, differing from each other by the location where in the system they occur:

- Type 1: Concurrent events that occur at different blocks in the model and that do not influence each other. There is no need to treat this case in a special way since Dymola/Modelica always looks at the model as a whole (simultaneous equations, see Section 4.1.1). Hence, it will update the variables of the critical blocks in parallel (in terms of simulation time) anyway.

- Type 2: Concurrent events that occur at the same block with two input ports (such as the Add-Block). In this case, the block has to provide a solution itself, e.g. either favour the processing of one of the external events or involve both of them in the computation of the new variable values.

- Type 3: Concurrent events that occur at a block with only one input port. Given the fact that there is only one input port, these events must be an external and an internal event, whereas the external event occurs at the same time when the block in question should execute its internal transition. Note that this scenario of an external and an internal event at the same block can also be seen as two internal transitions in two connected blocks (the internal transition of the first block will be recognised as an external event by the second block).
  This situation brings up the topic of priorities, which is strongly related to the differences between ModelicaDEVS and PowerDEVS, and which will be covered in a separate section (4.3.5).

- Type 4: Concurrent events that occur at the same block, caused by a loop. They are actually a special case of type-3 events, which however could cause even more troubles. Fortunately, the problem of events that instantaneously return to the same block they originate from is solved by Dymola itself, as will be seen in Section 4.3.4. Hence, a solution to the problem of type-3 events automatically solves the type-4 events issue.

Apparently, concurrent events only require additional attention if they occur at the same block. If the given block has two input ports, it has to provide a solution how to treat the two signals by itself. If it has only one input port, it will have to choose between the external event or the internal transition to process first. As already mentioned, this is related to the priority issue of the DEVS formalism and the problem of loops – depending on how the concurrent events have been generated.

4.3.4 Loops

Loops occur if a variable X is being used (either on the right hand side of an equation or in a when-statement) to update a variable Y that is used to update the variable X. Due to the fact that the view of Modelica is model-wide (it treats all equations that appear in the components of a model equally), such a loop may also be spread over several blocks: a variable X of block A is used to update another variable Y in a block B that is used again to update the variable

X of the block A (this corresponds to the problem in Section 4.3.5, Figure 4.2).
Such cycles are called algebraic loops. They may be broken by applying the `pre()` operator on certain variables of the loop: the `pre()` operator characterizes the left limit of a variable at a given time instant and therefore may break the loop by inserting an infinitesimally small time delay between two updates of a loop-variable (see [4] for more details).

Once the components are programmed correctly in terms of breaking potential loops by inserting the `pre()` operator in appropriate places, there is a second type of cycles that may occur: components that set their local variable `sigma` to zero during the external transition have to execute an internal transition immediately after (actually contemporarily to) the current external transition. Dymola solves such loops automatically by iterating until a consistent restart condition is found (using the infinitesimally small time delay induced by the `pre()` operator for variables that have to be updated twice or more times within one simulation step).

4.3.5 Priorities

In fact, there is only one single situation where the importance of priority settings between components comes in: two connected blocks, where both block A and block B have to execute an internal transition at the same time (see figure 4.1). Block A actually does not even feel

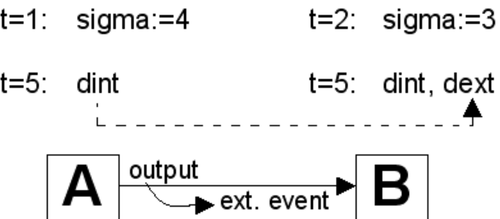


Figure 4.1: Concurrent events at block B.

the presence of the concurrent events, and just processes the internal transition. This however causes a priority problem in block B which was about to go through an internal transition itself and now receives an external event at the same time, since the internal transition of block A entails an output that appears as an external event at block B.

In our simple two-block-example there are only two possible priority orderings with the following consequences:

- If block A is prior to block B, block A will produce the output event before block B executes the internal transition. Hence, block B will first execute an external transition triggered by the external event it received from block A.
- If block B is prior to block A, it will first go through the internal transition and receive the external event right afterwards, when A will be allowed to execute its internal transition.

The problem of block priorities can be solved in two ways: by an explicit, absolute ordering of all components in a model, or by letting every block determine itself whether it processes

the external or the internal event first, in case both of them occur at the same time. While PowerDEVS makes use of the former approach (see Figure 3.3), ModelicaDEVS implements the latter: because of the block-to-block nature of the communication in coupled models, a global priority list of all components would be useless or at least associated with additional effort for checks on the priority relation between two active components. It is therefore much more convenient to take this decision locally by simply determining whether to execute the internal or the external transition first.

In ModelicaDEVS, there is only a hard coded version of the priority handling implemented: an internal transition is always prior to an external one.
The reason for this choice – the fixed ordering as well as the priority order of internal before external – lies in the way events are passed from block to block and in the inability of Dymola/Modelica to solve algebraic loops (see Section 4.3.4). Assume the following abstract code snippet containing the basic variables and when-statements to enable a block to recognise internal and external events and to produce output events.

```
equation
  dext = uEvent;
  dint = time>=pre(lastTime)+pre(sigma);

  when {dint} then
    yVal[1]=...;
    yVal[2]=...;
    yVal[3]=...;
  end when;
    yEvent=edge(dint);

  when {dext, dint} then
    if dint then
      sigma = ...;  //something...
    else
      sigma = ...;  //something else...
    end if;
    lastTime=time;
  end when;
```

There are two important facts to take into consideration: a) the boolean variable `dext` depends on `uEvent`, which is actually the output (the `yEvent` variable) of another block, and b) the variable `yEvent` of any component depends on the variable `dint`. Note that the way shown above is not the only one to program a block, but in any case the statements a) and b) hold, which is the critical point to the fixed priority order problem.
Consider now a model that – in case of a concurrent occurrence of two events – attempts to execute the external transition before the internal transition. Then, the first when-statement in the code above would have to be changed into

```
when (dint and not dext) then
```

and `yEvent` would have to be modified into

```
yEvent = edge(dint and not dext)
```
[5]

[5] Note that this assignment is actually invalid since the `edge()` operator expects a single variable as an argument. It however exactly demonstrates the meaning of the required modification. In correct Modelica, an auxiliary variable would have to be created, taking the value of `dint and not dext`, that then can be passed to the `edge()` operator.

This however would entail the update of the output variable `yEvent` to be dependent not only on `dint`, but on `dext`, too. Assume now a toy-scenario of a loop of two blocks A and B that execute their external transaction prior to the internal transaction and therefore have been programmed as described above, with the modified when-statement (Figure 4.2).

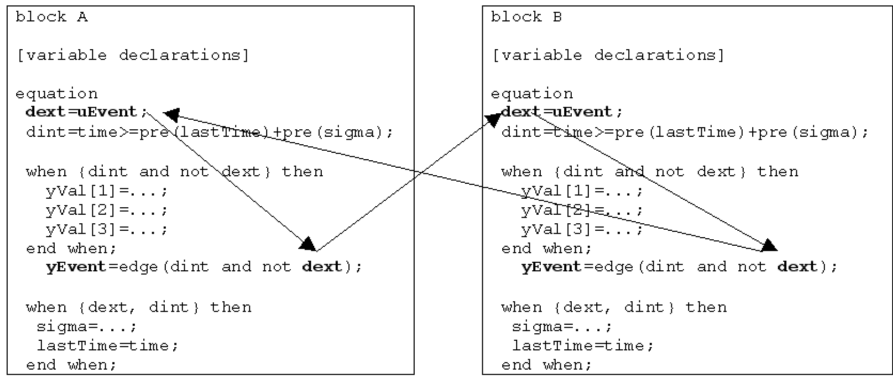


Figure 4.2: Algebraic loop involving the variables `dext`, `yEvent/uEvent` of two blocks connected in a loop.

Because of the cycle between the variables `yEvent`, `uEvent` and `dext` (the output `yEvent` of one block becomes the input `uEvent` of the other block), such a scenario results in an algebraic-loop-error message of Dymola (cf. Section 4.3.4).
More generally, we can say that **the output of a block must not depend directly on the input**, because a number n of blocks that are connected in a cycle (the output of block n is connected to the input of block 1) could cause the following dependency loop : output 1 depends on input 1; input 2 depends on output 1; output 2 depends on input 2; input 3 depends on output 2; ... ; input n depends on output $n-1$; output n depends on input n; input 1 depends on output n.

The cause for this issue is Modelica's principle of simultaneous equations and thus the problem cannot be avoided. For this reason, `yEvent` has to remain independent of `dext` and has to be updated in a way such that it depends solely on `dint`. Hence an internal transition has to be treated prior to an external one, because if `dint` becomes true, also `yEvent` does so. This however indicates the generation of an output event, and every output event **has** to be followed by an internal transition. If the external transition were treated with priority to the internal transition, it could give rise to a situation where the external transition is executed and an output is generated, which would be an illegal behaviour according to the DEVS formalism.

4.4 Hierarchic Models

Since Dymola/Modelica is already aimed at object-oriented modelling, which includes the reuse of autonomous models as parts of larger models, the issue of hierarchic systems did not need any particular treatment.
Furthermore, because of the equation-based simulation mechanism, Dymola/Modelica does

not handle hierarchical models differently to flat models: while simulating a model, Dymola collects the equations of **all** components constituting the model and evaluates them simultaneously. A second-level submodel contributes just some additional equations to the global equation "pool", but does not cause any further inconveniences, like in PowerDEVS, where hierarchical models entail a larger amount of messages passed between the hierarchy levels.

4.5 Time/State Events

As explained in Section 4.1, Dymola can trigger two types of events:

- State events that may require iteration to find a consistent restart condition (remember that the integration is interrupted and then continued with new starting conditions).
- Time events that make Dymola "jump" directly to the point in time when the time event will arise.

The only expressions that are responsible for activating the when-statements in the models,

```
dext = uEvent;
```

and

```
dint = time>=pre(lastTime)+pre(sigma);
```

both trigger time events and hence avoid the more expensive (in terms of computation time) state events.

Let us mention once more (see also the footnote on page 41) that a previous version of ModelicaDEVS used an approach that triggered only state events. According to performance comparisons carried out between the two versions, it has been found that the time event approach is roughly four times faster than its state event counterpart.

Chapter 5

ModelicaDEVS – the Library

This chapter presents the components of the ModelicaDEVS library. The scope is not only to provide a potential user of the library with a first overview of the available components and their use, but also to give some more general insight into the reasoning behind their concrete implementation.

The discussion of the blocks is conducted according to the structure given by their affiliation to the different packages, resulting in three main sections:

- Source blocks: blocks that have either no input port or an input port that accepts only continuous signals from standard Modelica components.
- Function blocks: blocks that have one or more DEVS input ports and one DEVS output port
- Sinks: blocks that accept DEVS signals but do not generate any output – or, if they do, it is again not a DEVS signal which could be processed by other ModelicaDEVS blocks.

. Furthermore, there is one section dedicated to the component WorldModel (within the *Miscellanous* package), which is not a common DEVS block but facilitates the model handling in terms of choosing what type of QSS a model shall belong to, and in the end a brief description of the *Templates* package is given.

5.1 WorldModel

The WorldModel component is a very simple, equation-less model that consists of only one parameter:

```
model worldModel
  extends Modelica.Blocks.Interfaces.BlockIcon;
  parameter Integer qss=1;
equation
  ...[annotation]...
end worldModel;
```

It has to be inserted in every model, and its parameter *qss* is used to set the parameter *method* in the model's others components.
But how is it possible to set a parameter by means of a parameter of another component? The solution in this context is the Modelica "inner/outer" language construct (cf. [15]): *"If a variable or a component is declared as outer, the actual instance is defined outside of the defining class and is determined by searching the object hierarchy upwards until a corresponding declaration with the inner prefix is found."*
Every block contains therefore an outer variable `world` of the type `worldModel` and uses the variable `world.qss` in order to set the parameter *method*:

```
model anyBlock

  parameter Integer method=world.qss "Use QSS1, QSS2 or QSS3";
  outer ModelicaDEVS.Templates.worldModel world;
```

By definition, the model itself has to contain a component called *world* of the type `worldModel` that is declared as "inner". In order to automatically guarantee these settings for all instances of the WorldModel component, the WorldModel features the following annotation properties:

```
annotation (defaultComponentName="world",
            defaultComponentPrefixes="inner",
            Diagram,
            Icon([Icon definitions])
);
```

The true value of *method* in any block that is present in a model that also contains the WorldModel is then determined by the *WorldModel.qss* parameter. Assume a simple toy model, only consisting of a Ramp block and the compulsory WorldModel component. Figure 5.1 shows how the *method* parameter of the Ramp block is defined by setting the *qss* parameter of the component world.

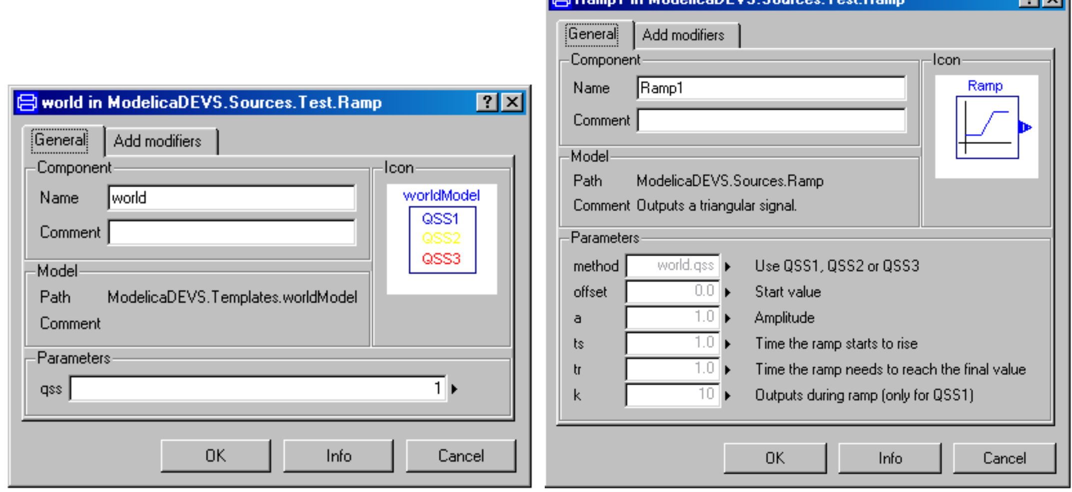


Figure 5.1: The parameter windows of the WorldModel and the Ramp block.

The *method* parameter determines whether to use QSS1, QSS2 or QSS3 specifications (see Section 2.3.2 for details on Quantised State Systems). The presence of the WorldModel component allows the user to switch the *method* parameter of all blocks in a model at once

instead of being forced to switch it for each block separately. Then again, it is still possible to assign a fixed value to the *method* parameter of certain blocks if this is desired: simply open the parameter window of the block in question and modify the parameter *method* manually. Thereby, *method* becomes independent of `world.qss`, and hence, a modification of *qss* in the WorldModel will not effect it anymore.

Note that blocks that lack a parameter *method* produce output events that are valid for any type of QSS, and are never influenced by the WorldModel.

5.2 Source Blocks

Currently, there are 13 source blocks available, whereby ten of them are “genuine” source blocks and the rest are transformator blocks that take input signals from non-DEVS models and transform them into signals that can be processed by subsequent DEVS blocks. Despite the fact that these components are not sources in a strict sense of the word[1], they are located in the *SourceBlocks* package since they act as sources to components in the ModelicaDEVS library. The fact that they accept standard Modelica signals is not visible to a ModelicaDEVS model: there is no possibility to send a DEVS signal to a transformator block which makes them look like source blocks to ModelicaDEVS components.

Note that the signals generated by “genuine” source blocks could also be obtained by taking the corresponding components from the standard Modelica library and transform their output by one of the transformator blocks (SamplerLevel, SamplerTime and SamplerTrigger, see Sections 5.2.11 - 5.2.11).

Every block in the *SourceBlocks* package generates a signal of a certain basic shape (which is indicated by the block’s name, such as “sine”, “step”, “triangular”, and many others. This basic shape can then be adjusted to the needs of the modeller by means of parameters that provide the possibility to set values for the amplitude, the frequency, and so on.
Most of the time, the signals allow for more than one suitable combination of parameters in order to describe them sufficiently. In particular, there are two cases that are worthwhile to take into consideration when determining the parameters through which the signal trajectory of a source block may be modified. The step and pulse signals shall serve as simple examples to illustrate the two cases:

- For the step signal, in order to provide parameters to set the step ground value and the step height, either the pair *lower/upper value* or the pair *offset/amplitude* can be used. Although the paradigm of the two alternatives is rather different (absolute values vs. relative values), they provide the same possibilities to modify the signal’s exact trajectory and hence, the choice of which one to use is rather a question of taste. In connection with other blocks, though, there is one argument that favours the choice of the *offset/amplitude* pair: in order to maintain a consistent structure throughout all blocks, the parameter sets of the various components are kept as similar as possible.

[1] A source block is a block that does not take input but only generates output.

Given that the parameter set of some signals (sine, triangular, etc.) by tradition contain the *amplitude* parameter (amongst others), it seemed to be more convenient to maintain this convention and to adapt the other components (such as the Step block, the Pulse block, etc.).

- To determine at what time instants the pulse signal has to jump to the upper value and when it has to fall back, either the pair *start/end time* or the pair *start time/time high* can be used.
  In this case, again, the latter pair was chosen, but for a different reason: if the user is allowed to determine the start and the end time explicitly by absolute values, it is possible that, by mistake, he sets the end time to a value smaller than the start time, which would not make sense and therefore would lead to wrong results. A solution to this problem would be a) to check always if the two values are correctly set and to print a message[2] if they are not, or b) to make use of the parameter pair *time start/time high* which leaves no room for unintentionally badly set parameters, anyway.

Due to the above considerations, the programming of some blocks in ModelicaDEVS compared to PowerDEVS are slightly different regarding the subject of parameters. Nonetheless, ModelicaDEVS provides the same modification possibilities, only the handling of the blocks differs.

Another difference between ModelicaDEVS and PowerDEVS is the dissimilar code structure of source blocks compared to function or sink blocks: given that the simulator framework of PowerDEVS (see Section 3.2.2) expects blocks to be constituted by the five functions `init()`, `ta()`, `dint()`, `dext()` and `lambda()`, PowerDEVS is forced to describe the behaviour of each block by means of these functions – even if it were possible to choose an easier approach – otherwise, they could not be used by the simulator framework. ModelicaDEVS, on the other hand, has to respect only one constraint: at every output instant, the values of `yVal[1]`, `yVal[2]` and `yVal[3]` have to be updated and the value of the variable `yEvent` has to be set to true in order to inform the next block about the changed output vector. The way **how** this is done does not matter. Thus in ModelicaDEVS, not all source blocks, but only those that produce output at non-equidistant time instants, follow the traditional DEVS way of generating output events: they make use of the variables `sigma` and `dint` in order to determine the next output time instant. The other components are programmed to compute these event instants in a more direct way, for example by means of a when-statement that is periodically activated in certain intervals:

```
when (sample(0,interval)) then
```

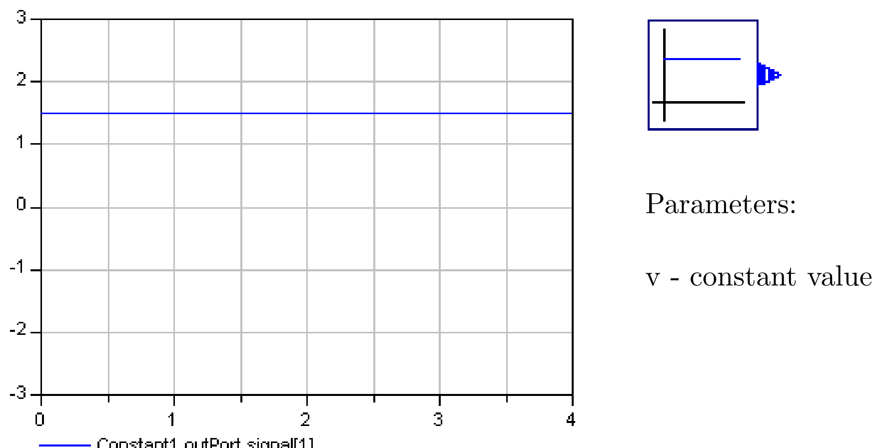
Note that also some of the function blocks could have been programmed in a simpler way than it is foreseen by the DEVS formalism. In that case however, the formal structure has been perpetuated because those blocks participate actively in the model and should respect the conventions of the DEVS formalism in order to participate in a genuine DEVS simulation, whereas source blocks just contribute input signals.

The following paragraphs present the different source components of the ModelicaDEVS library. To this end, an example of a possible output trajectory and the block icon are

[2] e.g. by using the `assert()` function of Modelica: `assert(`*condition*`, ''`*text to print*`'');`

given, and the block is described in detail by listing its parameters and revealing the output event instants. In case it is necessary, further explanations regarding the block behaviour/implementation are added.

5.2.1 Constant

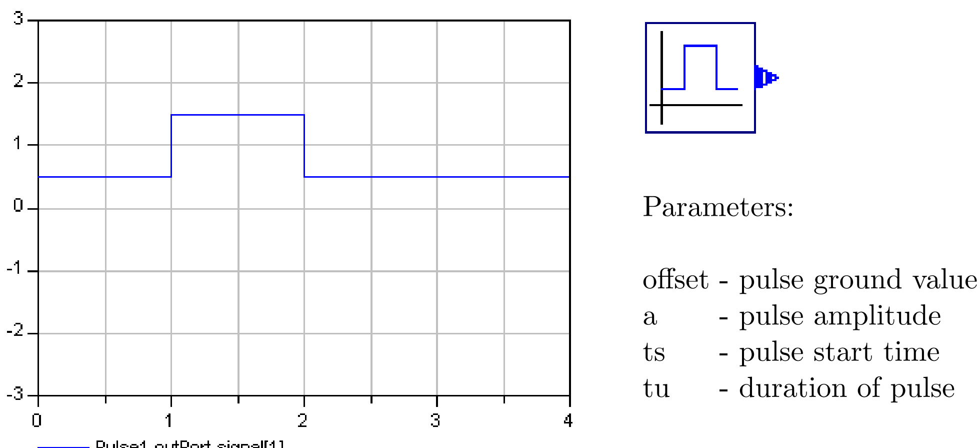


Parameters:

v - constant value

The only output event is generated at time $t = 0$ and takes the value determined by v.

5.2.2 Pulse



Parameters:

offset - pulse ground value
a - pulse amplitude
ts - pulse start time
tu - duration of pulse

Events are generated at time $t = 0$, $t = ts$ and $t = ts + tu$. The first and third event carry the value *offset*, the second one holds a value of *offset*+*a*.

Note that it is possible to assign a negative value to the amplitude, such that the pulse signal is rather an indentation than a bulge respective to the ground value.

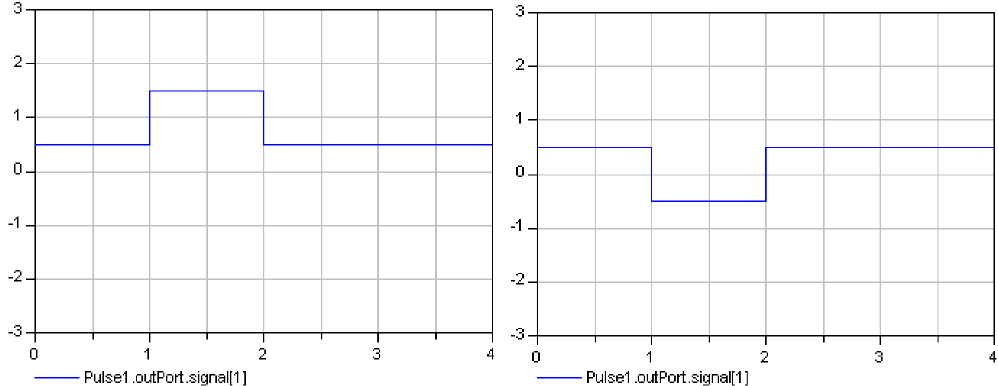
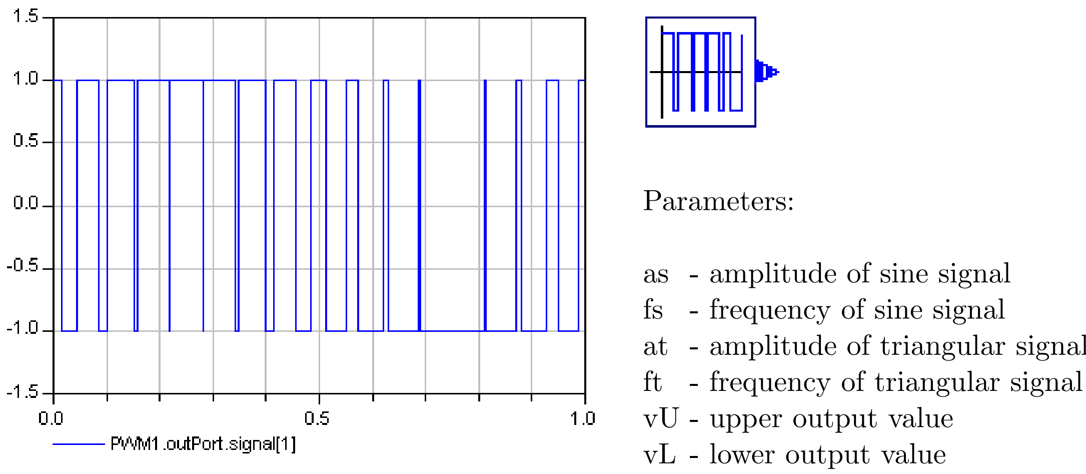


Figure 5.2: Difference of applying a negative amplitude instead of a positive one to a pulse signal.

5.2.3 PWM Signal



Parameters:

as - amplitude of sine signal
fs - frequency of sine signal
at - amplitude of triangular signal
ft - frequency of triangular signal
vU - upper output value
vL - lower output value

The PWM block compares a sine signal to a triangular signal and gives the following output: vU if the sine signal is bigger than the triangular signal and vL if the sine signal is smaller. Outputs are generated each time the two signals (sine/triangular) cross each other.

This block is an example demonstrating the possibility to reuse models as blocks within other models: instead of programming the PWM signal explicitly, it is actually an assembly of three other blocks: a Sine source, a Triangular source and a Comparator. The functionality of these three blocks is packed into one block – the PWM block – which can be reused as a component in other models. Figure 5.3 shows the block icon (on the left) and the block's interior structure (on the right) that generates the signal described above.

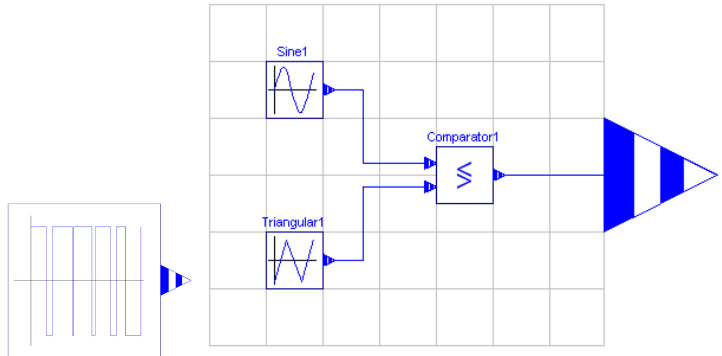
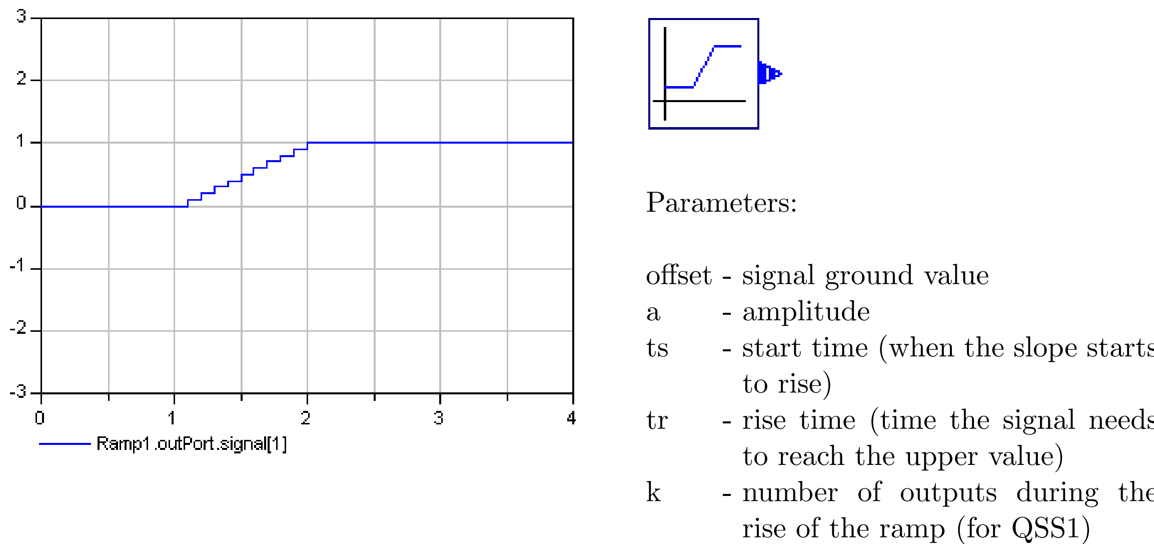


Figure 5.3: The outside and the inside of the PWM block.

5.2.4 Ramp



Parameters:

offset - signal ground value
a - amplitude
ts - start time (when the slope starts to rise)
tr - rise time (time the signal needs to reach the upper value)
k - number of outputs during the rise of the ramp (for QSS1)

The trajectory shown above is the one obtained when using the component in QSS1-mode.

The easiest way of describing a ramp signal would be by a) indicating the time instance when the signal starts to rise and b) giving the slope of the trajectory. However, provided that it is not possible to indicate a slope in QSS1, the ramp trajectory has to be described by means of a piecewise constant function. To this end, it is broken into pieces, which allows the component to launch output events during the rise of the ramp signal.
In PowerDEVS, the ramp signal is quantised, so the time instants when to emit the “ramp events” correspond to the time instances when the original trajectory would cross the next quantum level. Unfortunately, this approach causes problems when the amplitude is not a multiple of the quantum: the final value does not adopt its defined value (*offset*+*a*) but a value in its proximity that is a multiple of the quantum. Therefore, ModelicaDEVS does not make use of quantum steps, but generates equidistant outputs (this can be thought of as a sampling of the original ramp trajectory). However, except for the cases when *offset*+*a* is not a multiple of the quantum, ModelicaDEVS and PowerDEVS produce of course the same

results.
In QSS1, events are generated at time instants $t = 0$ and $t = ts + i \cdot \frac{tr}{k}$ for $i \in \{0, 1, 2, .., k\}$. The corresponding values are *offset* for $t = 0$ and *offset* $+i \cdot \frac{a}{k}, i \in \{0, 1, 2, .., k\}$ for the other time instants.

For QSS2 and QSS3, the signal is described by means of a value and a slope, where the latter one is zero before the time instant $t = ts$ and after the signal has reached its final value, and $\frac{a}{tr}$ in between. The output in Dymola, however, is not very illustrative, since the first output value jumps from the lower to the upper value at the time $t = ts$ and the second output value (the slope) jumps from zero to $\frac{a}{tr}$ at the time $t = ts + tr$ and back to zero after $t = ts + tr$. Figure 5.4 illustrates this behaviour.

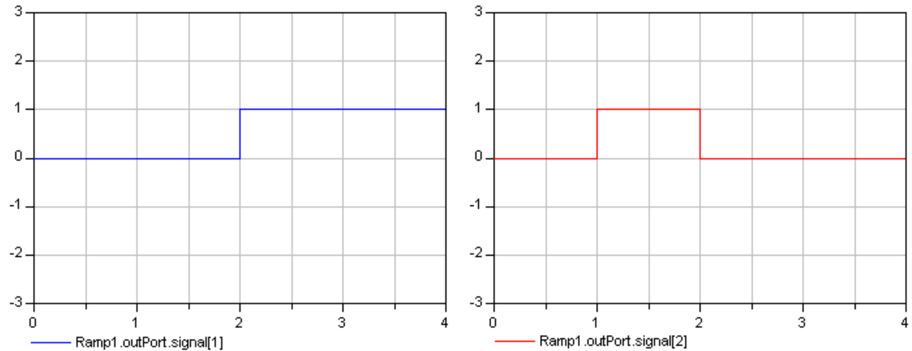
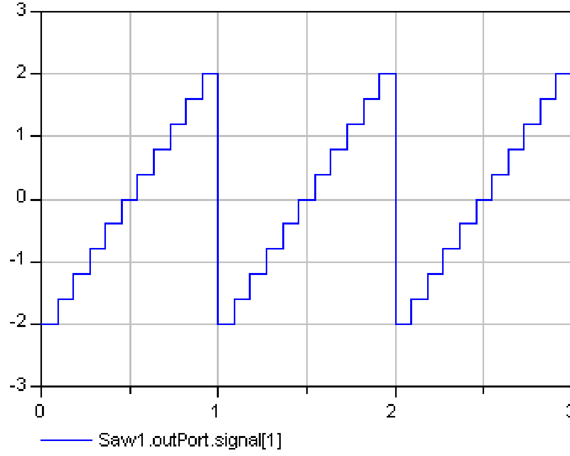
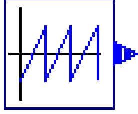


Figure 5.4: The output of the Ramp block in QSS2/QSS3-mode. The function value is plotted on the left (blue), the coefficient of the linear term of the Taylor series expansion (red). The coefficient of the quadratic term is always zero.

Assigning a negative value to the amplitude results in a mirroring of the signal at the straight line $y =$*offset* (see the Pulse block under 5.2.2 for an example).

5.2.5 Saw





Parameters:

a - amplitude
f - frequency
k - number of outputs per period (for QSS1)

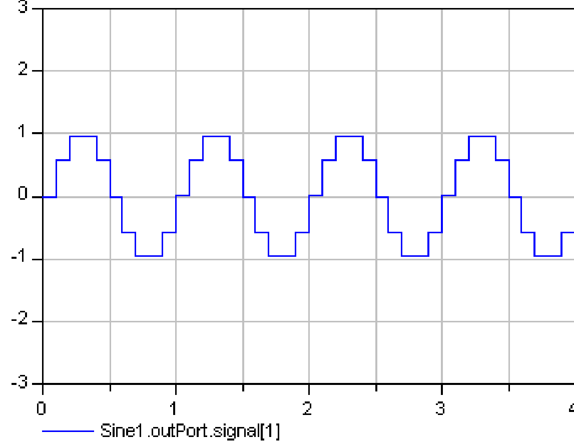
Again, QSS1 needs the signal to be described in terms of a piecewise constant stair function (see the description of the Ramp block under 5.2.4 for more details). The parameter k defines the time instants when to generate the outputs.
As can be seen in the above picture, the saw signal starts with the lower value ($-amplitude$) at the beginning of a period, but takes the upper value ($+amplitude$) before the end of the period. The reason for this is that it is not possible to make it take the upper value exactly when the period is over, since at that time instant it also would have to take the lower value in order to start the next period. Thus, only the start or the end of a slope can take the theoretically correct value, but not both.
The output events are again equidistantly distributed at instances $t = i \cdot \frac{1}{f \cdot k}, i \in \{0, 1, 2, .., k\}$.

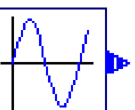
For QSS2 and QSS3, events are generated at time $t = i \cdot \frac{1}{f}, i \in \mathbb{N}$ (i.e., at the beginning of every new period). Since both the starting value and the slope are the same at every saw "tooth", the values of the first two entries in the output vector are constant. The signal can be reconstructed by means of the auxiliary variable `showEvents` that changes its value each time a new period starts.

The Saw block in PowerDEVS has an additional parameter that determines whether the slopes of the signal are rising or falling. In ModelicaDEVS, falling slopes can be obtained by assigning a negative value to the amplitude.

Assigning a negative value to the amplitude results in a reflection of the signal at the x-axis (see the Pulse block under 5.2.2 for an example).

5.2.6 Sine





Parameters:

a - amplitude
f - frequency
phi - phase
k - outputs per period
start - start time of the output series

The well known first three parameters describe the customary properties of a sine signal. The last two parameters are used to generate the output: since the signal has to be understandable by DEVS blocks, the originally continuous sine signal has to be broken down into events.
Parameter k declares how many outputs per period are desired (duration of a period = 1/f). These outputs can be seen as samples of a normal sine signal.
Parameter *start* indicates when the output series starts. This enables a sampling of the sine

signal not only at time instants $t = 0 + i \cdot \frac{1}{f \cdot k}, i \in \mathbb{N}$, but also at time instants $t = start + i \cdot \frac{1}{f \cdot k}$ where *start* is the sampling start time and acts as an offset to the multiples of $\frac{1}{f \cdot k}$.

Assigning a negative value to the amplitude yields a mirroring of the signal at the x-axis (see the Pulse block under 5.2.2 for an example).

Note that the output of the Sine block may slightly differ from the output of the corresponding PowerDEVS sine signal: when simulating a Sine block with $k = 10$, $a = f = 1$ and $phi = start = 0$, its output both in ModelicaDEVS and PowerDEVS looks more or less as shown in Figure 5.5. In particular, it can be seen that at time $t = 0.5$ the coefficient of the quadratic term of the sine signal Taylor series expansion crosses the zero-line. What remains hidden, however, is that the actual value is not 0 but 3.26e-14. This alone would not cause any problem yet, if it were not for PowerDEVS to assign this coefficient a slightly different value, namely -5.00424e-5. While even the mere difference in magnitude of 10^9 could cause different behaviours in blocks that use the sine signal as an input source, our case is even more problematic, because in PowerDEVS, the coefficient of the quadratic term of the Taylor series expansion takes a negative value.
In extreme cases, these variations can lead to fairly different simulation results of two analogous models, once built in PowerDEVS, once in ModelicaDEVS. Usually, the global trajectories of the systems remain more or less the same in both models, but locally, around the point where the variations in the two sine signals occur, there may be remarkable different output trajectories.
This is unfortunately not a problem that can be solved easily, since both approximative values for zero have their justification (they are close enough to zero to represent its theoretical value), and moreover, they depend on the internal definition of the `sin()` function (C++/Modelica specific).

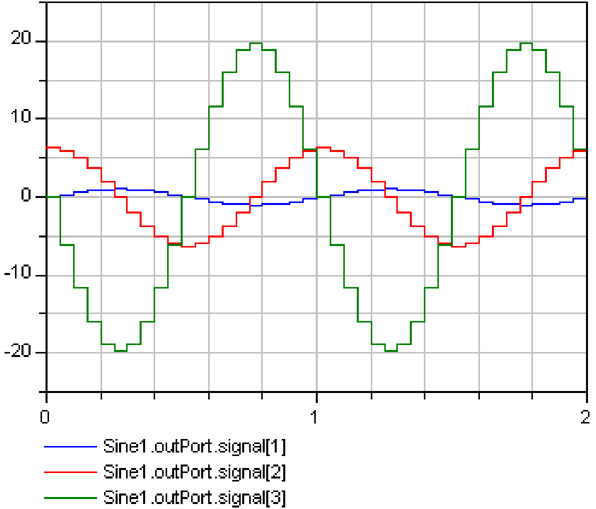
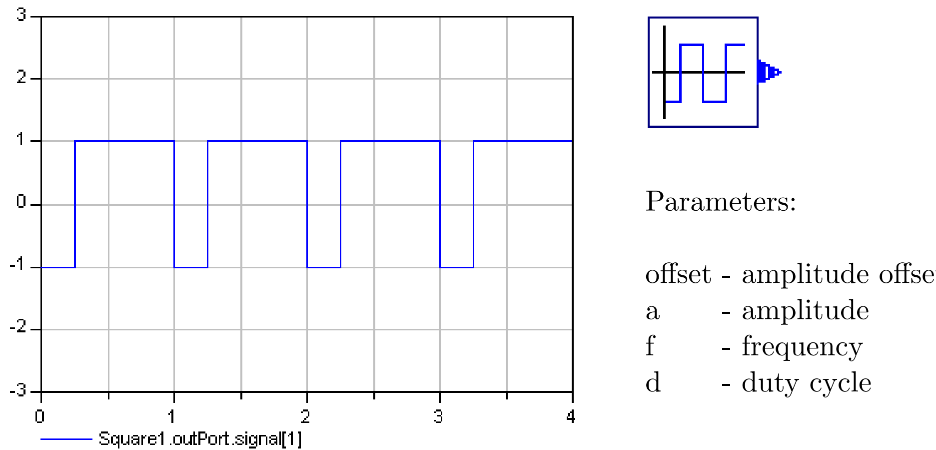


Figure 5.5: Sine output (blue), the coefficient of the linear term of the Taylor series expansion (red) and the coefficient of the quadratic term (green).

5.2.7 Square



Parameters:

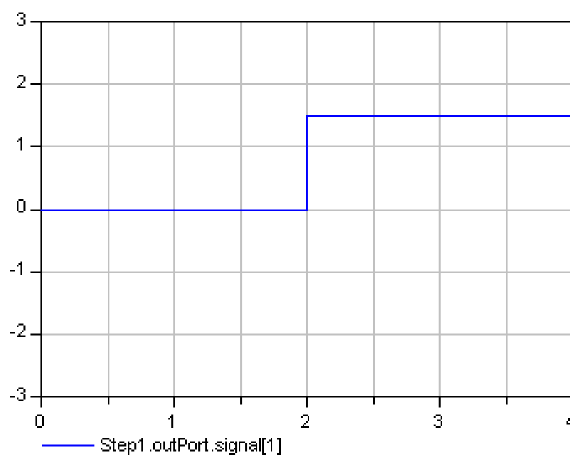
offset - amplitude offset
a - amplitude
f - frequency
d - duty cycle

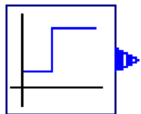
Events are generated at the beginning of every period ($t = i \cdot \frac{1}{f}, i \in \mathbb{N}$) and at the beginning of the duty cycle ($t = i \cdot \frac{1}{f} + \frac{1}{f(100-d)}, i \in \mathbb{N}$). The duty cycle indicates the percentage of the period in which the signal is positive, i.e. when the output event takes a value of $y = offset + a$. During the rest of the period, the signal is equal to $y = offset - a$. For the graph above a duty cycle of 75% has been chosen.

Assigning a negative value to the amplitude results in a mirroring of the signal at the straight line $y = offset$ (see the Pulse block under 5.2.2 for an example).

Note that the parameter *offset* is only present in the ModelicaDEVS Square block, but not in the original PowerDEVS version. This is due to the fact that during the testing phase (Chapter 6) it turned out to be convenient to have a block that produces an output that oscillates between 1 and 0 in order to imitate a boolean pulse.
However, in PowerDEVS such a boolean pulse can be represented by a Square block that oscillates, say, between -1 and 1, and a Comparator which compares the square signal to a constant value of 0.999 (slightly smaller than 1) and produces outputs which take values of either 0 or 1.

5.2.8 Step





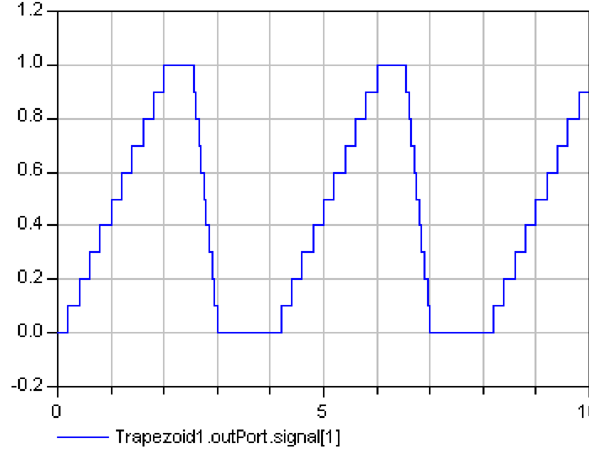
Parameters:

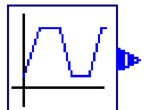
offset	- signal ground value
a	- amplitude
ts	- start time (when the signal executes the "step")

Two events are generated: the first one at $t = 0$ (value: *offset*), the second one at $t = ts$ (value: *offset+amplitude*).

Assigning a negative value to the amplitude is equal to reflect the signal at a straight line $y =$*offset* (see the Pulse block under 5.2.2 for an example).

5.2.9 Trapezoid





Parameters:

offset	- signal ground value
a	- amplitude
tr	- rise time (how long does it take to reach the upper value)
tf	- fall time (how long does it take to reach the lower value)
tu	- time up (how long does the signal stay on the upper value)
td	- time down (how long does the signal stay on the lower value)
k	- number of outputs during the slopes (for QSS1)

For the output shown above, the following parameter setting has been chosen : *offset=0*, $a = 1$, $tr = 2$, $tf = 0.5$, $tu = 0.5$, $td = 1$, $k = 10$.

For QSS1, the slopes need to be described by a piecewise constant stair function (see the description of the Ramp block under 5.2.4 for more details). The parameter k defines how many steps (events, in more technical terms) are used to describe a slope. Provided that the rising and the falling slopes do not necessarily have to be equally steep, we have to split our output events into two parts, where the first part includes events that occur during a rising slope, and the second part includes events that occur during a falling slope:

- Rising slopes generate events at time instants $t = j \cdot (tr + tu + tf + td) + i \cdot \frac{tr}{k}$ for $j \in \mathbb{N}_0$ and $i \in \{0, 1, 2, .., k\}$
- Falling slopes generate events at time instants $t = tr + tu + j \cdot (tr + tu + tf + td) + i \cdot \frac{tf}{k}$ for $j \in \mathbb{N}_0$ and $i \in \{0, 1, 2, .., k\}$

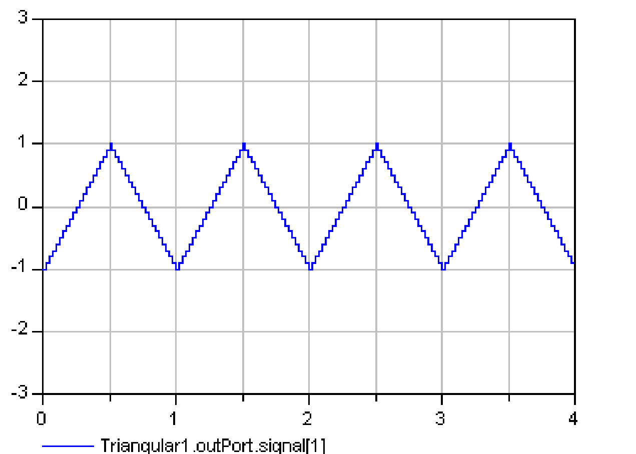
Note that – analogous to the Ramp block – the stairs representing the rising slopes start with a horizontal surface and end with a vertical step, such that the signal takes the correct values in the time instants when the slope starts and when it is finished (as opposed to the Saw block, see Section 5.2.5).

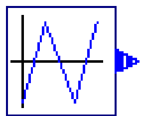
For QSS2 and QSS3, the output events occur when a slope starts or ends:

- Start of rising slope: $t = i \cdot (tr + tu + tf + td), i \in \mathbb{N}_0$. The first entry of the output event (vector) contains the value *offset*, the second one the steepness of the slope: $\frac{a}{tr}$.
- End of rising slope: $t = tr + i \cdot (tr + tu + tf + td), i \in \mathbb{N}_0$. The first entry of the output event contains the value a, the second one zero (the signal becomes a flat straight line, no slope anymore).
- Start of falling slope: $t = tr + tu + i \cdot (tr + tu + tf + td), i \in \mathbb{N}_0$. The first entry of the output event contains the value a, the second one the steepness of the falling slope: $\frac{a}{tf}$.
- End of falling slope: $t = tr + tu + tf + i \cdot (tr + tu + tf + td), i \in \mathbb{N}_0$. The first entry of the output event contains again the value *offset*, but with a the second output entry of zero.

Assigning a negative value to the amplitude results in a mirroring of the signal at a straight line at a y-value of *offset* (see the Pulse block under 5.2.2 for an example).

5.2.10 Triangular





Parameters:

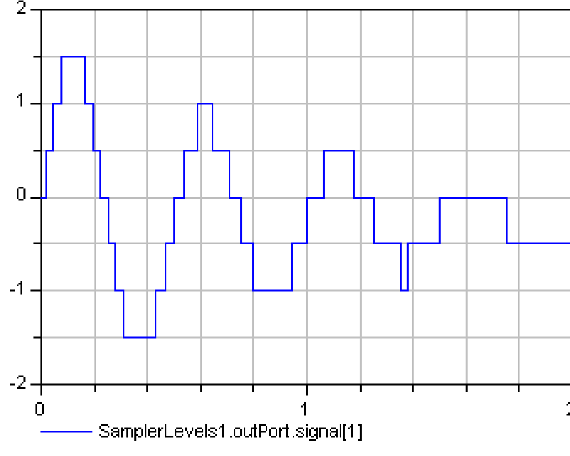
a - amplitude
f - frequency
k - number of outputs per slope (for QSS1)

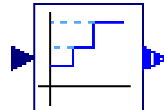
Again, QSS1 needs the signal to be described by means of a piecewise constant stair function (see the description of the Ramp block under 5.2.4 for more details). The parameter k defines the time instants when to produce the outputs.
Analogous to the Ramp or the Trapezoidal block and contrary to the Saw block, the stairs representing the slopes start with a horizontal surface and end with a vertical step, such that the signal takes the correct values at the time instants of the starting and end points of a slope. Event time instants are thus $t = i \cdot \frac{1}{2fk}, i \in \mathbb{N}_0$.

For QSS2 and QSS3, the output events occur at the peaks and the lowest points of the signal trajectory: at time instants $t = \frac{1}{2f} + i \cdot \frac{1}{f}, i \in \mathbb{N}_0$ the output vector takes a value of $(a, -\frac{4a}{f}, 0, true)$ and at time instants $t = i \cdot \frac{1}{f}, i \in \mathbb{N}_0$ the output vector is $(-a, \frac{4a}{f}, 0, true)$.

Applying a negative value instead of a positive one (both with the same absolute value) results in a mirroring of the original signal at the x-axis (see the Pulse block under 5.2.2 for an example).

5.2.11 SamplerLevel





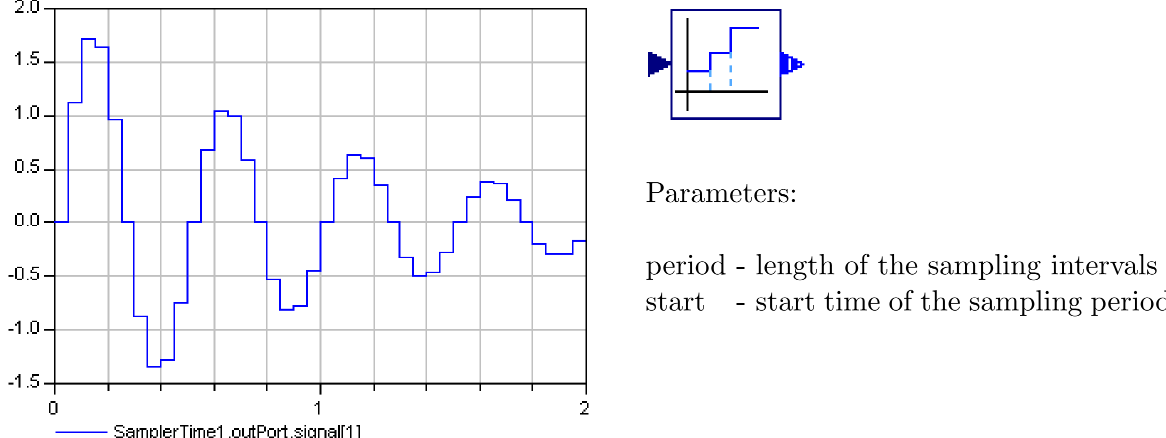
Parameters:

quantum - quantisation degree

For the graph shown above, a damped sine signal was connected to the input port of the SamplerLevel block. The quantum value was 0.5.

The purpose of the SamplerLevel block is to take a continuous signal and to transform it into a signal that is understandable by the DEVS components. To this end, the continuous input signal has to be transformed into events. In the case of the SamplerLevel block, the event instants are given by the points in time when the continuous signal reaches a new quantum level. The quantum levels can be thought to "slice" the original signal trajectory into horizontal stripes – see also Chapter 2.3.2 for details on quantisation.
The event instants are therefore not predictable without knowing the continuous input signal. The only information we have about the output event vectors is that their first entry is a multiple of the quantum (0.5 in this case).

5.2.12 SamplerTime



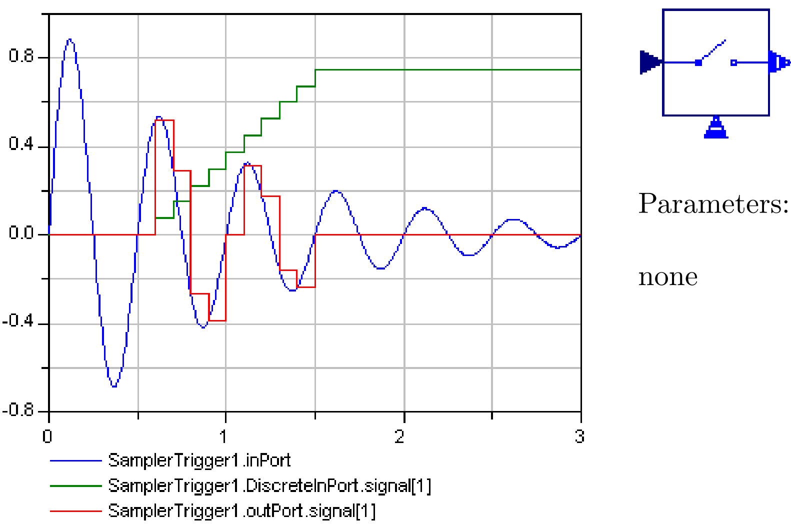
Parameters:

period - length of the sampling intervals
start - start time of the sampling period

Analogous to the SamplerLevel block, the SamplerTime block transforms a continuous signal into a DEVS signal. This time, though, the sampling takes place at discrete time instants. Therefore, the component has parameters for the sampling period and the sampling start time which define samplings at time instants $t = start + i \cdot period$.

In order to illustrate the difference between the SamplerTime and the SamplerLevel block, the same damped sine signal was used as an input to the SamplerTime block, just as it was the case for the SamplerLevel block. The chosen value for the period was 0.05. In contrast to the SamplerLevel block, the values of the first component of the output vectors are no longer equidistantly distributed, but in return, the event instances are multiples of 0.05.

5.2.13 SamplerTrigger



Parameters:

none

This block is a derivation of the CommandedSampler block in the FunctionBlocks package (Section 5.3.2). The only difference is that the SamplerTrigger does not take a discrete signal to sample but a continuous one. Note that sampling the continuous signal at discrete event instants transforms it into a DEVS signal, which is the scope of the Sampler blocks: generating DEVS signals from standard Modelica signals.

Since the time instants when the sampling has to take place are given by the discrete input function, the behaviour of this block does not need further specification by parameters, but rather an additional input port for the discrete function that controls the sampling: at each event that is sent to the SamplerTrigger through the additional input port, the SamplerTrigger samples the continuous input signal. Hence, actual trajectory of the additional input signal is of no importance – it is only used to tell the SamplerTrigger block when it has to sample the continous input signal.

The setup for the sample output above looks as shown in Figure 5.6: a damped sine signal serves as the continuous signal, and the output of the Ramp block in QSS1-mode triggers the sampling series. For a better understanding of this mechanism, both the continuous signal as well as the discrete ramp signal have been plotted: it can be easily seen that whenever the Ramp signal changes its value, which means that an event has been generated, the SamplerTrigger block takes the current value of the damped sine signal and holds it until the Ramp block sends the next event.

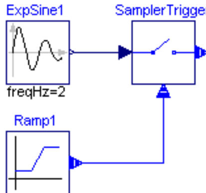


Figure 5.6: Setup for the SamplerTrigger example.

5.3 Function Blocks

Apart from the Integrator block – the core component of ModelicaDEVS – this package contains several blocks to build algebraic functions (binary and unary operator blocks), a Delay block, a Comparator and a Quantiser block.

Since the operator blocks work according to a common scheme, they are not discussed individually but bundled under paragraph 5.3.1. The other blocks are treated one by one since their mechanism is more complex and therefore calls for a more detailed explanation.

5.3.1 Unary and Binary Operators

Additionally to the common operator blocks that are also implemented in ModelicaDEVS, PowerDEVS features two more blocks that are designed to deal with functions, namely the NLfunction (non-linear function, $y = f(u)$) and the ImpFunction block (implicit function $0 = g(u, y)$). Both take a dynamic number of inputs $u_1..u_{10}$, a string[3] defining the function and compute the output y. Due to the dynamic nature of these blocks – the number of inputs is only determined at application time not at implementation time as with the other components – it would have been rather tedious/time consuming to implement their functionalities in Modelica, wherefore they are not included in the ModelicaDEVS library. In most cases, it is however possible to model their functionalities using the common unary and binary operator blocks.

5.3.1.1 Unary Operators

Unary operator blocks are very simple in their structure: they only take one input, apply a function and pass the result to the next block. Their internal and external transition merely consist of setting `sigma` to infinity or zero respectively. See the code of the Gain block for an example (assume that all variables are declared correctly):

```
equation
  dext= uEvent;
  dint= time>=pre(lastTime)+pre(sigma);

  when {dint} then
    yVal[1]= uVal[1]*g;  //g is the gain value set by the user
    yVal[2]= if method>1 then uVal[2]*g else 0;
    yVal[3]= if method>2 then uVal[3]*g else 0;
  end when;
    yEvent = edge(dint);

  when {dext, dint} then
    sigma  = if dint then Modelica.Constants.inf else 0;
    lastTime= time;
  end when;
```

[3]For instance "2*u0+sin(u1)". Note that this string uses the input values $u_1..u_{10}$.

Other blocks in this group are the Inverse block, the Power block and the Sin[4] block. The Power block features a parameter for the exponent value, and the Gain block has a parameter for the value by which the input shall be multiplied. The Inverse and the Sin blocks do not have any parameters.

Given that these blocks only depend on external events and lack a proper internal transition (it simply consists of setting the value of sigma to infinity), their behaviour could be described by a single when-statement that would become active when `dext` becomes true. But this a) would not correspond to the DEVS formalism, which defines both an external and an internal transition, and b) could cause algebraic loops (4.3.4) when connected in a cycle with a Comparator block (5.3.3).

5.3.1.2 Binary Operators

The general structure of the code of binary operator blocks will be illustrated by the example of the Add block:

```
equation
  dext1= uEvent1;
  dext2= uEvent2;
  dint= time>=pre(lastTime)+pre(sigma);

  when dint then
    yVal[1]= u0*pre(newVal11) + u1*pre(newVal21);
    yVal[2]= if method>1 then u0*pre(newVal12) + u1*pre(newVal22) else 0;
    yVal[3]= if method>2 then u0*pre(newVal13) + u1*pre(newVal23) else 0;
  end when;
    yEvent = edge(dint);

  when {dint, dext1, dext2} then
    e=time-pre(lastTime);

    newVal11= if dint then pre(newVal11) else
     if dext1 then uVal1[1] else pre(newVal11)+pre(newVal12)*e+pre(newVal13)*e*e;
    newVal12= if dint then pre(newVal12) else
     if dext1 then uVal1[2] else pre(newVal12)+2*pre(newVal13)*e;
    newVal13= if dint then pre(newVal13) else if dext1 then uVal1[3] else pre(newVal13);

    newVal21= if dint then pre(newVal21) else
     if dext2 then uVal2[1] else pre(newVal21)+pre(newVal22)*e+pre(newVal23)*e*e;
    newVal22= if dint then pre(newVal22) else
     if dext2 then uVal2[2] else pre(newVal22)+2*pre(newVal23)*e;
    newVal23= if dint then pre(newVal23) else if dext2 then uVal2[3] else pre(newVal23);

    sigma   = if dint then Modelica.Constants.inf else 0;
    lastTime=time;
  end when;
```

The mechanism is divided into two parts as recommended in Section 4.2: a first part that becomes active at the occurrence of an internal event, and a second part that is activated by an internal as well as an external event:

[4] Note that the Sin block and the Sine block from the *SourceBlocks* package are different blocks.

- The first when-statement can be simply thought of as the analogon to the lambda output function.

- The second when-statement represents the external and the internal transition. It shows one of the typical properties of the binary operator blocks: usually, only one of the two input ports receives a new signal whereas the other one remains silent, i.e. still holds the value that it had adopted the last time it received a signal. The evaluation of the output of the Add block could now simply depend on the old value of the inactive port and the new value that has arrived through the active one. However, in order to make the computation as accurate as possible, the old value of the silent port is replaced with an estimation of a value that the signal coming in through the silent port is likely to have adopted in the meantime (since the last time when there was an explicit external event). The estimation is carried out on the basis of the signal's previous Taylor series values (the coefficients to the constant, linear and quadratic terms).
  Figure 5.7 shows an example of possible inputs and outputs at the ports of an Add block. For the input signals, the current value as well as the coefficient of the first derivative (which is equal to the linear term of the Taylor series expansion) of the input function are graphically indicated. The output graph labelled "output" gives the true output involving the estimated value for silent ports. The graph labelled "alternative output" shows what the output would look like if no estimation for the function coming in through the silent port took place (which is the case in QSS1 where no higher-order Taylor series expansion is used).

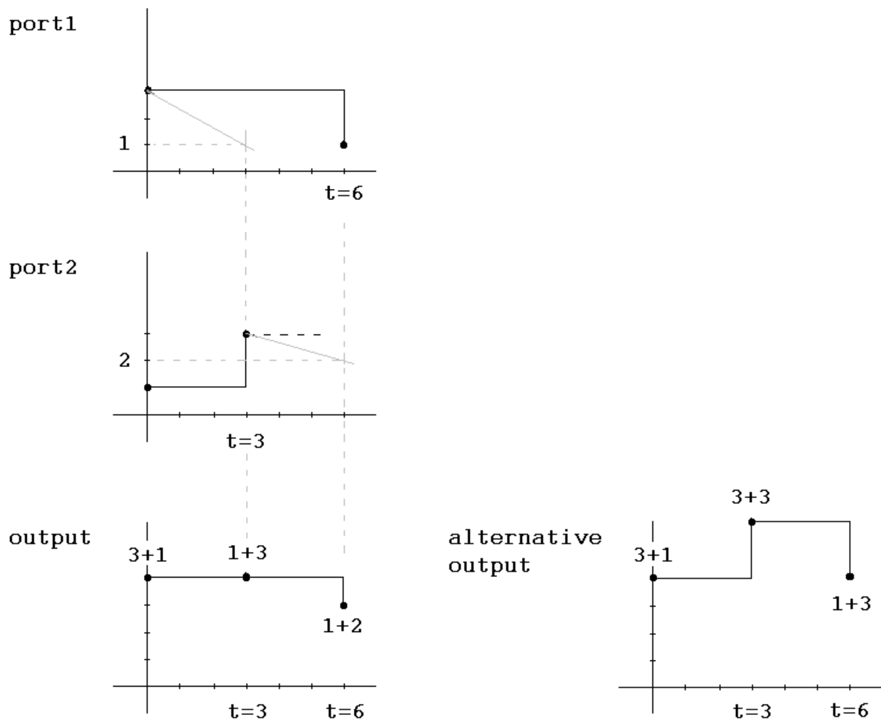
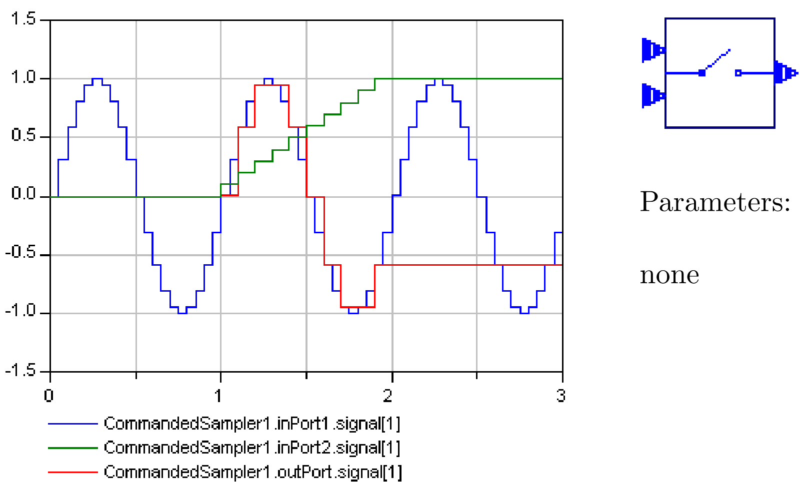


Figure 5.7: Sample input and output trajections at an Add block.

Other blocks in the group of binary operators are the Multiplier and the Divider blocks (where the Divider block is a multi-component model, consisting of a Multiplier and an Inverse block). In contrast to the Multiplier and the Divider blocks which have no parameters, the Add block accepts values to specify the coefficients for the two parts of the sum.

5.3.2 CommandedSampler



Parameters:

none

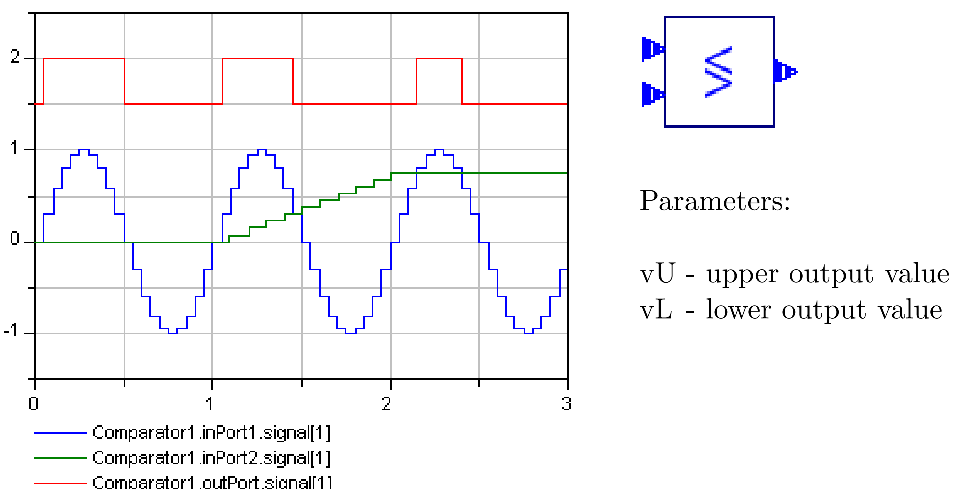
The CommandedSampler is fairly similar to the SamplerTrigger block (5.2.13). It takes two inputs where the first input is the signal from which the samples are taken, and the second input specifies the time instants when the sampling takes place. The second input is an arbitrary ModelicaDEVS block that generates a number of events, as it is in the nature of DEVS models. Usually, it is the values of the single events that we are interested in, given that they constitute the output trajectory. In this case however, for the CommandedSampler block to function correctly, it is only important to have **any** events at certain time instants: when the CommandedSampler receives an event from its second input port, it immediately takes a sample from the signal coming in through the first input port and holds this value until the next event is arriving at its second input port.
Note that both input signals are discrete-time variables in terms of Modelica, i.e. they are piecewise constant.

Normally, the sampling instants do not coincide with the event instants of the first port. This means that at a given sampling instant, the signal from the first port has been updated a certain amount of time ago and may have changed meanwhile (without producing an output event). Hence, analogously to the binary operator blocks (see Section 5.3.1, Figure 5.7), an approximation of the first input port's true value is estimated (on the basis of the linear and quadratic terms of the Taylor series expansion stored in the second and third entry of the input/output vecors).

The picture illustrating the behaviour of the CommandedSampler shows a Sine block (blue) and a Ramp block (green) connected to the input ports of the CommandedSampler. The Sine gives the signal that has to be sampled, the Ramp block (used in QSS1-mode) determines

the sampling instants: we know that every time the Ramp block changes its output value, there must have been an event. Hence, at each of these event instants, the output of the CommandedSampler block (plotted in red) takes the current value of the Sine signal, since it has been triggered by the event sent by the Ramp block to do so.

5.3.3 Comparator



Parameters:

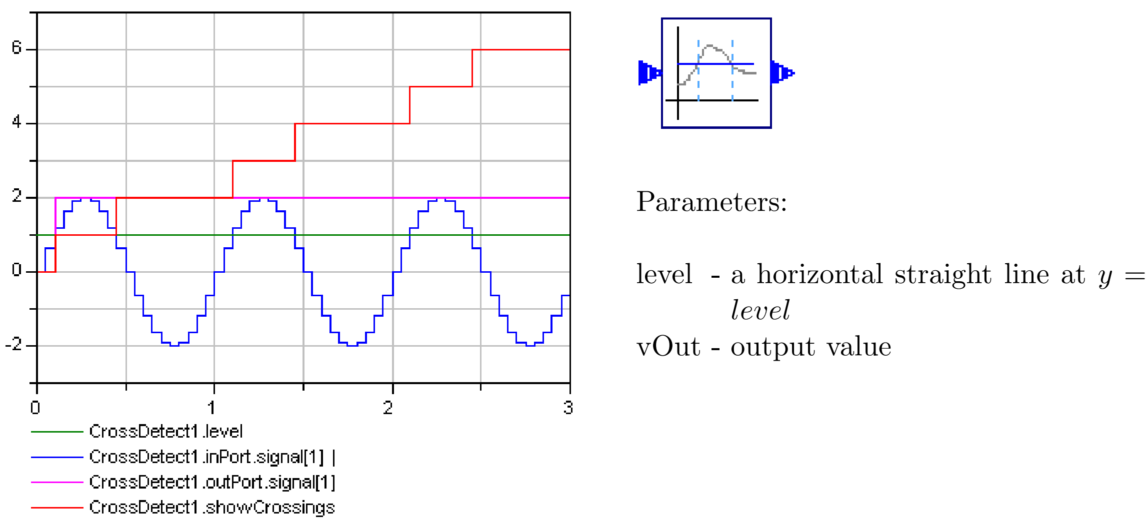
vU - upper output value
vL - lower output value

This block compares two input signals and produces an output event carrying a value that depends on the result of the comparison. The user is invited to specify an "upper" and a "lower" output value (vU and vL). Then, if the first input signal is bigger than the second one, the output takes the value of vU, and vL otherwise.

It is the only component that in case of concurrent transitions (cf. Section 4.3.3) executes the external transition before the internal one. This has the following reason: at each event instant, the comparator estimates the moment when the two input signals will cross the next time (the estimation makes use of the linear and quadratic terms of the Taylor series expansion). The chronological distance between the current and the next event is represented by the variable `sigma`. Suppose now the time instant when the two signals are supposed to cross has arrived. If simultaneously, one of the signals updates its value, it is of course more advantageous to use this new, true value instead of relying on the estimated one. Therefore, in case of concurrent internal and external transitions, the external transition is more important to process, because the values on which the scheduling of the internal transition has been calculated may have become outdated.
Note that because of the reversed priority order of the internal and the external transition, the Comparator block may not be connected in a cycle with itself, unless there is another component in between that breaks the algebraic loop (see Section 4.3.5 for details on algebraic loops).

The picture above shows the output of a Comparator (red) that compares a sine signal (blue) and the output of a Ramp block (green) run in QSS1-mode, such that there are output events during the ramp. vU and vL have been set to 2 and 1.5 respectively.

5.3.4 CrossDetect



Parameters:

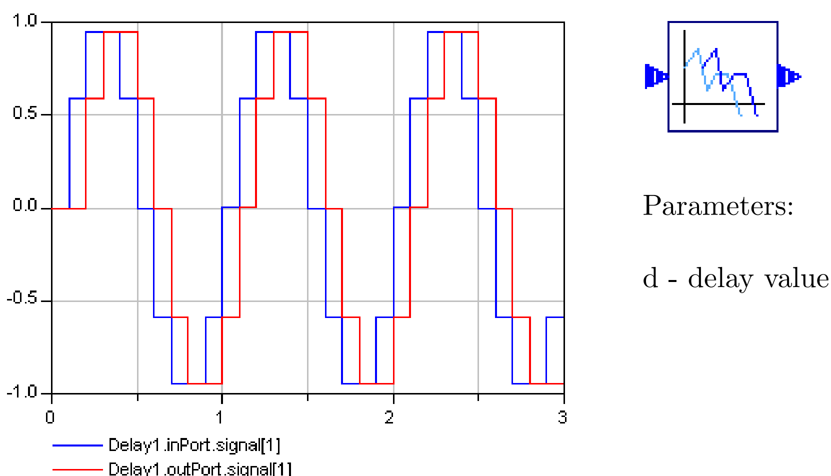
level - a horizontal straight line at $y = level$

vOut - output value

This block detects the time instants at which the input signal crosses the *level* specified by the user, and produces an output with a constant value *vOut*.
It may seem a bit strange that the generated output values are always the same (constant). Considering however that the CrossDetect block is aimed at detecting the time instants **when** a signal crosses the given level, the actual value of the output does not matter, but only the fact that there are output events in the appropriate moments. The interesting output variable is therefore the auxiliary variable `showEvents`. It is a very simple Integer variable that is incremented by 1 each time an output event is generated, the scope of which is just to have a variable with a value that changes at event instants.

The picture above illustrates the behaviour of a CrossDetect block that discovers the crossing instants of a given *level* (green) and the actual input signal, the sine signal (blue). The variable `CrossDetect.uVal[1]` is given in pink, but as can be easily seen it is of little use, since after the first crossing instant it just jumps from its initial value (zero) to *vOut* and stays there. For this reason, also the variable `showEvents`, the time component of the output vector, is depicted. Given the fact that this variable indicates the time instants when the sine signal crosses the level, it can be said to be the actual output of the CrossDetect block while the usual output variable (`CrossDetect.uVal[1]`) is rather a dummy variable, this time.

5.3.5 Delay



Parameters:

d - delay value

The Delay block takes an input signal and outputs the exact same trajectory a given amount of time later (specified by the parameter d).

The above picture shows such an input signal (blue) and the corresponding output signal (red) with a delay of 0.1.

Contrary to the other blocks, which have been programmed in Modelica exclusively, the Delay block exploits the possibility to use externally defined functions programmed in C. The reason for this is that in order to be able to output the values of the input signal d units of time later than they have actually arrived, the Delay block has to first store the incoming event values into an array and read them out again after d units of time. Given that at implementation time, it cannot be known how many values must be stored before the first value is read out again and therefore could be overwritten by a new value, the size of the array has to be sufficiently big such that also big delays are possible. Unfortunately, it is not recommendable to declare too big an array in Modelica, since arrays are treated as a set of scalars, and too many variables in a block make its compilation and simulation become very slow. For this reason, a number of external C functions have been declared, the scope of which is to take care of the implementation parts that involve arrays. Nevertheless, the governing of the block – when to call which transition – lies still within the Modelica model itself. By means of one of the functions defined in the Delay block, it shall be illustrated what it takes to use a C function within Modelica (cf. also [4]):

- Create a C++ file containing all the functions that you want to be able to use from Dymola. In the case of our Delay block the file's name is `DEVSdelay.cpp` and it defines several functions among which also the following one:

```
#ifndef DEVSdelay_C
#define DEVSdelay_C

#include <math.h>
```

```
double ARRu[1000], ARRmu[1000], ARRpu[1000], ARRtime[1000];
int counterIn=1;

double getSigmaDint(int counterOut, double tim, double delay){
  if ((counterOut+1)%1000==counterIn){
    return 4e10;
  }else{
    if (ARRtime[counterOut]+delay-tim < 0){
      return counterOut;
    }else{
      return ARRtime[counterOut]+delay-tim;
    }
  }
}

[...other functions...]

#endif
```

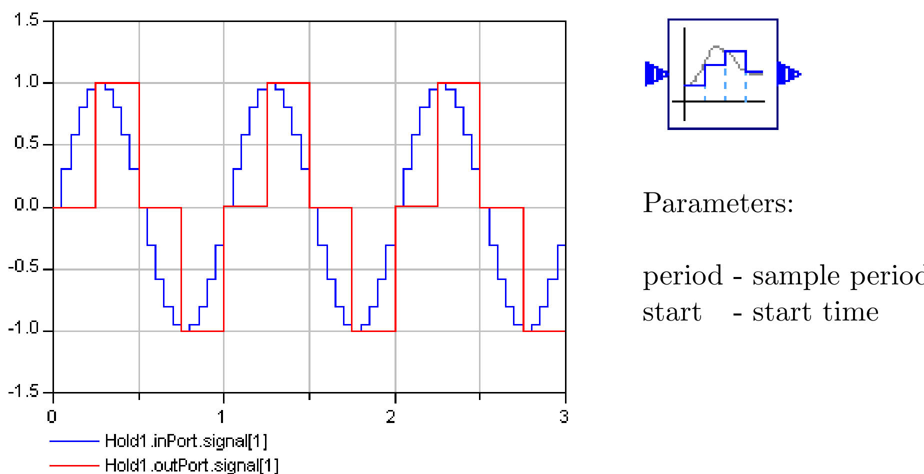
Note that it is important to protect Dymola from including the C++ file more than once. This is done using the `#ifndef` and `#endif` construct.

- Put the file into the folder Dymola/Source since this is where Dymola looks for the C code.
- Now it is possible to declare a Modelica function that acts as an interface to the C function.
  As we could see in the above code snippet, the function `getSigmaDint()` expects three arguments and has a return value. This means we have to declare three input variables and one output value in our Modelica function. The variable declaration section of a function is usually followed by the algorithm section. In the case of an external defined function, this section is replaced by two declarations (lines 6 and 7) that tell Dymola a) that the function is an external one and b) where to find the corresponding C code.

```
1 function getSigmaDint
2   input Integer counterOut;
3   input Real tim;
4   input Real delay;
5   output Real sigma_o;
6 external "C";     //instead of the "algorithm" declaration
7   annotation(Include="#include <DEVSdelay.cpp>");
8 end getSigmaDint;
```

The function `getSigmaDint()` can now be used as if its whole body had been written in Modelica.

5.3.6 Hold



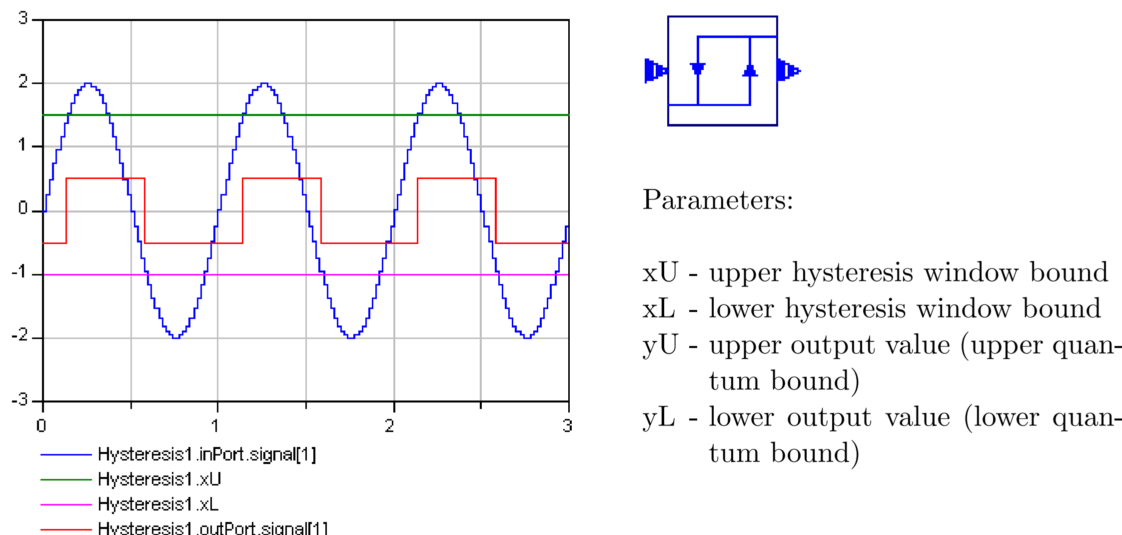
Parameters:

period - sample period
start - start time

The Hold block (called *SampleHold* in PowerDEVS) samples an input signal at periodical time instants which are defined by the parameters *period* and *start*: $t_{sample} = start + period \cdot i$ for $i \in \mathbb{N}_0$. It holds the sampled value until the next sampling instant.

The picture above shows a sine signal (blue) that is sampled (red) at a frequency of $T = 0.25$.

5.3.7 Hysteresis



Parameters:

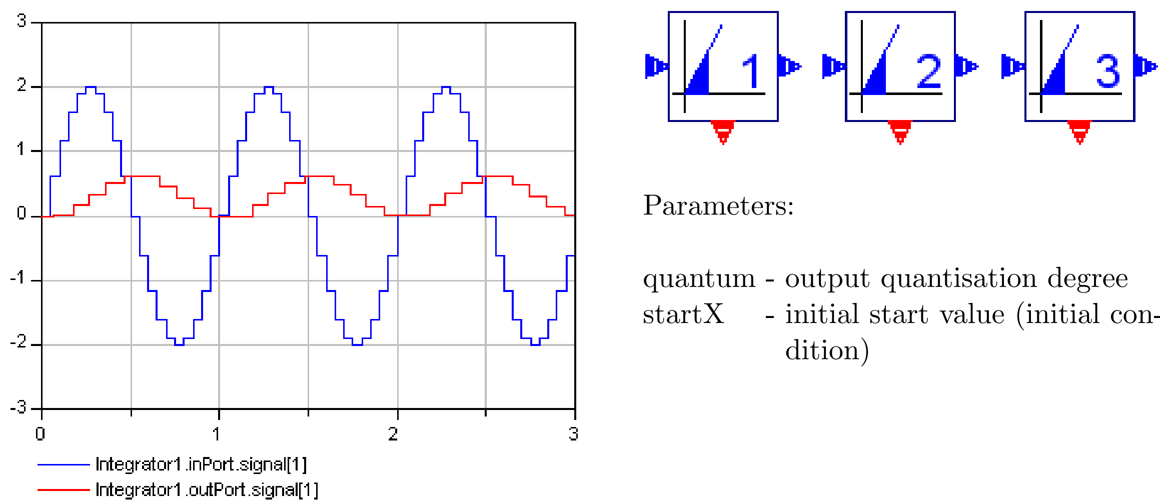
xU - upper hysteresis window bound
xL - lower hysteresis window bound
yU - upper output value (upper quantum bound)
yL - lower output value (lower quantum bound)

The Hysteresis block is a very simple hysteretic quantisation function that maps the trajectory of the input function to only two values (yL and yU). However, the resulting quantisation function is hysteretic: when the input function crosses the value xU from below, the block produces an output of the value yU. Analogously, when the signal crosses xL from above, the Hysteresis block produces an output with the value yL. If the signal crosses either xU from

above or xL from below, nothing happens. This means in particular that the input function can take the same value twice, each time though triggering a different output event.
See also Section 2.3.2 for more details about hysteretic quantisation functions.

The picture above shows an input function in the form of a sine signal (blue) and the output of the Hysteresis block (red) with the following parameter setting: $xU = 1.5$, $xL = -1$, $yU = 0.5$ and $yL = -0.5$.

5.3.8 Integrator



Parameters:

quantum - output quantisation degree
startX - initial start value (initial condition)

The Integrator is, so to speak, the core component of ModelicaDEVS. Its purpose is to integrate a (piecewise constant) function originating from another DEVS block that is connected to the Integrator's input port.

Actually, the library contains three different Integrator blocks – one for each type of QSS. The reason for not having merged them into a single block is that they heavily differ from each other, so putting them together would have led to a block the code of which would have become very complex and thus uncomfortable to read.
However, in order to still be able to exploit the functionality of the WorldModel block (switching between different types of QSS, see Section 5.1), there is an "integrator-wrapper" component that features the same ports and parameters as an Integrator does, but has an additional parameter to choose between the QSS1, QSS2 or QSS3 Integrator. Figure 5.8 shows the internal structure of the model: all three types of Integrator blocks are present, and each of them has its port connected to the corresponding port of the wrapper component. Their instantiation however depends on the value of the *method* parameter (i.e. `world.qss`). See the code snippet below for an illustration:

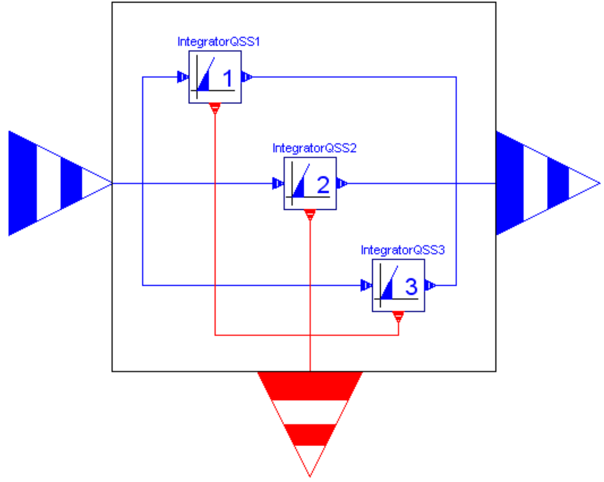


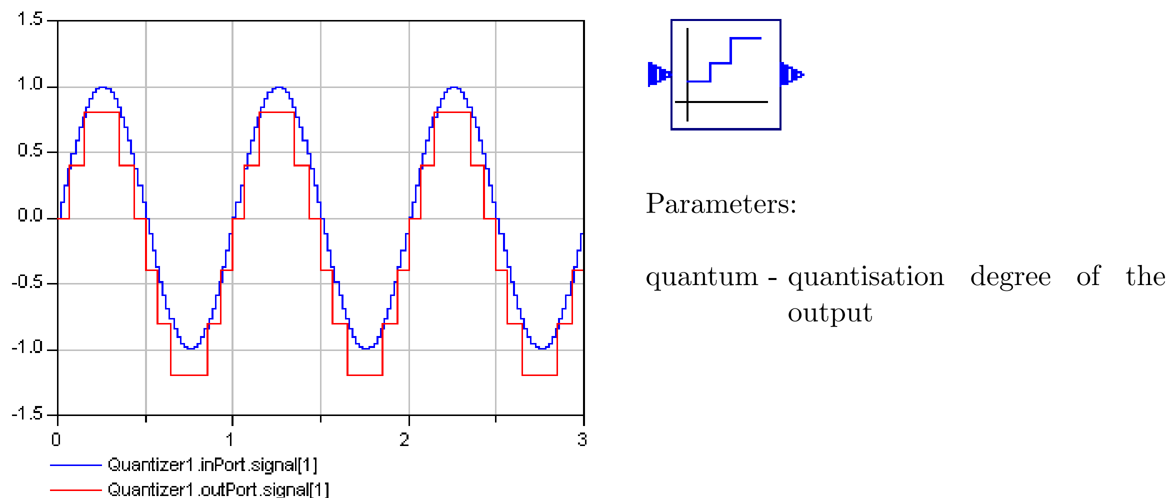
Figure 5.8: The structure of the Integrator wrapper block.

```
1 block Integrator
2   extends ModelicaDEVS.Interfaces.DDBlockIntegratorMulti;
3   parameter Real quantum= 0.1 "Quantum";
4   parameter Real startX=0 "Start Value of x";
5
6   IntegratorQSS1 IntegratorQSS1_1(quantum=quantum,startX=startX) if method == 1;
7   IntegratorQSS2 IntegratorQSS2_1(quantum=quantum,startX=startX) if method == 2;
8   IntegratorQSS3 IntegratorQSS3_1(quantum=quantum,startX=startX) if method == 3;
```

If *method* takes a value of 1, only the QSS1 Integrator is instantiated (line 6), if *method* equals 2, only the QSS2 Integrator is instantiated (line 7), and the same holds for a *method* value of 3 (line 8). It is implied that connections leading to a block are only instantiated if the block itself has been instantiated.
As stated before, the integrator wrapper block has the same parameters (*quantum* and *startX*) as a real Integrator (lines 3 and 4). This circumstance allows the wrapper to initialise the instantiated Integrator block with the appropriate parameter values. It is not very illustrative an example, given that the parameters of a specific Integrator and the integrator wrapper block have the same names, and thus the parameter initialisation unfortunately results in an instruction like `quantum=quantum` and `startX=startX`. Nevertheless, it should be understandable how parameter propagation within a model works (see also Section 5.1 or 5.3.11 for other examples).

The picture illustrating the behaviour of the Integrator shows a sine signal (blue) and the integrated sine function (red).

5.3.9 Quantiser



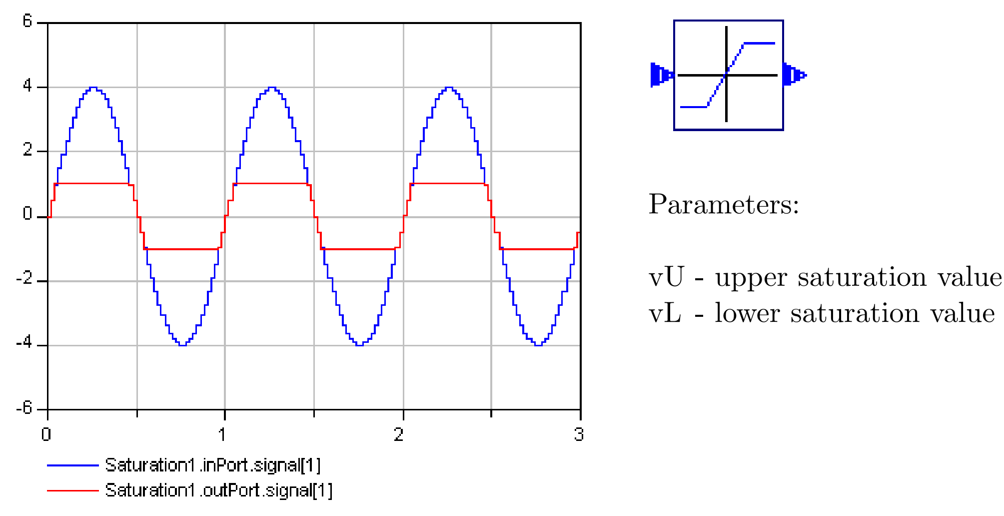
Parameters:

quantum - quantisation degree of the output

A quantiser relates its inputs and outputs by a quantisation function as defined in Section 2.3.2. In the particular case of ModelicaDEVS/PowerDEVS, the Quantiser block is a non-hysteretic function similar to `floor()` that maps the input signal to the quantisation levels that are equidistantly distributed at values $Q_k = \pm k \cdot quantum$ with $k \in \mathbb{Z}_0$.
In other words, if we assume a *quantum* value of 0.1, then the y-axis can be thought to be "cut" into slices of height 0.1, since the quantisation levels are located at values of $y = 0.1 \cdot i$ for $i \in \mathbb{Z}$ – at equidistant y-values starting from zero in both the positive and negative directions. The Quantiser block generates an output event whenever the input signal crosses a quantisation level. The output event takes the value of the current function value; hence it is a multiple of the *quantum* parameter value.
Note that this block is somewhat similar to the CrossDetect block, that however has a fixed output value and only a single level where crossings have to be detected.

The picture above shows a sine signal quantised with a quantum of 0.4. The sine signal itself is plotted in blue, the quantised version in red.

5.3.10 Saturation



Parameters:

vU - upper saturation value
vL - lower saturation value

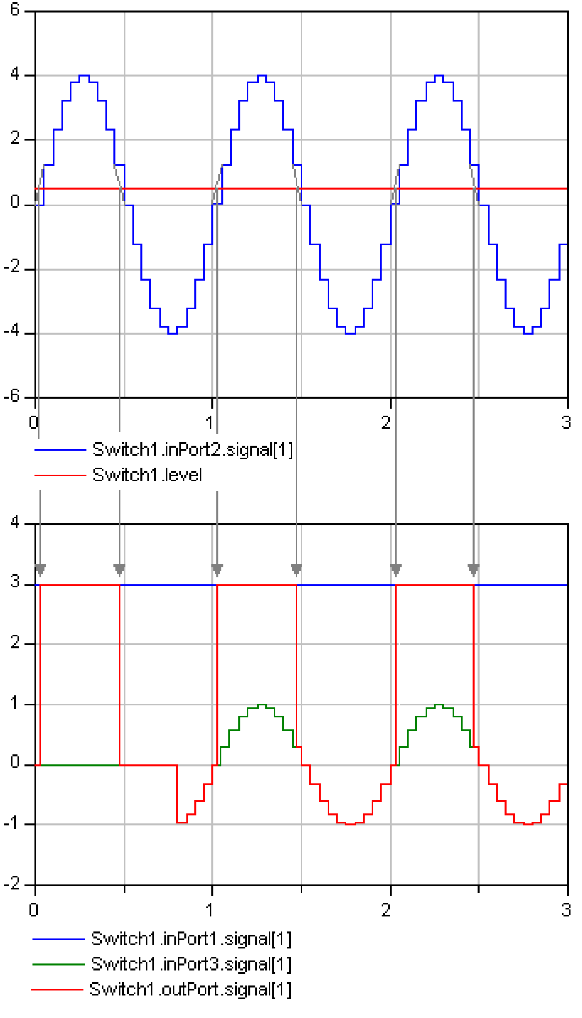
This block basically just propagates the input signal to its output port, but – as its name suggests – truncates it if it takes values outside a certain region. The band where the signal is forwarded without any modification, is defined by the two parameters vL and vU.

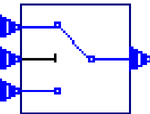
The picture above shows a sine signal (blue) and its truncated version (red). The saturation values are $vU = 1$ and $vL = -1$.

Note that it makes no sense to interchange the upper and lower saturation value. Therefore, ModelicaDEVS guards against such a situation by using the `assert()` function to stop the compilation of the model and print out an error message for the user in case of $vL > vU$.

```
equation
  assert(vU>vL, "Saturation block: vU should be bigger than vL");
  ...[usual instructions for block behaviour]...
```

5.3.11 Switch





Parameters:

level - switching level

The Switch block can be thought to direct either the first or the third input port to its output port. The second input port takes the decision which of the input ports is actually propagated: if the value of the second input port is bigger than a given level, the Switch propagates the first input port, otherwise the third one.

The two pictures above illustrate the behaviour of a Switch block.

- The first picture shows the trajectory of the second input port (a sine signal plotted in blue) and the switching level (red) given by the parameter *level*. The crossing instants of the sine signal with the switching level determines the time instants when the Switch has to flip, thereby changing the source (port 1 or port 3) for its output port.
- The second picture shows the first input port (blue) that is simply a constant value at $v = 3$ and the third input port that is a sine signal that starts only at time $t = 0.8$ (see also 5.2.6 for details on the parameter *start*). The output of the Switch block is then plotted in red. It can be easily seen that when the sine signal in the first picture is bigger than the switching level, the output is constant at the value 3, and when the sine signal drops below the switching level, the Switch output equals the trajectory of the sine signal of the third input port.

Contrary to PowerDEVS, where this block is an atomic model, ModelicaDEVS makes use of the fact that the DEVS formalism allows for hierarchical models: by putting a multi-component model into one of the predefined block hull templates it can be used as a normal block for further models. The interior life of the Switch block looks as shown in Figure 5.9.

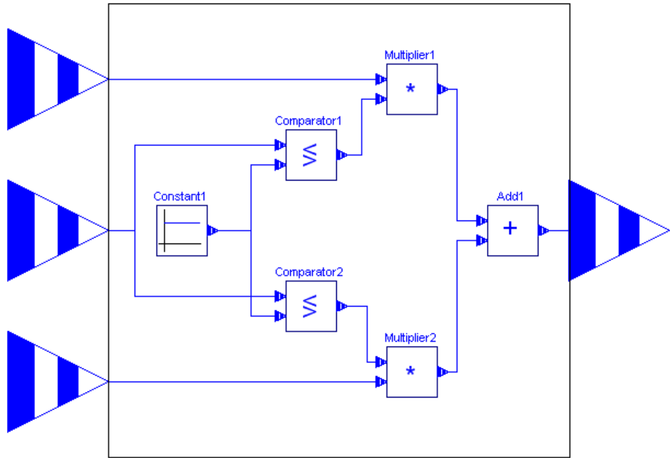


Figure 5.9: The Switch block consists of a set of coupled atomic blocks.

Let us have a look at the purpose of each of the internal blocks.

- The Constant block represents the level to which the second port is compared. Note that since the level is set by the user by means of a parameter (`parameter Real level=0 "Switching level.";`), this value has to be passed to the Constant block which yields the following declaration: `Sources.Constant Constant1(v=level)`.

- The two Comparator blocks evaluate if the value of the second input port is above or below the specified level. To this end, the upper comparator emits an output value of 1 if the value of the second port is bigger than the level and 0 otherwise. The lower Comparator emits an output value of 1 if the value of the second port is smaller or equal than the level and 0 otherwise. In other terms, the Comparators declare which of the two switch input signals (port 1 or port 3) to propagate to the output port: if for instance, the upper comparator emits 0, the lower one by definition emits 1 and vice-versa.
  Note the settings of the output values of the Comparators: $Comparator1.vU = 1$, $Comparator1.vL = 0$, $Comparator2.vU = 0$, $Comparator2.vL = 1$.

- The Multiplier blocks receive the input of the first and the third port, respectively, and a value of the Comparators that determines whether the particular input signal has to be multiplied by 0 or 1. If, for instance, the input of the second Switch port is smaller or equal than the specified level, the upper Comparator emits 0 and the lower one emits 1. Hence, the signal of the third port would be forwarded since its value is multiplied by 1 whereas the upper one is zeroed-out by multiplying its value by 0.

- The Add block merges the two outputs coming from the Multiplier blocks (one of which is always zero and the other one carries the value from one of the input ports of the Switch block).

5.4 Sink Blocks

The package of sink blocks consists of only one component: the Interpolator. It may seem a bit strange that, although it produces output, the Interpolator claims to be a sink. This is however justified considering the fact that even if it accepts DEVS signals, it does not generate any and therefore looks like a sink to a DEVS model. It only produces output targeted at non-DEVS models.

The following sections discuss the need/advantage of such a component, and show how it works.

5.4.1 The Interpolator as a simple Interface

One of the Interpolator's purposes is to provide an interface from a ModelicaDEVS model to a conventional Modelica model by transforming the signals coming from DEVS blocks into signals for non-DEVS blocks. Given that DEVS blocks have input and output ports consisting of a vector of size four[5] whereas normal Modelica blocks have scalar inputs and outputs, the first task of the interpolator is a simple type conversion, such that DEVS signals can be passed to other Modelica blocks.
The Interpolator can be said to be the counterpart to the sampler blocks presented under 5.2.11 - 5.2.13 which take a signal of type real and transform it into a DEVS event.

[5] See Section 4.2.1 for a syntactically correct description of ports.

The simplest way to pass a DEVS-input-signal to a conventional Modelica block is to just forward a truncated version of the input DEVS vector, namely its first entry.

The next section shows however why this approach is not the best one possible, and why it is more advantageous to slightly change the input signal before passing it to the next block.

5.4.2 Function Smoothening

The second purpose of the Interpolator (apart from the type conversion)is to smoothen the piecewise constant trajectory of DEVS state variables. Of course, conventional Modelica blocks are able to process a piecewise constant signal, but this would yield a very poor approximation of the true function.
Remember that simulating a continuous system using the DEVS formalism means to numerically approximate the theoretical trajectories of the state variables. The output values generated by DEVS blocks are then points on that approximating curve. When displayed in Dymola, the output points are connected by piecewise constant straight lines and thus give rise to the well known piecewise constant output function of ModelicaDEVS blocks. Note that given the discrete nature of the state variables, these stair-like trajectories are a precise reproduction of the virtual block behaviour: the state variables keep a certain value for a given amount of time and then jump to another value (see Figure 5.10).

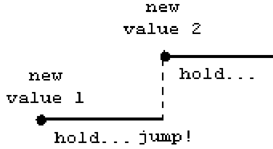


Figure 5.10: State variable trajectory.

A smoother approximation of the true function could be obtained if we were to interpolate two output values by a higher-order curve than a piecewise constant function.
Unfortunately, given the aforementioned discrete nature of the state variables, it makes no sense for Dymola to connect two output events by a straight line, or even a quadratic or cubic curve. So, if we want to have smoother output, we have to interpolate “by ourselves” (see Figure 5.11). Interpolation in this case results in a function consisting of consecutive interpolation segments that are of a higher approximation order than a piecewise constant function.

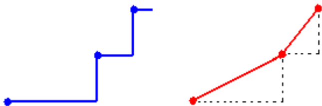


Figure 5.11: Common Dymola output (blue) vs. Interpolator output (red), using first-order approximation (linear function).

For an illustration of the output of an Interpolator (connected to an Integrator) see Figure 5.12, where the green line is the discrete output of the integrator, the red line is the (linear)

interpolation and the blue line is the real solution, computed with the DASSL method of Dymola. The simulated model is the one shown in Figure 3.1.

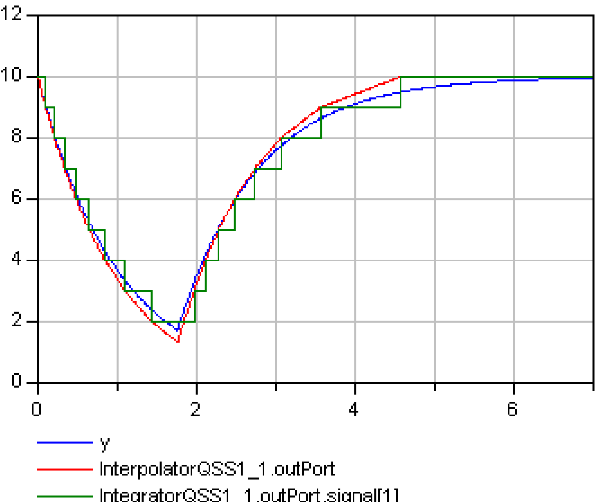


Figure 5.12: The output of the Integrator (green line), the interpolated result (red line) and the numerical integration with DASSL (blue line).

After having established the advantages of an Interpolator component, it will now be explained in more detail how it works. Since the Integrator is the only block the output of which is meaningful to interpolate, let us examine the collaboration of an Interpolator that receives signals from an Integrator block. For simplicity reasons, we shall only consider the first-order case where the connection between two output points are straight lines. Quadradic and cubic interpolation works analogously.

Two points x and $x(t + h)$ can be linearly connected if x and the first derivative $\dot{x}$ are known. Hence, this information has to be given to the Interpolator, in order to make it able to interpolate the output of the Integrator.
Given that the input of an Integrator block is the derivative of its output, the QSS1 (linear approximation) Integrator passes the current function value `X` and its own input value `uVal[1]` to the Interpolator.

```
yIntVal[1]= X;        //coefficient of the constant term of the Taylor series
yIntVal[2]= uVal[1]; //coefficient of the linear term of the Taylor series
yIntVal[3]= 0;        //coefficient of the quadratic term of the Taylor series
yIntVal[4]= 0;        //coefficient of the cubic term of the Taylor series
```

From these values the Interpolator is able to compute the trajectory from the current to the next output value of the Integrator (i.e., interpolate the two points):

```
y=uVal[1] + uVal[2]*(time-pre(lastTime));
```
[6]

where `Interpolator.uVal[1]` contains `Integrator.X`, and `Interpolator.uVal[2]` contains the value of `Integrator.uVal[1]`.

[6] Note that the instructions shown here are not actual code of the Interpolator block. At this point however, they are probably more suited to illustrate the way the Interpolator works than the real formulae would be.

Note that by passing the values `X` and `uVal[1]` to the Interpolator, the Integrator implicitly assumes that there will be no external event between the current event and the next internal one. This assumption is too simplifying of course, but it is the only possible approximation. In the case however that the Integrator indeed has to execute an external transition before it reaches the moment when – according to the value of `sigma` – it would have to execute its next internal transition, the slope of the interpolation probably changes, because due to the external transition that the Integrator has to execute, also the value of `uVal[1]` is likely to change. Thus, in the case of an external event, the Integrator simply sends a new set of `X` and `uVal[1]` to the Interpolator, in order to inform it about the changed situation.
At every arrival of an external event, the Interpolator interrupts the current simulation, adjusts the value of the interpolation slope to the new value received from the Integrator, and restarts the interpolation applying the new slope.

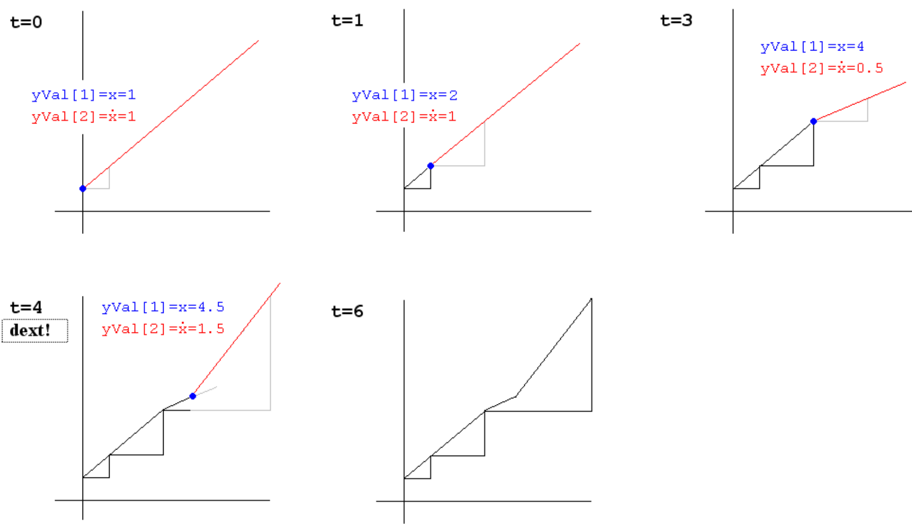


Figure 5.13: Slope adaption by the interpolator because of an external event in the block, the output of which is being interpolated.

Figure 5.13 sketches a scenario where an Integrator executes internal transitions at time instants $t = 0$, $t = 1$, $t = 3$ and $t = 6$ and receives an external event at $t = 4$. At every step, the illustration shows the output vector that the Integrator passes to the Interpolator. The variable `yVal[1]` holds the actual integration value, `yVal[2]` stores the first derivative (which is equal to the Integrator's input port `uVal[1]`).
The trajectory of the Interpolator until $t = 3$ is rather easy to comprehend. At $t = 3$ however the output of the Integrator reaches a value of 4, the estimated next value would be 5, reached in 2 more time steps ($\sigma = 2$). While the Integrator is "waiting" for these two time steps to pass, it receives an external event, so `uVal[1]` is updated (from 0.5 to 1.5), and by executing the external transition also `X` changes (from 4 to 4.5). When the Interpolator receives the new (external) event at $t = 4$, it substitutes the old interpolation slope by the new one. We can identify this substitution by the buckling in the trajectory of the last picture of Figure 5.13 at time $t = 4$.

5.4.3 Interpolator Specific Ports

Given that the Interpolator cannot work with the normal output vectors of DEVS blocks, but requires additional information about the possible future state trajectory, the Integrator features a second output port supplementary to the common one. Through this port, the Integrator emits events consisting of the following four values:

- The current value of the signal that is equal to the first entry of the normal output vector.
- The coefficient of the linear term of the Taylor series expansion; in other words, the first derivative.
- The coefficient of the quadratic term of the Taylor series expansion; in other words, the second derivative divided by 2.
- The coefficient of the cubic term of the Taylor series expansion; in other words, the third derivative divided by 6.
  Note that the third derivative of the Integrator output can be obtained by means of the third entry of the input vector: since the task of the Integrator block is to integrate the input function, the Integrator input vector can be thought of as the derivative (component-wise) of the output vector. As stated before, the output vector of a block holds the coefficients of the first three terms of the Taylor series, this means in particular that the third entry contains the second derivative devided by two. To the Integrator, this third entry looks like the **third** derivative devided by two. Hence, in order to pass the third derivative devided by six to the Interpolator, the Integrator simply devides the third entry of its input port by 3. Figure 5.14 illustrates this circumstance.

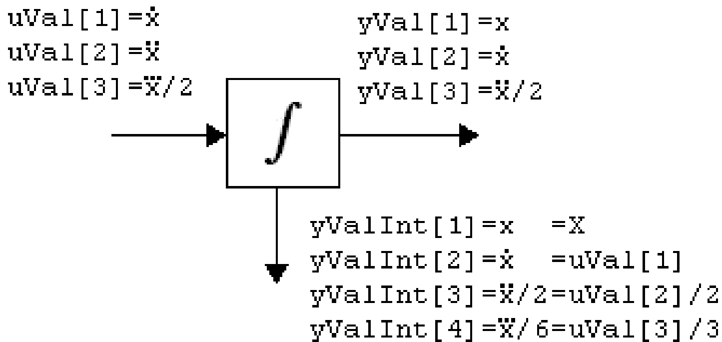


Figure 5.14: Integrator input and output vectors.

- The usual boolean value, which informes the Interpolator that block X has just generated an output event that needs to be processed.

The dissimilarity of the normal output/input vectors and the special vector that is required by the Interpolator is also highlighted graphically in the layout of the blocks: instead of the common blue input ports, the Interpolator has a red input port that is connectable only to the red output port of the Integrator.

5.5 Templates

This package contains the DEVS block specific connectors (input/output ports), and several templates for the different block types. The following list gives an overview over the available block templates:

- BlockIcon: this is a simple model that is inherited by almost every other block. Its purpose is merely to draw the square shape shared by most ModelicaDEVS blocks.
- Continuous-Discrete blocks (CDBlock, CDBlockSpecial): continuous-discrete blocks have a real-type input port, connectable to the output ports of `Modelica/Blocks` components, and a discrete DEVS output port, connectable to DEVS input ports.
  Due to the different types of their input/output ports, the continuous-discrete blocks are used to connect non-DEVS blocks to DEVS blocks (SamplerLevel and SamplerTime, see 5.2.11 and 5.2.12, respectively).
  The CDBlockSpecial is a partial model similar to the CDBlock, but with an additional input port – it is only used for the SamplerTrigger block (see Section 5.2.13).
- DEVS source blocks (DSource): this type of block is used for the most part of the ModelicaDEVS source blocks (except for the SamplerLevel, SamplerTime and SamplerTrigger which are of the type continuous-discrete).
- Discrete-Discrete blocks (DDBlock, DDBlockSpecial, DoubleDDBlock, TripleDDBlock): discrete-discrete blocks are used for pure DEVS blocks that take and generate DEVS signals only.
  The DDBlockSpecial is a normal discrete-discrete block, but features an additional output port through which signals for the Interpolator can be sent (see Section 5.4.3).
  The DoubleDDBlock and TripleDDBlock are used for DEVS models that have more than one input port (e.g. the Add or the Switch block).
- Discrete-Continuous blocks (DCBlock): only the Interpolator block is of this type, because it is the only component that takes DEVS signals – sent through the special (red) output port of other blocks – and generates real-type signals that can be processed by non-DEVS Dymola components.
  The discrete-continous blocks are the counterpart of the continuous-discrete blocks.
- Continuous-Continuous blocks (CCBlock): this type of template is an extension of the `Modelica.Blocks.Interfaces.SISO` model and can be said to be a hull for a DEVS model that shall be inserted as a black box into a model consisting of blocks from the `Modelica/Blocks` library. Its only difference to the original block template is a) the WorldModel component it contains (remember that every model consisting of ModelicaDEVS blocks has to include the WorldModel), and b) the parameter $qssMethod$ (it should be possible from the outside to set the parameter of the WorldModel and thereby influence the QSS mode of the model components. See lines 3+4 in the code snippet below for an illustration how the chosen QSS mode is propagated into the DEVS model).
  The following declarations give a full description of the CCBlock:

```
1 partial block CCBlock
2   extends Modelica.Blocks.Interfaces.SISO
3   parameter Integer qssMethod=3 "Use QSS1, QSS2 or QSS3";
```

```
4   inner Miscellaneous.worldModel world(qss=qssMethod);
5 end CCBlock;
```

Note that if the DEVS model does not have to appear as a black box, the aforementioned CDBlock and DCBlock can be used to create an interface to the non-DEVS models.

- Electrical-Electrical blocks (EEBlock): electrical-electrical blocks are the analogon to CCBlocks – they are used to wrap electrical components built from ModelicaDEVS blocks.
  From an electrical signal to a DEVS signal, two type conversions have to be made: the first one is from electrical to real-type, the second one from real-type to DEVS. Hence, between the electrical positive/negative pins and the first/last DEVS component, there is an additional RealSignal connector (see Figure 5.15 and the code snippet below).

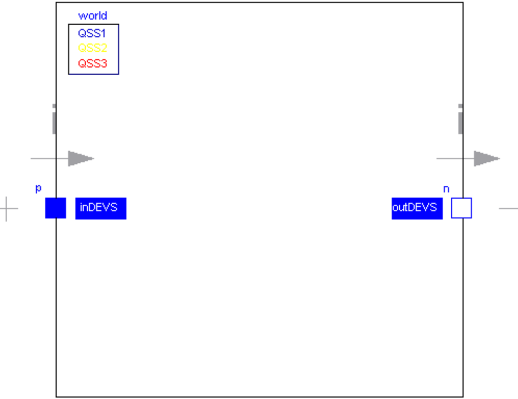


Figure 5.15: The layout of the EEBlock template.

Note that the first/last DEVS block still has to be a transformation block that transforms the real-type signal to a genuine DEVS signal or vice-versa.
The following code snippet gives the declarations of the EEBlock:

```
1  partial model EEBlock
2    extends Modelica.Electrical.Analog.Interfaces.OnePort;
3    parameter Integer qssMethod=3 "Use QSS1, QSS2 or QSS3";
4    inner Miscellaneous.worldModel world(qss=qssMethod);
5
6  protected
7    Modelica.Blocks.Interfaces.RealSignal inDEVS ;
8    Modelica.Blocks.Interfaces.RealSignal outDEVS;
9
10 equation
11
12   inDEVS=p.i;
13   v=outDEVS;
14
15 end EEBlock;
```

Due to the need of a electrical-to-real type conversion, virtual connections from the electrical pins to the real-type ports (inDEVS and outDEVS) have to be drawn. This

is done by the two equations on line 12+13.
It is important to mention that this kind of electrical-electrical block assumes the current to be given and calculates the voltage[7].

[7] Modelica's concept of equations instead of assignments (see Section 4.1.1) entails non-directed data flow, if not otherwise stated by declaring variables to be of type `input` or `output`. DEVS blocks, on the other hand, are always directed. Hence, assumptions about the flow of data have to be made when transforming from electical Modelica components to ModelicaDEVS blocks.

Chapter 6

Testing and Efficiency

This chapter presents various sections that examine a) the possibilities regarding the simulation of mixed/hybrid systems in ModelicaDEVS, and b) the run-time efficiency in comparison with other modelling software systems (Dymola and PowerDEVS).

6.1 Testing

6.1.1 Block Testing

The testing of the single blocks has been done twofold:

- Every block by its own or in the simplest possible compound with other blocks (e.g. function blocks cannot be tested alone since they need input).
- In a more complex model, for example to test the behaviour within loops.

In either case it was checked event by event whether an equivalent PowerDEVS model gave the same results. If there were discrepancies, both the code of Dymola and PowerDEVS were examined for errors in order to assure that the implementation of ModelicaDEVS does not base upon a possible erroneous implementation of the corresponding block in PowerDEVS. However, most of the time the compared results matched to a precision of up to 5 decimal digits. If there were discrepancies, they could be traced back to rounding or Dymola/PowerDEVS specific inaccuracies (see the Sine block under 5.2.6 for an example).

6.1.2 Testing of Mixed Systems

According to Section 1.2, ModelicaDEVS should also be able to be used together with conventional Dymola components. The final goal is of course to model hybrid systems (see Section 6.1.3), but to this end it is first of all necessary to be able to build mixed systems[1].

[1]The term "hybrid system" denotes systems with mixed ModelicaDEVS/Dymola integrators, whereas "mixed system" simply refers to a system that contains both Dymola and ModelicaDEVS blocks, but not necessarily mixed integrators. Hence, a hybrid system is always a mixed system but not vice-versa.

Figure 6.1 shows an example of a simple electrical circuit modelled in Dymola, and in a mixed version with a ModelicaDEVS capacitor.

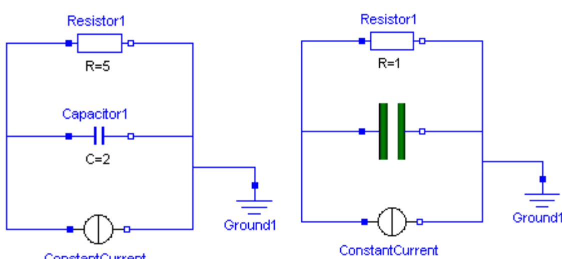


Figure 6.1: Two versions of a simple electrical circuit: once modelled in Dymola exclusively, once with the capacitor replaced by ModelicaDEVS blocks.

Figure 6.2 illustrates the implementation of the ModelicaDEVS capacitor. In order to be connectable to blocks from the electrical library of Modelica, it has to extend the EEBlock of the ModelicaDEVS library (see Section 5.5 for details on the transformation from electrical signals to real-valued signals within the EEBlock template). This block looks like the *TwoPin* block interface provided by Dymola on its outside, but internally performs a type conversion of the input and output signals: the two blocks inDEVS and outDEVS are used to transform the signals from the electrical pins to signals of type real and vice-versa. The second type conversion is done by means of the SamplerTime[2] and the Interpolator blocks that act as interfaces to non-DEVS components with real-valued input/output signals (see Chapter 5). The full textual description of the ModelicaDEVS capacitor is given below:

```
1  model CapacitorDEVS
2    extends ModelicaDEVS.Templates.EEBlock;
3    parameter Modelica.SIunits.Capacitance C=1 "Capacitance";
4    parameter Real period=1e-6 "Sampling period";
5
6    SourceBlocks.SamplerTime SamplerTime1(period=period);
7    FunctionBlocks.Integrator Integrator1;
8    SinkBlocks.Interpolator Interpolator1;
9    FunctionBlocks.Gain Gain1(g=1/C);
10
11 equation
12   connect(SamplerTime1.outPort, Gain1.inPort);
13   connect(Gain1.outPort, Integrator1.inPort);
14   connect(Integrator1.outPortInterpolator, Interpolator1.inPortInterpolator);
15   connect(inDEVS, SamplerTime1.inPort);
16   connect(Interpolator1.outPort, outDEVS);
17 end CapacitorDEVS;
```

[2]Note that it would have been more appropriate to use the SamplerLevel block, because the scope of DEVS integration is to by-pass the time discretisation. However, as we will see later, it was necessary to program a numerical version of the sampler block, in order to enable mixed simulations. Unfortunately, the numerical version of the SamplerLevel block led to aborted simulations due to "inconsistent restarting conditions". With a numerical version of the SamplerTime block on the other hand, the simulation could be performed without encountering any difficulties. Hence for the sake of consistency, it was decided to only use time sampler blocks instead of a SamplerLevel for the CapacitorDEVS block and a SamplerTimeNumerical for the CapacitorDEVSNumerical block.

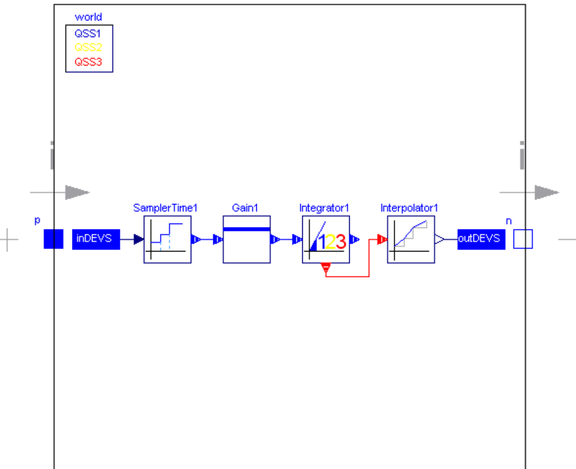


Figure 6.2: The internal structure of the ModelicaDEVS capacitor.

The SamplerTime block (line 6) samples the signal of the electrical current at a period specified by the parameter *sampling* (line 4), and the Gain block (line 9) multiplies the incoming signal by the value of $\frac{1}{C}$, where C is again specified by a parameter (line 3). Taken as a whole, the ModelicaDEVS blocks constitute nothing else than the well known capacitor formula

$$i = C\frac{dv}{dt}$$

in an algebraically modified version:

$$v = \frac{1}{C}\int i$$

Until now, it may seem very simple to replace a component from the standard electrical Dymola library by a ModelicaDEVS equivalent. The problem is however that the electrical components do not assume a certain data flow direction since they are described by acausal equations[3]. The data flow of ModelicaDEVS components on the other hand is directed, since DEVS components feature input and output ports instead of electrical bi-directional pins. This means that our ModelicaDEVS capacitor has to turn acausal equations into causal ones: it assumes it is given the current i, and hence it computes the voltage v. Unfortunately, such a capacitor would not work anymore if we were to connect it to a voltage source instead of a current generator.

An even more severe problem is caused by the SamplerTime block applying the `der()` operator to the signal that it receives through its input port (lines 2 and 8 in the code snipped below):

```
1 equation
2   derivative=der(u); //used to produce der(der(u))
3   yEvent= sample(start,period);
```

[3]An acausal equation $x = y$ is an equation in its literal sense, so it can be understood as $x = y$ or $y = x$. A causal equation on the other hand assigns a variable a certain value, so the meaning of $x = y$ is actually $x := y$. Note that causal equation are also often called assignments.

```
5   when sample(start,period) then
6     yVal[1]= u;
7     yVal[2]= if method>1 then derivative else 0;
8     yVal[3]= if method>2 then der(derivative) else 0;
9   end when;
```

Given that the input of the SamplerTime block depends on the output of the Interpolator in the DEVS capacitor, Dymola would have to differentiate the output of the Interpolator, which unfortunately it is not able to, and hence aborts the simulation with a “Failed to differentiate the equation in order to reduce the DAE index.” error message.
An attempt to solve this problem was made using Dymola’s “User specified Derivatives” feature described in the Dymola Handbook [4]: functions for the first and second derivatives have been inserted into the Interpolator, but due to unknown reasons, this did not solve the issue neither.

In order to be able to perform mixed simulations nonetheless, another trick has been applied: supplementary to the standard SamplerTime block that uses the Modelica `der()` operator, an additional block has been programmed: the SamplerTime**Numerical** block avoids the problem caused by the `der()` operator by means of the `delay()` function that is used to differentiate numerically (concept of difference equations). Instead of the first and second derivatives of the input signal, the SamplerTimeNumerical returns a numerical approximation:

```
du = delay(pre(u),d);
d2u= delay(pre(u),2*d);

yVal[1]= pre(u);
yVal[2]= if method>1 then (pre(u)-du)/d else 0;
yVal[3]= if method>2 then (pre(u)-2*du+d2u)/(d*d) else 0;
```

Using the new sampler block, the mixed simulation could be carried out without any problems, and the results differ only slightly from the simulation with conventional Dymola components (see Figure 6.3).

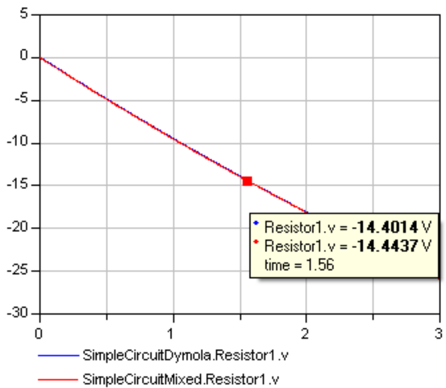


Figure 6.3: The simple electrical circuit of Figure 6.1 simulated with standard Dymola (blue) and mixed Dymola-DEVS components (red).

6.1.3 Testing of Hybrid Systems

An example of a hybrid system – a mixed integration, in other words – is any electrical circuit with at least one ModelicaDEVS capacitor/inductor and at least one Dymola capacitor/inductor. The reason for this restriction is that the mathematical definition of capacitors and inductors both contain differential equations, so if we have for example one conventional inductor, which uses the Modelica `der()` operator, and a ModelicaDEVS capacitor, which uses an Integrator block as described under 5.3.8, we have the required situation of a mixed integration.
As an example for such a hybrid system, we use the flyback converter (described in more detail in Section 6.2.1) with a voltage source connected to one side, and a load to its other. The capacitor in the secondary winding is replaced by an equivalent ModelicaDEVS capacitor, just like in the smaller example in Section 6.1.2.
Note that again, it is the numerical version of the ModelicaDEVS capacitor (CapacitorDEVSNumerical) that has to be used in order not to obtain "DAE index reduction" error messages (see the previous section for more details regarding this error).

Figure 6.4 shows the output of the mixed simulation compared to the result of the standard Dymola simulation. Just like it was the case with the simpler example from Section 6.1.2, the output of the hybrid simulation differs only slightly from the Dymola simulation. Thus, it is also possible to perform not only accurate mixed, but also hybrid simulations.

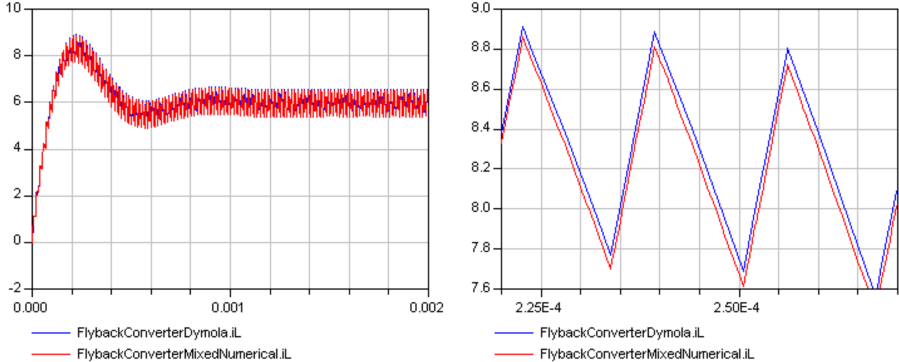


Figure 6.4: The flyback converter simulated with standard Dymola (blue) and mixed Dymola-DEVS components (red).

6.2 Efficiency

This section analyses the performance of the new ModelicaDEVS library in comparison with the PowerDEVS software and conventional Dymola simulation. To this end, a system with frequent switching operations was modelled using each of the three possibilities (PowerDEVS, ModelicaDEVS and Dymola), and the simulation times of the three models were compared against each other.
The chosen system is the flyback converter example presented in [7].

The following sections discuss the modelling of the flyback converter in the respective modelling environments and conclude with a comparison of the resulting simulation times.

6.2.1 Flyback Converter Scheme (Dymola)

The flyback converter can be used to transform a given input voltage to a different output voltage. It belongs to the group of DC-DC converters.
A very simple electrical circuit with a voltage source connected to the primary winding of the converter and a load to its secondary winding looks as shown in Figure 6.5.

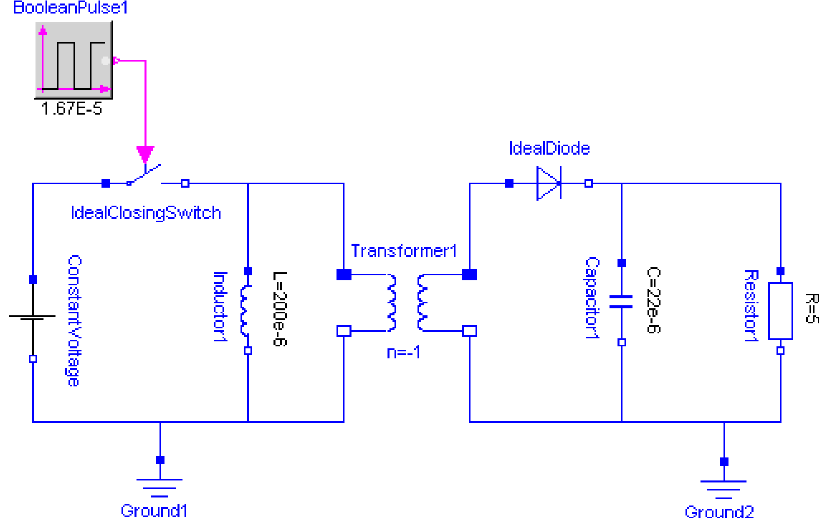


Figure 6.5: The flyback converter scheme modelled in Dymola.

The operation of the flyback converter consists of two phases:

- Phase 1: When the switch is closed, the diode is locked and current flows through the inductor on the left side of the transformer (primary winding), setting up a magnetic field. The consumer load (a resistor in our case) is driven by the energy stored in the capacitor.
- Phase 2: When the switch is opened, the voltage source is separated from the circuit. The magnetic field of the inductance is responsible for creating a current that flows through the diode now, since it cannot flow via the voltage source anymore. The inductor serves as a source of energy, which means that it charges the capacitor and runs the load.

The two phases are repeated in high frequency (controlled by the boolean signal that is connected to the switch), thereby providing the load with an almost constant current.

Figure 6.6 shows the first two milliseconds of a simulation run of the flyback converter circuit given in Figure 6.5. Additionally, it also shows a more detailed view (only 0.19 milliseconds)

in order to illustrate the influence of the switch on the trajectory of the voltage/current at the load: when the switch (green) is closed[4], the resistor is driven by the capacitor, so the voltage (blue) and the current (red) at the resistor decreases slowly (phase 1). When the switch is open, it is the voltage source to power supply the resistor, so the resistor voltage/current increases (phase 2).

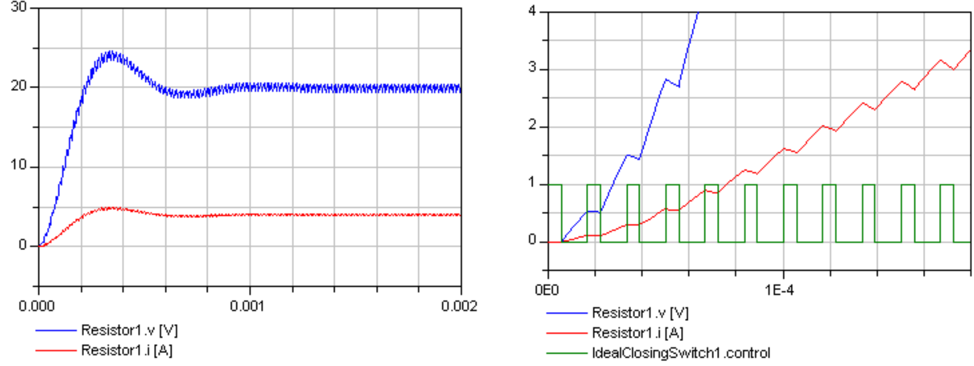


Figure 6.6: The flyback converter output. Once for a period of 2ms (on the left), and in a more detailed view (on the right).

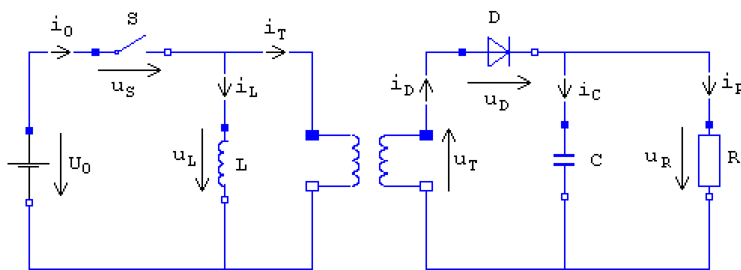


Figure 6.7: The flyback converter scheme with current and voltage indications.

The converter circuit can be described by the following equations (consider Figure 6.7 for a definition of the variables):

[4] Given the fact that it is an Ideal**Closing**Switch, it is closed if the variable *control* equals 1.

$$
\begin{aligned}
U_0 &= constant \\
0 &= \text{if } open_1 \text{ then } i_0 \text{ else } u_S \\
u_L &= L \cdot \frac{di_L}{dt} \\
i_C &= C \cdot \frac{du_R}{dt} \\
u_R &= R \cdot i_R \\
0 &= \text{if } open_2 \text{ then } i_D \text{ else } u_D \\
open_2 &= u_D < 0 \text{ and } i_D \leq 0 \\
u_T &= -u_L \\
i_T &= -i_D \\
i_0 &= i_L + i_T \\
i_D &= i_C + i_R \\
u_0 &= u_S + i_L \\
0 &= u_T + u_D + u_R
\end{aligned}
$$

Note that these equations give an acausal description of the flyback converter, which is not what we will need to set up a flyback model in ModelicaDEVS and PowerDEVS. This issue is explained in more detail in the next section.

6.2.2 Flyback Converter Block Diagram (PowerDEVS/ModelicaDEVS)

In order to be able to model the flyback converter in ModelicaDEVS or PowerDEVS, we need to map the behaviour of the converter to a block diagram, which then can be reproduced by means of the components from the PowerDEVS/ModelicaDEVS libraries. Block diagrams are obtained by causalising the equations of the system (circuit) according to the Tarjan algorithm as explained in [3].
Unfortunately in our case, the presence of the switch coupled with the directed input-output data flow intrinsic to DEVS models cause an unsolvable problem if we just attempt to build our block diagram from the common set of equations given in Section 6.2.1. It is therefore necessary to split the equations in two sets, one for each switch position (for each operation phase of the converter), thereby eliminating the switching component. Figure 6.8 shows the respective electrical circuits for the two switch positions (closed/open).

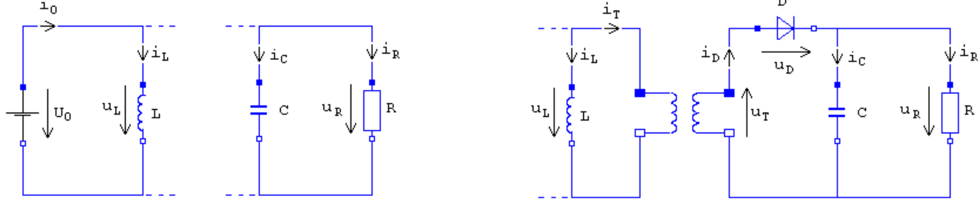


Figure 6.8: The flyback converter schemes for the closed and open switch case.

The following sets of equations describe the two operation phases of the converter:

Switch closed

$$
\begin{aligned}
U_0 &= constant \\
u_L &= U_0 \\
u_L &= L \cdot \frac{di_L}{dt} \\
i_C &= C \cdot \frac{du_R}{dt} \\
u_R &= R \cdot i_R \\
i_R &= -i_C \\
i_0 &= i_L
\end{aligned}
$$

Switch open

$$
\begin{aligned}
i_D &= i_L \\
i_D &= i_C + i_R \\
u_L &= L \cdot \frac{di_L}{dt} \\
i_C &= C \cdot \frac{du_R}{dt} \\
u_R &= R \cdot i_R \\
u_R &= -u_L
\end{aligned}
$$

Now we have to causalise[5] the equations. To this end, we apply the Tarjan algorithm [3]: we mark all variables appearing in the equations with labels "solve for" and "known". By convention [2], known variables are underlined and variables for which a certain equation has to be solved are put into square brackets.
The following rules help a) to find a good starting point for applying the algorithm, and b) to choose the next variable to be marked:

- Outputs of integrators are considered to be known. Hence, if we have an equation like $u_L = L\frac{di_L}{dt}$, all variables i_L occurring in other equations can be underlined (remember that known variables are marked by underlining them).
- An equation that contains only one unknown variable has to be solved for the particular variable that therefore can be put in square brackets.
- A variable that occurs in only one equation has to be obtained from that equation, i.e., the equation has to be solved for the particular variable. Thus, the variable can again be put into square brackets.
- As soon as a variable has been put into square brackets, all its further occurrences in other equations can be labelled "known" since the value of that variable will be obtained from the equation where it is labelled "solve for".

The causalised version of the two equation sets listed before looks as follows (in order to be able to trace the algorithm, the variables are numbered in the order they were underlined or bracketed):

Switch closed

$$
\begin{aligned}
[U_0]^{(1b)} &= \underline{constant}^{(1a)} \\
[u_L]^{(1d)} &= \underline{U_0}^{(1c)} \\
\underline{u_L}^{(1e)} &= L \cdot \left[\frac{di_L}{dt}\right]^{(1f)} \\
\underline{i_C}^{(2e)} &= C \cdot \left[\frac{du_R}{dt}\right]^{(2f)} \\
\underline{u_R}^{(2a)} &= R \cdot [i_R]^{(2b)} \\
\underline{i_R}^{(2c)} &= -[i_C]^{(2d)} \\
[i_0]^{(1h)} &= \underline{i_L}^{(1g)}
\end{aligned}
$$

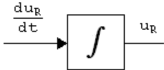
Switch open

$$
\begin{aligned}
[i_D]^{(1b)} &= \underline{i_L}^{(1a)} \\
\underline{i_D}^{(1c)} &= [i_C]^{(2d)} + \underline{i_R}^{(2c)} \\
\underline{u_L}^{(3c)} &= L \cdot \left[\frac{di_L}{dt}\right]^{(3d)} \\
\underline{i_C}^{(2e)} &= C \cdot \left[\frac{du_R}{dt}\right]^{(2f)} \\
\underline{u_R}^{(2a)} &= R \cdot [i_R]^{(2b)} \\
\underline{u_R}^{(3a)} &= -[u_L]^{(3b)}
\end{aligned}
$$

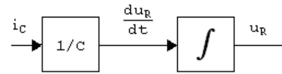
The two sets of causalised equations define exactly the way variables depend on each other, and we are able to build the required block diagram(s). In order to illustrate how to get from

[5] Causalising means to determine which equation we solve for which variable.

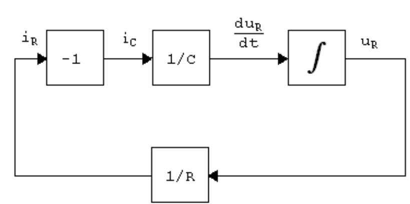
a set of equations to a block diagram, the subsequent table depicts the various steps during the synthesis of the block diagram representing the first set of causalised equations.



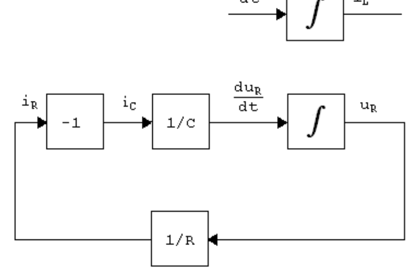
1. Although it theoretically does not matter with which block/equation we start, it is recommended to first insert the integrators.



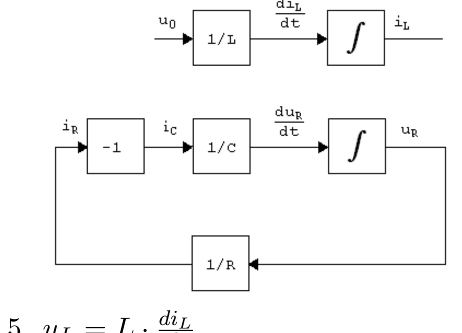
2. Our first equation we want to represent preferably depends on the integrator variables: $i_C = C \cdot \frac{du_R}{dt}$



3. As a next step we include the equations $i_R = -i_C$ and $u_R = R \cdot i_R$



4. There are no dependencies anymore, so we insert an integrator again.



5. $u_L = L \cdot \frac{di_L}{dt}$

6. Finally, we represent the equations $u_L = U_0$ and $i_0 = i_L$, and thereby complete the block diagram of the first equation set.

Figure 6.9 shows the two (finished) block diagrams, where the second one has been built according to the same method that has been applied for the first one.
Note that the two resulting diagrams are not completely different from each other but share common parts, which will be helpful as we shall see soon.
The last step towards a complete single block diagram representing the flyback converter, is to merge the two partial diagrams presented above. To this end, recall that initially, we split the original set of equations to eliminate the switch, or, in other terms, to obtain a diagram for each switch position. Hence, while merging the two diagrams, we have to re-insert the switching component. The switch can be said to alternate between the two diagrams: when it is open, the first block diagram is valid, when it is closed, the second one becomes active.
Fortunately, the two diagrams share a similar structure, which eases the merging process a lot: normally we would have to build both of the models and in some way deactivate one of

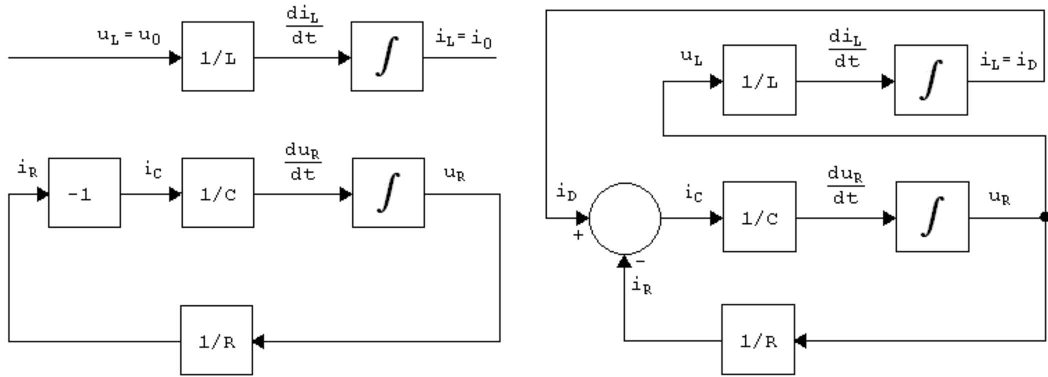


Figure 6.9: The block diagrams for the two equation sets.

them when the other becomes active. In our case however, we just look for a way how to transform one of the diagrams into the other one. We find that the two critical variables are u_L and i_C since their definition in the first diagram is different from their definition in the second one:

- When the switch is closed (first diagram), u_L equals u_0, thus the constant value given by the parameters of the model. The current i_C depends only on i_R instead of a second variable as it is the case for the situation of an open switch.
- When the switch is open (second diagram), u_L is equal to $-u_R$ and i_C is determined by both i_D and i_R.

Hence, if we place our switching component in such a way that it switches a) the value for u_L between u_0 and $-u_R$, and b) the value of i_C between $-i_R$ and $i_D - i_R$, we obtain a model that fully represents the flyback converter.
Such a model is shown in Figure 6.10. Note that there are still two switches, which however correctly represent the single switch in the original flyback converter, since they flip at the same time and thus could be coupled.

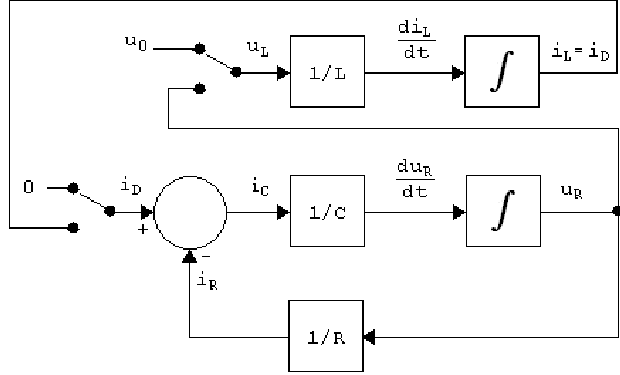


Figure 6.10: The flyback converter final block diagram.

Figure 6.11 shows the model of 6.10 built in ModelicaDEVS.

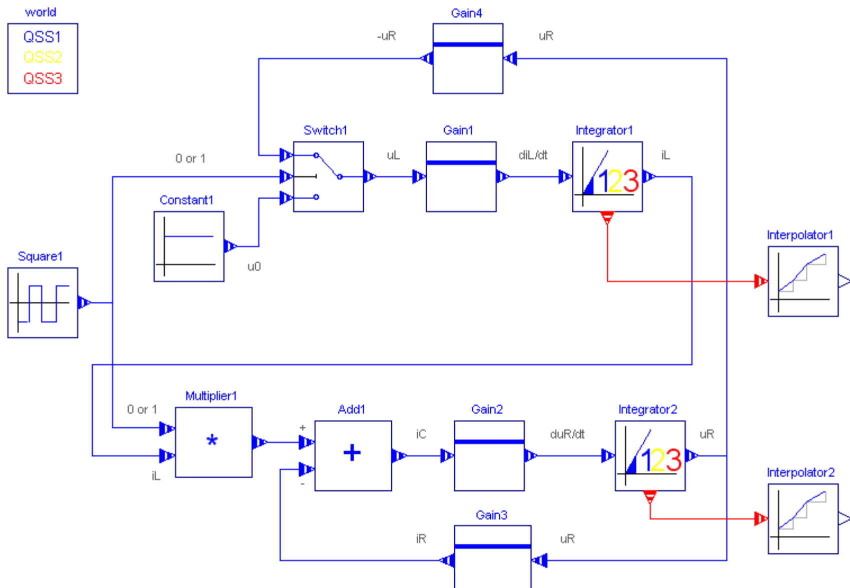


Figure 6.11: The flyback converter modelled in ModelicaDEVS.

Although its structure should be clear to a large extent, the switching parts may need a brief explanation.

- The Square block controls the flipping frequency of the two switches. It generates a signal that oscillates between 1 and 0 (1 = switch open, 0 = switch closed).
- The upper switch in Figure 6.10 is modelled by a Switch block with the parameter *level* set to zero[6]. The Constant block represents the input voltage u_0.
  Given that the switch either directs the first or the third input port to its output port, depending on the output of the Square block, this set-up provides the Gain1 block with either the constant value (u_0) or the signal from the Integrator2 block (u_R).
- The lower switch is represented by a simple Multiplier block the output of which represents the current i_D. Due to the multiplication of the output of Integrator1 (i_L) by the signal from the Square block (0 or 1), i_D oscillates between 0 (phase 1, switch closed) and the value it is provided by Integrator1 (phase 2, switch open).
  This precisely corresponds to the real situation where the current through the diode is zero during phase 1 (diode locked), and non-zero during phase 2.

Figure 6.12 shows the flyback converter model built in PowerDEVS. Except from the part that controls the switches, its structure is equal to the one of the corresponding ModelicaDEVS model. The reason for the switch control to be different than in ModelicaDEVS is

[6]It is important to mention that a *level* value of 1 would **not** yield the same results since the Switch connects the first input port to the output port if the second input port is bigger than the *level* value and the third input port otherwise, namely if the second input port is smaller or equal than the value of *level*. Hence, the switch would not flip at all.

that the Square block in PowerDEVS – contrary to the ModelicaDEVS implementation (see Section 5.2.7) – does not have an *offset* parameter and hence it is not possible to generate a signal that oscillates between 0 and 1 directly, since this would require oscillation around 0.5. The PowerDEVS equivalent of a ModelicaDEVS Square block that oscillates between 0 and 1 is the following construction: a Square block that oscillates between, say, -1 and 1, a Constant block generating a value slightly smaller than the amplitude (0.99 in our case), and a Comparator that produces outputs of 0 if the Square block value is bigger than the constant value, and 1 otherwise.

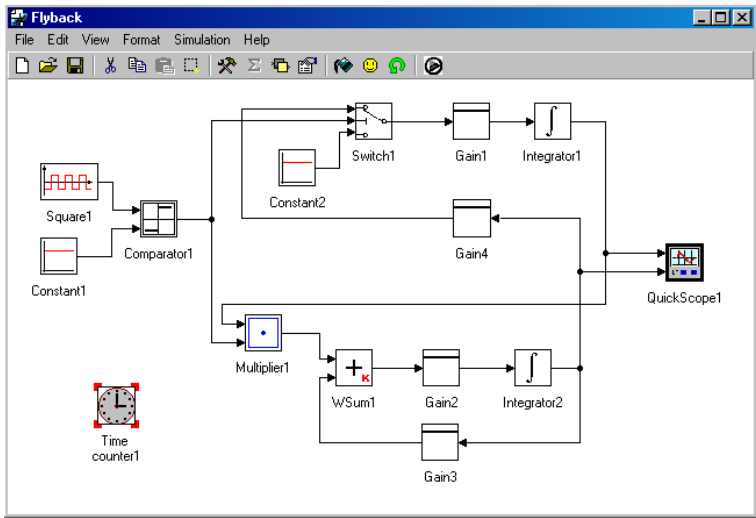


Figure 6.12: The flyback converter modelled in PowerDEVS.

6.2.3 Performance Comparison

6.2.3.1 General Remarks

Before the actual performance comparison between Dymola, PowerDEVS and ModelicaDEVS is done, a number of general remarks about particularities of the different systems regarding time measurement shall be made.

Dymola

In Dymola, the time elapsed during a simulation is indicated in the *Message* window. The speed of a simulation can be influenced by the number of stored variables[7]. For a faster simulation it is thus recommended to limit the number of stored variables to the minimum.

[7]This setting can be made in the *Experiment setup* window, in the *Output* tab.

Using other integration methods instead of LSODAR may speed up the simulation, but the results may not be as accurate anymore (e.g. with Euler), because methods other than LSODAR and DASSL do not fully support event handling. If this problem is tried to be solved by using smaller step size, the results are again accurate, but due to the smaller steps, the computation time increased and the simulation takes just as long (or even longer) as it does using LSODAR.

PowerDEVS

In order to measure the time needed for a simulation, a PowerDEVS model can be equipped with a so called TimeCounter component that prints the elapsed time during the simulation of a certain model to a separate text file. Note that the final time indicated in the TimeCounter block should be **smaller than** the simulation final time (as opposed to equal to).

As mentioned in Chapter 3, PowerDEVS features two ways of presenting the simulation results: the ToDisk block produces ASCII files, while the Qscope block generates a graphical representation of the state variables trajectories. Given the fact that the TimeCounter counts the time from the beginning of the simulation until the specified final time (note: this is not necessarily equal to the final simulation time), and the graphics are generated during the simulation, using a Qscope block slows the simulation down (up to a factor of 10). For performance measurements it is therefore recommended to use ToDisk blocks only.

ModelicaDEVS

A ModelicaDEVS simulation does not use the Modelica `der()` operator, but the ModelicaDEVS Integrator block, and therefore, no integration in terms of Dymola should take place. However, simulation with integration methods other than LSODAR or DASSL (e.g. Euler, Mexx, Rkfix4)[8] yields different results for different integration methods. This is due to the fact that not all integration methods fully support discrete systems or show difficulties processing time events, and although the ModelicaDEVS simulation does not make use of the `der()` operator this causes problems: the time-advance function of Dymola is intrinsicly coupled to the chosen integration method. Hence, it is acutally the time-advance functions of integration methods other than LSODAR or DASSL that do not support time events.

6.2.3.2 Comparison

The scope of this section is to compare Dymola, PowerDEVS and ModelicaDEVS with respect to run-time efficiency. The chosen simulation model is the flyback converter described in the previous sections.

The Dymola and the ModelicaDEVS model were simulated with the LSODAR integration

[8] Many of the currently available solvers will be removed and new ones will be added for Dymola 6, so the situation regarding integration methods will change anyway.

method for two reasons:

- Only DASSL or LSODAR may be used for the simulation of ModelicaDEVS models. Thus, in order to have a fair comparison, it is recommended to choose the same integration method for the standard Dymola simulation.
- LSODAR is roughly two times faster than DASSL. For this reason, LSODAR has been chosen, such that Dymola/ModelicaDEVS have the best possible basis regarding the comparison with PowerDEVS.

The following table gives the average simulation CPU time[9] for a simulation of 0.002 seconds[10] of the flyback converter model in Dymola, ModelicaDEVS and PowerDEVS.

		CPU time [s]	**time events**	**result points**
Dymola		0.062	239	738
	QSS1	3.55	6363	11829
ModelicaDEVS	QSS2	0.688	958	2299
	QSS3	0.656	833	2164
PowerDEVS		0.018	N/A	N/A

The above table shows a clear ordering of the three different systems in terms of performance: PowerDEVS is faster than Dymola, which in turn is faster than ModelicaDEVS.

The fact that PowerDEVS is up to a factor of 4 faster than the corresponding Dymola simulation is probably ascribable to the two different simulators of PowerDEVS and Dymola. Given that Modelica evaluates the equations of a model simultaneously, it has to process all variables at each event instant. PowerDEVS on the other hand just updates the variables of the currently active component.
Thus in this case, the simultaneous equation evaluation is not necessarily an advantage.

The reason for ModelicaDEVS being so much slower than Dymola is due to the fact that ModelicaDEVS creates a lot more time events than the conventional Dymola simulation (833 and 239 (for QSS3), respectively). This however cannot be avoided since these events are created due to the definition of the DEVS formalism[11]. However, the problem is that at each of these time events, the Dymola simulator has to save all variables, which of course is time consuming. The fact that the ModelicaDEVS simulation contains a lot more (factor 3) variables than the corresponding standard Dymola simulation even worsens the situation, because this means that ModelicaDEVS simulations feature more variables that have to be stored at more time events than it is the case in a standard Dymola simulation.
The same issue (number of time events) seems also to be responsible for the significant difference in terms of run-time efficiency between the QSS1 method and the higher-order ap-

[9]Tested on a IntelCeleron 2.6 GHz Laptop with 256MB RAM. Note that the CPU time results vary for different computer systems – nevertheless, the relative ordering remains the same.

[10]In order to make sure that there is no initialisation overhead, the same model has also been simulated for 0.5s and 1s. From the obtained results could be deduced that this is not the case for either of the three simulation systems (PowerDEVS, ModelicaDEVS and standard Dymola).

[11]A decrease of time events could only be achieved by increasing the *quantum* parameter of Integrator blocks, which unfortunately may lead to inaccurate results and thus is not recommended.

proximation methods QSS2 and QSS3. Figure 6.13 illustrates the difference in the output of a QSS1 Integrator versus a QSS3 Integrator: during a period where the higher-order Integrator (QSS3) only creates one event, the first-order Integrator (QSS1) generates roughly ten of them.

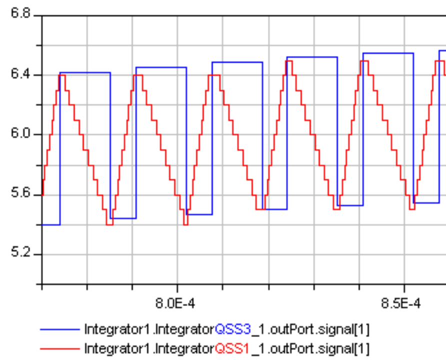


Figure 6.13: Simulation in QSS3 mode vs. simulation in QSS1 mode.

Chapter 7

Conclusion

The result of this master's thesis project is a new DEVS library "living" within the environment of Dymola, featuring the same components as PowerDEVS, such that it can be used to simulate continuous systems.

The following sections discuss the degree of realisation of the goals described in Section 1.1 and Appendix A.

7.1 Summary

7.1.1 Modelling

Due to the implemented set of DEVS components, it is possible to model any system that is describable by algebraic and/or differential equations.
The only drawback is that DEVS blocks can only model systems that have been transformed into a block diagram. Unfortunately, it is not always straightforward to obtain the block diagram representation of a system. However, this is a problem related to the DEVS formalism, given the fact that DEVS models impose a certain input-output direction on the data flow of the simulation, while the equation described models of Dymola/Modelica may be non-directed, acausal models.

7.1.2 Mixed Systems

Thanks to the four transformation blocks (SamplerLevel, SamplerTime, SamplerTrigger and the Interpolator) it is possible to embed ModelicaDEVS models into Dymola models or vice-versa.
Unfortunately, this may cause several difficulties as will be elaborated in more detail in Section 7.1.4.

7.1.3 Performance

During the performance comparison (Section 6.2.3.2) between ModelicaDEVS, PowerDEVS and Dymola, it has been found that, indeed, the DEVS formalism is very well suited for

simulating systems with frequent switching operations: simulations with PowerDEVS gave very good results compared to pure Dymola simulations (in terms of performance).
ModelicaDEVS on the other hand seems to suffer from the variable bookkeeping at time events which makes the simulation become a factor of ten slower than conventional Dymola, although the ModelicaDEVS code does not generate any (or very few) time consuming state events.

So, the initial idea of being able to save time because of Modelica's parallel variable update instead of the message passing between coordinators and simulators in PowerDEVS, has failed: ModelicaDEVS is about 40 times slower than PowerDEVS, and 10 times as slow as standard Dymola.

7.1.4 Substitution of Dymola components

Another goal was to allow for the substitution of Dymola models by ModelicaDEVS blocks, exploiting the object-oriented properties of Dymola/Modelica. While in theory, the ModelicaDEVS library provides interface blocks from ModelicaDEVS to standard Dymola and vice-versa, and the substitution itself is not a problem at all, it is still not that simple to substitute a non-directed data flow model by an input-output-directed block: the block has to assume a certain data flow direction which is actually not given by the original model. For example, the ModelicaDEVS capacitor (see Section 6.1.2), which was meant to replace the capacitor from the standard electrical library, assumes it is given the current i and it computes the voltage v. The original capacitor however does not have such a restriction but computes either i or v, depending on which of the two it is given. The ModelicaDEVS capacitor can only be used if such an assumption holds, i.e., if it is for example connected to a current generator (as opposed to a voltage source).

Thus – beside the fact that pure Dymola seems to be more efficient anyway (see Section 6.2.3.2 – since the replacement of Dymola components by ModelicaDEVS models is only possible under certain limitations, it is therefore not recomendable to use this possibility, unless it is for “theoretical” purposes like investigating the behaviour of the DEVS formalism under certain circumstances that require the mixing of DEVS and standard Dymola components.

7.2 Outlook and Open Problems

7.2.1 Extension of ModelicaDEVS

The main scope of this thesis was to implement a PowerDEVS-like Modelica library in order to allow the simulation of continuous systems by the DEVS formalism within the Dymola environment. But of course, the DEVS formalism allows also for purely discrete models such as client-server or monitoring systems. It would therefore be possible to extend the current ModelicaDEVS library with additional blocks such as different types of servers, client generators, random number generators, etc.
This would give rise to models containing both common[1] DEVS and continuous Dymola

[1] e.g. the aforementioned client-server system that is not related to integration.

parts, by-passing however the need of converting the continuous part into a block diagram representation, as it would be the case in other DEVS simulation software systems that cannot simulate equations discribing continuous systems directly.

7.2.2 Index Reduction Problem

As described in Section 6.1.2, there is still an unsolved problem regarding the differentiation of ModelicaDEVS signals. Since Dymola cannot know how to differentiate the values of a ModelicaDEVS event, it has to be given the derivatives explicitly.
According to the Dymola Handbook [4], this should be possible providing "user defined derivatives". The main idea of this feature is to allow the user to explicitly declare the derivative functions of any user defined function.
The instructions in the Dymola tutorial have been strictly complied with, but it did not solve the problem for ModelicaDEVS. Although it is now possible to perform a mixed/hybrid simulation using a numerical differentiation scheme, it would be preferable to have an analytical rahter than a numerical solution to this problem. Due to lack of time however, this problem is left for a potential future improvement of the ModelicaDEVS library.

Bibliography

[1] Cassandras, C.G. and S. Lafortune (1999), Introduction to discrete event systems, Kluwer Academic Publishers, Boston.

[2] Cellier, F.E. (1991), Continuous System Modeling, Springer-Verlag, New York.

[3] Cellier, F.E. and E. Kofman (2005), Continuous System Simulation, Springer-Verlag, New York.

[4] Dymola User's Manual. Is distributed with the Dymola software, available in the folder "Documentation".

[5] Fritzson, P. (2004), Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, Wiley Interscience.

[6] Gill, A. (1962), Introduction to the Theory of Finite-state Machines, McGraw-Hill.

[7] Glaser, J.S., F.E. Cellier, and A.F. Witulski (1995), Object-Oriented Switching Power Converter Modeling Using Dymola With Event-Handling, Proc. OOS'95, SCS Object-Oriented Simulation Conference, Las Vegas, NV, pp.141-146.

[8] Harel, D. (1987), Statecharts: A visual Formalism for complex systems, The Science of Computer Programming, 8, pp.231-274.

[9] Kelton, D.W. and A.M. Law (1991), Simulation Modeling and Analysis, McGraw-Hill, New York.

[10] Kim, T.G. (1994), DEVSim++ User's Manual. C++ Based Simulation with Hierarchical Modular DEVS Models, Korea Advance Institute of Science and Technology.

[11] Kofman, E. and S. Junco (2001), Quantised State Systems. A DEVS Approach for Continuous Systems Simulation, Transactions of SCS. 18(3). pp.123-132.

[12] Kofman, E., A Second Order Approximation for DEVS Simulation of Continuous Systems, Simulation (Journal of The Society for Computer Simulation International), 78(2), pp 76-89.

[13] Kofman, E., M. Lapadula, and E. Pagliero (2003), PowerDEVS: A DEVS-based Environment for Hybrid System Modeling and Simulation, Technical Report LSD0306, LSD, Universidad Nacional de Rosario, Argentinien. Submitted to Simulation.

[14] Kofman, E., A Third Order Discrete Event Method for Continuous System Simulation. Part II: Applications. Technical Report LSD0502, LSD, Universidad Nacional de Rosario. Accepted in RPIC'05.

[15] Modelica Tutorial (by the Modelica Association). Is distributed with the Dymola software, available in the folder "Modelica/Documentation" or available at "http://www.modelica.org/documents/ModelicaTutorial14.pdf".

[16] Norris, J.R. (1997), Markov Chains, Cambridge University Press.

[17] Petri, C.A. (1962), Communication with Automata[2], PhD Thesis, Darmstadt Inst. of Technology, Bonn, Germany.

[18] Zeigler, B.P. (1976), Theory of Modeling and Simulation, John Wiley and Sons, New York.

[19] Zeigler, B.P. (1984), Multifacetted Modelling and Discrete Event Simulation, Academic Press, London.

[20] Zeigler, B.P., J.S. Lee (1998), Theory of quantized systems: formal basis for DEVS/HLA distributed simulation environment, SPIE Proceedings, Vol. 3369, pp. 49-58.

[21] Zeigler, B.P., H. Sarjoughian (2000), Introduction do DEVS Modelling and Simulation with JAVA: A Simplified Approach to HLA-Compliant Distributed Simulations. Arizona Center for Integrative Modelling and Simulation http://www.acims.arizona.edu/SOFTWARE/software.shtml#DEVSJAVA.

[22] Ziemer, R.E., W.H. Tranter, D.R. Fannin (1998), Signals and Systems: Continuous and Discrete, Prentice Hall, New Jersey.

[2] Original german title: Kommunikation mit Automaten.

Appendix A

Official Task Description

DEVS is a modeling formalism for the mathematical description of discrete event-oriented systems, based on a re-interpretation of classical systems theory [1]. A system (model) has inputs and outputs; however, the DEVS system does not receive time-continuous signals through its inputs, but rather discrete event descriptions. Similarly, it also sends event descriptions to other systems (models) through its outputs.

The DEVS formalism is object-oriented, and lends itself to the description of complex systems by means of hierarchically structured models [2].

Although it has been evident for many years that the DEVS formalism may be employed in the simulation of continuous systems as well, it was demonstrated only quite recently that this approach indeed offers some advantages over traditional approaches that make use of classical numerical ODE solvers [3].

When a continuous system is being simulated using a digital computer, the problem must in any case be discretized, as digital computers are incapable of processing continuous signals directly. To this end, conventional approaches discretize the time axis, whereas they leave the state variables themselves continuous (i.e., real-valued). In contrast, the DEVS approach discretizes the state variables, whereas it leaves the time axis continuous.

An implementation of the DEVS formalism, that is particularly well suited for this type of applications, is PowerDEVS, coded in C++, software developed particularly for the simulation of continuous systems by means of the DEVS formalism [4].

A disadvantage of PowerDEVS may be, that all continuous signals must be discretized using the same (DEVS) approach. However, it may be of interest to discretize some subsystems using conventional approaches, whereas others are being discretized using the DEVS formal-

ism.

To this end, it may be interesting to consider a re-implementation of PowerDEVS in a modeling and simulation environment that enables the software developer to combine various modeling approaches with each other flexibly and conveniently.

Dymola is a graphical object-oriented modeling environment, designed for modeling and simulation of physical systems [5]. The user interface of Dymola is highly flexible and enables the software developer to implement alternative modeling approaches elegantly and easily. For this reason, Dymola is very well suited for the proposed re-implementation of PowerDEVS.

The proposed research attempts a re-implementation of PowerDEVS [4] in Dymola [5].

Start with implementing the currently available PowerDEVS software on your computer, and simulate some applications that you can find in the references [3,4].

The PowerDEVS software consists of two completely separate parts. One part implements the DEVS formalism, whereas the other part supports the graphical user interface (GUI). In the re-implementation, only the former part must be considered, since Dymola already offers a highly flexible graphical user interface.

Re-implement the PowerDEVS interpretation of the DEVS formalism in the Dymola environment, using the alphanumerical programming language Modelica, that is built into Dymola [6].

It should be recognized that the simulation of continuous systems by means of the DEVS formalism can be re-interpreted as a traditional simulation using another integration algorithm. For this reason, it is justified to request that this new approach be integrated with the Dymola environment as seemlessly as possible. To this end, it may be useful to define an alternate der() operator – you could use for example the diff() operator, such that all you need to do when switching from the simulation of a model by means of any of the built-in integrators to a DEVS simulation, would be to replace the der() operator by the diff() operator. In the future, DEVS simulation could be introduced in a completely transparent way as an additional integration algorithm. However, this requires access to Dymola at a level that the standard user doesn't have access to. For this reason, the adaptation of Dymola to incorporate DEVS integration internally will have to be done by Dynasim at a later time. The quantisation can be hidden behind the diff() operator, as we already know that every DEVS integrator must be quantised.

Simulate the same examples, that you had simulated earlier using the PowerDEVS software, once again, now using the new Dymola implementation, and verify that both software tools indeed generate identical resultats.

Program a higher-order model, requiring more than one integrator, and discretize a subset of the integrators by means of DEVS, whereas the remaining integrators are being discretized using conventional methods (e.g., DASSL), and verify that the mixed simulation works adequately.

References

1. *Zeigler, B.P. (1976), Theory of Modelling and Simulation, Wiley, New York.*
2. *Concepción, A.I. and Zeigler, B.P. (1988), "DEVS Formalism: A Framework for Hierarchical Model Development," IEEE Trans Software Engineering, 14(2), pp. 228-241.*
3. *Cellier, F.E. and E. Kofman (2005), Continuous System Simulation, Springer-Verlag, New York.*
4. *Kofman, E., M. Lapadula, and E. Pagliero (2003), PowerDEVS: A DEVS-based Environment for Hybrid System Modeling and Simulation, Technical Report LSD0306, LSD, Universidad Nacional de Rosario, Argentina. Submitted to Simulation.*
5. *Brück, D., H. Elmqvist, H. Olsson, S.E. Mattsson (2002), Dymola for Multi-Engineering Modeling and Simulation, Proc. 2nd International Modelica Conference, Oberpfaffenhofen, Germany, pp. 55:1-55:8.*
6. *Tiller, M.M. (2001), Introduction to Physical Modeling with Modelica, Kluwer Academic Publishers, Boston.*