MASTERTHESIS

An Approach to Develop a User Friendly Way of Implementing DEV&DESS Models in PowerDEVS

Carried out at the Institut for

Analysis and Scientific Computing

Vienna University of Technology
supervised by

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Felix Breitenecker,

Dipl.-Ing. Irene Hafner,
and Dipl.-Ing. Matthias Rer
by

Franz Josef Preyser

Rudmanns 59
A-3910 Zwettl

Vienna, July 31, 2015

Acknowledgement

I would like to express my gratitude to Prof. Dr. Felix Breitenecker and Dipl.-Ing. Nikolas Popper for offering the possibility for me to write this diploma thesis.

Further, many, many thanks to my supervisors Dipl.-Ing. Irene Hafner and Dipl.-Ing. Matthias Rer as well as to my colleague Dipl.-Ing. Carina Pll who tirelessly corrected my thesis over and over again until really very short before the deadline.

The same gratitude belongs to my girlfriend Theresa Rauch, BSc. who always supported me with more than hundred percent, especially in the final phase.

Another name that should be mentioned here is Arthur Schmidt from the University of Applied Sciences Technology, Business and Design Wismar who gave the right hint for this topic.

Moreover my gratitude goes to Bernhard Heinzl, Peter Smolek, Ines Leobner an the rest of the BaMa team as well as to Johannes Weber for hours of inspiring discussions and conversations.

Finally, I want to thank my parents for supporting me financially, particularly during the last phase of my studies as well as my uncles Johann and Josef Kirchner.

Kurzfassung

Im Bereich der Simulation von Produktionsprozessen wird neben den blichen logistischen Aspekten das Thema Energieverbrauch immer wichtiger. Dabei entstehen sogenannte *hybride* Modelle. Das sind Modelle, die sich weder rein diskret noch rein kontinuierlich beschreiben lassen. Die genaue Spezifikation hybrider Modelle sowie eine solide Handhabung in der Simulation stellen eine ausgesprochen schwierige Aufgabe dar. Gleichzeitigkeit von Ereignissen, Zustandsereignisse sowie Rckkopplungen in gekoppelten Modellen sind einige der wichtigsten Ursachen dafr.

Ziel dieser Diplomarbeit ist es, zuerst formale Beschreibungsmethoden fr hybride Modelle vorzustellen um danach die Implementierungsmglichkeiten fr derartige Modelle in einem konkreten Simulator zu untersuchen. Als Beschreibungsformalismus wird der auf Bernard Zeiglers Formalismen DEVS und DESS fr ereignisdiskrete bzw. kontinuierliche Modelle aufbauende DEV&DESS - Formalismus von Herbert Praehofer gewt. Eine Beschreibung dieser Formalismen ist in Kapitel 2 zu finden, wo aurdem QSS (Quantised State System) erkl wird. QSS ist eine Sammlung von Verfahren zur ereignisdiskreten Beschreibung und numerischen Integration von kontinuierlichen Signalen und ist daher im Zusammenhang mit hybrider Modellierung von besonderem Interesse, vor allem da er dazu verwendet werden kann, kontinuierliche Modelle in ereignisdiskrete Modelle umzuwandeln.

PowerDEVS ist ein Simulator, der sowohl den DEVS - Formalismus als auch QSS untersttzt und somit sehr vielversprechend im Hinblick auf die Implementierungsmglichkeiten von hybriden Modellen ist. Da *PowerDEVS* in weiterer Folge verwendet wird, wird in Kapitel 3 eine Einfhrung in dieses open-source Simulationstool gegeben.

Im 4. Kapitel wird auf diverse Problematiken eingegangen, die im Zuge der DEVS - Modellierung, speziell bei gekoppelten Systemen, auftreten. Die Ursachen dieser Probleme liegen in der korrekten Auflsung von gleichzeitigen Ereignissen in Kombination mit Rckkopplungen. Der DEVS-Formalismus berlt diese Auflsung dem Modellierer, erzwingt jedoch keine explizite Formulierung des Modellverhaltens im Falle von gleichzeitigen Ereignissen. An diesem Schwachpunkt von DEVS setzt die Erweiterung Parallel DEVS (P-DEVS) an. Zur Simulation eines P-DEVS Modells wird allerdings ein eigener P-DEVS Simulator bentigt. PowerDEVS arbeitet jedoch mit dem urspranglichen DEVS Formalismus. Um trotzdem die angesprochenen Probleme in den Griff zu bekommen, werden diese in Kapitel 4 schrittweise identifiziert und es werden Lsungsanse entwickelt. Am Ende entsteht ein neuer generischer PowerDEVS Bibliotheksbaustein namens *Atomic PDEVS block*, bei dessen Verwendung das Verhalten eines Modellblocks gemem P-DEVS-Formalismus definiert werden kann.

Unter Verwendung von QSS fr die kontinuierlichen Anteile (Differentialgleichungsmodelle) werden diese in PowerDEVS ebenfalls in ereignisdiskrete DEVS Modelle umgewandelt. Somit knnen sowohl kontinuierlicher Anteil als auch diskreter Anteil eines hybriden Modells in PowerDEVS implementiert und simuliert werden. Jedoch existiert noch kein Bibliotheksblock mittels dem eine direkte Umsetzung eines DEV&DESS Modells in PowerDEVS mglich ist. Die Entwicklung eines solchen Bausteins namens *Atomic DEV&DESS block* ist Thema des 5. Kapitels, das mit der Implementierung eines konkreten DEV&DESS Modells unter Verwendung dieses Bausteins abschlie.

Abstract

In the area of production process simulation, beside the common logistical optimisation goals, the factor energy consumption is gaining more and more importance. Since energy consumptive processes are usually described continuously, whereas logistic models are purely discrete, considering both leads to *hybrid* models. A model is called hybrid when its behaviour neither can be described purely continuously nor purely discretely or at least, not without a lot of additional effort. The exact and well defined formulation of the behaviour of hybrid models as well as their solid handling in simulation are very challenging tasks. Concurrent events, state events, and feedback loops in coupled models are the major causes for problems.

The goal of this diploma thesis is to introduce formalisms for describing hybrid models and to investigate the possibilities for implementing such models in a specific simulator.

As description formalism DEV&DESS, introduced by Herbert Praehofer, has been selected. DEV&DESS is based upon the two formalisms DEVS and DESS introduced by Bernard Zeigler. Using DEVS discrete event models can be described, whereas DESS is designated for the description of continuous models.

The description of these formalisms represents the first part of Chapter 2. In its second part QSS (Quantized State System) is introduced. QSS denotes a set of methods for describing and numerically integrating continuous signals in a discrete event manner. Thus, it is capable of transforming continuous models into DEVS models which can be utilized for the simulation of hybrid systems.

The simulator PowerDEVS supports both, the DEVS formalism, and QSS. Therefore, it seems to be quite promising for implementing and simulating hybrid models. Since PowerDEVS will be used in the following chapters, an introduction is given in Chapter 3.

When creating and simulating coupled DEVS models, it turns out to be quite difficult to formulate a DEVS description leading to exactly the intended behaviour, particularly in situations with concurrent events. This is caused by the way concurrent events are resolved in coupled DEVS. Parallel DEVS (P-DEVS) denotes an extension of DEVS which exactly addresses this drawback. However, PowerDEVS does not support the implementation of P-DEVS models.

In Chapter 4 the particular problems that occur with coupled DEVS models in combination with concurrent events and feedback loops are systematically identified and a solution approach for each of them is developed. This process concludes with the definition of the generic PowerDEVS library block *Atomic PDEVS block*. It supports model description in P-DEVS manner and is supposed to solve the problems outlined.

In PowerDEVS continuous models can be created graphically as block diagrams comparable to Simulink or Dymola. However, there exists no library block which allows to implement a DEV&DESS model directly. So the topic of Chapter 5 is the development of such a generic library block named *Atomic DEV&DESS block* which is based on the Atomic PDEVS block introduced in Chapter 4. Finally it is demonstrated how to implement a specific DEV&DESS model in PowerDEVS using the Atomic DEV&DESS block.

Contents

CHAPTER 1

Motivation and Introduction

Due to recent political and societal developments as well as rising energy-prices, it becomes more and more interesting for industrial companies to optimize their production processes not only in terms of logistical issues(including time, space and production goods) but also taking energy consumption aspects into account. Additionally, low energy consumption accompanied with low CO_2 emissions can be used for advertising of the produced goods in a beneficial way. Motivated by the reasons mentioned above *BaMa project* came into being (see section 1.1) and with it this diploma thesis.

Usually modelling and simulation of logistical parts of production processes is done in a purely discrete way, whereas most energy consumptive processes like all thermodynamical activities are of continuous nature. Therefore, in the BaMa project a model description is needed capable of describing both the discrete and the continuous parts. Such models consisting of discrete as well as continuous parts are called *hybrid*.

That is where the *DEV&DESS* (Discrete Event & Differential Equation System Specification) modelling formalism comes into play(see section 2.1). As it will be seen in section 2.1 the DEV&DESS modelling formalism does not only fulfil the requirement of being able to describe hybrid systems but is also fully hierarchical which is needed for the cube approach in BaMa as will be explained in section 1.1.

However, a modelling formalism without any way to implement and simulate models formulated with it would not be very useful for BaMa. Most simulators for logistical problems though, are neither capable of simulating continuous model parts nor do they support the DEVS formalism. On the other hand, a lot of simulators originating in the continuous area of simulation in fact do support the discrete model parts as well, but still the handling of events there is quite unsatisfying. This is due to the very basic mechanism of solving ODEs (Ordinary Differential Equations) that actually forms the simulation engine in these tools and that leads to a quite expensive iterative search for each *state event* appearing in a simulation run. A more detail discussion about that follows in section 1.2.

QSS (Quantized State Systems – [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]) is the name of set of methods for discretising and numerically integrating continuous signals in a discrete event manner and thus,

is supposed to overcome the problems of common continuous simulation engines (see section 2.2).

As shown in [**?**], it is possible to implement a DEV&DESS formulated model in SIMULINK using the SimEvents- and Stateflow-toolboxes without having those troubles with state events. However, there is the need to program an ODE solver on your own for each single atomic DEV&DESS model. Further there exists a Modelica DEVS library called *ModelicaDEVS* that makes it possible to implement and simulate causal hybrid models in *Dymola* and that even works with QSS. Yet, in [**?**] it is stated that event handling is kind of slow there compared to *PowerDEVS* [**?**]. The difference lies in the simulation engine. PowerDEVS uses the DEVS simulation engine proposed by Zeigler in [**?**].

So in this diploma thesis a possibility of how to implement and simulate a system model formulated in DEV&DESS in PowerDEVS is demonstrated. PowerDEVS will be described in detail in chapter 3.

1.1 The BaMa Project

BaMa stands for 'Balanced Manufacturing' and is the name of a research-project funded by the FFG (Austrian Research Promotion Agency). Several institutes of the Technical University of Vienna and dwh GmbH – Simulation Services as well as some industrial partners are working on the project right now.

The goal of this project is the development of a methodology on how to model and simulate production processes with a focus on energy-consumption and the CO_2 emission connected to it. The idea is to segment a whole factory into hierarchically structured parts, called *cubes*, i.e. one cube can contain several other cubes but must not overlap with other cubes. Each cube has a well defined interface to its neighbour cubes including energy in- and outputs as well as in- and outputs for *entities*. Entities are all kinds of production goods which finally end up in the end-product.

Since a cube has energy inputs and outputs, the energy consumption of the cube can be calculated by subtracting the output-energy from the input-energy. However, the basic principle of BaMa cubes is that no energy is lost(there has to be an energy balance). Therefore, the energy consumed by the cube is assigned to all entities which are currently located inside that cube. Through this mechanism, if an entity leaves the factory as end-product, it carries a value with it, representing the overall amount of energy that was necessary to produce it. Exactly the same principle can be applied to optimization-factors like production-costs, time, and CO_2-consumption. Figure 1.1 shows an illustration of such a BaMa cube and its interfaces.

Actually the goals of BaMa go even further than just simulating production processes in a factory. In addition, a methodology on how to optimize those processes in terms of costs as well as energy-, time- and CO_2-consumption with respect to given boundary conditions, while using knowledge gained from the simulation, is planned to be developed.

Finally, the third goal is to make it possible to load actual measurements into the simulation-model to parametrize it accordingly to the current state of the factory. So the overall goal is to end up with the definition of a *BaMa-Toolchain* consisting of three modules: *Monitoring, Prediction, Optimization*. For further details on BaMa-project see [**?**].

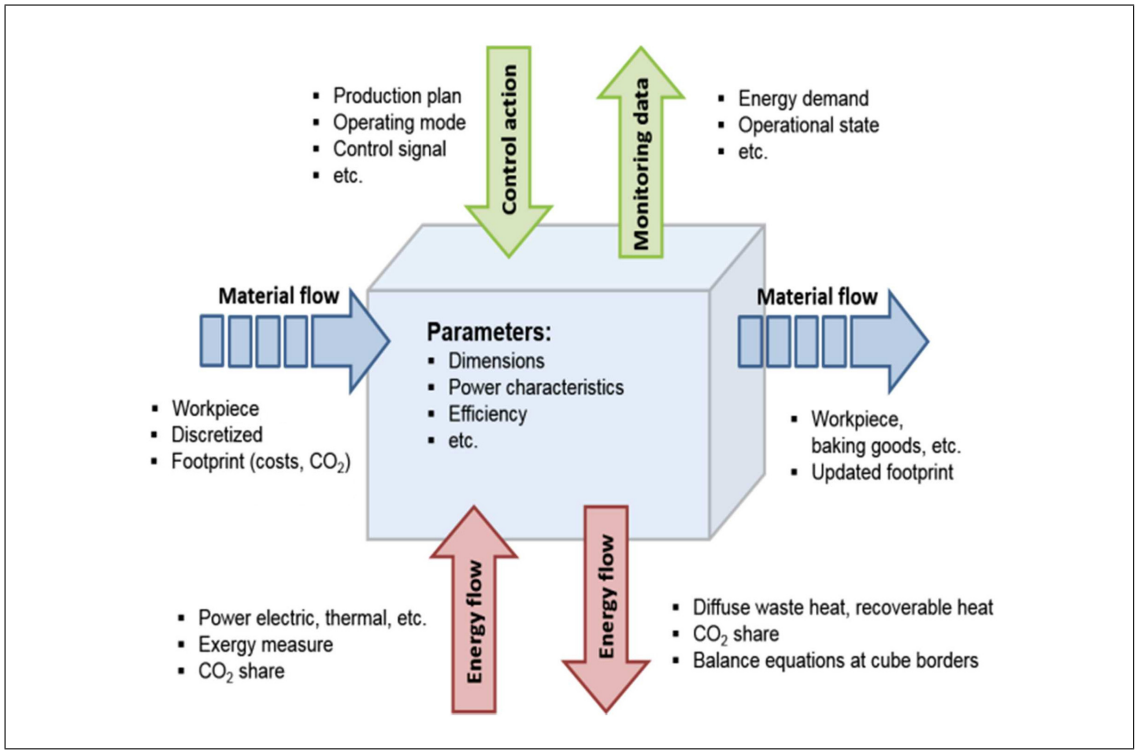


Figure 1.1: Illustration of a BaMa–cube

1.2 Shortcomings with Established Simulators

The common working principle of simulators for continuous systems is to deduce an ODE from the graphically or textually entered model and to numerically integrate it using an *ODE solver*. There exists a variety of different numerical algorithms for ODE solver like explicit/implicit Euler, Heun, Runge-Kutta ...(see [**?**], [**?**]). So the mathematical model the simulation engine works with is an initial value problem(IVP) of the form:

$$\begin{cases} \dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}, t) \\ \mathbf{x}(t_0) = \mathbf{x}_0 \end{cases} \tag{1.1}$$

where t denotes time and $\mathbf{x}$ denotes the potentially vectorial system state. Generally bold letters will be used for vectorial variables.

Applying any numerical integration method $\mathbf{\Psi}$ on that IVP results in a series of approximative solution samples $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_, \ldots \mathbf{x}_N$ which obey the following recursion:

$$\mathbf{x}_{n+1} = \mathbf{\Psi}(\mathbf{x}_{n+1}, \mathbf{x}_n, \ldots, \mathbf{x}_{n-k-1}, t_n, t_{n-1}, \ldots, t_{n-k-1}, \mathbf{f}, h_n) \tag{1.2}$$

where $k \in \mathbb{N}$ tells us that the integration-method in use is a k-step-method. Whether Ψ really makes use of $\mathbf{x}_{n+1}$ or not depends on Ψ being an implicit or an explicit integration method. The values $\mathbf{x}_n$ are numerical approximations for $\mathbf{x}(t_n)$, where $h_n = t_{n+1} - t_n$ can either be constant

(fix step solver) or vary with n depending on the size of the local integration error(variable step solver).

Now, if during a simulation you need to trigger an event at a certain point in time t_e you have to force the ODE solver to calculate a solution sample at $t_n = t_e$. This is actually not much of a problem because you just have to adapt the step-size to end up with a solution-sample at time t_e. Events of this kind are called *time events*. However, if you have to handle *state events*, i.e. an event whose occurrence time t_e depends on the model state $\mathbf{x}$ itself, problems arise.

For example imagine the probably most simple case of a state event where the event-time t_e is defined as point in time where one of the state variables x_i has its next zero crossing ($\mathbf{x} = (x_1, x_2, x_3, \ldots, x_d)$), i.e. $x_i(t_e) = 0$. Since you do not have the analytical solution $x_i(t)$ but only approximative samples $x_i(t_n)$ of it at specific points in time t_n you only know, that the state event has happened, when the new sample $x_i(t_{n+1})$ has another sign than its predecessor $x_i(t_n)$, aside from the unlikely case when the solver accidentally calculates a sample exactly at the event time t_e or at least very close to it.

Now the usual way to encounter that problem is to iteratively search in the interval $[t_n, t_{n+1}]$ for the time t_e of the zero crossing of x_i. Common methods for this iterative search are *Newton, Bisection, Secant-Method or Regula-Falsi*. However, all of these methods need to calculate a solution sample of the whole state $\mathbf{x}$ for each single iteration, even if $\dot{x}_i(t) = f_i(\mathbf{x}, t)$ possibly only depends on x_i, that is $\dot{x}_i(t) = f_i(x_i, t)$, or only on a few entries of $\mathbf{x} = (x_1, x_2, x_3, \ldots, x_d)$. This and the iterative nature of the event location make state events in those simulators for continuous systems unattractive. QSS is promising a way out of this dilemma like we will see in section 2.2.

CHAPTER 2

Theoretical Basis

2.1 Modelling Formalism

Bernard Zeigler proposed in his book 'Theory Of Modeling and Simulation' [**?**] a classification of dynamic system models into three basic types:*Discrete Event* -, *Discrete Time* - and *Differential Equation* - systems(DEV,DTS,DES). The first one means system models which are usually simulated using an event scheduler. The second one denotes system models where changes of state values are happening in equidistant instances of time, like in digital circuits used in all digital computers. The third one covers purely continuous models described with differential equations.

For each type Zeigler introduced a system-specification-formalism called DEV-, DTS- and DES-Specification (DEVS,DTSS and DESS) and he showed that actually DTSS is a subtype of DEVS. That is, for each system model in form of a DTSS exists a DEVS describing exactly the same system-behavior.

In addition, a fourth formalism called *DEV&DESS* was introduced by Praehofer [**?**], with DEV&DESS standing for Discrete Event and Differential Equation System Specification. DEV&DESS is intended to describe *hybrid systems*. In this context, *hybrid system* means a system consisting of both a discrete and a continuous part. However, DEV&DESS describes only a method for formally specifying the behaviour of a hybrid system, but it does not tell how to deduce a simulatable model from such a specification. Since simulation is performed mostly on digital computers which work in completely discrete way, discretisation is necessary for each DESS and for the DESS-part of each DEV&DESS to be able to simulate them. For pure DESS models usually ODE solver algorithms are used to numerically solve the differential equations, i.e. to simulate the DESS model. Therefore, the DESS model in combination with the used ODE solver constitutes a DEVS model, approximating the DESS model. This resulting DEVS model, as each DEVS model, can then be simulated error free on a digital computer, apart from the error due to the finite representation of real numbers.

Very important properties of all four mentioned formalisms are their hierarchical nature and their 'closure under coupling'. That is, an atomic model of each formalism has inputs and out-

puts which can be coupled with inputs and outputs of other atomic blocks or with the inputs and outputs of an overlying non-atomic model which inhabits these atomic models (hierarchically). The resulting overlying model now behaves exactly like an atomic model (closure under coupling) of the particular formalism and therefore again can be coupled with other atomic and non-atomic models.

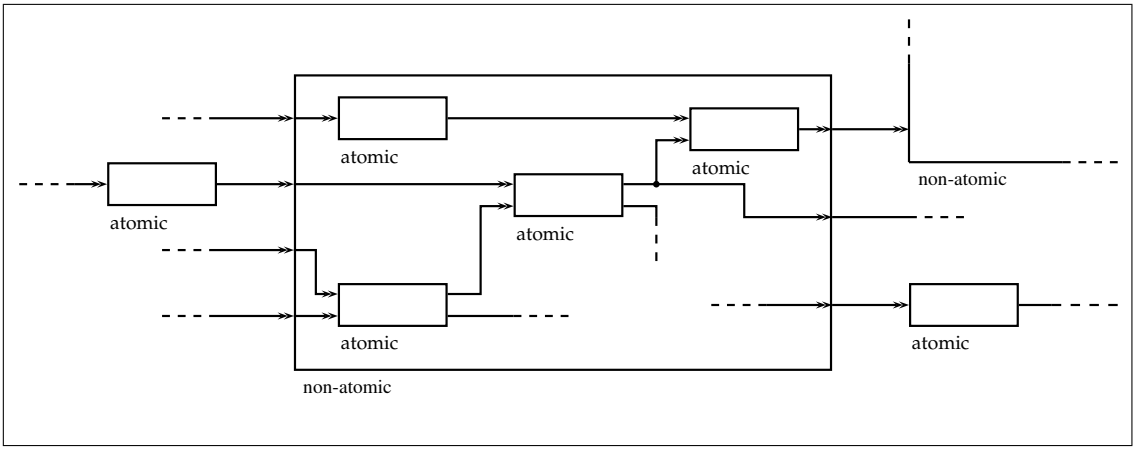


Figure 2.1: Graphical illustration of the hierarchical coupling-property of each of the formalism DEVS, DTSS, DESS and DEV&DESS.

Over years many extensions and specialisations of DEVS have been developed. Ideas from one of them called *Parallel DEVS (P-DEVS)* will be used in section 4.1 to overcome some modelling difficulties arising when simulating coupled DEVS models.

Since ultimately we are interested in DEV&DESS and its simulation, in the following we will shortly describe first the DEVS- and P-DEVS- then the DESS- and finally the DEV&DESS-formalism.

2.1.1 Modelling Formalism for Discrete Systems

2.1.1.1 Atomic DEVS

DEVS stands for Discrete Event System Specification and is the name of a formalism for describing models of systems that allow changes only at discrete points in time. Such changes are called *events*. Since DEVS models can be coupled, we distinguish *atomic* and *coupled* DEVS. An atomic DEVS is specified by the following 7-tuple.

$$DEVS_{atomic} =< X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta >$$

where

$$
\begin{aligned}
X &\ldots \text{set of possible inputs(e.g. } \mathbb{R}^n) \\
Y &\ldots \text{set of possible outputs(e.g. } \mathbb{R}^+ \times \mathbb{N} \times \mathbb{R}^m) \\
S &\ldots \text{set of possible states(=state space)} \\
Q &= \{(s,e) | s \in S, e \in [0, ta(s)]\} \\
\delta_{ext} : Q \times X \to S &\ldots \textit{external state transition function} \\
\delta_{int} : S \to S &\ldots \textit{internal state transition function} \\
\lambda : S \to Y &\ldots \textit{output function} \\
ta : S \to \mathbb{R}_0^+ \cup \infty &\ldots \textit{time advance function} \text{ (“lifespan of a state”)}
\end{aligned}
$$

Figure 2.2 shows a graphical illustration of an atomic DEVS and its working principle.

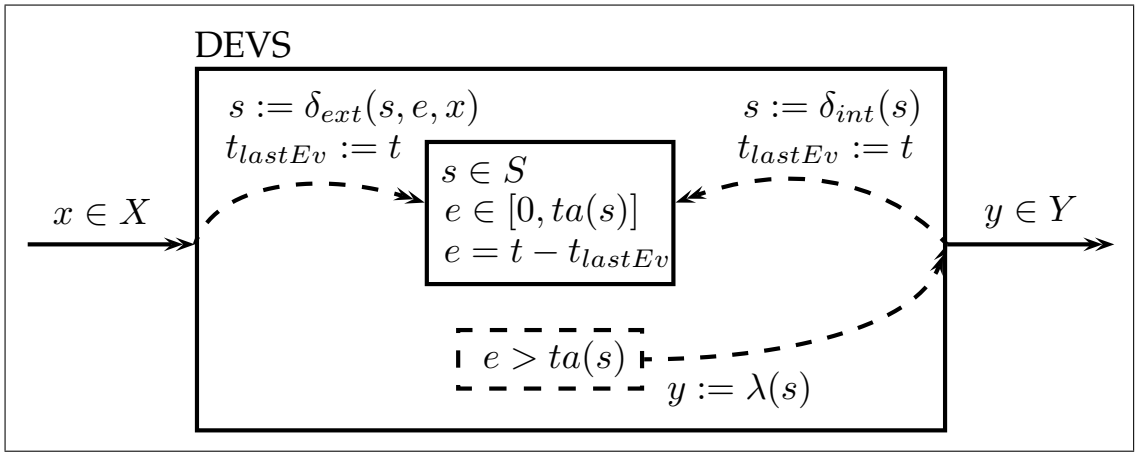


Figure 2.2: Graphical illustration of an atomic DEVS.

As already mentioned, in a system described by a DEVS, only at discrete points in time something can change. Such changes can be caused by two things: either an *input message* arrives on one of the input ports or an internal *time event* occurs.

In the first case, as result of the arrival of a value x at the input, the *external state transition function* δ_{ext} is executed and updates the internal state s with respect to x, the old value of s and the time e since the last event occurrence. Afterwards the value of e is set back to zero.

In the second case, the value of e reaches the upper limit for the age of the current state s which is given with $ta(s)$. As consequence, first the *output function* λ is executed, which may lead to an output y ($y = \emptyset$ is also possible, meaning no output) and then the *internal state transition function* δ_{int} is executed. δ_{int} again updates the internal state on basis of the old state value and resets e to zero.

A DEVS is called *legitimate*, if for each possible set of initial conditions, there is only a finite number of events occurring during a finite amount of time. In this case, a DEVS is well defined.

2.1.1.2 Coupled DEVS

The graphical illustration in Figure 2.2 shows only one input port. Nevertheless, it is possible for an atomic DEVS to have as many input ports as necessary. Since the input values x are elements of a very general set of possible inputs X, for example X could be $\mathbb{R}^n \times \{0, 1, 2\}$, meaning that the corresponding DEVS possesses 3 inputs, with each being able to receive n dimensional real valued vectors. Each input message would then consist of a 2 tuple, where the first value is element of $\mathbb{R}^n$ and the second value gives the port of arrival (0,1 or 2). The same holds for the outputs and the set of possible output values Y.

Now a network of coupled DEVS can be described by the following:

$$N =< X, Y, D, \{M_d\}_{d \in D}, \{I_d\}_{d \in D \cup \{N\}}, \{Z_{i,d}\}_{i,d \in D \cup \{N\}}, Select >$$

$$\begin{aligned}
X &\ldots \text{ set of possible inputs} \\
Y &\ldots \text{ set of possible outputs} \\
D &\ldots \text{ set of involved 'child DEVS' denominators} \\
M_d &\ldots \text{ child DEVS of } N \text{ for each } d \in D \\
I_d \subset D \cup \{N\} &\ldots \text{ influencer set of } d,\ d \notin I_d \\
Z_{i,d} &\ldots \text{ output translation function} \\
Select : 2^{D \cup \{N\}} \rightarrow D \cup \{N\} &\ldots \text{ tie breaking function}
\end{aligned}$$

for each $i \in I_d$:

$$Z_{i,d} : \begin{cases} X \rightarrow X_d & , if\ i = N \\ Y_i \rightarrow Y & , if\ d = N \\ Y_i \rightarrow X_d & , if\ i, d \in D \end{cases}$$

$Select$ is necessary to resolve simultaneous events. Imagine two atomic DEVS blocks with an output to input coupling, having an internal event at the same time. This would result in internal state transitions in both blocks and therefore possibly in an output of the first DEVS block. These three actions actually take place at the same time, but it makes a difference in which order they are executed. For example, let us assume that at first the internal state transition of the first block is executed, producing an output and therefore an external state transition in the second block. This external state transition in the second block changes its state s and thus $ta(s)$, which may lead to the internal state transition of the second block not being executed at all. On the other hand, if the internal state transition of the second block is treated at first, the second block would first update its internal state and may produce an output, which, however, does not affect the first block. Afterwards the internal state transition of the first block is executed, producing an output which triggers an external state transition in the second block.

So in the first case the new state of the second block would be $\delta_{ext}(s, e, x)$ or maybe $\delta_{int}(\delta_{ext}(s, e, x))$, however, in the second case the new state would be $\delta_{ext}(\delta_{int}(s), e, x)$.

DEVS is a very general formalism. As a result it can be shown that a lot of other discrete event formalisms, as for example Event Graphs, Statecharts, Petri Nets, and even Cellular Automata describe subclasses of the set of all systems describable with DEVS. That is why Zeigler proposes the so called DEVS Bus as common interface for multi formalism simulation. For more details see [**?**].

2.1.1.3 Atomic Parallel DEVS

The resolution of concurrent events in a coupled DEVS model is accomplished by the *select* function which defines a total order among all the DEVS blocks connected with each other in a coupled DEVS. According to this order, concurrent events are treated. Therefore, the behaviour of an atomic DEVS model in a coupled DEVS depends on the select function. Parallel DEVS (P-DEVS) is an approach to localize concurrent events handling to the atomic blocks themselves, making the behaviour of atomic P-DEVS independent from their coupling environment. An atomic P-DEVS is described by the following 8-tuple:

$$P - DEVS_{atomic} =< X, Y, S, \delta_{ext}, \delta_{int}, \delta_{conf}, \lambda, ta >$$

where

$$\begin{aligned}
X &\ldots \text{ set of possible inputs(e.g. } \mathbb{R}^n) \\
Y &\ldots \text{ set of possible outputs(e.g. } \mathbb{R}^+ \times \mathbb{N} \times \mathbb{R}^m) \\
S &\ldots \text{ set of possible states(=state space)} \\
Q &= \{(s, e) | s \in S, e \in [0, ta(s)]\} \\
\delta_{ext} : Q \times X^b \to S &\ldots \textit{external state transition function} \\
\delta_{int} : S \to S &\ldots \textit{internal state transition function} \\
\delta_{conf} : S \times X^b \to S &\ldots \textit{confluent transition function} \\
\lambda : S \to Y &\ldots \textit{output function} \\
ta : S \to \mathbb{R}_0^+ \cup \infty &\ldots \textit{time advance function} \text{ (“lifespan of a state”)}
\end{aligned}$$

As it can be seen there are only two differences recognisable compared to an atomic DEVS definition. The first is the domain of δ_{ext} which consists of Q and X^b. X^b is simply the set of all multisets (bags) over X. The reason for this will be explained later. The second is the additional confluent transition function δ_{conf}. It is a different treatment of concurrent internal and external events compared to DEVS that makes this δ_{conf} necessary. While with DEVS those concurrencies are resolved at the coupled DEVS level using the select function, with P-DEVS they are resolved individually for every single atomic P-DEVS using δ_{conf}. I.e., whenever at the same time both an external and an internal event is triggered, an atomic P-DEVS executes $\delta_{con}f$ which decides in each case separately how to handle the collision. For example, δ_{conf} could be defined as $\delta_{conf}(s, x^b) = \delta_{ext}(\delta_{int}(s), x^b, 0)$ which would mean: always treat first the internal event and afterwards the external event. On the other hand, δ_{conf} could also be defined as $\delta_{conf}(s, x^b) = \delta_{int}(\delta_{ext}(s, x^b, ta(s)))$ which would mean: always treat first the external event and afterwards the internal event. Anyway, δ_{conf} can also be defined completely differently and

its definition could also depend on the current state s in such a way, that for some states s first the internal and then the external transition function is executed and for other states the other way round. Or δ_{conf} does not make use of δ_{int} and δ_{ext} at all. Everything is possible.

There is a second difference between DEVS and P-DEVS which concerns execution of coupled models and causes that bag formulation of inputs and outputs. It will be addressed in the next subsection.

2.1.1.4 Coupled Parallel DEVS

The same way atomic DEVS can be coupled, atomic P-DEVS can be coupled, which results in a coupled P-DEVS, described by the following 6-tuple.

$$N =< X, Y, D, \{M_d\}_{d \in D}, \{I_d\}_{d \in D \cup \{N\}}, \{Z_{i,d}\}_{i,d \in D \cup \{N\}} >$$

$$\begin{aligned} X &\ldots \text{ set of possible inputs} \\ Y &\ldots \text{ set of possible outputs} \\ D &\ldots \text{ set of involved 'child P-DEVS' denominators} \\ M_d &\ldots \text{ child P-DEVS of } N \text{ for each } d \in D \\ I_d \subset D \cup \{N\} &\ldots \text{ influencer set of } d,\ d \notin I_d \\ Z_{i,d} &\ldots \text{ output translation function} \end{aligned}$$

for each $i \in I_d$:

$$Z_{i,d} : \begin{cases} X^b \to X_d^b & , if\ i = N \\ Y_i^b \to Y^b & , if\ d = N \\ Y_i^b \to X_d^b & , if\ i, d \in D \end{cases}$$

While in coupled DEVS at one instance of time for each arriving input message at an atomic DEVS the external transition function is applied separately, in coupled P-DEVS all concurrent input messages of an atomic P-DEVS are gathered in a bag and afterwards δ_{ext} or δ_{conf} is applied only once on the whole bag. The attentive reader may ask now, how can these bags of input messages be provided without executing δ_{int} or δ_{conf} of the imminent blocks, since we know from DEVS that λ is only executed right before an internal transition (imminent blocks are atomic DEVS, that are currently scheduled for an internal event). The answer is, that when simulating a coupled P-DEVS model, first of all each atomic block that experiences an internal event at the current simulation time (=imminent block) will execute λ but won't continue with executing δ_{int} or δ_{ext} immediately. Thus all outputs can be gathered before executing the state transitions.

And still there remains a question: what if there are Mealy type blocks among the receiving blocks, i.e. blocks that immediately produce output in reaction of a received input? Output messages produced by such blocks are not available before executing δ_{ext} or δ_{conf}. Actually I didn't find an answer to that in literature. The way the P-DEVS simulator proposed by Zeigler in [**?**] handles this is to simply treat those, due to execution of δ_{ext} or δ_{conf} newly created

imminents separately in a second repetition. Figure 2.3 shows an example where this may lead to an unintended behaviour.

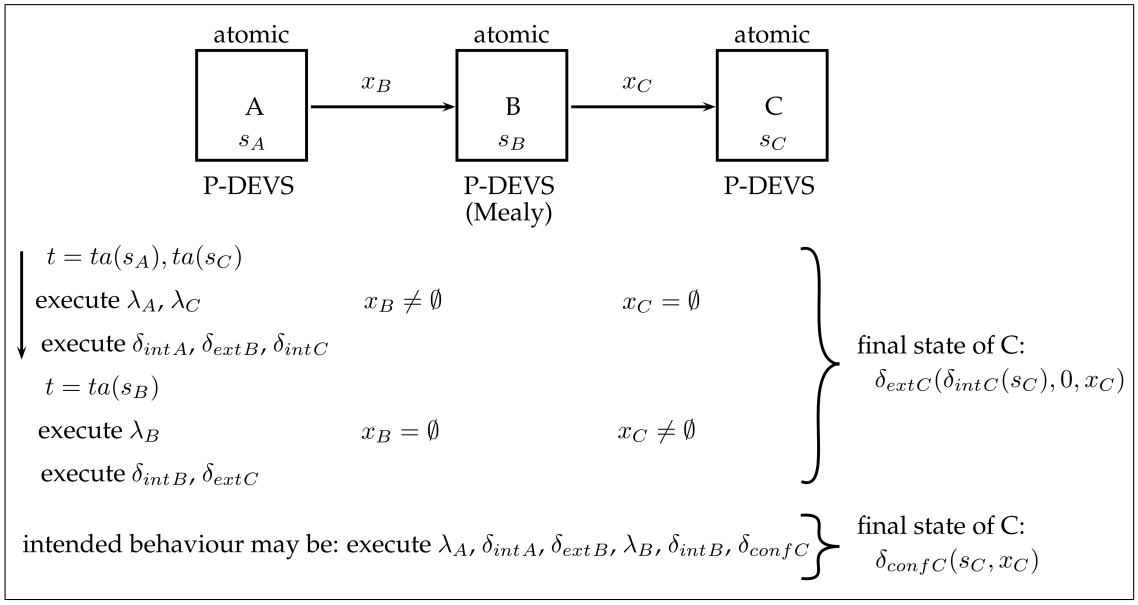


Figure 2.3: An example where the in [**?**] proposed P-DEVS simulator may lead to an unintended behaviour.

A possible way out of this dilemma will be discussed in section 4.1.

2.1.2 Modelling Formalism for Continuous Systems

2.1.2.1 Atomic DESS

DESS stands for Differential Equation System Specification and is the name of a formalism for describing models of systems with purely continuous behaviour. An atomic DESS is described by the following 5-tuple:

$$DESS_{atomic} =< X, Y, Q, f, \lambda >$$

where

X ... set of possible inputs(e.g. $\mathcal{C}\left(\mathbb{R}_0^+; \mathbb{R}^n\right)$)

Y ... set of possible outputs(e.g. $\mathcal{C}^1(\mathbb{R}_0^+; \mathbb{R}^m)$)

Q ... set of possible states(=state space)

$f : Q \times X \to Q$ ... rate of change function ('right side' of an ODE system)

$\lambda : Q \to Y$ ... output function (Moore type)

or

$\lambda : Q \times X \to Y$ ... output function (Mealy type)

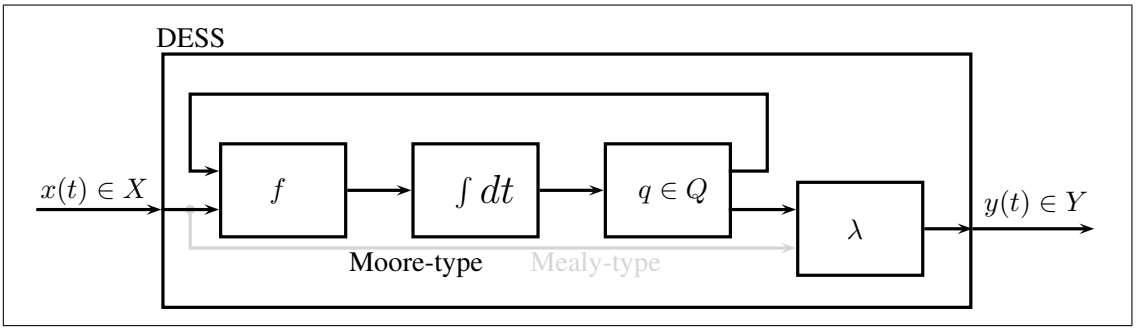


Figure 2.4: Graphical illustration of an atomic DESS.

The requirements for the input signal $x(t)$ are to be bounded and piecewise continuous and thus, for coupling reasons, the output signal $y(t)$ has to fulfil these requirements as well. Since the core issue of a DESS actually is to solve the ODE

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

for the existence of a unique solution $q(t)$ we also need f to fulfill the Lipschitz condition:

$$\|f(q, x) - f(q', x)\| < k \cdot \|q - q'\| \quad \forall q, q' \in Q \text{ and } x \in X \tag{2.1}$$

for a constant $k \in \mathbb{R}^+$.

2.1.2.2 Coupled DESS

The coupling mechanism for DESS works exactly in the same manner as for DEVS. Therefore, a coupled DESS is described by a tuple:

$$N =< X, Y, D, \{M_d\}_{d \in D}, \{I_d\}_{d \in D \cup \{N\}}, \{Z_d\}_{d \in D \cup \{N\}} >$$

The meaning of the tuple entries is the same as in section 2.1.1.2 except for Z_d, now called *interface map* for d:

for $d \in D \cup \{N\}$: $Z_d : \underset{i \in I_d}{\times} YX_i \to XY_d$, where

$$YX_i = \begin{cases} X & , if\ i = N \\ Y_i & , if\ i \in D \end{cases}$$

and

$$XY_d = \begin{cases} Y & , if\ d = N \\ X_d & , if\ d \in D \end{cases}$$

The difference to the output translation function $Z_{i,d}$ of section 2.1.1.2 is that now for the calculation of the input signal of block d all output signals of each influencing block $i \in I_d$ are needed, whereas with coupled DEVS, each influencing block $i \in I_d$ can produce an input message for block d independently of the other influencer. Therefore, when coupling DEVS

blocks it is possible to connect several outputs of different blocks to one input of another block. However, if those outputs provide messages at the same time, it is essential in which order they are treated at the receiving input. This order can be controlled with the tie breaking function $Select$. In case of P-DEVS there is no order among the concurrent input signals, since they are treated all at once. Concerning DESS, simultaneousness is a thing we do not need to worry about.

One issue that must be taken care of when coupling DESS are *algebraic loops*. As depicted in figure 2.4 we distinguish two kinds of DESS: Moore and Mealy type. Since with Mealy type DESS, the input directly influences the output of the system, when coupling Mealy type DESS, a circular dependency of output from input values – a so called algebraic loop – can arise. For this reason, to be able to ensure a coupled DESS to be well defined, each circular coupling needs to contain at least one Moore type block, i.e. the coupled DESS needs to be free of algebraic loops.

Actually the legitimacy-criterion for DEVS is closely related to the algebraic loop issue mentioned in the context of DESS. As DEVS blocks of type Mealy, that are blocks that may immediately response to an input with an output signal, coupled in a circular way may also lead to an illegitimate model. However, not every illegitimate coupled DEVS model contains an algebraic loop. It is even possible for a DEVS model to be illegitimate without having a single Mealy block in it. For example when an atomic DEVS walks through a series of inner states $(s_i)_{i\in\mathbb{N}}$ with $ta(s_i) > 0\ \forall i \in \mathbb{N}$ and $\sum_{i\in\mathbb{N}} ta(s_i) < \infty$. Therefore, an illegitimate coupled DEVS model does not exactly correspond to a DESS model with algebraic loop in it.

2.1.3 Modeling Formalism for Hybrid Systems

2.1.3.1 Atomic DEV&DESS

When modelling large and complex systems, pure discrete or pure continuous models may not suffice anymore to map each detail of interest in the real system onto the model. Therefore, a formalism for hybrid systems is needed. As the name already tells, DEV&DESS formalism is a composition of DEVS and DESS and therefore capable of describing hybrid systems.

An atomic DEV&DESS can be described by the following 11-tuple:

$$DEV\&DESS_{atomic} =< X^{discr}, X^{cont}, Y^{discr}, Y^{cont}, S, \delta_{ext}, C_{int}, \delta_{int}, \lambda^{discr}, f, \lambda^{cont} >$$

where

$$X^{discr}, Y^{discr} \ldots \text{set of possible discrete inputs and outputs}$$
$$X^{cont}, Y^{cont} \ldots \text{set of possible continuous inputs and outputs}$$
$$S = S^{discr} \times S^{cont} \ldots \text{set of possible states(=state space)}$$
$$Q = \{(s^{discr}, s^{cont}, e) | s^{discr} \in S^{discr}, s^{cont} \in S^{cont}, e \in \mathbb{R}_0^+\}$$
$$\delta_{ext} : Q \times X^{cont} \times X^{discr} \to S \ldots \text{external state transition function}$$
$$\delta_{int} : Q \times X^{cont} \to S \ldots \text{internal state transition function}$$
$$\lambda^{discr} : Q \times X^{cont} \to Y^{discr} \ldots \text{discrete output function}$$
$$\lambda^{cont} : Q \times X^{cont} \to Y^{cont} \ldots \text{continuous output function}$$
$$f : Q \times X^{cont} \to S^{cont} \ldots \text{rate of change function (“right side” of an ODE system)}$$
$$C_{int} : Q \times X^{cont} \to \{true, false\} \ldots \text{state event condition function}$$

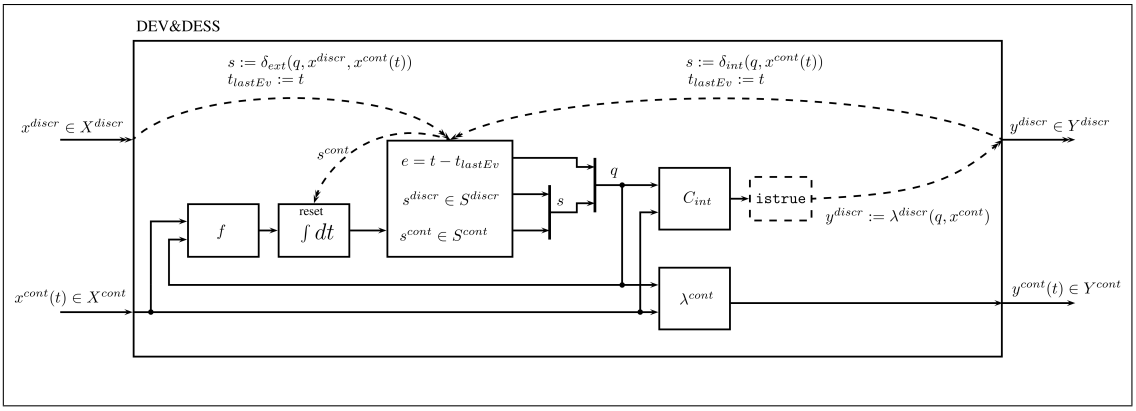


Figure 2.5: Graphical illustration of an atomic DEV&DESS.

The meaning of all sets and functions listed above can be looked up in the corresponding section for DEVS (see 2.1.1.1) and in the section for DESS (see 2.1.2.1), with on exception: C_{int}. C_{int} is a function of the actual state q and continuous input value $x^{cont}(t)$ and is responsible for triggering internal events, which then may cause a discrete output $y^{discr} = \lambda^{discr}(q, x^{cont})$ and definitely results in the execution of δ_{int}.

Therefore, internal events in DEV&DESS are not exclusively dependable on time, as it is the case with DEVS, but may also be triggered because of the system state s reaching a certain threshold. Events of the latter type are called *state events*. Since the state transition functions δ_{int} and δ_{ext} update the whole state, including its continuous part, they may lead to a discontinuous change in s^{cont}. Thus, as s^{cont} is the output of an integrator, this integrator needs to be reset, each time an external or internal event occurs.

The last distinguishing feature of the whole DEV&DESS from its components DEVS and DESS is the dependency of δ_{int} and λ^{discr} on the actual continuous input value.

For DEV&DESS to be well defined, we need to fulfil both, the requirements for the DEVS part, and the requirements for the DESS part. Therefore for each possible input trajectory and initial state, during a finite interval of time only a finite number of events is allowed to happen. Furthermore again the function f has to meet the Lipschitz requirements (2.1) and the continuous input and output signals need to be bounded and piecewise continuous.

2.1.3.2 Coupled DEV&DESS

Since atomic DEVS can be coupled with each other and atomic DESS can be coupled with each other, also atomic DEV&DESS can be coupled. However there are some restriction concerning the coupling of continuous outputs with discrete inputs. A coupled DEV&DESS can be described by the following 7-tuple.

$$N =< X^{discr} \times X^{cont}, Y^{discr} \times Y^{cont}, D, \{M_d\}_{d\in D}, \{I_d\}_{d\in D\cup\{N\}}, \{Z_d\}_{d\in D\cup\{N\}}, Select >$$

where

$$\begin{aligned}
X^{discr}, Y^{discr} &\ldots \text{set of possible discrete inputs and outputs}\\
X^{cont}, Y^{cont} &\ldots \text{set of possible continuous inputs and outputs}\\
D &\ldots \text{set of involved 'child DEV\&DESS' denominators}\\
M_d &\ldots \text{child DEV\&DESS of } N \text{ for each } d \in D\\
I_d \subset D \cup \{N\} &\ldots \text{influencer set of } d,\ d \notin I_d\\
Z_d &\ldots \text{interface map for } d\\
Select : 2^{D\cup\{N\}} \to D \cup \{N\} &\ldots \text{tie breaking function}
\end{aligned}$$

The meaning of all the terms listed above are already known either from the coupled DEVS definition in section 2.1.1.2 or from the coupled DESS defnition from section 2.1.2.2. But there are some restrictions, concerning the coupling of discrete outputs with continuous inputs and vice versa.

Therefore, the interface map Z_d is divided into two component functions. One for the calculation of the discrete inputs of block d:

$$Z_d^{discr} : \underset{i\in I_d}{\times} YX_i \to XY_d^{discr}$$

and one for the calculation of the continuous inputs

$$Z_d^{cont} : \underset{i\in I_d}{\times} YX_i \to XY_d^{cont}$$

Then, we need to define how to interpret a connection from a discrete output to a continuous input and the other way round:
Discrete output signals, actually are only existent at the instance of time when they are produced. The rest of the time, the value of the output signal is the empty set $\emptyset$ or non existent. However, to enable connections between discrete outputs and continuous inputs, we define discrete outputs

to be piecewise constant. So the value of a discrete output at a time between two output events is always the value of the last output event. Therefore, it is allowed to connect discrete outputs arbitrarily to continuous inputs. The other way round is not that easy, and it is necessary to apply restrictions. Thus, continuous outputs are only allowed to be connected to discrete inputs, if they are piecewise constant.

One could think of a connection from discrete to continuous being realized by putting an additional DEV&DESS block in between that receives the discrete output at its discrete input and forwards it to its continuous output. The other way around works as well.

As DEV&DESS sums up the functionality of both sides, the discrete and the continuous one, the modeller has to deal with the requirements of each formalism as well. On the one hand the modeller needs to take care not to produce algebraic loops, and on the other hand he also needs to think of how to define the tie breaking function $select$ for the model to produce the desired behaviour.

As Zeigler showed [**?**], all three basic formalisms, DEVS, DTSS, and DESS describe subclasses of the set of DEV&DESS describable systems. Therefore, DEV&DESS is perfectly suited to formally describing hybrid models of real systems in a simulator independent way. Nevertheless, when it comes to simulate a DEV&DESS on a digital computer, it is necessary to discretise the DESS part resulting in a pure DEVS. In which way this discretisation is carried out already depends on the simulator in use and the possibilities for numerical ODE treatment it provides . Common ODE solvers are working in a discrete time fashion, meaning that they compute solution samples of the whole ODE in constant or variable time steps. An alternative and rather young method is called QSS and will be described in detail in section 2.2. QSS is actually a more extensive method for numerically solving an ODE than ODE solvers, because it separately integrates each single dimension of the ODE and therefore applies different step sizes for each dimension. Furthermore, in higher order QSS methods, there is more effort put into the description of continuous signals than with ODE solvers. Thereby, continuous signals are not only described by their actual value but also by their first, second, third,... derivative, i.e. by values describing the short term evolution of the signal. Of course this does not come for free and the price to pay is a higher memory consumption and the requirement of some kind of event scheduling support from the simulator. Anyway, as for the simulation of hybrid systems an event scheduler is indispensable, QSS is perfectly suited for implementing a simulation model of a DEV&DESS.

2.2 QSS – Quantized State System

QSS stands for *Quantized State System* and the underlying idea was first proposed by Zeigler as method to create a DEVS representation of a continuous system. Kofman [**?**] improved that method and today there already exists a variety of enhancements [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**].

In Zeigler's definition of *DESS*(*Differential Equation System Specification*), a continuous system needs to be able to cope with bounded piecewise continuous input signals. Zeigler's idea was now to discretize a DESS model S simply by adding a *Quantizer* in succession of S. This Quantizer just discretises all output signals of S. As mentioned above, a DESS can be fed with piecewise constant signals, and therefore, the coupled system of S plus Quantizer works like a

discrete event system (see Figure 2.6). Note that DEVS signals actually only exist at event times and have the value $\emptyset$ all the other time. However, for coupling purposes with DESS and *DTSS* (*Discrete Time System Specification*), they can be interpreted as piecewise constant(see [**?**]). Therefore, every DESS followed by a quantizer represents a DEVS. The input quantizer in Figure 2.6 actually represents the output quantizer of the discretised DESS preceding in the coupling scheme or it just quantizes an overall system input signal.

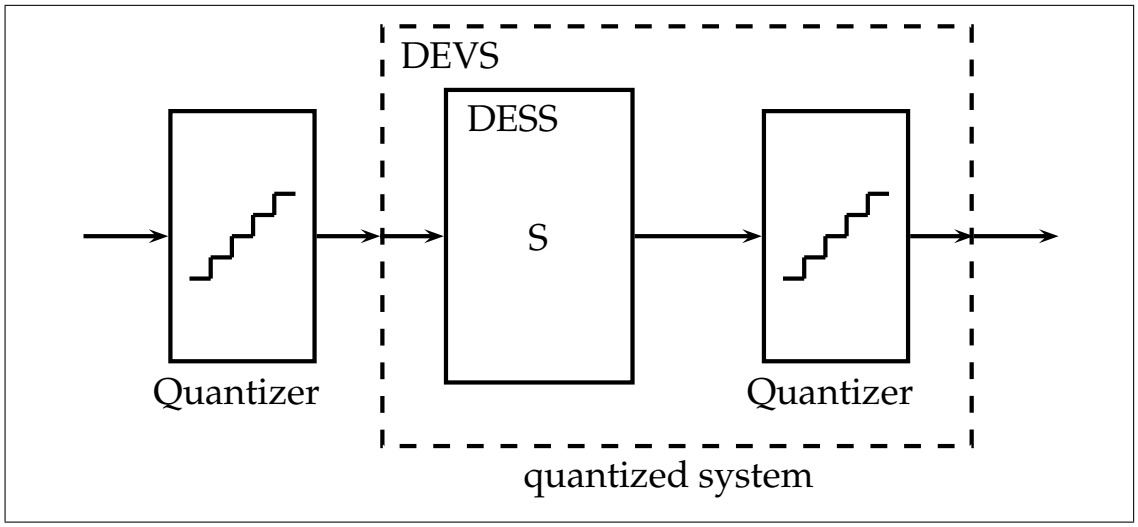


Figure 2.6: Illustration of a quantized System

2.2.1 Quantizers and Quantization Methods

Quantizers actually origin from the field of electrical engineering and their task is to map analogue signal values within a given range onto a finite set of digital values. Quantizers as Zeigler describes them are doing quite the same. They convert a continuous input signal into a piecewise continuous step function where the step height is a value out of a given finite set. Usually the values of this set are equidistant and the difference between two such neighbouring values is called *Quantum*. The step height of a QSS quantized signal at a given point in time corresponds to the highest possible step height that is smaller than the continuous signal value at that time. This leads to a maximum absolute error of one quantum.

Actually when quantizing electrical signals, the actual step height is the nearest neighbour of the current continuous signal value in the set of possible step heights, which leads to a maximum absolute error of half a Quantum. However, the reason for the rather simple quantization method used with QSS may be that it makes the expansion to QSS methods of higher order more straight forward.

Zeigler's simple quantization method has one drawback: if the continuous signal value is right at the border to the next step heights, that is, around that value where the output of the quantizer changes its value, it may happen that the quantizers output value permanently jumps from the lower to the higher level and back again. To overcome that, Kofman introduced a hysteresis at the switching levels (illustrated in Figure 2.7). But this was only the first enhancement.

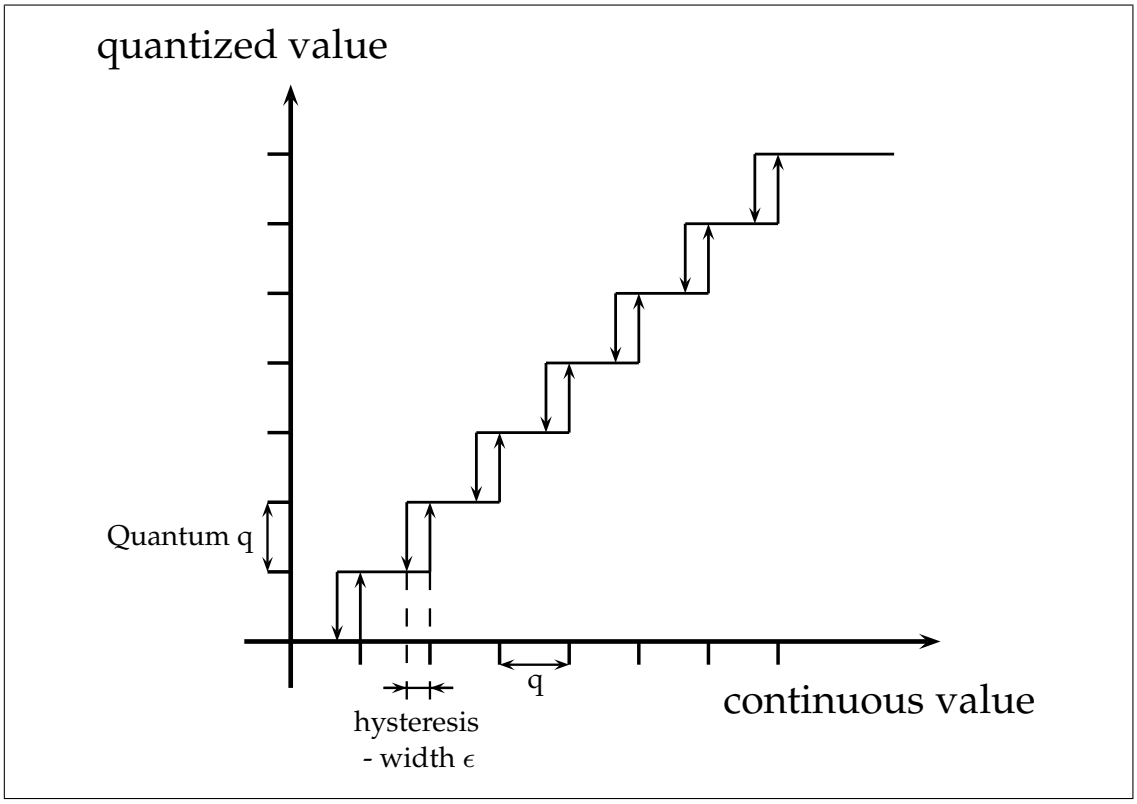


Figure 2.7: Illustration of the woriking principle of a QSS quantizer.

2.2.1.1 QSS

The abbreviation QSS is used with different meanings. Originally it is the name for a system discretised in a specific way (a system with quantised state). Then the name is used for the concrete quantisation method used thereby(see Figure 2.7) but also as general term for the whole class of quantisation methods consisting of the original one (of first order) and all extensions of it (of higher order, see section 2.2.1.2). The output of such a quantisation is a quantised signal which is called QSS signal. Thus, QSS also denotes a method of describing continuous signals. Since each QSS order produces a different description of a continuous signal, QSS again denotes also the whole class of continuous signal description methods corresponding to the class of quantisation methods. However, different quantisation methods may use the same QSS signal description method. Furthermore, each QSS order comes with a native variable step size integration method of continuous signals. The method for QSS of first order is called QSS integration but again also the whole class of integration methods of the different QSS orders.

2.2.1.2 QSSn – QSS of Order n

The next step from QSS to QSS2 is quite interesting. For this purpose, we regard the QSS quantized signal not only as piecewise constant approximation of the continuous signal but as

piecewise Taylor approximation of order zero at the particular point in time where the last step happened. So each time the difference between continuous signal value and step height becomes equal to the quantum q or $-\epsilon$, the Taylor expansion point is updated. See Figure 2.8 for an example.

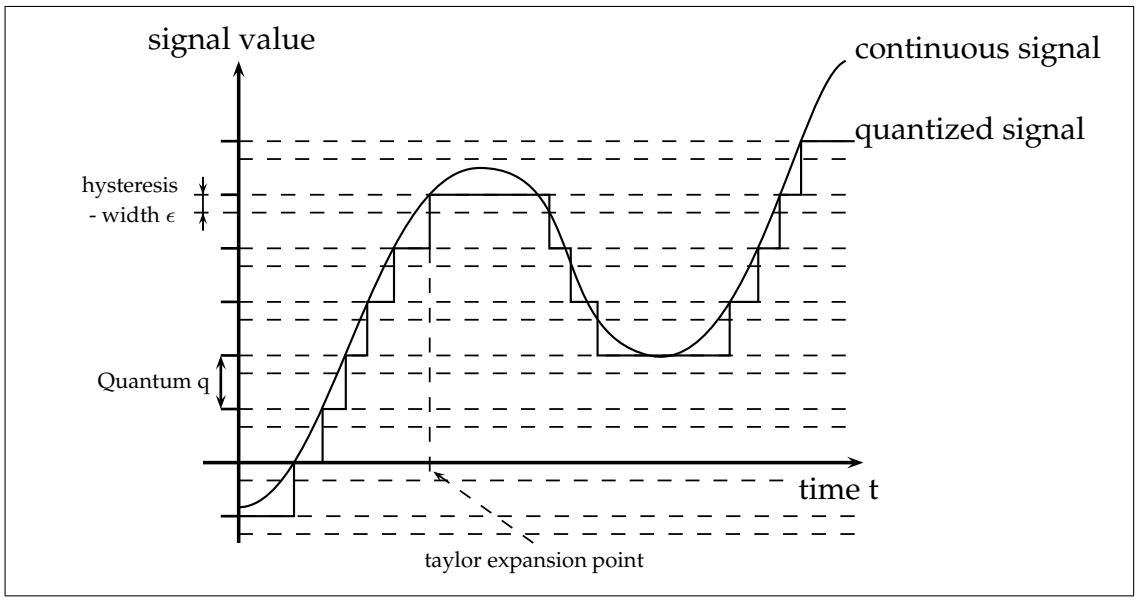


Figure 2.8: QSS quantization of a continuous signal.

So the extension to QSS2 is obvious now. Just describe the continuous signal not with piecewise Taylor approximations of order zero, but of order one. That is, with piecewise linear segments. And again, whenever the distance from linear approximation to continuous signal exceeds the quantum q, the Taylor expansion point is updated to the point of the continuous signal where this exceeding first occurs. See figure 2.9 for an example.

Exactly the same principle works with QSSn, $n = 3, 4, \ldots$, meaning that QSSn signals consist of piecewise polynomial segments of degree $n - 1$. Note that QSSn signals are actually piecewise continuous but not continuous, because there are points of discontinuity whenever the taylor expansion point is updated. An advantage of QSSn, $n \geq 2$ is that there is no need for a hysteresis any more, since after an update of the taylor expansion point, the difference of continuous and quantized signal is zero and therefore, small fluctuations of the continuous signal do not lead to further expansion point updates. A disadvantage of QSSn, $n \geq 2$ is that discretisation of DESS systems does not work that straight forward any more as it was with QSS. This is because with QSSn, $n \geq 2$, the description of signals now consists of a n-tuple instead of a single value and therefore cannot be fed directly into the inner DESS system S as shown in Figure 2.6. However, this problem simply can be fixed by sampling the piecewise polynomial QSSn signal from the output of the left quantizer in Figure 2.6 resulting in a QSS signal that can be fed into the DESS. Figure 2.10 illustrates that.

That is, if the S&H block (Sample & Hold) in Figure 2.10 for example receives a DEVS signal (a_k, b_k) at time t_k and another one (a_{k+1}, b_{k+1}) at time t_{k+1}, it produces at the ouput

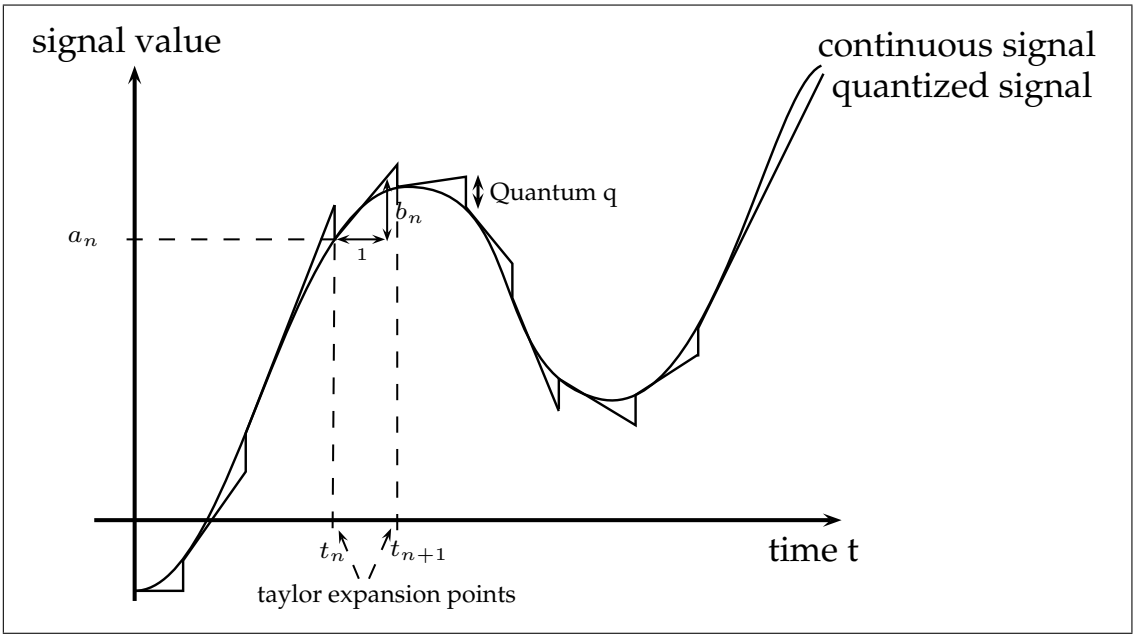


Figure 2.9: QSS2 quantization of a continuous signal. The linear approximation of the continuous signal between t_n and t_{n+1} is $a_n + b_n \cdot (t - t_n)$.

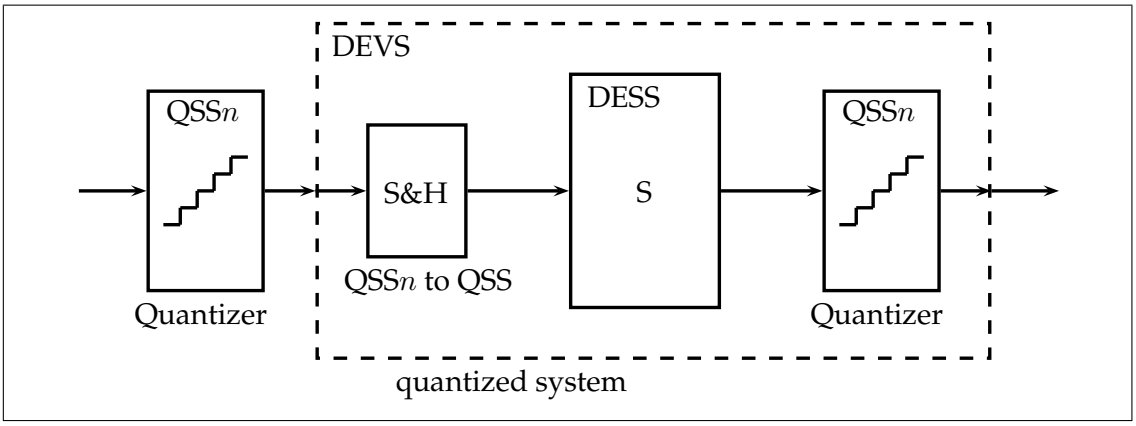


Figure 2.10: Illustration of a way discretise a DESS using QSSn.

samples $a_k + b_k \cdot (t_j - t_k)$ at times t_j with $t_j \in [t_k, t_{k+1})$ for $j \in \{0, 1, \ldots, m\}$ (m is an arbitrary non negative integer determining the sample rate).

The advantage of this type of discrete description of continuous signals is that, knowing the actual QSSn representation of the signal, one does not only know the current value of the signal, but also how it will evolve in the near future. This makes it possible to calculate the points in time when for example a continuous state variable reaches some concrete threshold triggering a state event instead of iteratively searching for them.

Additionally to the mentioned QSSn quantization methods there also exists a logarithmic quantization method, where the size of the quantum is not constant but depends on the actual

size of the continuous signal (see [**?**]). This is interesting, because the principle of QSS1 actually is the same as fixed point representation of reals in digital computers, whereas logarithmic quantisation can be compared with floating point representation (see 2.2.3).

2.2.2 Integration of QSS signals

As we learned before, QSS signals are actually piecewise polynomial:

$$f(t) = a_k + b_k \cdot (t - t_k) + c_k \cdot (t - t_k)^2 \dots$$

for $t \in (t_k, t_{k+1}]$ and $k \in \mathbb{N}$. The integration of a polynomial of degree m simply results in a polynomial of degree $m + 1$:

$$w(t) = w(t_k) + \int_{t_k}^{t} f(\tau)d\tau = w(t_k) + a_k(t - t_k) + \frac{b_k}{2} \cdot (t - t_k)^2 + \frac{c_k}{3} \cdot (t - t_k)^3 \dots$$

Therefore the numerical solution of IVPs like (1.1) is quite simple. However, for coupling reasons, one QSS simulation model should use the same order of QSSn everywhere in the model. Therefore, the output of an integrator block receiving a polynomial of degree m should also be of degree m, which can be achieved by simply cutting off the output polynomial after the m-th coefficient. This again introduces a quantization error. To bound that error, we just treat the correct output polynomial of order $m + 1$ the same way we treat other continuous signals entering the QSS: we quantize it.

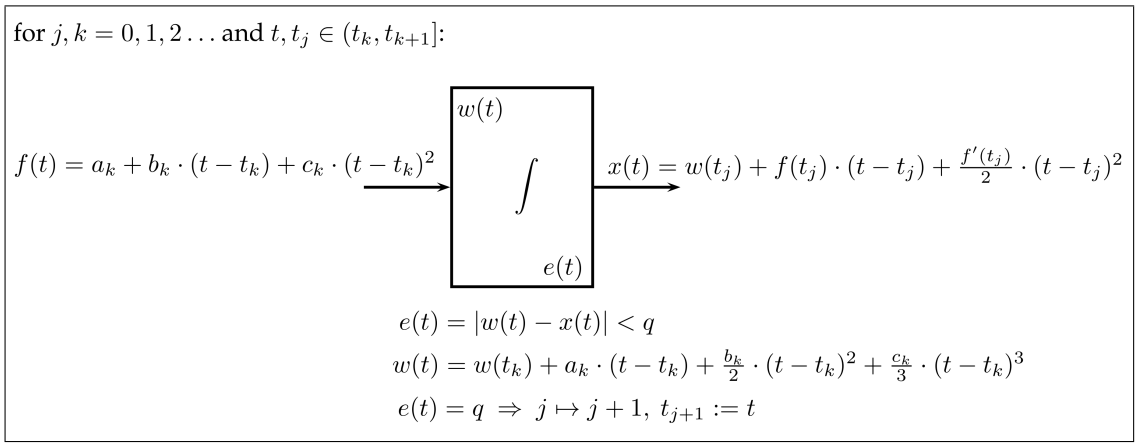


Figure 2.11: The working principle of a QSS3 integrator.

Figure 2.11 illustrates the working principle of an ordinary QSS3 integrator. Every time the distance between polynomial $w(t)$ of degree $m + 1$ and polynomial $x(t)$ of degree m exceeds the quantum q, the Taylor expansion point of the output signal $x(t)$ is updated resulting in a new output polynomial and an incrementation of j. Actually this could be interpreted as state event. With defining the absolute difference of $x(t)$ from $w(t)$ as inner state $e(t)$ of the integrator, we end up with a state event each time $e(t)$ reaches the quantum q. Thus, the smallest positive zero crossing of the so called *event function* $e(t) - q$ marks the time of the state event. Since $x(t)$ and

$w(t)$ are polynomials we perfectly know, as long as their degree is smaller than 5, the time of the state events have not to be searched for iteratively using Newton, Regular Falsi or something similar but simply can be determined by calculating the zeros of the polynomials $w(t) - x(t) - q$ and $w(t) - x(t) + q$. Exactly this is one big advantage of QSS regarding state events.

Apart from that straight forward integration with successive quantization explained above, there also exist methods *BQSS*,*CQSS* ([**?**]) as well as *LIQSS*,*LIQSS2* ([**?**]) for stiff systems. However, those methods actually do not change the way, the input polynomial $f(t)$ is integrated, but the quantization of the integrated input polynomial $w(t)$ works in a different way there.

2.2.3 Quantization in Non-QSS Simulators

Due to the fact that for every digital computer, the number of digits it can work with is limited, actually any numerical software running on a digital computer works with quantized numbers. Therefore, each implementation of an IVP like (1.1) in a tool like SIMULINK or Dymola, i.e. the simulation of the model described by the IVP in one of those tools, in fact is based on a kind of QSS representation of all continuous signals appearing there. There are two basic types of quantization:

1. fixed point representation:

   $$x = d_1 d_2 \dots d_m \,.\, d_{m+1} d_{m+2} \dots d_{m+n}$$

   with a fix quantum size $q = 10^{-n}$ and

2. floating point representation:

   $$x = s_1 s_2 \dots s_m \times 10^{e_1 e_2 \dots e_n}$$

   with a variable quantum size $q = 10^{e_1 e_2 \dots e_n}$ depending on the actual size of x (comparable with logarithmic quantization [**?**]).

So QSSn actually can be seen as an enhancement of the quantization methods used in digital computers, which is aimed at representing not only constant real numbers but also whole real valued signals, meaning real values and their short term evolution.

2.2.4 Similarities of QSSn and Explicit One-Step ODE Solvers

Now let us take a look at the parallels of ODE solvers and QSS methods. We start with a scalar IVP like:

$$\begin{cases} \dot{x}(t) = f(x,t) \\ x(t_0) = x_0 \end{cases} \tag{2.2}$$

which we want to solve numerically using QSSn and compare the resulting formulas with those of explicit one step ODE solvers which are defined as:

$$x_{k+1} = x_k + \Phi(x_k, t_k, f) \cdot h_k \qquad k = 0, 1, \dots N \tag{2.3}$$

where

$$t_0 < t_1 \cdots < t_N, \quad h_k = t_{k+1} - t_k$$

We will start with applying simple QSS onto (2.2). Figure 2.12 illustrates that with a signal flow graph. Actually with QSS, the values passed on from one block to the next are piecewise constant and therefore, between to changes of the quantization level, x_k and $f_{k,m}$ are independent from time. However, the graphic is designed to explain QSSn as well and that is why x_k and $f_{k,m}$ are drawn time dependent.

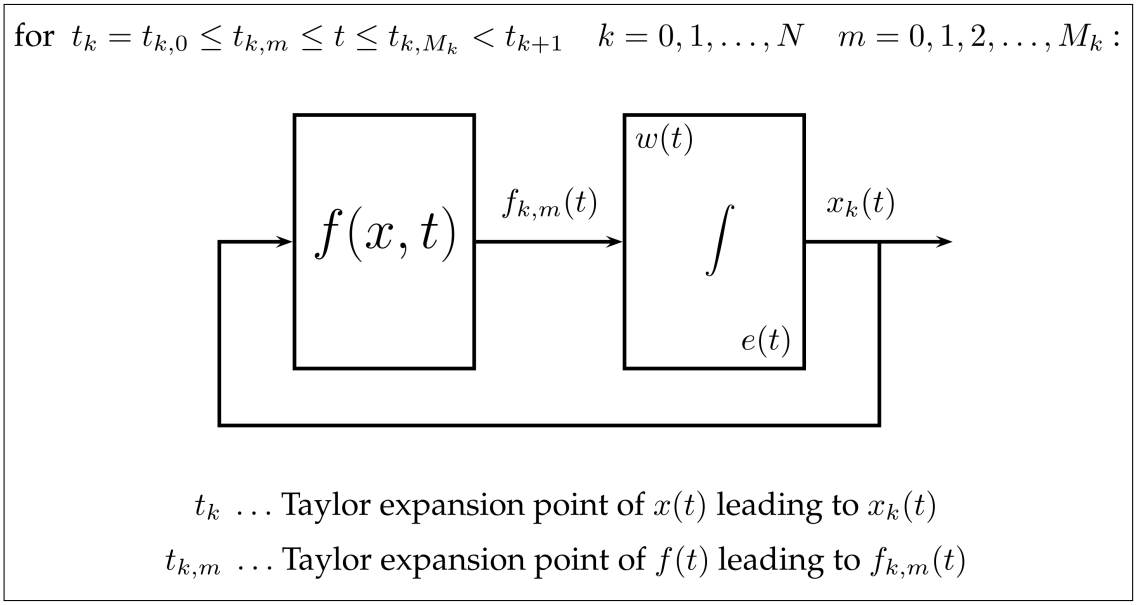


Figure 2.12: Block diagram, illustrating the QSSn integration of an initial value problem.

Initially the integrator outputs the quantized initial value x_0. Let assume, that x_0 can be represented exactly with QSS. Then $f(x_0, t_0)$ is calculated, leading to the quantized value $f_{0,0}$ that is fed into the integrator. The integration of $f_{0,0}$ leads to the polynomial of first degree

$$w(t) = w(t_0) + f_{0,0} \cdot (t - t_0)$$

($w(t_0) = x_0$) that describes a more precise representation of the desired solution $x(t)$ than the piecewise constant output x_k does. This polynomial $w(t)$ is then used to get a local error estimation

$$e(t) = |w(t) - x(t)| = |f_{0,0} \cdot (t - t_0)|.$$

By bounding this error estimation by the quantum q, the time t_1 for next integrator output update or in other words the step size $h_1 = t_1 - t_0$ can be calculated:

$$t_1 = t_0 + \frac{q}{|f_{0,0}|}$$

There are two sources for change of the quantized value of $f(x,t)$: a change of x and change of time t. That is why quantized f is equipped with two indices in Figure 2.12. The first is incremented each time t_k the integrator updates its output value x_k. The second one is incremented

each time $t_{k,m}$, f changes due to the direct time dependency of f and is reset to zero whenever the integrator updates its output value (which leads to an incrementation of the first index).

So every time the output value of the f block changes, the polynomial $w(t)$ changes, and therefore, time t_1 has to be recalculated. If, for example quantized f changes at $t = t_{0,1} > t_0$, then $w(t)$ changes to

$$w(t) = w(t_{0,1}) + f_{0,1} \cdot (t - t_{0,1})$$

Nevertheless, it can be seen that $w(t)$ is continuous although the integrators output $x(t)$ is not.

Finally, if there is no further change of f until time t_1 is reached, the integrator output is updated with

$$x_1 = w(t_1)$$

and the whole process is repeated until the designated end time is reached.

The calculation of the integrator step size depending on an error estimation strongly reminds of variable step size ODE solver. Now let us deduce a recursive formula for x_{k+1}.

Let's assume that the actual output of the integrator is x_k, the actual quantized value of f is $f_{k,m}$, and the actual value of w is $w(t) = w(t_{k,m}) + f_{k,m} \cdot (t - t_{k,m})$. Further, let's assume that the time for the next output update of the integrator is calculated with t_{k+1} and that there will be no change of quantized f until t_{k+1} – in this case we define $M_k = m$. Then the next integrator output value can be calculated as:

$$\begin{aligned}
x_{k+1} &= w(t_{k+1}) \\
&= w(t_{k,M_k}) + f_{k,M_k} \cdot (t_{k+1} - t_{k,M_k}) \\
&= w(t_{k,M_k-1}) + f_{k,M_k-1} \cdot (t_{k,M_k} - t_{k,M_k-1}) + f_{k,M_k} \cdot (t_{k+1} - t_{k,M_k}) \\
&\vdots \\
&= w(t_k) + f_{k,0} \cdot (t_{k,1} - t_{k,0}) + f_{k,1} \cdot (t_{k,2} - t_{k,1}) + \cdots + f_{k,M_k} \cdot (t_{k+1} - t_{k,M_k}) \\
&= w(t_k) + \left(f_{k,0} \, \frac{t_{k,1} - t_{k,0}}{t_{k+1} - t_k} + f_{k,1} \, \frac{t_{k,2} - t_{k,1}}{t_{k+1} - t_k} + \cdots + f_{k,M_k} \, \frac{t_{k+1} - t_{k,M_k}}{t_{k+1} - t_k} \right) \cdot (t_{k+1} - t_k) \\
&= x(t_k) + \left(f_{k,0} \, \frac{t_{k,1} - t_{k,0}}{t_{k+1} - t_k} + f_{k,1} \, \frac{t_{k,2} - t_{k,1}}{t_{k+1} - t_k} + \cdots + f_{k,M_k} \, \frac{t_{k+1} - t_{k,M_k}}{t_{k+1} - t_k} \right) \cdot h_k
\end{aligned}$$

where $f_{k,m} = f(x_k, t_{k,m})$. Looking at the last line of this formula, one can see that it looks exactly like the recursion for a one step ODE solver (2.3) with

$$\Phi = f_{k,0} \, \frac{t_{k,1} - t_{k,0}}{t_{k+1} - t_k} + f_{k,1} \, \frac{t_{k,2} - t_{k,1}}{t_{k+1} - t_k} + \cdots + f_{k,M_k} \, \frac{t_{k+1} - t_{k,M_k}}{t_{k+1} - t_k} \tag{2.4}$$

This means that QSS integration can be seen as one step, variable step ODE solver with the speciality of changing the calculation method of Φ after each step. More detailed, in each step $x_k \to x_{k+1}$, Φ is a convex combination of a set of samples of $f(x_k, \cdot)$ between t_k and t_{k+1}. Amount M_k and position in time of those samples depend on the rate of change of $f(x_k, \cdot)$ in the interval (t_k, t_{k+1}). For the special case that $f(x_k, t)$ does not change for $t \in (t_k, t_{k+1})$, i.e. $M_k = 0$, Φ would become:

$$\Phi = f_{k,0} = f(x_k, t_k)$$

and therefore:

$$x_{k+1} = x_k + f(x_k, t_k) \cdot h_k \tag{2.5}$$

which actually is nothing else than the recursion of explicit Euler method. Therefore, assuming f depending only on $x(t)$, we get that QSS integration is exactly the same as variable step explicit Euler method.

For QSSn integration the signals $f_{k,m}(t)$, $w(t)$, and $x_k(t)$ for $t \in (t_{k,m}, t_{k,m+1})$ have the form

$$\begin{aligned}
f_{k,m}(t) &= f_{k,m} + f'_{k,m} \cdot (t - t_{k,m}) + \frac{f''_{k,m}}{2} \cdot (t - t_{k,m}) + \cdots + \frac{f^{(n-1)}_{k,m}}{(n-1)!} \cdot (t - t_{k,m})^{n-1} \\
w(t) &= w(t_{k,m}) + f_{k,m} \cdot (t - t_{k,m}) + \frac{f'_{k,m}}{2} \cdot (t - t_{k,m})^2 + \cdots + \frac{f^{(n)}_{k,m}}{n!} \cdot (t - t_{k,m})^n \text{ and} \\
x_k(t) &= x_k(t_k) + f_{k,0} \cdot (t - t_k) + \cdots + \frac{f^{(n-1)}_{k,0}}{(n-1)!} \cdot (t - t_k)^{n-1}
\end{aligned}$$

with $f^{(j)}_{k,m} = \frac{d^j f}{dt^j}(x_k(t_{k,m}), t_{k,m})$. The final result for the recursion is then

$$\begin{aligned}
x_{k+1} = x_k &+ \left(f_{k,0} \frac{t_{k,1} - t_{k,0}}{t_{k+1} - t_k} + f_{k,1} \frac{t_{k,2} - t_{k,1}}{t_{k+1} - t_k} + \cdots + f_{k,M_k} \frac{t_{k+1} - t_{k,M_k}}{t_{k+1} - t_k} \right) \cdot h_k \\
&+ \left(\frac{f'_{k,0}}{2} \left(\frac{t_{k,1} - t_{k,0}}{t_{k+1} - t_k} \right)^2 + \frac{f'_{k,1}}{2} \left(\frac{t_{k,2} - t_{k,1}}{t_{k+1} - t_k} \right)^2 + \cdots + \frac{f'_{k,M_k}}{2} \left(\frac{t_{k+1} - t_{k,M_k}}{t_{k+1} - t_k} \right)^2 \right) \cdot h_k^2 \\
&+ \left(\frac{f''_{k,0}}{6} \left(\frac{t_{k,1} - t_{k,0}}{t_{k+1} - t_k} \right)^3 + \frac{f''_{k,1}}{6} \left(\frac{t_{k,2} - t_{k,1}}{t_{k+1} - t_k} \right)^3 + \cdots + \frac{f''_{k,M_k}}{6} \left(\frac{t_{k+1} - t_{k,M_k}}{t_{k+1} - t_k} \right)^3 \right) \cdot h_k^3 \\
&\vdots \\
&+ \left(\frac{f^{(n-1)}_{k,0}}{(n-1)!} \left(\frac{t_{k,1} - t_{k,0}}{t_{k+1} - t_k} \right)^{n-1} + \cdots + \frac{f^{(n-1)}_{k,M_k}}{(n-1)!} \left(\frac{t_{k+1} - t_{k,M_k}}{t_{k+1} - t_k} \right)^{n-1} \right) \cdot h_k^{n-1}
\end{aligned}$$

which again can be identified as, let us call it *variable structure* one step, variable step ODE solver. Assuming that $f(x_k, t)$ is not changing for $t \in (t_k, t_{k+1})$ it follows

$$x_{k+1} = x_k + f(x_k, t_k) \cdot (t_{k+1} - t_k) + \frac{\frac{df}{dt}(x_k, t_k)}{2} \cdot (t - t_k)^2 + \cdots + \frac{\frac{d^{n-1} f}{dt^{n-1}}(x_k, t_k)}{(n-1)!} \cdot (t - t_k)^{n-1} \tag{2.6}$$

which is also known as Taylor method among ODE solvers.

If the investigation is now extented to d dimensional IVPs, two things can be recognized:

First, QSSn treats each index of the vectorial values $\mathbf{x}_k$, $f(\mathbf{x}, t)$ separately, resulting in different step sizes for each single dimension, whereas ODE solvers treat the vectorial ODE as a whole and therefore choose one step size for all dimensions.

Second, the function blocks $f_i(\mathbf{x}, t)$ in the QSSn signal flow graph now change their output value not only due to time changes and changes of x_i but also due to changes of all other entries of the vector $\mathbf{x} = (x_1, x_2, \ldots, x_d)$. See Figure 2.13.

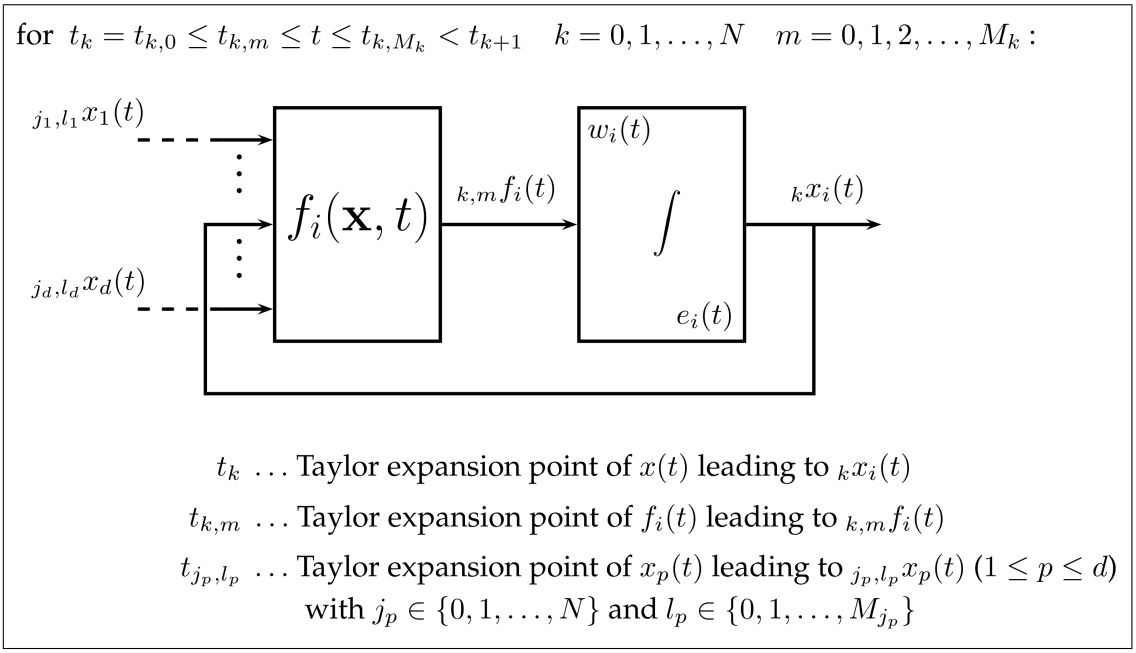


Figure 2.13: Signal flow graph of QSSn applied to a vectorial initial value problem.

However, the changes of f_i due to changes of x_j, $j \neq i$ can be seen as changes due to time changes. Therefore, QSSn applied on vectorial IVPs corresponds to applying a separate variable-step one-step ODE solver on each dimension.

One last point which should be mentioned in this context is that calculation of arithmetic expressions consisting of QSSn signals, as for example the function $f(x, t)$ may be, can only be performed free of error if they consist of nothing else but summation and multiplication with constant values. This is because the set of all polynomials of degree n is a vector space and each operation on that vector space that would lead out of it, e.g. division or inversion of polynomials, is accompanied with an error, since the resulting QSSn signal would be only a projection of the correct result into the vector space of polynomials of degree n. This drawback could be eased when such calculations would be performed in a vector space of polynomials of higher degree and then this more precise result would be simply quantized the same way a continuous signal entering the QSS is quantized. That is, each time t the distance of the higher degree polynomial to the n-degree polynomial reaches the quantum, the n-degree polynomial is replaced by the n-degree Taylor approximation of the higher degree polynomial with expansion point t.

CHAPTER 3

PowerDEVS

PowerDEVS[1] [**?**] is an open source software tool for the simulation of dynamical systems. Models can be entered as block diagrams as known from other simulators simulators like SIMULINK, XCOS or Dymola. However, in PowerDEVS the behaviour of each single block strictly follows a corresponding DEVS or a corresponding coupled DEVS and the coupling of blocks follows the rules of coupled DEVS. Although, as long as the modeller only uses predefined library blocks to build up a model by establishing the desired input-output couplings, there is no need to know anything about DEVS at all. Not until the library blocks do not suffice anymore to model the desired behaviour, knowledge about DEVS is necessary. In this case a new block has to be created, which in in PowerDEVS can be done by defining the block's functionality as DEVS. The DEVS of the new block then can be implemented straight forward as C++ code using the *atomic editor* which will be introduced in one of the next sections.

Since in background PowerDEVS works with a pure DEVS model, the kind of systems that can be simulated are discrete event systems. As discussed in chapter 2 this also includes all discrete time systems, like e.g. digital circuits and continuous systems described as differential equations and solved using ODE solver algorithm. However, PowerDEVS supports a different method to discretise continuous systems: QSS. QSS transforms the continuous system not into a discrete time system but directly into a discrete event system. Thus, all the blocks for continuous systems simulation that come with the PowerDEVS library use QSS. The first time the modeller is confronted with that fact is when he uses an integrator-block in his model. The integrator block provides a parameter for setting the concrete QSS-method to be used.

PowerDEVS consists of four independent parts:

- model editor
- atomic editor
- structure generator

[1] `http://sourceforge.net/projects/powerdevs/`

- preprocessor

Further, PowerDEVS closely cooperates with the open source numerical computations software *Scilab*[2]. Scilab is very similar to Octave[3] or MATLAB[4] and its user interface consists of a console where variables of all kinds can be defined and are stored in the Scilab workspace.

A simulation model created in PowerDEVS is able to connect to a running Scilab session via TCP[5] and thus, it can access the Scilab workspace and make use of the Scilab command interpreter. To give an example, the simulation model can read simulation and model parameters from Scilab workspace at the beginning of the simulation and write simulation results back to workspace when it is finished. Furthermore, all numerical methods Scilab offers can be used in PowerDEVS. This is utilised particularly for the interpretation of inputs entered as parameter values for PowerDEVS blocks. Most of the predefined library blocks let their parameter values be interpreted by Scilab and use the Scilab return value for the actual parameter value. If the entered parameter value is a number, Scilab just returns that number. However, if the entered value for example is something like `A*sin(x)`, `tan(1/2)` or `exp(-5*y)` with `A`, `x` and `y` being Scilab workspace variables, Scilab interprets this expression and returns a numerical value.

Scilab also offers the possibility to write scripts. In combination with the fact that PowerDEVS simulation models can be executed from Scilab console, this could be used to write a Scilab script that executes a sequence of simulations with varying simulation and model parameters and subsequently does the postprocessing of the simulation results.

3.1 Model Editor

The PowerDEVS model editor is the user interface for the modeller to graphically create simulation models. Figure 3.1 shows the model interface with an implemented bouncing ball model. All blocks in the model are predefined library blocks. From the simulation engine's point of view, the whole model is one coupled DEVS.

The block 'reset v(t) on bounce' in Figure 3.1 is also a coupled DEVS. Its internal structure is depicted in Figure 3.2. In Figure 3.2 the library 'Basic Elements' is open. This library consists of four blocks. The 'Atomic' block represents an atomic DEVS template, i.e. an atomic DEVS whose functionality still needs to be programmed in C++ programming language. It can be used to create a custom block. The other three blocks, 'Coupled', 'Inport', and 'Outport' are needed for coupled DEVS. To create a coupled DEVS, a 'Coupled' block has to be dragged and dropped into the model. With a right click on the 'Coupled' block and selecting 'Open coupled', the coupled model can be opened, which results in an empty model worksheet. Then arbitrary library blocks can be dragged and dropped into the empty worksheet and coupled with each other. For

[2] `http://www.scilab.org/`

[3] `http://www.gnu.org/software/octave/`

[4] `http://www.mathworks.com/products/matlab`

[5] TCP ... Transmission Control Protocol; "TCP provides reliable, ordered, and error-checked delivery of a stream of octets between applications running on hosts communicating over an IP network. TCP is the protocol that major Internet applications such as the World Wide Web, email, remote administration and file transfer rely on."(en.wikipedia.org, 2015)

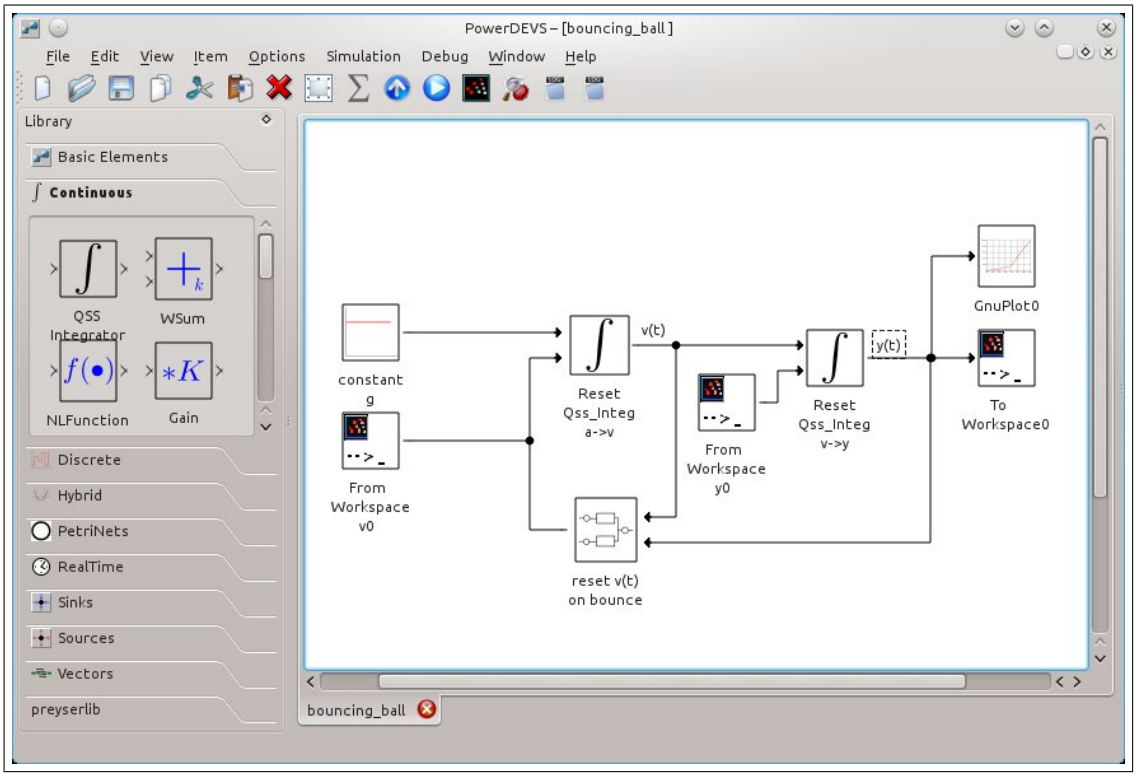


Figure 3.1: The PowerDEVS model editor with a bouncing ball model. The block 'reset v(t) on bounce' represents a coupled model, comparable to a SIMULINK subsystem. On the left the different libraries can be seen. Currently the library for continuous systems is open.

each 'Inport' and 'Outport' block inserted into the coupled model, one hierarchical level above the 'Coupled' block is equipped with an additional input port and output port respectively. In this way arbitrary hierarchical DEVS models can be created.

Figure 3.3 shows the dialogue, opened when double clicking on an integrator block in a PowerDEVS model. Here model parameter values can be entered. As mentioned above, most parameters allow any expression interpretable by Scilab to be entered. When creating a custom block, the amount, names, and types of parameters shown in this dialogue can be defined.

Figure 3.4 shows the simulation dialogue window. It is opened, when clicking onto the blue play symbol in the model editors tool bar. When clicking on that button, the structure generator and the preprocessor create the simulation model from the C++ source code describing the single blocks and from the coupling information given with the drawn connections.

From the simulation dialogue window a simulation run can be started using start time `t=0` and the value entered in the 'Final Time' input field as simulation end time. Additionally, the number of successive simulation runs can be defined as well as the maximum of concurrent events that are allowed to happen. That is, if there are more than the given maximum number of events triggered at the same time, the model is considered *illegitimate* and therefore, the simulation is aborted. Further, there is the possibility to stop the simulation after a given amount

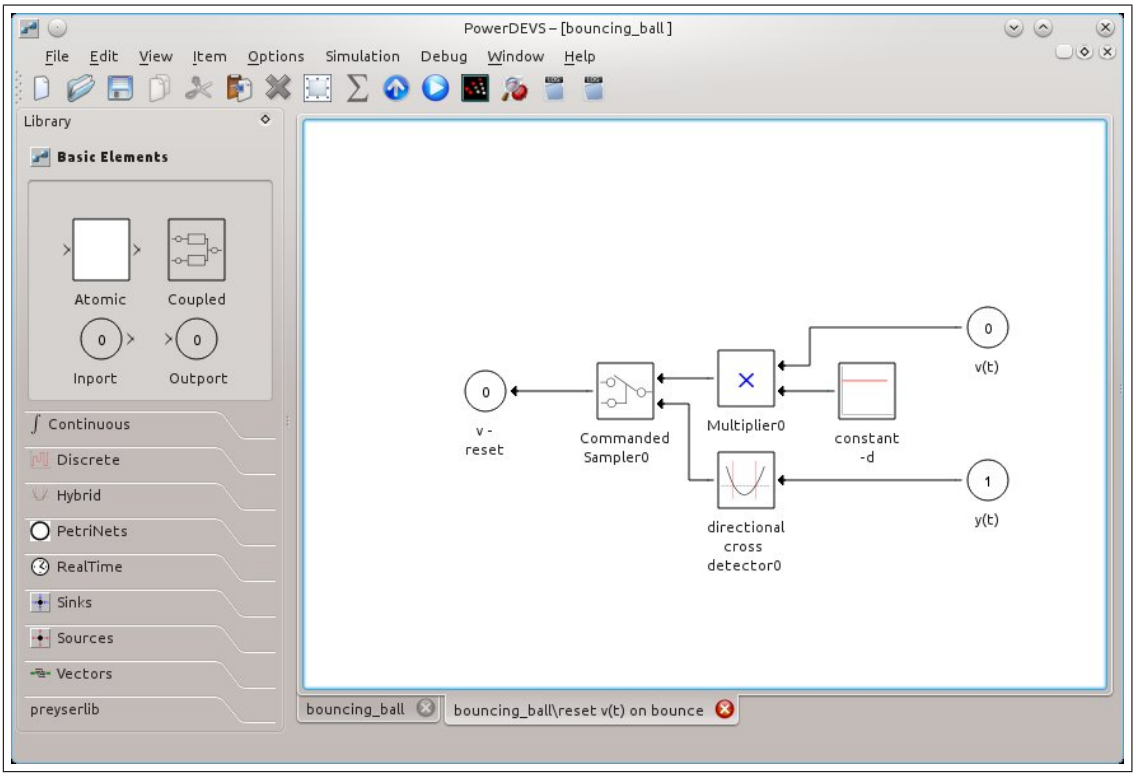


Figure 3.2: The coupled model ‘reset v(t) on bounce’ from the bouncing ball model depicted in Figure 3.1.

of events was triggered (‘Perform’ text field). This can be achieved by clicking onto the ‘Step(s)’ button instead of onto ‘Run Simulation’. With each further click on that button the simulation is continued until again the number of entered events were triggered.

PowerDEVS is especially designed to support real time simulation as well. This is what the button ‘Run Timed’ is for. It can be used to run the simulation in real time. That is, one second in simulation time corresponds to one second in real time. Of course, a more or less correct correspondence between simulation time and real time can only be assumed when working on a real time operating system.

In the bouncing ball example in Figure 3.1, there are two ‘From Workspace’ blocks and one ‘To Workspace’ block. They are used to read the initial values for ball height `y0` and ball velocity `v0` from Scilab workspace and to write the simulation result, ball height `y(t)`, back into the Scilab workspace.

For the parameter value of the block ‘constant -d’ in Figure 3.2 the letter `d` is entered. This also causes PowerDEVS to search in Scilab workspace for a variable with the name `d` (`d` is the damping factor the bouncing ball velocity is multiplied with on each bounce). Figure 3.5 shows the graphical user interface of Scilab, how the initial values and parameters used in the bouncing ball example are entered and a Scilab plot of the simulation results that were written back into Scilab workspace.

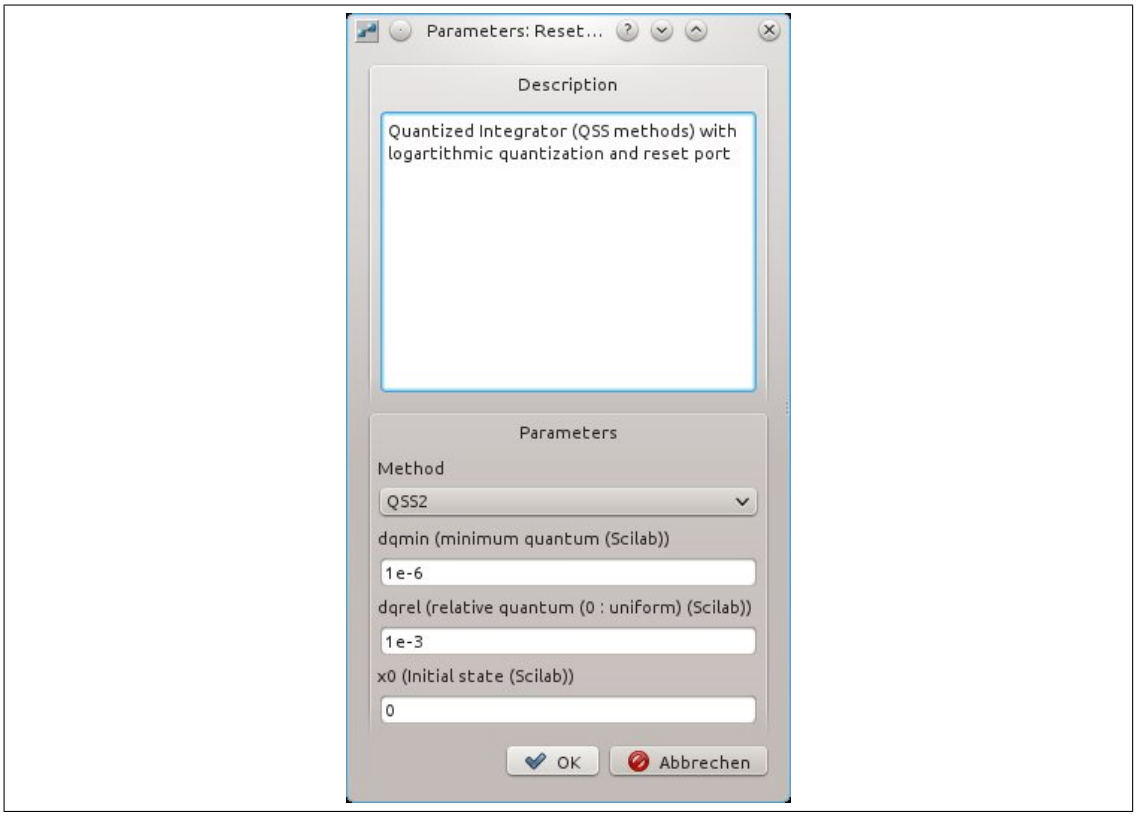


Figure 3.3: The dialogue that is opened when double clicking on a PowerDEVS non-coupled block, in this case an integrator block. Here block parameter values can be entered.

The select function for coupled models is realized as priority list in PowerDEVS. Figure 3.6 shows the bouncing ball model with its priority list. Every time two blocks in a coupled model experience an internal event at the same time, the execution of the internal events is conducted in the order according to the priority list. Since execution of an internal event may result in an output message of the corresponding block and thus result in an input message at another block, a change in the priority list definitely is capable of changing the models behaviour.

For example, if the two concurrent blocks supply the same receiver block with input messages $x1$ and $x2$, then the internal state of the receiving block may depend on the order in which $x1$ and $x2$ arrive. In case of $x1$ arriving before $x2$, the internal state s of the receiving block will be calculated as $s = \delta_{ext}(\delta_{ext}(s, x1), x2)$, whereas it will be calculated as $s = \delta_{ext}(\delta_{ext}(s, x2), x1)$ if $x2$ arrives before $x1$. Therefore, the definition of the priority list is a very important part of the modelling process of a coupled DEVS model. Maybe this is even the most difficult part of it, because it can become quite extensive to consider all possible combinations of concurrent signals in the model and how they should be resolved. This is exactly one point which will be dealt with in section 4.1.

The library opened in Figure 3.6 is named 'Vector' library and provides blocks working with vectorial signals, i.e. the coupling lines between blocks from that library transport vectorial

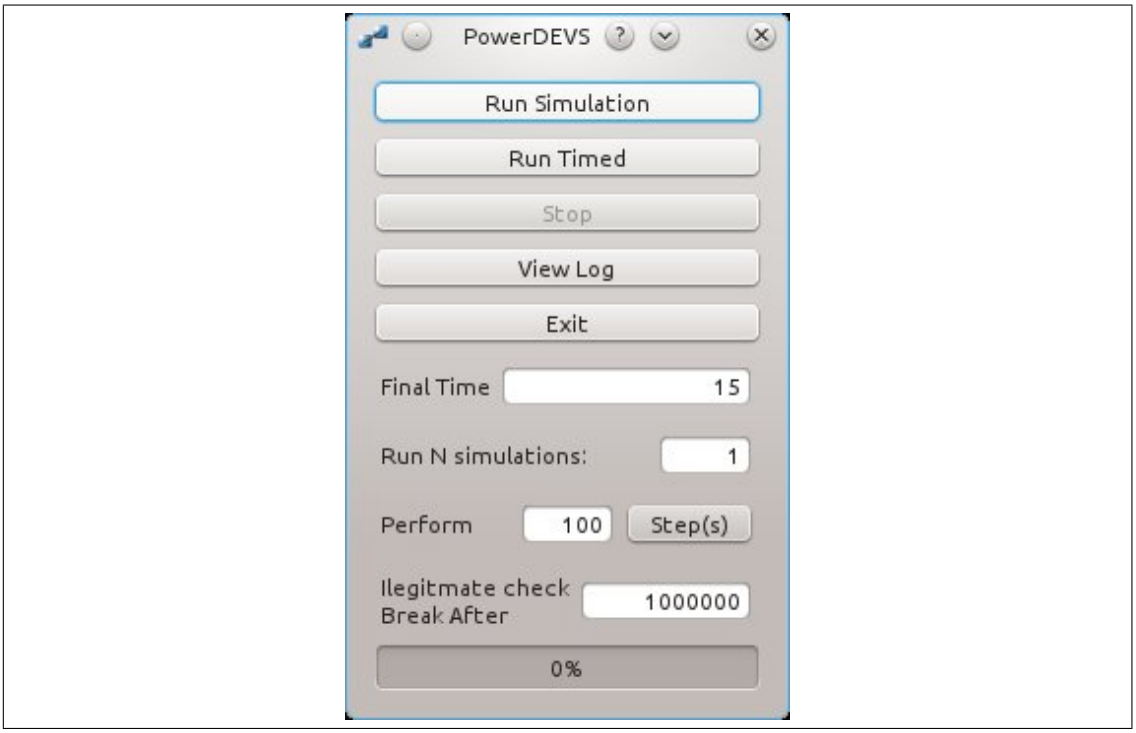


Figure 3.4: The dialogue that is opened when starting the simulation via clicking onto the blue play symbol in the model editors tool bar. Here simulation parameters can be entered and the simulation can be started.

values. However, in the concrete implementation again only scalar messages are sent on those lines, but each equipped with an attribute called *index* that defines the index of the vectorial value this scalar value belongs to.

When speaking about scalar values in combination with continuous signals in PowerDEVS, actually n-tuples are meant, with n corresponding to the order of QSS method in use (QSSn). That is, if there is a message sent at time t consisting of a n-tuple and an index value of i, this means that the Taylor polynomial of degree $n-1$, with expansion point t of the i-th index of the vectorial signal on that line, has the coefficients given with the n-tuple.

Thus, single indices of vectorial signals can change independently and at different instances of time and therefore, each change of an index produces an external event at the receiving block. So vectorial signals in PowerDEVS are perfectly suited for producing concurrent input events, e.g. when several or all indices of a vectorial signal change at the same time. However, those concurrent input events have to lead to the same final state of the receiving block, no matter in which order they are processed, which makes the creation of consistently working, custom atomic blocks quite hard.

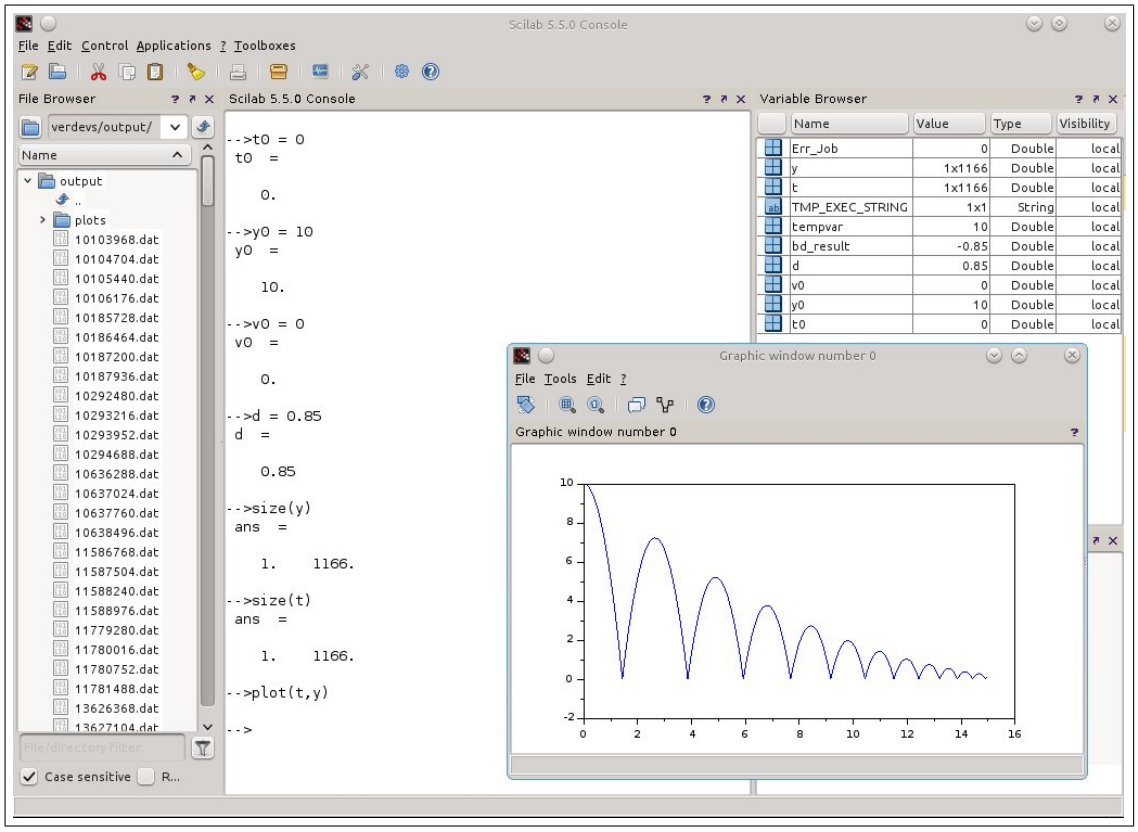


Figure 3.5: The Scilab user interface with the *Console* in the middle an the currently defined workspace variables in the *Variable Browser* top right. The first four commands in the Console define initial time `t0`, initial height `y0`, initial velocity `v0` and the damping constant of the bouncing ball model depicted in Figure 3.1. The bouncing ball model writes its simulation results, consisting of ball heights `y` at specific time instances `t` back to the Scilab workspace. Bottom right, the result of the Scilab plot command can be seen.

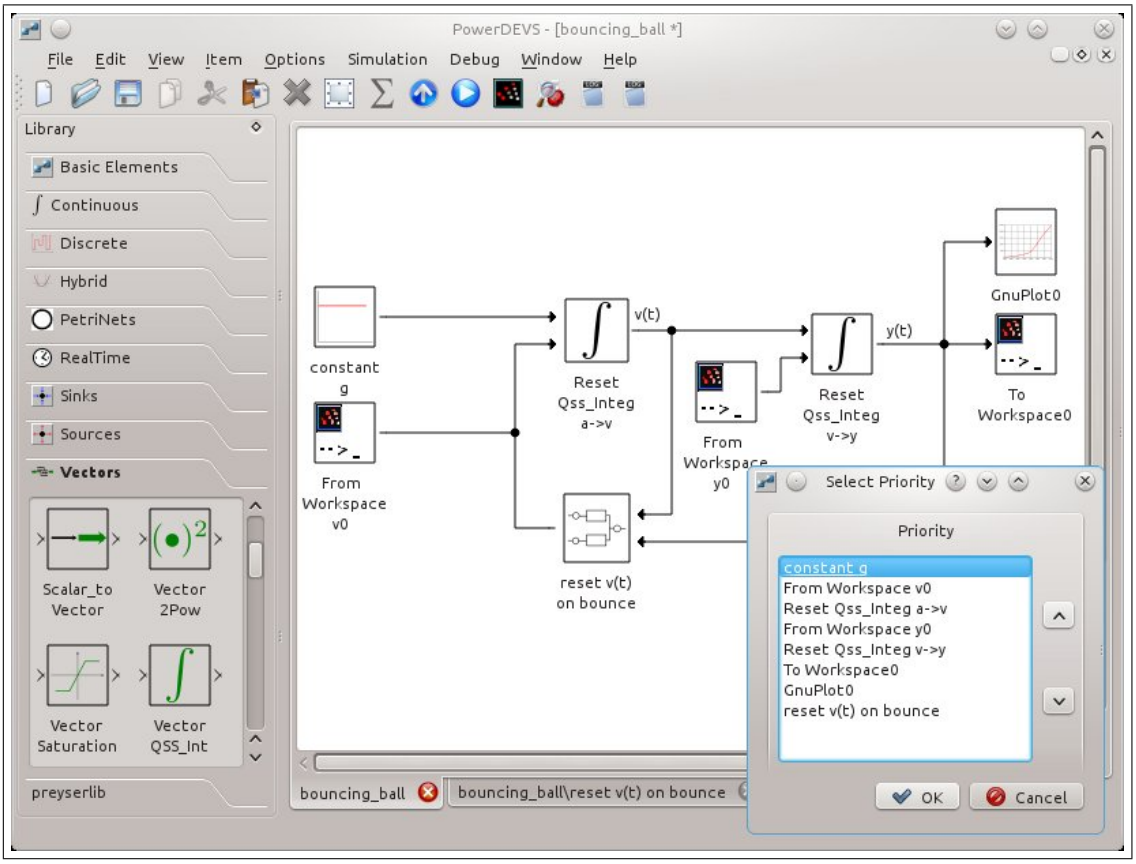


Figure 3.6: The PowerDEVS model editor with the priority list and the 'Vectors' library opened.

3.2 Atomic Editor

As well as all other mentioned modules of PowerDEVS (model editor, structure generator and preprocessor), the atomic editor is a stand-alone program. However, it can be started from the model editor by right clicking onto a model block and selecting 'Edit Code'. Though this works only as long as the chosen block already has an assigned source code file pair consisting of a `.h` and a `.cpp` file. Otherwise, when creating a completely new block starting with the 'Atomic' block from the 'Basic Elements' library, new source files need to be created. Therefor the atomic editor has to be used. This can be done from the model editor by right clicking onto the block, selecting 'Edit...', then switching to the 'Code' tab and clicking onto the button 'new file', which opens the atomic editor with blank tabs as depicted in Figures 3.7 and 3.8. The block edit dialogue is depicted in Figure 3.9. After saving the newly created code files, they still need to be linked to the atomic block in the model editor it was started with. This again can be achieved in the 'Code' tab of the block edit dialogue. There the source code, which describes the functionality of the block, needs to be selected (see Figure 3.9c).

The block edit dialogue also provides the means to set the desired number of block input ports and output ports, to enter a block description, and to select a block icon (see Figure 3.9a) as well as to define the block parameters (see Figure 3.9b). As block icons, .svg files can be

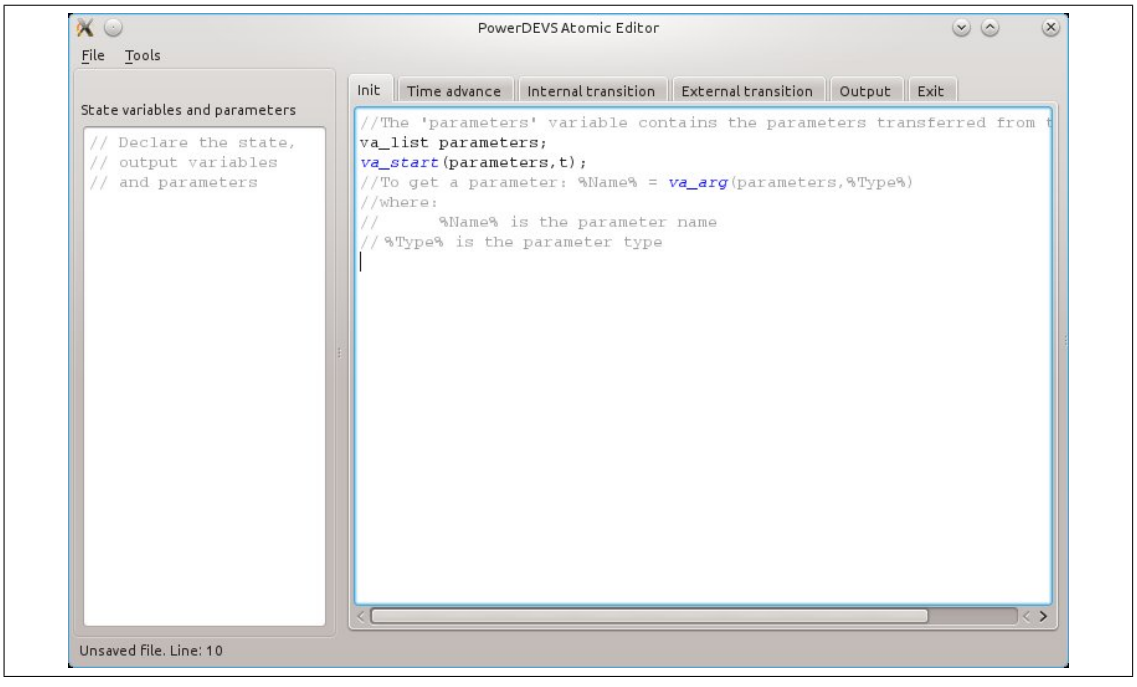
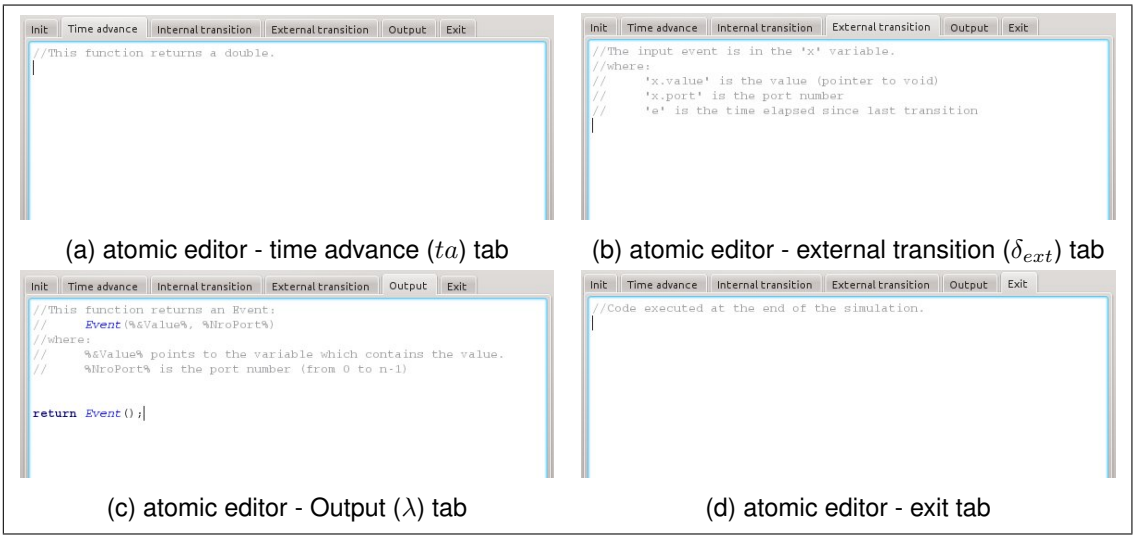


Figure 3.7: PowerDEVS atomic editor. It is used to define new model blocks by formulating their DEVS in C++. The C++ code describing τ, δ_{int}, δ_{ext} and λ as well as an `init` and an `exit` routine is entered in the appropriate tab of the atomic editor.



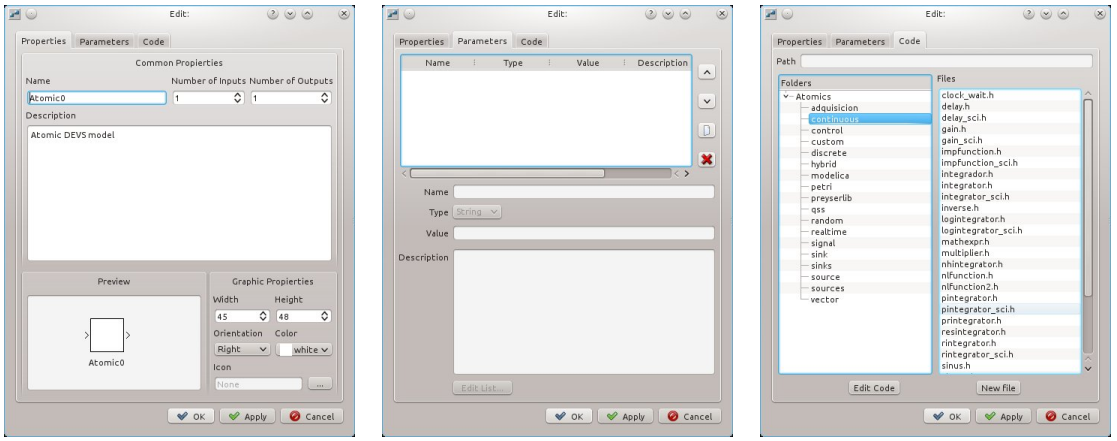
(a) atomic editor - time advance (ta) tab

(b) atomic editor - external transition (δ_{ext}) tab

(c) atomic editor - Output (λ) tab

(d) atomic editor - exit tab

Figure 3.8: The picture shows four of the six tabs of the atomic editor which need to be filled out by the modeller. The missing ones are the init tab and the one for defining the internal transition. The init tab is shown in Figure 3.7. The internal transition tab is initially completely blank and therefore not depicted.



(a) (b) (c)

Figure 3.9: The 'Edit' dialogue of a model block in the model editor. (a) shows the properties tab, (b) the parameters tab and (c) the code tab.

used, which for example can be created using the open source tool *LaTeXDraw*[6].

Figure 3.7 shows the atomic editor with completely new source code files opened. As already mentioned, the atomic editor is designed to directly enter a DEVS using C++ language. There are seven input areas:

[6] http://latexdraw.sourceforge.net/

- State variables and parameters
- Init
- Time advance (ta)
- Internal transition (δ_{int})
- External transition (δ_{ext})
- Ouput (λ)
- Exit

The simulation engine of PowerDEVS strictly sticks to the proposal in [**?**]. Therefore, for each atomic block in PowerDEVS, which corresponds to an atomic DEVS, there is a new C++ class defined, which is derived from the class `Simulator`. This `Simulator` base class dictates each derived class to overload the functions: `init`, `ta`, `dint`, `dext`, `lambda`, and `exit`, each of which corresponding to one of the tabs in the atomic editor.

Every coupled system in a PowerDEVS model is represented by an instance of the class `Coupling` that also is derived from `Simulator`. Thus, from outside, a `Coupling` instance looks like an atomic and therefore again can be part of another `Coupling` instance. On top of this hierarchical structure comes the class `RootCoupling`, derived from `Coupling`, and finally there is the class `RootSimulator` (see Figure 3.10). For further details it is referred to the source code itself which, as PowerDEVS is open source, is freely available on `sourceforge.net`.

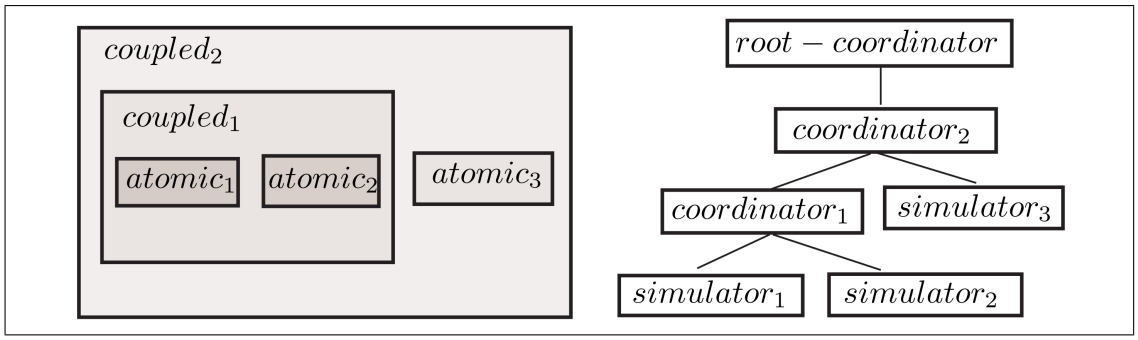


Figure 3.10: Illustration of the hierarchical structure of a PowerDEVS model and the corresponding class structure of the simulation model. 'coordinator' corresponds to the class `Coupling`. 'root–coordinator' corresponds to the classes `RootCoupling` and `RootSimulator`. (picture taken from [**?**])

Next, it will be explained shortly, how to use the single tabs depicted in Figure 3.7 and in Figure 3.8.

State variables and parameters. The C++ code written in this part of the atomic editor comes, as it is, into the section of the C++ class definition where member variables are defined. Thus, parameter variables, system state variables, and all kind of variables that need to be accessible from the other tabs (=member functions) have to be defined here.

Init. The C++ code inserted into the 'Init' tab is executed right before the simulation is started. Usually it is used to read and process parameter values, entered in the block edit dialogue (see 3.9b) and to store those values in variables defined in the 'State variables and parameters' section. Figure 3.7 shows the initial content of the 'Init' tab. The two lines of code there read the parameter values from the input mask and store them into the variable `parameters`. The other lines are comments which explain how to fetch a parameter value out of `parameters`. When retrieving the single parameter values form `parameters`, they are delivered in the order they appear in the block edit dialogue. For example, if the block has defined three parameters `a`, `b`, and `c`, then the first execution of `%Name% = va_arg(parameters,%Type%)` returns `a`, the second returns `b`, and the third returns `c`. To give an example, if the corresponding block has two parameters. The first one of type 'Value' and the second one of type 'String'. Then their values can be retrieved by typing:

```
a = (double)va_arg(parameters,double);
b = (char*)va_arg(parameters,char*);
```

where the variables `a` and `b` need to be defined either in 'State variable and parameters' or in 'Init' before those two lines.

Time advance. The C++ code inserted here is used to calculate the live time of the current state. Thus, the last line in this tab should be

```
return(sigma);
```

with `sigma` being a non-negative double value and giving the time to the next internal event provided that no external event is triggered for that time.

Internal transition. Here the behaviour of the block in case of an internal event has to be programmed. That is, the new internal state needs to be calculated depending on the old state.

External transition. Here the behaviour of the block in case of an external event has to be programmed. That is, the new internal state needs to be calculated depending on the old state and the currently received input message. This input message is stored in the variable `x` which is an instance of the C++ class `Event`. This class has an integer member variable `port` and a member variable `value` of type `void*`. Thus, `x.port` stores the input port where the message arrived. `x.value` points to the message itself which can be of arbitrary type. However, this presumes that the receiver always knows the exact type the messages it receives.

Output. Here the functionality of the output function λ has to be implemented. Thus, an object of type `Event` needs to be created and returned. The class `Event` has a constructor expecting any kind of pointer as first argument and an integer defining the output port as second argument. The pointer is exactly the `void` pointer which will be received in 'External transition' of the receiving block.

Exit. This is the place to deposit code that should be executed when the simulation is finished. For example, if there was memory allocated in this block during simulation or in 'Init', this is the right place to free it.

3.3 Structure Generator and Preprocessor

The graphical model created with the model editor is stored in a .pdm file. It stores block names, block positions in the model, the routing between the blocks, and already the hierarchical structure. Further, the parameter types and values as well as the assigned code file paths for each block are stored. The Structure Generator then reduces this .pdm file to a .pds file that only stores hierarchical model structure as well as for each block its source file path and the parameter list `Parameters` and the necessary coupling information. Listing 3.1 shows the .pds file for the bouncing ball example depicted in Figure 3.1.

After the Structure Generator has created the .pds file, the Preprocessor uses that information to create a file named 'model.h' and a makefile named 'Makefile.include', both located in the folder 'powerdevs/build/'. In 'model.h' a class named `Model` is defined that is derived form `RootSimulator` and that represents the whole model. The makefile is used to create a stand alone executable by linking the object files (stored in 'powerdevs/build/objs/') that were compiled from source files needed in the model. The executable is named 'model' and is located in 'powerdevs/output/'. Also located in the folder 'powerdevs/output/' are two log files named 'compiled.log' and 'pdevs.log'. As the name suggests, the first one stores the compiler feedback from creating the object files and the executable and the second one stores log entries that were written during simulation using the command `printLog(C-String, ...)`. `printLog` has exactly the same interface as the well known standard C library command `printf` with the exception that the first argument (file pointer) is not needed.

The executable 'model' can then be executed completely independent from PowerDEVS. However, if somewhere in the model, a block makes use of Scilab, a running Scilab session is needed and as already mentioned before, most library blocks make use of Scilab. Anyway, as long as there is no communication with Scilab taking place during the simulation, the executable can be run as stand alone program.

```
Root-Coordinator
 {
  Simulator
   {
    Path = sources\constant_sci.h
    Parameters = "-9.81"
   }
  Simulator
   {
    Path = preyserlib/fromworkspace.h
```

```
    Parameters = "t0","v0","QSS"
   }
  Simulator
   {
    Path = qss/res_qss_integrator.h
    Parameters = "QSS3","1e-6","1e-3","0"
   }
  Simulator
   {
    Path = preyserlib/fromworkspace.h
    Parameters = "t0","y0","QSS"
   }
  Simulator
   {
    Path = qss/res_qss_integrator.h
    Parameters = "QSS3","1e-6","1e-3","0"
   }
  Simulator
   {
    Path = sinks/toscilab_offline.h
    Parameters = "t","y"
   }
  Simulator
   {
    Path = sinks/gnuplot.h
    Parameters = 1.000000e+00,"set xrange [0:%tf] @ set grid @ set title 'bouncing ball'","with
        lines title 'ball height y(t)'","","","",""
   }
    Coordinator
     {
      Simulator
       {
        Path = sources\constant_sci.h
        Parameters = "-d"
       }
      Simulator
       {
        Path = qss/qss_multiplier.h
        Parameters = "Purely static","1e-6","1e-3"
       }
      Simulator
       {
        Path = preyserlib/directionalcrossdetect.h
        Parameters = "0","1","-1"
       }
      Simulator
       {
        Path = qss/command_sampler.h
        Parameters =
       }
      Simulator
       {
        Path = qss/command_sampler.h
        Parameters =
       }
      Simulator
       {
        Path = sources\constant_sci.h
        Parameters = "0"
       }
      EIC
       {
        (0,0);(1,0)
        (0,1);(2,0)
       }
      EOC
       {
        (3,0);(0,0)
       }
      IC
       {
        (1,0);(3,0)
        (0,0);(1,1)
```

```
        (5,0);(4,0)
        (2,0);(4,1)
        (2,0);(3,1)
      }
    }
  Simulator
   {
    Path = qss/samplehold.h
    Parameters = "0.001","0"
   }
  EIC
   {
   }
  EOC
   {
   }
  IC
   {
    (0,0);(2,0)
    (3,0);(4,1)
    (8,0);(6,0)
    (2,0);(7,0)
    (2,0);(4,0)
    (1,0);(2,1)
    (7,0);(2,1)
    (4,0);(7,1)
    (4,0);(8,0)
    (4,0);(5,0)
   }
 }
```

Listing 3.1: The .pds file of the bouncing ball model depicted in Figure 3.1.

3.4 Simulation Process

As already mentioned in section 3.2, the atomic editor creates a C++ class from the code entered into the tabs, which is derived from the predefined class `Simulator`. This class `Simulator` has, among others, the member variables `t`, `tn`, `tl`, and `e`. Thus, those member variables are accessible in each member function of the derived class and therefore in each of the tabs 'Init', 'Time advance', 'Internal transition', 'External transition', 'Output', and 'Exit'. In the following instead of 'Time advance', 'Internal transition', 'External transition', and 'Output' also the terms ta, δ_{int}, δ_{ext}, and λ, denominating their DEVS meanings will be used.
`t`, `tn`, `tl`, and `e` have the following functions:

- `t` always gives the actual simulation time.
- `e` stores the time since the last time δ_{int} or δ_{ext} has been called. Attention, the value for `e` is not updated until right before the next call of one of those two functions.
- `tl` gives the time of the last execution of δ_{int} or δ_{ext} and is updated right after each call of δ_{int} or δ_{ext}.
- `tn` gives the time when the next internal transition will be called. Right after the execution of δ_{int} or δ_{ext}, ta is called and after that call, `tn` is updated.

So the execution order in case of an internal event is as follows

1. call of λ
2. recalculation of `e`
3. call of δ_{int}
4. `tl := t`
5. call of ta
6. `tn := t + ta(.)`

and in case of an external event it is as follows

1. recalculation of `e`
2. call of δ_{ext}
3. `tl := t`
4. call of ta
5. `tn := t + ta(.)`

Finally some concrete cases of coupled systems simulation will be worked through to demonstrate the execution order of δ_{int}, δ_{ext}, and λ of the involved blocks. Thus, we first introduce the terms *mealy type* and *moore type* in connection with DEVS.

Definition 3.4.1. *Mealy Type DEVS*
A DEVS is called Mealy Type DEVS *or* of Type Mealy*, if there exists an internal state s and an external input x in such a way that $ta(\delta_{ext}(s, x)) = 0$. Thus, a model described by a mealy type DEVS may produce an output as immediate response to an input.*

Definition 3.4.2. *Moore Type DEVS*
A DEVS is called Moore Type DEVS *or* of Type Moore*, if it is not of type mealy.*

Since a DEVS is either moore type or mealy type and DEVS is closed under coupling, from the perspective of a PowerDEVS block, the rest of the model surrounding it is again a DEVS of type moore or of type mealy. Thus, there are actually only four possible cases. Either the block itself is moore and the surrounding system is also moore, or the surrounding system is mealy, or the block is mealy and the surrounding system is moore, or block and surounding system are mealy. The latter case results in an algebraic loop and therefore potentially in an illegitimate model and therefore is not investigated further.

If only all possible coupling combinations are of interest, the second and third cases are the same. Thus, in the following the first and the second case will be investigated (see Figures 3.11 and 3.12).

3.4.1 Moore - Moore Example

First a moore - moore coupling like depicted in Figure 3.11 will be examined. The states of the left system are denoted with s_k^1 and the states of the right system are denoted with s_k^2. Further, x_k are the input messages of the left system and y_k are the input messages of the right system ($k \in \mathbb{N}$). It is started with initial states s_0^1 and s_0^2.

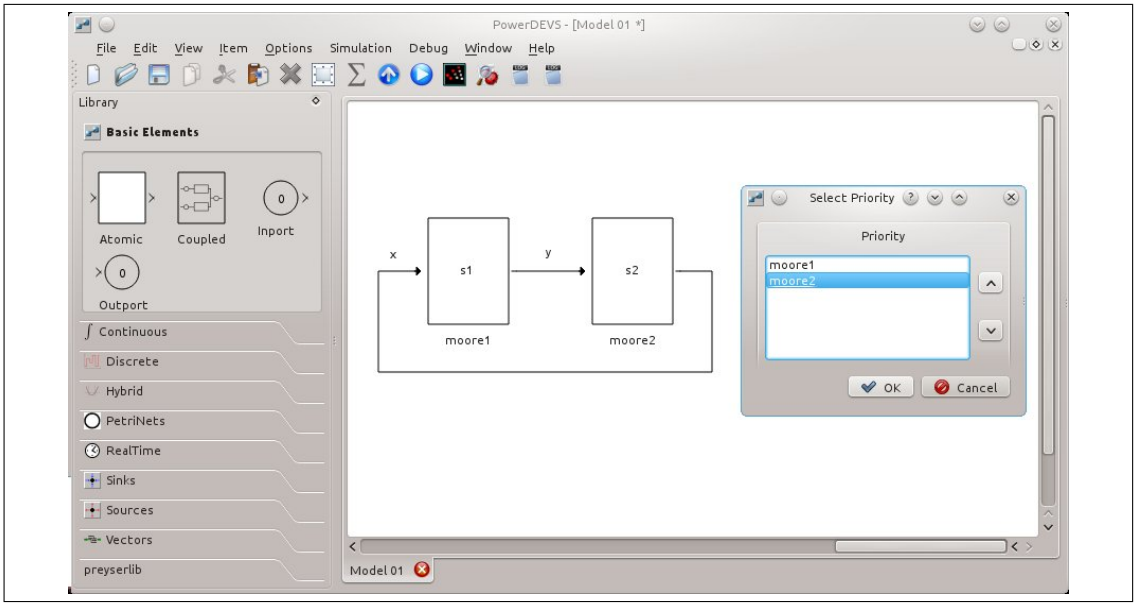


Figure 3.11: PowerDEVS simulation example: a moore block coupled with a moore type surrounding system.

In Table 3.1 the sequence of different function calls is listed, assuming a simulation run where *moore1* has internal events at $t = 1$ and $t = 2$ and *moore2* has an internal event at $t = 2$. Thus, both moore blocks have internal events at $t = 2$. It can be seen that already in this simple example the final states of both systems depends on the priority order. Even more, it depends on the priority order if δ_{int} at $t = 2$ is executed at all.

t	moore1, moore2	moore2, moore1
0	$init^1()$ $init^2()$ $ta^1(s_0^1)$ $ta^2(s_0^2)$	$init^2()$ $init^1()$ $ta^2(s_0^2)$ $ta^1(s_0^1)$
1	$y_1 := \lambda^1(s_0^1)$ $s_1^2 := \delta_{ext}^2(s_0^2, y_1)$ $ta^2(s_1^2)$ $s_1^1 := \delta_{int}^1(s_0^1)$ $ta^1(s_1^1)$	$y_1 := \lambda^1(s_0^1)$ $s_1^2 := \delta_{ext}^2(s_0^2, y_1)$ $ta^2(s_1^2)$ $s_1^1 := \delta_{int}^1(s_0^1)$ $ta^1(s_1^1)$
final states	$s_1^1 = \delta_{int}^1(s_0^1)$ $s_1^2 = \delta_{ext}^2(s_0^2, \lambda^1(s_0^1))$	$s_1^1 = \delta_{int}^1(s_0^1)$ $s_1^2 = \delta_{ext}^2(s_0^2, \lambda^1(s_0^1))$
2	$y_2 := \lambda^1(s_1^1)$ $s_2^2 := \delta_{ext}^2(s_1^2, y_2)$ $ta^2(s_2^2)$ $s_2^1 := \delta_{int}^1(s_1^1)$ $ta^1(s_2^1)$	$x_1 := \lambda^2(s_1^2)$ $s_2^1 := \delta_{ext}^1(s_1^1, x_1)$ $ta^1(s_2^1)$ $s_2^2 := \delta_{int}^2(s_1^2)$ $ta^2(s_2^2)$
final states	$s_2^1 = \delta_{int}^1(s_1^1)$ $s_2^2 = \delta_{ext}^2(s_1^2, \lambda^1(s_1^1))$	$s_2^1 = \delta_{ext}^1(s_1^1, \lambda^2(s_1^2))$ $s_2^2 = \delta_{int}^2(s_1^2)$
3	$exit^1()$ $exit^2()$	$exit^2()$ $exit^1()$

Table 3.1: The behaviour of a moore - moore coupling with internal events of *moore1* at $t = 1$ and $t = 2$ and an internal event of *moore2* at $t = 2$. Simulation end time is 3. In the left column *moore1* has a higher priority than *moore2*. In the right column the priority of *moore2* is higher.

3.4.2 Moore - Mealy Example

Now a moore - mealy coupling like depicted in Figure 3.12 will be examined. The states of the left system are denoted with s_k^1 and the states of the right system are denoted with s_k^2. Further, x_k are the input messages of the left system and y_k are the input messages of the right system ($k \in \mathbb{N}$). It is started with initial states s_0^1 and s_0^2.

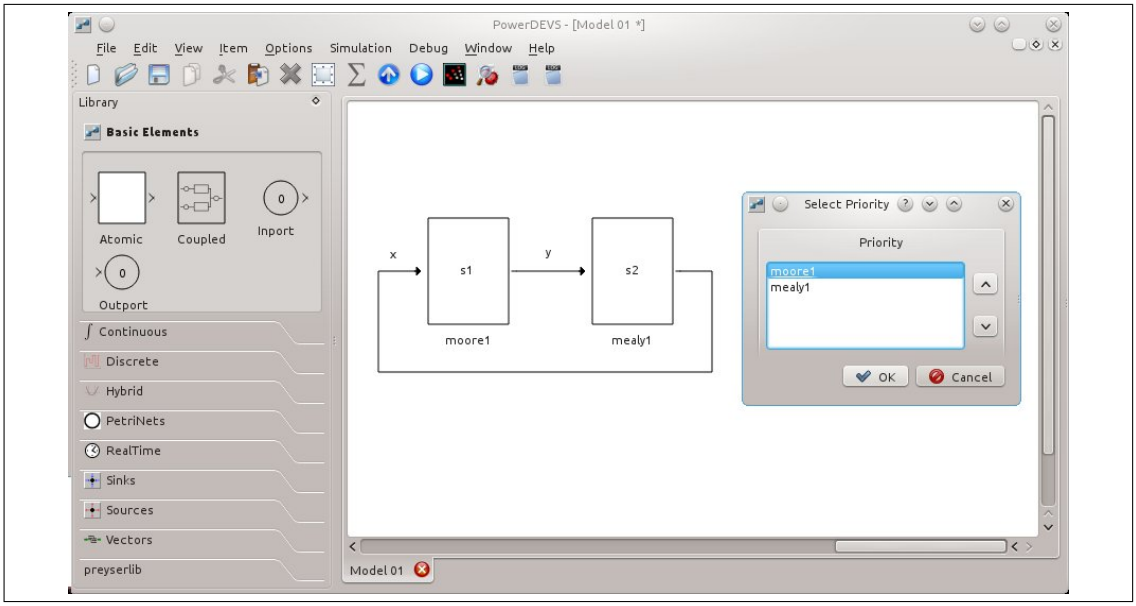


Figure 3.12: PowerDEVS simulation example: a moore block coupled with a mealy type surrounding system (or the other way round).

In Table 3.2 the sequence of different function calls is listed, assuming a simulation run where *moore1* has internal events at $t = 1$ and $t = 2$ and *mealy1* has an internal event at $t = 2$. Thus, both blocks have internal events at $t = 2$. Again, it can be seen that the final states of both systems depend on the priority order.

It turns out to be quite difficult to program a library block, i.e. a block that behaves always in the same way, no matter how the surrounding system looks like, and independently from its place in the priority list.

t	moore1, mealy1	mealy1, moore1
0	$init^1()$ $init^2()$ $ta^1(s_0^1)$ $ta^2(s_0^2)$	$init^2()$ $init^1()$ $ta^2(s_0^2)$ $ta^1(s_0^1)$
1	$y_1 := \lambda^1(s_0^1)$ $s_1^2 := \delta_{ext}^2(s_0^2, y_1)$ $ta^2(s_1^2)\ (= 0)$ $s_1^1 := \delta_{int}^1(s_0^1)$ $ta^1(s_1^1)\ (> 0)$ $x_1 := \lambda^2(s_1^2)$ $s_2^1 := \delta_{ext}^1(s_1^1, x_1)$ $ta^1(s_2^1)\ (> 0)$ $s_2^2 := \delta_{int}^2(s_1^2)$ $ta^2(s_2^2)\ (> 0)$	$y_1 := \lambda^1(s_0^1)$ $s_1^2 := \delta_{ext}^2(s_0^2, y_1)$ $ta^2(s_1^2)\ (= 0)$ $s_1^1 := \delta_{int}^1(s_0^1)$ $ta^1(s_1^1)\ (> 0)$ $x_1 := \lambda^2(s_1^2)$ $s_2^1 := \delta_{ext}^1(s_1^1, x_1)$ $ta^1(s_2^1)\ (> 0)$ $s_2^2 := \delta_{int}^2(s_1^2)$ $ta^2(s_2^2)\ (> 0)$
final states	$s_2^1 = \delta_{ext}^1(\delta_{int}^1(s_0^1), \lambda^2(\delta_{ext}^2(s_0^2, \lambda^1(s_0^1))))$ $s_2^2 = \delta_{int}^2(\delta_{ext}^2(s_0^2, \lambda^1(s_0^1)))$	$s_2^1 = \delta_{ext}^1(\delta_{int}^1(s_0^1), \lambda^2(\delta_{ext}^2(s_0^2, \lambda^1(s_0^1))))$ $s_2^2 = \delta_{int}^2(\delta_{ext}^2(s_0^2, \lambda^1(s_0^1)))$
2	$y_2 := \lambda^1(s_2^1)$ $s_3^2 := \delta_{ext}^2(s_2^2, y_2)$ $ta^2(s_3^2)\ (= 0)$ $s_3^1 := \delta_{int}^1(s_2^1)$ $ta^1(s_3^1)\ (> 0)$ $x_2 := \lambda^2(s_3^2)$ $s_4^1 := \delta_{ext}^1(s_3^1, x_2)$ $ta^1(s_4^1)\ (> 0)$ $s_4^2 := \delta_{int}^2(s_3^2)$ $ta^2(s_4^2)\ (> 0)$	$x_2 := \lambda^2(s_2^2)$ $s_3^1 := \delta_{ext}^1(s_2^1, x_2)$ $ta^1(s_3^1)\ (> 0)$ $s_3^2 := \delta_{int}^2(s_2^2)$ $ta^2(s_3^2)\ (> 0)$
final states	$s_4^1 = \delta_{ext}^1(\delta_{int}^1(s_2^1), \lambda^2(\delta_{ext}^2(s_2^2, \lambda^1(s_2^1))))$ $s_4^2 = \delta_{int}^2(\delta_{ext}^2(s_2^2, \lambda^1(s_2^1)))$	$s_3^1 = \delta_{ext}^1(s_2^1, \lambda^2(s_2^2))$ $s_3^2 = \delta_{int}^2(s_2^2)$
3	$exit^1()$ $exit^2()$	$exit^2()$ $exit^1()$

Table 3.2: The behaviour of a moore - mealy coupling with internal events of *moore1* at $t = 1$ and $t = 2$ and an internal event of *mealy1* at $t = 2$. Simulation end time is 3. In the left column *moore1* has a higher priority than *mealy1*. In the right column the priority of *mealy1* is higher.

CHAPTER 4

Problems, Shortcomings and Solution Approaches

As already mentioned in chapter 3, in PowerDEVS it can become quite unhandy to build a coupled model with exactly the intended behaviour. Concurrent events and *zero time feedbacks* are introducing difficulties, especially when trying to design a universally applicable library block, i.e. a block whose surrounding system and position in the priority list is not known in advance.

The solution approach of Parallel DEVS (P-DEVS) is to locate the concurrency resolution not at the global level but locally at each single block. The approach presented in this section will make use of that idea. For simulation of P-DEVS though, a specific P-DEVS simulation engine is necessary, but PowerDEVS works with a DEVS simulation engine. Therefore, P-DEVS cannot be implemented in PowerDEVS directly.

Another point is the way PowerDEVS passes on messages between blocks. Although it is the most flexible way one can think of, it is also some kind of uncomfortable when reusability of programmed blocks is intended. Since the type of messages passed on is completely arbitrary, interconnection between blocks from different libraries or between blocks that were programmed by different developers is quite error-prone.

Although PowerDEVS is particularly designed for simulating hybrid models, there exists no atomic DEV&DESS block in the PowerDEVS library. Therefore, when having a formal DEV&DESS of a model, so far there is no way to directly implement it in PowerDEVS.

In the following sections a general concept is developed of how to tackle the problems connected with programming an 'Atomic' DEVS in PowerDEVS and thereby handling concurrent input signals. Furthermore, a more specialized but also more secure and more comfortable (concerning reusability) way to exchange messages between PowerDEVS blocks is searched for. The development of this general concept happens stepwise and finally results in an extended PowerDEVS atomic DEVS block (named *Atomic PDEVS*) that should relieve the modeller of concerns about concurrency handling. That is, this Atomic PDEVS block can be used as a starting point for programming custom PowerDEVS blocks with the advantage that the modeller

does not have to think about all the possible orders in which concurrent input messages may be processed.

The Atomic PDEVS block is introduced in three steps. In each step a concrete problem is identified followed by a solution approach which is finally incorporated into the Atomic PDEVS block. For better understanding all three steps will be accompanied with a practical example that displays all the problems that are intended to be solved.

Of course, a formal proof of the correctness of the Atomic PDEVS block would be needed. However, this is work that still needs to be done.

After the development of the Atomic PDEVS block, the problem of the message formats in PowerDEVS is tackled in section 4.2.

Finally, in section 5 a way to implement a DEV&DESS directly in PowerDEVS is presented. For this purpose an *Atomic DEV&DESS* PowerDEVS block is developed which is based on the Atomic PDEVS block.

4.1 Concurrent Input Signals and Consistency Problems

From the point of view of a PowerDEVS block, there are two possible sources for input signals that can be distinguished. The first is an internal transition of another block. The second is a zero time feedback of one of the output ports of the block itself. I.e. one of the output ports of the block is coupled to a DEVS (atomic or coupled) of type mealy, which again has an output port that is coupled back to an input port of the first block. If at the same simulation time arrive more than one input messages at a DEVS block, despite their concurrency, there exists a defined order in which those messages are processed. This order is dictated by the `Select` function or by the priority list in PowerDEVS.

As already indicated in section 3.1, when designing a DEVS, an unconsidered processing order of concurrent input messages can lead to an unintended evolution of the state of the system. But not only the internal state of a PowerDEVS block can easily evolve in an unintended way due to concurrent input messages but also its output messages. An example is given by a mealy type block with initial inner state s_0 and two concurrent input messages x_1 and x_2. First, it is assumed that the blocks producing x_1 and x_2 have higher priority than the mealy block, leading to an evolution of the inner state as follows:

$$\begin{aligned} s_1 &= \delta_{ext}(s_0, x_1) \text{ with } ta(s_1) = 0 \\ s_2 &= \delta_{ext}(s_1, x_2) \text{ with } ta(s_2) > 0 \end{aligned}$$

Thus, the mealy block does not produce any output message at all. Now it is assumed that the mealy block has a higher priority than the block producing x_2, resulting in the following sequence of transitions:

$$\begin{aligned} s_1 &= \delta_{ext}(s_0, x_1) \text{ with } ta(s_1) = 0 \\ &\qquad \lambda(s_1) \\ s_2 &= \delta_{int}(s_1) \text{ with } ta(s_2) > 0 \\ s_3 &= \delta_{ext}(s_2, x_2) \text{ with } ta(s_3) > 0 \end{aligned}$$

Thus, whether a mealy block produces output messages or not also may depend on the priority list.

So, it turns out to be a quite challenging task to design a coupled DEVS model that shows the intended behaviour in every possible situation. In the following, systematically, concrete sources of the difficulties that appear are identified and mechanisms to counter them are developed. For a better understanding, in parallel a concrete example will accompany the process. Stepwise a more and more 'concurrency stable' DEVS for the concrete example is formulated which finally serves as template for defining arbitrary 'concurrency stable' DEVS.

4.1.1 Concurrent Input Signals

4.1.1.1 Problem Identification

When designing a new atomic PowerDEVS block with more than one input port or with a vectorial input port, it can become quite difficult to do this in a way that results in the intended block behaviour for all possible combinations of concurrent input messages. This is due to the fact, that δ_{ext} modifies the internal state on each reception of an input message and these modifications need not to be commutative. That is, $\delta_{ext}(\delta_{ext}(s, x_1), x_2)$ does not need to be equal to $\delta_{ext}(\delta_{ext}(s, x_2), x_1)$.

To concretise the problem the specific example depicted in Figure 4.1 will be investigated. The intended behaviour of the example is described in the caption of the figure. The block in

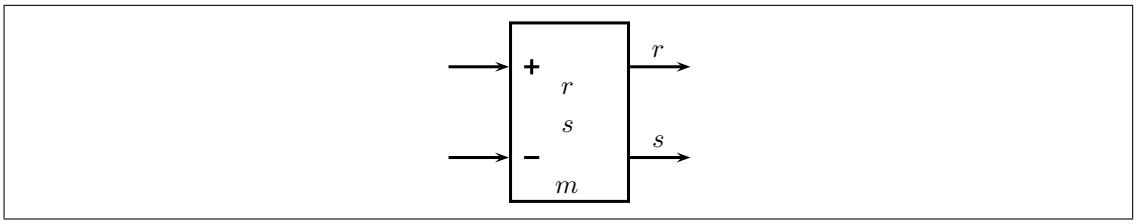


Figure 4.1: Illustration of an atomic DEVS example that accumulates positive and negative inputs in its inner state s and outputs the value of s every second. Whenever s becomes bigger than a given upper bound m, s is output at the reset output port and afterwards it is reset to zero.

the example will be referred to as *summation block*. In a first approach its formulation as DEVS looks like the following:

$$S = \left\{(r, s, \sigma) \mid r, s \in \mathbb{R},\ \sigma \in \mathbb{R}_0^+\right\}$$
$$X = \left\{(x, p) \mid x \in \mathbb{R}_0^+,\ p \in \{0, 1\}\right\},\ Y = \{(y, p) \mid y \in \mathbb{R},\ p \in \{0, 1\}\}$$

$$(r, s, \sigma) = \delta_{ext}(r, s, \mathbf{x}, e) = \begin{cases} (0, s + x, \sigma - e) & \text{if } \mathbf{x} = (x, 0) \ \wedge \ s + x < m \\ (s + x, 0, 0) & \text{if } \mathbf{x} = (x, 0) \ \wedge \ s + x \geq m \\ (0, s - x, \sigma - e) & \text{if } \mathbf{x} = (x, 1) \end{cases}$$

$$\lambda(r, s) = \begin{cases} (s, 1) & \text{if } r = 0 \\ (r, 0) & \text{if } r \neq 0 \end{cases}$$

$$(r, s, \sigma) = \delta_{int}(r, s) = \begin{cases} (0, s, 1) & \text{if } r = 0 \\ (0, 0, \sigma) & \text{if } r \neq 0 \end{cases}$$

$$ta(r, s, \sigma) = \sigma$$

At first glance, the definition looks quite promising, however, there is a special case in which the model may not behave as intended. As long as there is only one input message at a time, everything is fine but as soon as there arrive input messages both at input port 0 and at input port 1, troubles may arise.

To give an example, it is assumed that the initial state s is a little bit smaller than the upper bound m when there arrives an input value x_0 at input port 0 with $s + x_0 \geq m$. At the same time there also arrives an input value $x_1 > x_0$ at input port 1. Now, what the block is ought to do is to add $x_0 - x_1 < 0$ to the current state value s. If the block producing x_1 has higher priority than the block producing x_0 the DEVS works as intended. However, if the block producing x_0 has higher priority, x_0 is processed first, leading to a growth of s above the upper bound m and thus to a reset of the inner state. So, assuming that $s + x_0 - x_1 < m$, the two possible values for s after that instant of time are: $s + x_0 - x_1$ and $-x_1$. Further, in the second case an output at the reset port is produced that is not produced in the first case.

Moreover, there is another shortcoming in the DEVS stated above. The model actually is of type moore when only looking at output port 1, however, it is of type mealy when also looking at output port 0. Therefore, when there is an internal event (output of s at output port 1) at the same time as there is an external event that causes a reset (output of s at output port 0), it again depends on the priority list, whether there is an output only at port 0 or at both, port 0 and port 1. Actually the concrete behaviour in such a case is not defined at all in the caption of Figure 4.1. The possibilities are either to

- first output s at output port 1 and then $s + x_0$ at output port 0, before s is reset to zero, or
- to output $s + x_0$ at output port 0, then to reset s to zero and then to output zero at output port 1, or
- or to output $s + x_0$ at both output ports before s is reset to zero.

It can be seen that it is quite easy to formulate a complete DEVS for a system, since with the definition of δ_{ext}, λ, δ_{int} and ta for the described model the behaviour in every situation imaginable is determined. However, this does not mean that this behaviour is the one the modeller intended. Actually, exactly because the modeller does not have to define the behaviour for all the special situations explicitly, it is very likely to overlook one and therefore to produce a faulty DEVS.

4.1.1.2 Solution Approach.

The source of the problem pointed out above is the fact that concurrent input messages are not treated concurrently but sequentially. The concrete order of treatment depends on the `Select` function or on the priority list in case of PowerDEVS. However, the priority list is a property of the whole coupled model and therefore it is difficult to take it into account when designing an atomic block.

So the solution is to somehow buffer concurrent input messages, comparable to bags in P-DEVS, and treat them all at once. The idea followed here is to create an input buffer for each input port and for each index of vectorial input ports and to use δ_{ext} only for storing the received input message in the corresponding buffer. The state modifications are shifted into the internal transition. Thus after each external transition the DEVS switches to a *transitory state* (a state s with $ta(s) = 0$) resulting in an execution of δ_{int} in which the state changes related to the external event are accomplished. For this purpose some kind of flag f is needed to indicate whether the call of δ_{int} is due to an external event or due to an internal event or due to both. The latter case also exists in P-DEVS and is called *confluent transition* δ_{conf}. So in the first solution approach presented in the following, in δ_{ext} only input messages are buffered and in δ_{int} the actual jobs of δ_{ext}, δ_{int} and δ_{conf} are implemented.

Now, with the modifications described above and given that all DEVS that supply other blocks with input messages have lower priority than the blocks they supply, every block receiving messages will buffer all them before it executes its internal transition. In the internal transition then, all received messages are treated at once and therefore, there is no processing order of concurrent messages to be considered by the modeller. Furthermore, the distinction between internal, external *and* confluent transition forces the modeller to think about the behaviour of the DEVS in case of a simultaneous occurrence of an internal and an external event. Therefore, the suggested modification solves the problem of concurrent input messages in a coupled model, as long as there are no feedbacks in it. This is because no feedbacks imply linear signal flow and therefore, all blocks can be arranged in the priority list according to that linear signal flow path. This again results in a coupled model, in which all blocks supplying other blocks with input messages have lower priority than the blocks they supply.

The modifications applied to the summation block example are leading to the following modified DEVS:

$$S = \{(r, s, x_0, t_0, x_1, t_1, f, \sigma, \sigma_n) \mid r, s \in \mathbb{R},\ x_0, x_1 \in \mathbb{R}_0^+,\ f \in \{INT, EXT, CONF\},\ t_0, t_1, \sigma, \sigma_n \in \mathbb{R}_0^+\}$$

$$X = \left\{(x, p) \mid x \in \mathbb{R}_0^+,\ p \in \{0, 1\}\right\},\ Y = \{(y, p) \mid y \in \mathbb{R},\ p \in \{0, 1\}\}$$

$$\mathbf{q} = (\tilde{\mathbf{q}}, e) = (r, s, x_0, t_0, x_1, t_1, f, \sigma, \sigma_n, e)$$

$$\delta_{ext}(\mathbf{q}, \mathbf{x}) = \begin{cases} (r, s, x, t, x_1, t_1, CONF, 0, \sigma - e) & \text{if } \mathbf{x} = (x, 0) \ \wedge \ f = INT \ \wedge \ \sigma = e \\ (r, s, x, t, x_1, t_1, EXT, 0, \sigma - e) & \text{if } \mathbf{x} = (x, 0) \ \wedge \ f = INT \ \wedge \ \sigma > e \\ (r, s, x_0, t_0, x, t, CONF, 0, \sigma - e) & \text{if } \mathbf{x} = (x, 1) \ \wedge \ f = INT \ \wedge \ \sigma = e \\ (r, s, x_0, t_0, x, t, EXT, 0, \sigma - e) & \text{if } \mathbf{x} = (x, 1) \ \wedge \ f = INT \ \wedge \ \sigma > e \\ (r, s, x, t, x_1, t_1, f, 0, \sigma_n) & \text{if } \mathbf{x} = (x, 0) \ \wedge \ f \neq INT \\ (r, s, x_0, t_0, x, t, f, 0, \sigma_n) & \text{if } \mathbf{x} = (x, 1) \ \wedge \ f \neq INT \end{cases}$$

$$\lambda(\tilde{\mathbf{q}}) = \begin{cases} (s, 1) & \text{if } (f = CONF \ \vee \ f = INT) \ \wedge \ r = 0 \\ (r, 0) & \text{if } r \neq 0 \\ \emptyset & \text{else} \end{cases}$$

$$\delta_{int}(\tilde{\mathbf{q}}) = \begin{cases} (0, s, x_0, t_0, x_1, t_1, INT, 1, 1) & \text{if } f = INT \\ (0, s_n, x_0, t_0, x_1, t_1, INT, \sigma_n, \sigma_n) & \text{if } f = EXT \ \wedge \ s_n < m \\ (s_n, 0, x_0, t_0, x_1, t_1, INT, 0, \sigma_n) & \text{if } f = EXT \ \wedge \ s_n \geq m \\ (0, s_n, x_0, t_0, x_1, t_1, INT, 1, 1) & \text{if } f = CONF \ \wedge \ s_n < m \\ (s_n, 0, x_0, t_0, x_1, t_1, INT, 0, 1) & \text{if } f = CONF \ \wedge \ s_n \geq m \\ (0, s, x_0, t_0, x_1, t_1, f, \sigma_n, \sigma_n) & \text{if } r \neq 0 \end{cases}$$
$$\text{with } s_n = s + \mathbb{1}_{\{0\}}(t - t_0) \cdot x_0 - \mathbb{1}_{\{0\}}(t - t_1) \cdot x_1$$

$$ta(\tilde{\mathbf{q}}) = \sigma$$

where the pair (x_i, t_i) stores the input at input port i that was received at time t_i. $\mathbb{1}_{\{0\}}(z)$ denotes the indicator function being one for $z = 0$ and zero otherwise. s_n denotes the new value of s after adding and subtracting newly received inputs. σ_n stores the calculated duration until the next internal event is triggered. This duration cannot always be assigned to σ immediately since there may be the need for an additional call of λ or of the internal transition function before time is allowed to advance. This occurs for example after an external transition that leads to a transitional state, as it is the case when the summation block is reset.

The modified DEVS looks quite complicated at first glance. However, the modeller only has to define δ_{int}, whereas δ_{ext} is the same for every model and can easily be extended to more inputs than two. When looking at δ_{int}, the three cases internal, external and confluent transition can be seen.

Since all inputs are treated at once, the modeller is forced to think about how the model shall behave when several concurrent input messages arrive. Moreover, the order of arrival of concurrent input messages is not important anymore as long as they arrive before the internal transition is executed. The more input ports (and indices of vectorial input ports) there are, the higher is the number of possible arrival orders which do not have to be considered anymore.

One important difference to P-DEVS is that for each input port and index only the last arriving message of all is stored in the input buffer, whereas with P-DEVS all arriving messages are stored in bag. So here 'newer' messages override 'older' ones, whereas in P-DEVS they are added into the bag as additional input messages.

A DEVS of a block becomes quite extensive when adding more and more features. This is due to the fact that even when only one entry of the vectorial inner state needs to be changed in δ_{ext} or in δ_{int} the whole new state must be written. Additionally, distinction of cases is also more space expensive than with programming languages, for in programming languages nested case distinctions are possible. For this reason, from now on the DEVS of the summation block example and of its extensions will be stated in C++ programming language in the way it would be entered into PowerDEVS atomic editor.

4.1.1.3 PowerDEVS Implementation.

As explained in section 3.2 there are 6 areas in the atomic editor which need to be filled with C++ source Code. The 'State variables and parameters' area is used for variable definitions. It will be referred to as *definitions area*. The 'Init' tab serves for variable initialisation and for fetching block parameter values that have been entered using the block parameters dialogue in the model editor. It will be referred to as *init function*. The 'Time advance' tab represents $ta(s)$ and will be referred to as *time advance function*. The 'Internal transition' tab, representing δ_{int}, will be referred to as *internal transition function*. Note that in the following it will be distinguished between δ_{int} and the internal transition function. Since the actual, state modifying functionality of δ_{ext}, δ_{int}, and δ_{conf} is located in the internal transition function, with δ_{ext}, δ_{int}, and δ_{conf} it will be referred to the section of code in the internal transition function that implements the corresponding δ-function. These sections of code can be identified by an enclosing `if` condition of the form `if(flag=='e')`, `if(flag=='i')` and `if(flag=='c')` respectively. The 'External transition' tab originally represents δ_{ext} but is only used for buffering input messages in the solution approach developed here. It will be referred to as *external transition function*, not to be confused with δ_{ext} which describes the corresponding code section in the internal transition function. The 'Output' tab represents λ, however, in the course of the next sections its functionality will be extended a little bit. Thus, the actual duty of λ, the calculation of the output messages, will correspond to only a section of code in the 'Output' tab. The 'Output' tab will be referred to as *output function*. The last tab is the 'Exit' tab which will be referred to as *exit function*.

```
1 double in_array[2]; // input messages
2 double t_in_array[2]; // arrival times of input messages
3 double sigma, sigma_n;
4 double m, r, s;
5 char flag;          // possible values: 'i','e','c'
```

Listing 4.1: The PowerDEVS definitions area for the summation block example.

Listings 4.1–4.6 show the C++ code for each of the 6 tabs, describing the modified DEVS of the summation block example. At first, all necessary state variables, parameter variables, and

auxiliary variables have to be defined in the definitions area of the atomic editor (see Listing 4.1).

Further, some variables need to be initialized (Listing 4.2). `t` is the current simulation time that is available in every tab of the atomic editor. The arrival times of the messages in the input buffer are initialized with `t-1` because initially there is no valid entry in the input buffer and if the arrival time of an input is set to a value that is lower than `t` then the input is considered old and is not used.

The upper bound `m` is defined as PowerDEVS block parameter and thus read from the `parameters`-list (line 5) that is handed on to the init function as function argument. `sigma` is initialised with `1` and thus at simulation time `t=1` the sum `s` is output for the first time. The value of `flag` is initialised with `'i'`. This is because when an internal event occurs, first the output function is executed which does not change `flag`, and then the internal transition function is executed where `flag` needs to have value `'i'`. On the other hand, when an external event occurs first, then in the external transition function `flag` will be set to `'e'` or `'c'` and afterwards again the output function followed by the internal transition function is executed.

```
1 t_in_array[0] = t-1;
2 t_in_array[1] = t-1;
3 sigma = 1;
4 sigma_n = 1
5 m = (double)va_arg(parameters,double); // a model/block parameter
6 r = 0;
7 s = 0;
8 flag = 'i'
```

Listing 4.2: The PowerDEVS init function for the summation block example.

The actual calculation of the time advance function $ta(s)$ is done in the internal and external transition functions and its result is stored in the state variables `sigma` and `sigma_n` respectively. Thus, the only code in the time advance function is a return statement with `sigma` as argument (see Listing 4.3).

```
1 return(sigma);
```

Listing 4.3: The PowerDEVS time advance function for the summation block example.

The calculation of the next state value in case of any event is done in the internal transition function depicted in Listing 4.4. There are four major cases to be distinguished in this concrete example. First there are the three cases in which the internal transition function is executed immediately after an internal or after an external event. In these three cases `r` is equal to zero and `flag` has one of the values `'i'`, `'e'` or `'c'`. Depending on the value of `flag` the next state value is computed by δ_{int}, by δ_{ext} or by δ_{conf}. If thereby a reset is carried out, `sigma` stays zero and `r` is set to `s`. As consequence, the output function is executed once more to output `r` at the reset output port and then also the internal transition function is executed again. However, this time `r` is not equal to zero and therefore, the `if` condition in line 1 is fulfilled,

which represents the fourth case. In the fourth case `r` is set back to zero and `sigma` is set to value `sigma_n` that has been calculated formerly.

```
1  if(r!=0) {
2    r = 0;
3    sigma = sigma_n;
4  } else if(flag=='i') {
5    sigma = 1;
6  } else if(flag=='e' || flag=='c') {
7    if(flag=='c') {
8      sigma_n = 1;
9    }
10   if(t_in_array[0]==t) {
11     s = s + in_array[0];
12   }
13   if(t_in_array[1]==t) {
14     s = s - in_array[0];
15   }
16   if(s>=m) {
17     r = s;
18     s = 0;
19   } else {
20     sigma = sigma_n;
21   }
22   flag = 'i';
23 }
```

Listing 4.4: The PowerDEVS internal transition function for the summation block example.

```
1  in_array[x.port] = *(double*)x.value;
2  t_in_array[x.port] = t;
3  if(tl<t) {
4    sigma_n = sigma - e;
5    if(t+sigma_n==t) {
6      flag = 'c';
7    } else {
8      flag = 'e';
9    }
10 }
11 sigma = 0;
```

Listing 4.5: The PowerDEVS external transition function for the summation block example.

Listing 4.5 shows the external transition function. In the first two lines the received input message and the current time are stored in the input buffers. `tl` gives the time at which the previous event happened and is provided by PowerDEVS. If this is the first time the external transition function is executed at current simulation time, `tl` is lower than `t` and therefore, `e` is not zero. Thus, the time to the next internal event is the old duration `sigma` reduced by `e`, i.e. `sigma_n = sigma-e`. However, if `t+sigma_n` numerically cannot be distinguished from

the actual simulation time `t` (i.e. `sigma_n` is smaller than the current quantum of the floating point quantisation), the system is concurrently experiencing an internal event, hence, a confluent event and therefore `flag` is set to `'c'`. Otherwise it is a pure external event. At the end of the external transition, `sigma` is set to zero since each external transition leads to a transitional state, as explained in section 4.1.1.3.

Listing 4.6 shows the output function. In case of `flag=='e'` the output function does not do anything. This is because first δ_{ext} has to be calculated in the internal transition function before a possible mealy output is produced.

```
1 if(flag=='c' || flag=='i') {
2   if(r==0) {
3     return Event(&s,1);
4   } else {
5     return Event(&r,0);
6   }
7 }
8 return Event();
```

Listing 4.6: The PowerDEVS output function for the summation block example.

4.1.2 Feedback, Zero Time Feedback

4.1.2.1 Problem Identification

As *feedback* we denote a coupling from a block back to another block that is located 'upstream' the signal flow compared to the first block. If the coupled DEVS, consisting of all blocks located between output and input of the block that receives feedback, is of type mealy, it is spoken of *zero time feedback*. The special thing about a zero time feedback is that the block that receives the feedback needs to execute λ and therefore also δ_{int} before it receives the input message resulting from the feedback. Therefore, it is not possible to gather all input messages before treating them all at once in the internal transition function because the execution of the same is required in the first place to produce some of the input messages.

Further, this problem can not only occur with zero time feedbacks but also with ordinary feedbacks. If incidentally at the same time at which other external events happen, there occurs an internal transition in a block where a feedback input signal originates, exactly the same problem occurs. This problem cannot be resolved by modifying the priority list. Figure 4.2 shows the example from the preceding subsection extended in way that introduces the mentioned feedback problems. The simple multiplier block that is added is of type mealy and therefore causes a zero time feedback.

4.1.2.2 Solution Approach

The solution approach is not to let state changes become effective until all messages in the whole model and at current simulation time are produced, i.e. not until time actually advances. For this purpose, the whole internal state is backed up before the internal transition is allowed to change

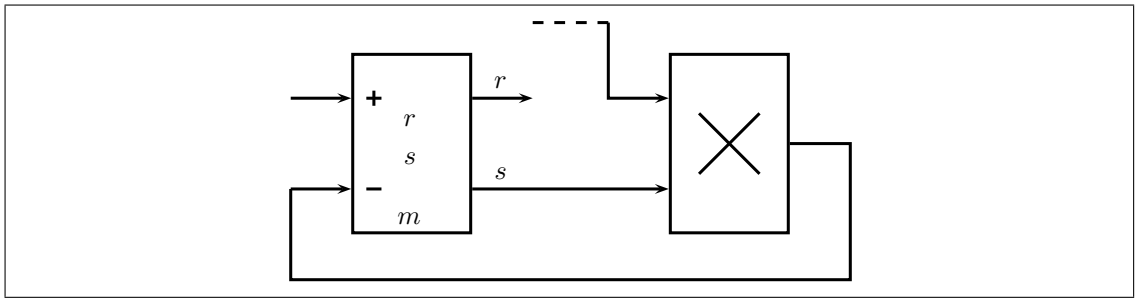


Figure 4.2: The example depicted in Figure 4.1 extended by a multiplier block that feeds back its output to the summation block, resulting in a *zero time feedback*.

it. The new state and the outputs are then always calculated on basis of the backup state. That is, when the state is denoted s and the backup is denoted s_{old}, the internal transition and λ look like the following:

$$s = \delta_{int}(s_{old}, \mathbf{x})$$
$$y = \lambda(s_{old}, \mathbf{x})$$

Here, both δ_{int} and λ depend not only on the internal state s, but also on the current input $\mathbf{x}$. This is because it is continued with the solution approach of the preceding section where the external transition function stores every input in the input buffer which, in turn, is part of the internal state. Therefore, the definition of δ_{int} and λ as input dependant is not contradictory to the definition of DEVS.

Anyway, if the set of current input messages $\mathbf{x}$ now changes due to a feedback after the internal state s has already been calculated, s simply is discarded and recalculated. The recalculation again uses s_{old}, which did not change, and $\mathbf{x}$ which is extended by the input message produced by the feedback.

So, in the approach described above, if an input messages arrives after the internal transition has already been executed, simply the whole internal transition is executed again. This leads to an iterative process, that is repeated as long as there arrive new messages at the input:

1. execution of the external transition function and storage of received inputs as long as blocks with higher priority (upstream the signal flow) produce input messages.

2. execution of the output function, on basis of the old state s_{old} and of the inputs received so far and stored in $\mathbf{x}$.

3. execution of the internal transition function, where δ_{int}, δ_{ext} or δ_{conf} calculates the new state.

4. if, due to produced outputs and zero time feedbacks or due to ordinary feedbacks, new input messages arrive, go back to 1.

Now the question arises, when to back up the system's state. The answer is, it is backed up before the internal state is modified for the first time at the current instant of time. The only place where the actual internal state (not counting input buffer or the like) is modified is the internal transition. Thus, the state must be backed up before the internal transition is executed for the first time (at current simulation time). To recognize whether it is the first time, the variable `tl`, provided by the PowerDEVS simulation engine, can be used. `tl` stores the time when the last event of the corresponding block happened and is updated right after each external and internal transition.

Since in the solution approach developed so far each external transition immediately leads to the execution of the internal transition, the old state has to be backed up either in the external transition or at the beginning of the internal transition, depending on what is executed first. The condition for being the first event at a time `t` is `tl<t`.

As the output function is called right before each internal transition, the state backup can be located there as well instead of in the internal transition. Actually this is even the better choice because in λ, which is part of the output function, already the state backup s_{old} is needed. Of course, this is not conform with the DEVS formalism since in DEVS there is no mechanism for λ to change the state, which s_{old} is part of. Anyway, although very laboriously, it also would be possible to implement a distinction of cases in λ, to use s in case of `tl<t` and to use s_{old} in case of `tl=t`. Thus, the backup of the old state could be shifted into the internal transition function as well. However, this actually shows that there is no conflict between doing the state backup in the output function and staying DEVS conform.

Another issue that must be discussed appears when more than one output message at different output ports or at the same output port, but with different indices, is to be produced. As in PowerDEVS the output of a message is accomplished by a return statement there can only be output one message per call of the output function which, in turn, is always followed by a call of the internal transition function. However, each sending of an output message, due to a zero time feedback, may immediately result in a new input message and therefore, in a recalculation of the inner state. Thus, unnecessary state calculation may be saved, when first outputting all produced messages, before any state changes are accomplished in the internal transition function.

So, the idea is to buffer all output messages in the same way, input messages are buffered, and then to repeat the loop 'output function – internal transition function'(in the following referred to as *output loop*) as long as there are *pending outputs*. Pending outputs are outputs that were produced and buffered at current simulation time, but not sent so far. During this loop, in the internal transition function simply nothing is done and in each call of the output function one pending output message is sent. If there arrive new input messages during the execution of the output loop, they are served immediately by an execution of the output function, however they do not disturb the execution of the output loop. If, during the output loop, the set of current input messages $\mathbf{x}$ has been altered though, $\lambda(s, \mathbf{x})$ is calculated again and the output loop is restarted. This process is repeated until the set of input messages $\mathbf{x}$ does not change anymore. Only then, the new state s is calculated in the internal transition function by applying δ_{int}, δ_{ext} or δ_{conf} onto s_{old} and the set of input messages $\mathbf{x}$. However, even if a new state has already been calculated, it still is possible that the set of current input messages changes due to an input coming from a block with lower priority. Anyway, in response simply $\lambda(s, \mathbf{x})$ is calculated again followed by a

repetition of the output loop and a recalculation of s.

Therefore, the iterative process running in each atomic block, each time an event occurs, can be described more detailed as follows:

1. **external transition function:** (skipped on pure internal events) If the received message differs from the last reception with same arrival time at the same port and index, add it to $\mathbf{x}$ and set σ to zero.

2. **time advance function:** If there is a new input message, go back to 1. Otherwise, if $\sigma = 0$, go to 3. Otherwise: finished.

3. **output function:** If there is no pending output message and $\mathbf{x}$ changed, calculate output messages on basis of the old state s_{old} and the current inputs $\mathbf{x}$: $\mathbf{y} = \lambda(s_{old}, \mathbf{x})$.

4. **output function:** If there is a pending output message, output one of them.

5. **internal transition function:** If there are no pending output messages and no newly received input messages, calculate new state s and new value for σ by computing $\delta_{int}(s_{old})$, $\delta_{ext}(s_{old}, \mathbf{x})$ or $\delta_{conf}(s_{old}, \mathbf{x})$. Otherwise, do nothing.

6. **internal transition function:** Go back to 2.

4.1.2.3 PowerDEVS Implementation

Listings 4.9–4.13 show the DEVS for the summation block, based on the sources in section 4.1.1.3 and extended according to the solution approach for feedback signals suggested in section 4.1.2.2. In this solution approach each output port needs a kind of flag describing the current output status to be able to determine, after the calculation of λ, whether there is an pending message at that port or not. Additionally, there has to be a mechanism to find out whether there is any pending message at any output port. Also it is necessary for each input port to store the time of the last change of the input messages, buffered at that port. As it will emerge in section 4.1.3.3, it also will become necessary to store the time of last change for each output port as well.

Thus, to simplify the handling of inputs and outputs, the C++ classes `InOutput` and `InOutputVector` are introduced in Listings 4.7 and 4.8.

```
1  class InOutput {
2    public:
3      double value;
4      double last_change_time;
5      bool already_treated;
6
7      InOutput() {
8        last_change_time = -1;
9        already_treated = false;
10     }
11     void set(double val,double t) {
12       value = val;
```

```
13        last_change_time = t;
14        already_treated = false;
15      }
16      double* treat() {
17        already_treated = true;
18        return(&value);
19      }
20      bool isEqualTo(double val,double t) {
21        if(last_change_time!=t) return(false);
22        return(value==val);
23      }
24  };
```

Listing 4.7: Defnition of the class `InOutput` which is used to handle inputs and outputs, their last change time and, in case of outputs, their sent status.

The class `InOutput` represents one input port or one output port. Each port has a last received `value`, a time of last change `last_change_time` and a flag that indicates whether the last value change has been already treated. Since in PowerDEVS the simulation start time is always zero, `last_change_time` is initialized with `-1`. This prevents the corresponding input / output port to be wrongly regarded as having received a message at `t=0`, in case of an event at the very beginning of the simulation. `already_treated` is initialized with `false` to also prevent the corresponding input / output port to be initially wrongly regarded as having stored an untreated message. When using the methods `set` and `get` to modify the `value` attribute, `last_change_time` and `already_treated` are modified automatically accordingly.

Due to repetitions of the output loop of some blocks, at other blocks the same input messages may arrive several times. This would lead to unnecessary additional external events at those blocks which, in turn, would unnecessarily restart the output loop there. To prevent such an behaviour, each arriving input message that has already been received at the same simulation time is ignored. To check whether an input message is equal to the message that is stored in the input buffer, the member function `isEqualTo` is used.

```
1  class InOutputVector:public std::vector<InOutput> {
2    public:
3      int untreated_entry_changes;
4
5      InOutputVector() {
6        untreated_entry_changes = 0;
7      }
8      void setAt(int c,double val,double t) {
9        if(true==(*this)[c].already_treated) {
10         untreated_entry_changes++;
11       }
12       (*this)[c].set(val,t);
13     }
14     double* treatAt(int c,double val) {
15       if(false==(*this)[c].already_treated) {
16         untreated_entry_changes--;
17       }
```

```
18        return((*this)[c].treat());
19      }
20      void treatAll() {
21        for(int i=0; i<this->size(); i++){
22          (*this)[i].already_treated = true;
23        }
24        untreated_entry_changes=0;
25      }
26  };
```

Listing 4.8: Defnition of the class `InOutputVector` which is used to handle a whole vector of inputs and outputs, their last change time and, in case of outputs, their sent status.

The class `InOutputVector` describes the whole input / output interface of a block, as for a better understanding at the moment it is assumed that there are only scalar input and output ports. In the final version though, presented in section 4.3, the input and output ports will be considered to be vectorial and therefore, `InOutputVector` will represent only one single port there. For the whole interface a class named `InOutputArray` will be introduced then.

`InOutputVector` is derived from the class `vector` of the standard C++ library. This generic class can be used to implement arrays of arbitrary type with dynamic size. Thus, a variable of type `vector` or of an derived type provides the `[.]` operator to access the entries of the underlying array, exactly like with ordinary arrays. Further, `InOutputVector` provides the methods `setAt` and `treatAt` to modify its entries. These methods simply call the `set` or `treat` method of the corresponding array entry and thereby, automatically update the attributes `last_change_time` and `already_treated` of that entry. Additionally, `InOutputVector` has the attribute `untreated_entry_changes` that indicates whether there is an entry in the array with attribute value `already_treated=false`. `untreated_entry_changes` is also updated automatically when using the methods `setAt` and `treatAt`.

```
1 InOutputVector in_array; // input message buffer
2 InOutputVector out_array; // output message buffer
3 double sigma, sigma_n;
4 double m, r, s;
5 double r_old, s_old;
6 char flag;          // possible values: 'i','e','c'
7 int outport;
```

Listing 4.9: The PowerDEVS definitions area for the summation block example with feedback handling.

Listing 4.9 shows the definitions area. Compared to Listing 4.1 it can be seen that the input buffer `in_array` is now an instance of the class `InOutputVector` and therefore, no additional array `t_in_array` for the arrival times is needed anymore. Further, also for the output ports a buffer `out_array` is defined now. Moreover, there are backup variables `s_old` and `r_old` for `s` and `r`.

```
1 in_array.resize(2);
2 out_array.resize(2);
3 sigma = 1;
4 sigma_n = 1
5 m=(double)va_arg(parameters,double); // a model/block parameter
6 r = 0;
7 s = 0;
8 outport = -1;
```

Listing 4.10: The PowerDEVS init function for the summation block example with feedback handling.

Listing 4.10 shows the init function. In lines 1 and 2 the size of the input and of the output interface is defined, i.e. the size of the input and of the output array. The initialisation of the arrival times of the input ports is done in the constructor of the class `InOutput` as for each entry of the input array the constructor is called either by the `resize` command or already at the definition of `in_array` by the constructor of the class `vector`. Since the code in the time advance function is exactly the same as in Listing 4.3, it is not depicted here again.

```
1 if(out_array.untreated_entry_changes==0 && in_array.untreated_entry_changes
    ==0) {
2   if(flag=='i') {
3     sigma_n = 1;
4   } else if(flag=='e' || flag=='c') {
5     if(flag=='c') {
6       sigma_n = 1;
7     }
8     if(in_array[0].last_change_time==t) {
9       s = s_old + in_array[0].value;
10    }
11    if(in_array[1].last_change_time==t) {
12      s = s_old - in_array[0].value;
13    }
14    if(s>=m) {
15      s = 0;
16    }
17  }
18  sigma = sigma_n;
19 }
```

Listing 4.11: The PowerDEVS internal transition function for the summation block example with feedback handling.

Listing 4.11 shows the internal transition function. If there is anything done at all during execution of the internal transition function depends on whether there are pending outputs or newly received inputs or not (see line 1). That is, the external transition function only calculates a new state value if the output loop has been completed without receiving new input messages in the meanwhile. The source code responsible for calculating a new state value does not differ much

from that in Listing 4.4. However, the calculation of the reset output is shifted into the output function here.

```
1  if(tl<t) { // if first event at current time
2    r_old = r;
3    s_old = s;
4    flag = 'e';
5    sigma_n = sigma - e;
6    if(sigma_n==0) {
7      flag = 'c';
8    }
9  }
10 if(false == in_array[x.port].isEqualTo(*(double*)x.value,t)) {
11   in_array.setAt(x.port,*(double*)x.value,t);
12   if(flag=='i') {
13     flag = 'c';
14   }
15   sigma = 0;
16 }
```

Listing 4.12: The PowerDEVS external transition function for the summation block example with feedback handling.

Listing 4.12 shows the external transition function. The first block of code, enclosed by the if statement `if(tl<t)`, is only executed when the first event at a concrete simulation time is an external one. In this case the backup of `s` and `r` is carried out and it is checked whether an internal event will be triggered too at the same time. If this is the case, `flag` is set to `'c'`, otherwise it is set to `'e'`.

The second block of code treats the arrived input message but only if it was not already received before. That is, if a message with equal value was already received at the same simulation time and at the same port, the newer equal one is discarded. If the first event at current simulation time has been an internal one, `flag` has the value `'i'` when the external transition function is entered for the first time. Therefore, it has to be altered to `c`. Further, `sigma` has to be set to zero when a new input message arrives, as the actual treatment of external events, i.e. the calculation of δ_{ext} is accomplished in the internal transition function.

```
1  if(tl<t) { // if first event at current time
2    r_old = r;
3    s_old = s;
4    flag = 'i';
5    sigma = 0;
6  }
7  if(out_array.untreated_entry_changes==0) {
8    if(flag=='i') { // create outputs if pure internal event
9      out_array.setAt(1,s_old,t); // output s
10   } else if(flag=='e' || flag=='c') { // create outputs if pure ext. event
11     double s_ = s_old;
12     if(flag=='c') {
13       out_array.setAt(1,s_old,t); // output s
```

```
14      }
15      if(t_in_array[0]==t) {
16        s_ = s_ + in_array[0].value;
17      }
18      if(t_in_array[1]==t) {
19        s_ = s_ - in_array[0].value;
20      }
21      if(s_ > m) {
22        out_array.setAt(0,s_,t); // output reset
23      }
24    }
25    in_array.treatAll();
26    outport = -1;
27  }
28  while(out_array.untreated_entry_changes>0) { // output next new entry
29    outport++;
30    if(false==out_array[outport].already_treated) {
31      return Event(out_array.treatAt(outport),outport);
32    }
33  }
34  return Event();
```

Listing 4.13: The PowerDEVS output function for the summation block example with feedback handling.

Listing 4.13 shows the output function. The first block of code (lines 1 to 6) is only executed when the first event at a concrete simulation time is an internal one. In this case the backup of `s` and `r` is carried out, `flag` is set to `'i'` and `sigma` is set to zero.

The second block of code (lines 7 to 27) calculates λ but only if there are not any pending outputs. As it can be seen, there are the three cases `flag=='e'`, `flag=='i'`, and `flag=='c'` that are distinguished. Therefore, it is possible to define λ for each of this cases separately. Thus, λ is also kind of separated into λ_{ext}, λ_{int} and λ_{conf}. λ_{int} is supposed to depend only on the old state s_{old} and not on the values in the input buffer. If λ_{ext} or λ_{conf} uses values of the input buffer for the calculation of the output messages, the resulting system is of type mealy. Otherwise it is of type moore.

To be able to monitor whether new input messages have arrived during the execution of the output loop, `in_array.untreated_entry_changes` is set to zero immediately after the calculation of λ. This is accomplished by executing `in_array.treatAll()` in line 25 which additionally sets the attribute `already_treated` to true for each entry of `in_array`. Thus, since `in_array.untreated_entry_changes` is incremented at each arrival of a new input message that changes `already_treated` of the corresponding input port to true, if `in_array.untreated_entry_changes` is still zero after the output loop is finished, it is known that there has not arrived any new input message in the meanwhile.

Since the execution of the second block of code starts the output loop, at its end `outport` is set to minus one.

The third block of code (lines 28 to 33), the while loop, carries out the outputting of the output messages that were calculated before in the second block of code. Since `outport` has been initialised with minus one and the incrementation of `outport` is conducted right at

the beginning of each while loop iteration, it is started at output port zero. The condition for the while loop to be entered is that there are still pending outputs left. An output is pending, if its flag `already_treated` has the value `false`. When using the method `setAt`, as done in lines 13 and 22, this flag is automatically set to `false` as well as the number of pending outputs, stored in `out_array.untreated_entry_changes`, is increased automatically then. In the same way, `already_treated` is set to `true` and `out_array.untreated_entry_changes` is decremented automatically when calling the method `treatAt`, as done in line 31.

Finally, when the last pending output message is sent, in the succeeding execution of the internal transition function, the new internal state value is calculated in one of the δ-functions and `sigma` is set to a positive value again. If afterwards there is no external transition arriving anymore at current simulation time, time will be advanced.

4.1.3 Message Retrieving

4.1.3.1 Problem Identification

Although the solution approach of section 4.1.2 already looks quite promising, there is still one problem concerning mealy type blocks. Considering a scenario in which a block of type mealy receives a new input message, after it has already executed its output loop and thereby has produced a set of output messages $\mathbf{y}_0$, it may occur that in the recalculation of the output loop a different set of output messages $\mathbf{y}_1$ is calculated. This is no problem as long as the set of output ports used in $\mathbf{y}_0$ is a subset of the set of output ports used in $\mathbf{y}_1$. However, if in the first output loop, a message has been sent on an output port on which no message is sent in the second output loop, then this message is not valid anymore and therefore has to be retrieved.

An example is given with the coupled system depicted in Figure 4.2:
When the summation block receives an input message x_0 at input port 0 at the same time an internal event is triggered, the old state s_{old} is output at output port 1. In the special case, when $s_{old}+x_0$ exceeds the upper bound m, a reset is carried out as well and thus, $s_{old}+x_0$ is output on output port 0. However, since output port 1 is the source of a zero time feedback, as consequence of the output at that port a new input x_1 will arrive at input port 1 (minus - input port). Now, if $s_{old}+x_0-x_1$ is smaller than the upper bound m, a reset will not take place anymore. Therefore, the former output on output port 0 was wrong and needs to be retrieved.

4.1.3.2 Solution Approach

The idea for countering this last problem is the following: the same way, the internal state of each block is backed up at the very first event of each instant of time, also the input buffer is backed up. However, the backup is only necessary when an old input value is to be overwritten for the first time at current simulation time. Additionally, all output messages are equipped with a retrieve-flag that can be set to true or false. Thus, the arrival of an input message with a retrieve-flag that is true leads to the restoring of the backup for the according input port and index and of the time of last change at that port. So, in the following re-execution of the output loop, it is as if this retrieved message has never been received.

To learn which output messages need to be retrieved, for each output port it is stored at which time a message was sent there the last time. After each recalculation of λ, it is checked if there has been calculated a new output message for each output port at which a message was sent during the former output loop. If there is a port for which no new output message has been calculated, the former sent message needs to be retrieved. Thus its retrieve-flag is set to true and it is added to the pending outputs.

Summing up, the working principle of an atomic block in case of an event can roughly be described by the following:

1. **external transition function:** (skipped on pure internal events) If the received message differs from the last reception with same arrival time at the same port and index, add it to $\mathbf{x}$ and set σ to zero.

2. **time advance function:** If there is a new input message, go back to 1. Otherwise, if $\sigma = 0$, go to 3. Otherwise: finished.

3. **output function:** If there is no pending output message and $\mathbf{x}$ changed, calculate output messages on basis of the old state s_{old} and the current inputs $\mathbf{x}$: $\mathbf{y} = \lambda(s_{old}, \mathbf{x})$. If there are output ports where, at current simulation time, a message has been sent from, but for which there is no message included in $\mathbf{y}$, set the retrieve flag of the old messages at those ports to $true$ and add them to $\mathbf{y}$.

4. **output function:** If there is a pending output message, output one of them.

5. **internal transition function:** If there are no pending output messages, calculate new state s and new value for σ by computing $\delta_{int}(s_{old})$, $\delta_{ext}(s_{old}, \mathbf{x})$ or $\delta_{conf}(s_{old}, \mathbf{x})$. Otherwise, do nothing.

6. **internal transition function:** Go back to 2.

There remains one special case to be considered regarding message retrieving. If there have been no other input messages than the one that is retrieved, it may occur that the whole external event which may have caused output messages and state changes already, needs to be withdrawn. When taking a closer look at that problem, there are two cases to be distinguished. In the first case also an internal event is triggered at the same time. This case is the less complicated one. The event simply has to be altered from a confluent to a pure internal one (by changing the value of the event type indicating flag from $'c'$ to $'i'$). Afterwards the output loop has to be repeated.

In the second case there has been a pure external event which either has already been treated completely or the output loop is currently executed when the retrieve message arrives. Now, the whole external event with all its consequences has to be withdrawn. In the following this event will be referred to as *withdraw event*.

4.1.3.3 PowerDEVS Implementation

Listings 4.16–4.19 show the DEVS for the summation block, based on the sources in section 4.1.2.3 and extended according to the solution approach for retrieving messages suggested in

section 4.1.3.2. The code for the time advance function is omitted as it has not changed since section 4.1.1.3.

```
1  struct DEVSMessage {
2    double value;
3    bool retrieve;
4  };
5
6  class InOutput {
7    public:
8      double value;
9      double value_bk;
10     double last_change_time;
11     double last_change_time_bk;
12     bool already_treated;
13
14     InOutput() {
15       last_change_time = -1;
16       double last_change_time_bk=-1;
17       already_treated = false;
18     }
19     void set(double val,double t) {
20       value = val;
21       last_change_time = t;
22       already_treated = false;
23     }
24     void setbk(double val,double t) {
25       if(last_change_time < t) {
26         value_bk = value;
27         last_change_time_bk = last_change_time;
28       }
29       value = val;
30       last_change_time = t;
31       already_treated = false;
32     }
33     void restoreBackup() {
34       value = value_bk;
35       last_change_time = last_change_time_bk;
36       already_treated = true;
37     }
38     double* treat() {
39       already_treated = true;
40       return(&value);
41     }
42     bool isEqualTo(double val,double t) {
43       if(last_change_time!=t) return(false);
44       return(value==val);
45     }
46 };
```

Listing 4.14: Definition of the struct `DEVSMessage` describing input and ouput messages and redefintion of the class `InOutput`(formerly defined in Listing 4.7).

Listings 4.14 and 4.15 show the definition of the structure `DEVSMessage` and of the classes `InOutput` and `InOutputVector` which were already introduced in section 4.1.2.3. However, `InOutput` and `InOutputVector` have been extended a bit.

The structure `DEVSMessage` is used for all input and output messages. Its attribute `retrieve` indicates whether the message is a valid one or just serves to retrieve a formerly sent message.

The extensions in class `InOutput` regard the ability to keep a backup of `value` and of `last_change_time` and to make that backup automatically when `value` is overwritten for the first time at current simulation time. For this purpose the function `setbk` is implemented. For restoring the backup the function `restoreBackup` can be used.

```
1  class InOutputVector:public std::vector<InOutput> {
2    public:
3      int untreated_entry_changes;
4      int current_values;
5
6      InOutputVector() {
7        untreated_entry_changes = 0;
8      }
9      void setAt(int c,double val,double t) {
10       if(true==(*this)[c].already_treated) {
11         untreated_entry_changes++;
12       }
13       if((*this)[c].last_change_time < t) {
14         current_values++;
15       }
16       (*this)[c].set(val,t);
17     }
18     void setAtbk(int c,double val,double t) {
19       if(true==(*this)[c].already_treated) {
20         untreated_entry_changes++;
21       }
22       if((*this)[c].last_change_time < t) {
23         current_values++;
24       }
25       (*this)[c].setbk(val,t);
26     }
27     void restoreBackupAt(int c){
28       if(false==(*this)[c].already_treated) {
29         untreated_entry_changes--;
30       }
31       if((*this)[c].last_change_time_bk < (*this)[c].last_change_time) {
32         current_values--;
33       }
34       (*this)[c].restoreBackup();
35     }
36     double* treatAt(int c) {
37       if(false==(*this)[c].already_treated) {
38         untreated_entry_changes--;
39       }
40       return((*this)[c].treat());
41     }
```

```
42      void treatAll() {
43        for(int i=0; i<this->size(); i++){
44          (*this)[i].already_treated = true;
45        }
46        untreated_entry_changes=0;
47      }
48      void setUntreatedAt(int c) {
49        if(false != (*this)[c].already_treated) {
50          untreated_entry_changes++;
51          (*this)[c].already_treated=false;
52        }
53      }
54  };
```

Listing 4.15: The class `InOutputVector` which has also be extended compared to Listing 4.8

The extensions in class `InOutputVector` concern again the making and restoring of backups but they also introduce the variable `current_values`. `current_values` is intended to store the number of input ports which have received new values at current simulation time. This number is needed when an input message is retrieved to see whether there have been received other input messages at current simulation time than the one that is retrieved. If there has not been received any others, either `flag` has to be set from `'c'` to `'i'` and the output loop has to be restarted, or, if `flag='e'`, the whole external event has to be withdrawn.

Therefore, `current_values` is incremented each time an entry of the `InOutput` array is changed for the first time at current simulation time and it is decremented at each backup restoration that decreases the attribute `last_change_time`. Furthermore, `current_values` is reset to zero at the very first event, internal or external, at a concrete simulation time.

```
1 DEVSMessage output_msg;
2 double sigma_n_bk;
```

Listing 4.16: Defnition of a variable which is used for sending output messages of type `DEVSMessage` and of a backup variable for `sigma_n` which is needed when due to a message retrieval a former external event becomes a *withdraw event*.

The two variable declarations in Listing 4.16 are needed additionally to the declarations made in Listing 4.9. The object `output_msg` is used for each output message that is sent. `sigma_n_bk` is needed as backup for `sigma_n` in case of a withdraw event.

```
1 if(out_array.untreated_entry_changes==0 && in_array.untreated_entry_changes
      ==0) {
2   if(flag=='i') {
3     sigma_n = 1;
4   } else if(flag=='e' || flag=='c') {
5     if(flag=='c') {
6       sigma_n = 1;
7     }
```

```
8       if(in_array[0].last_change_time==t) {
9         s = s_old + in_array[0].value;
10      }
11      if(in_array[1].last_change_time==t) {
12        s = s_old - in_array[0].value;
13      }
14      if(s>=m) {
15        s = 0;
16      }
17    } else if(flag=='n') {
18      s = s_old;
19      r = r_old;
20      sigma_n = sigma_n_bk;
21    }
22    sigma = sigma_n;
23  }
```

Listing 4.17: The PowerDEVS internal transition function for the summation block example with message retrieve mechanism.

Listing 4.17 shows the internal transition function. The only difference to Listing 4.11 consists of the additional case `flag=='n'` in line 17. `'n'` stands for 'no event', i.e. due to input messages that have been retrieved, formerly an event has been triggered which needs to be withdrawn now. Thus, the tasks to be done in this case are to restore the old state.

Listing 4.18 shows the external transition function. The things that are new in the first block of code (lines 1 to 11) are the backup of `sigma_n` and the reset of `x.array.current_values` to zero. The other new part in it, compared to Listing 4.12, is the block of code from line 12 to line 23. In line 12 simply an object of type `DEVSMessage` is created to store the received input message that needs to be of type `DEVSMessage` as well. Afterwards, it is checked if the received message is a retrieve message.

If this is not the case, everything works like in section 4.1.2.3, apart from the fact that in line 25 `setAtbk` is called instead of `setAt`. The difference is that `setAtbk` automatically creates a backup of the old input value at the corresponding input port before overwriting it, but only if the new message is the first to arrive at that input port at current simulation time. `setAt` does not make any backups.

However, if the received message is a retrieve message, the last backup for the corresponding input port has to be restored and it has to be checked whether any other input messages that arrived at current simulation time are left. This is accomplished by simply checking the value of `in_array.current_values`. If this value is greater than zero, then the only thing to do is setting `sigma` to zero. This is to make sure that the output function is executed once more and the output loop is repeated.

```
1 if(tl<t) { // if first event at current time
2   r_old = r;
3   s_old = s;
4   flag = 'e';
5   sigma_n = sigma - e;
6   sigma_n_bk = sigma_n;
```

```
7    if(sigma_n==0) {
8      flag = 'c';
9    }
10   in_array.current_values = 0;
11 }
12 DEVSMessage input_msg = *(DEVSMessage*)x.value;
13 if(input_msg.retrieve == true) {
14   in_array.restoreBackupAt(x.port);
15   if(in_array.current_values==0) {
16     if(flag=='c') {
17       flag = 'i';
18     } else if (flag=='e'){
19       flag = 'w';
20     }
21     sigma=0;
22   }
23 } else {
24   if(false==in_array[x.port].isEqualTo(input_msg.value,t)) {
25     in_array.setAtbk(x.port,input_msg.value,t);
26     if(flag=='i') {
27       flag = 'c';
28     }
29     sigma = 0;
30   }
31 }
```

Listing 4.18: The PowerDEVS external transition function for the summation block example with message retrieve mechanism.

If `in_array.current_values` is zero, there are two cases to be distinguished. In the first case also an internal transition is triggered at current simulation time. Therefore, `flag` needs to be changed from `'c'` to `'i'` and the output function has to be executed once more to start a pure internal event. In the second case it has been a pure external event until the event causing message was retrieved and therefore, the event type changes to withdraw event. So, the initial state consisting of `s_old`, `r_old`, and `sigma_n_bk` has to be restored. Moreover, output messages that may have already been sent in reaction to the input message need to be retrieved. Therefore, `flag` is set to `'w'` which causes λ to do nothing but to change `flag` to `'n'` at its next execution. As `sigma` is zero, the output function is called once more and therefore, the outputs that need to be retrieved are identified and the output loop is started. As `flag` was set to `'n'` (meaning no event) by λ, in following calls of the output function in the context of the output loop, λ will not be executed anymore. When all outputs are retrieved, in the internal transition function, the old system's state is restored and `sigma` is set to `sigma_n_bk` and thus, the whole external event has been withdrawn successfully.

```
1 if(tl<t) { // if first event at current time
2   r_old = r;
3   s_old = s;
4   flag = 'i';
5   sigma = 0;
```

```
6    in_array.current_values = 0;
7  }
8  if(out_array.untreated_entry_changes == 0 || flag=='w') {
9    if(flag=='i') { // create outputs if pure internal event
10     out_array.setAt(1,s_old,t); // output s
11   } else if(flag=='e' || flag=='c') {
12     double s_ = s_old;
13     if(flag=='c') {
14       out_array.setAt(1,s_old,t); // output s
15     }
16     if(t_in_array[0]==t) {
17       s_ = s_ + in_array[0].value;
18     }
19     if(t_in_array[1]==t) {
20       s_ = s_ - in_array[0].value;
21     }
22     if(s_ > m) {
23       out_array.setAt(0,s_,t); // output reset
24     }
25   }
26   if(flag=='w') {
27     flag = 'n';
28   }
29   for(outport=0; outport<out_array.size(); outport++) {
30     if(out_array[outport].already_treated == true) {
31       if(out_array[outport].last_change_time == t) {
32         out_array[outport].last_change_time = t - 1;
33         out_array.setUntreatedAt(outport);
34       }
35     }
36   }
37   in_array.treatAll();
38   outport = -1;
39 }
40 while(out_array.untreated_entry_changes>0) {
41   outport++;
42   if(false == out_array[outport].already_treated) {
43     output_msg.value = *(out_array.treatAt(outport));
44     if(out_array[outport].last_change_time<t) {
45       output_msg.retrieve = true;
46     }
47     return Event(&output_msg,outport);
48   }
49 }
50 return Event();
```

Listing 4.19: The PowerDEVS output function for the summation block example with message retrieve mechanism.

Listing 4.19 shows the output function. The differences to Listing 4.13 consist basically of the mechanism for identifying output messages that need to be retrieved (lines 29 to 36), of case `flag=='w'` (line 26), and of the slightly different output while loop (lines 40 to 49).

The meaning of `flag=='w'` has already been described above. The mechanism for identifying retrieve output messages is implemented by a for loop. In this for loop simply all output ports are parsed and it is checked whether the ports' attribute `last_change_time` is equal to the current simulation time `t` although there was no new value computed for that port during the preceding calculation of λ, i.e. `already_treated==true`. In this case, at the corresponding output port a retrieve-message needs to be sent which is marked by setting the ports' `last_change_time` to `t-1`. Thus, the retrieve ports can be identified later in the output while loop.

4.2 Message Transmission and Reusability of Library Blocks

As already mentioned earlier, an output message of a PowerDEVS block essentially consists of a pointer of type `void` and of an integer determining the output port. Those two values are stored in form of an instance of the class `Event`. Thus, in the output function, with the line

```
return Event(&value, port);
```

a new instance of `Event` is created and handed on to an instance of the class `Coupling`. The class `Coupling` represents a coupled model and therefore, it is responsible for delivering output messages to corresponding recipients. A recipient either can be an input port of a block that is also located inside the coupled model or it can be an instance of `Coupling` that represents a coupled model one hierarchical level above. `port` determines the output port, at which the message is to be sent. `&value` denotes the address of the variable `value` that is assigned to the `void` pointer.

From output port and coupling the receiving blocks and their corresponding input ports are determined. For each receiving block and input port, the sent instance of `Event` is copied (flat copy) and its integer value is set to the input port where it arrives. This instance is available then, as variable `x` in the external transition function of the receiving block.

When handing over the address of a variable to the `Event` constructor in the return statement of the output function, it is important that this variable is not defined locally in the output function. This is because otherwise the memory allocated for this variable is freed after the execution of the return statement and thus, a pointer arrives at the receiving external transition function which points to a location in memory that is not allocated anymore or maybe already allocated for something else.

```
1 class Event
2 {
3  /*! Tells which synchronization should be used for this event */
4  RealTimeMode mode;
5 public:
6  /*! The value carried by an event is defined to be a void pointer for
       flexibility.
7  *   Most of the blocks in the library (by convention) use this value
       pointing to a double array.
```

```
8  *   Therefore an input value can be retrieved with:
9  * double *v = (double*) event.value;
10 *
11 * v[0] is the value
12 * v[1] is the first derivate
13 */
14 void *value;
15 /*! The port value is used to know from which port the event came from */
16 Port port;
17 /*! This bool value is used in the RTAI distribution. When set, indicates
       that an external input event
18 *    ocurred while synchronzing this event, then, it should not be
       propagated. */
19 bool interrupted;
20 Event();
21 Event( void*, Port);
22 virtual ~Event();
23
24 /*! A null event is defined as one with value=NULL. */
25 void setNullEvent();
26 bool isNotNull();
27
28 void setRealTimeMode(RealTimeMode m) { mode=m; };
29 RealTimeMode getRealTimeMode() { return mode; };
30 /*! Retrieves the i'th double value of this event */
31 double getDouble(int i) { return ((double*)value)[i]; };
32 /*! Retrieves the first double value of this event */
33 double getDouble() { return getDouble(0); };
34 /*! Retrieves the first int value of this event */
35 double getInt()   { return ((int*)value)[0]; };
36 void   setDouble(double &v) { value=&v; };
37 void   setInt(int &v) { value=&v; };
38 /*! This is used internally by the simulation engine to notify an external
       input event */
39 void setInterrupted() { interrupted=true; }
40 bool isInterrupted() { return interrupted; }
41 };
```

Listing 4.20: The C++ class `Event` as it is defined in th file 'event.h' located in the subdirectory 'engine' of the folder 'powerdevs'.

Listing 4.20 shows the C++ definition of the class `Event`. It can be seen that there are the member variables `void *value` in line 14 and `Port port` in line 16. `Port` is defined in 'types.h' and is nothing else but integer.

As PowerDEVS is especially designed for simulating hybrid systems using QSS, usually the member variable `value` points to a double array (usually of size 10). The entries of such a double array are then representing the coefficients of the Taylor polynomial that describes the QSS signal. When vectorial signals are used, `value` points to an instance of the class `vector`, depicted in Listing 4.21.

```
1 class vector
2 {
3 public:
4  double value[10];
5  int index;
6 };
```

Listing 4.21: The C++ class `Event`.

4.2.1 Creating Deep Copies of Input Messages of Unknown Type

4.2.1.1 Problem Identification

When receiving an input message `x`, there are two possible ways to treat it in the external transition function: either a deep copy is made, or it is continued to use the memory pointed to by `x.value`. While the first method is quite straight forward, the second method comes with some issues that need to be considered.

The first issue is that the block that allocated the memory needs to take care of not freeing it as long as it is in use in some other block.

The second issue is that the memory may be edited at different blocks in a coupled model and possibly at the same simulation time. Therefore the consistency of the memory's content needs to be taken care of. Thus, the safe and easy way is to make a deep copy of an input message at each receiving block which then can be altered by each block arbitrarily. However, to be able to make a copy the concrete type of the input message needs to be known. As the received value is only a pointer of type `void*`, the concrete type of an input message cannot be learned from the input message itself, but needs to be known in advance when programming the regarded PowerDEVS block.

Furthermore, sometimes it is needed that a block modifies some attributes of a received input message and then outputs again the modified message. Examples with such behaviour often occur when simulating production processes. Thereby, often workpieces are simulated that wander from one workstation to the next. When a workpiece arrives at a workstation it gets modified and then is handed on to the next workstation or it is merged with other workpieces or divided into smaller workpieces and the results are handed on. Anyway, the most logical method to represent such workpieces (also called entities) in PowerDEVS would be as instances of a particularly designed C++ class describing a workpiece and its attributes. Thus, instances of this class, representing concrete workpieces with concrete attribute values, could be handed on from block to block via their addresses in memory. Each block makes a deep copy of every instance it receives, modifies it, and hands on the address in memory of the modified instance.

Now, in one production process, there may appear a variety of different workpieces with different attributes. Therefore it seems to be reasonable to create a class `Entity` which serves as base class to derive different workpiece classes from. However, those different types of workpieces may walk through the same workstations, or at least through workstations with the same or very similar tasks. So, for reusability reasons, it would be wise to program those workstations

in a way they can handle a whole class of different workpieces, e.g. all workpieces whose C++ classes are derived from a concrete base class.

With the goal to program a generic workstation that can handle all workpieces whose representing C++ class is derived from a certain base class, the creation of a deep copy of a received workpiece instance becomes a problem. As the workstation may need to pass on an instance that is of the same class of which the arrived message has been, this concrete class has to be known when making the deep copy. The generic workstation though, only knows that the class of the received workpiece is derived from a certain base class but it does not know its concrete type. So programming a generic workstation seems to be in conflict with making deep copies. Fortunately C++ supports *polymorphism* which can be utilized to overcome that problem.

4.2.1.2 Solution Approach

As mentioned above, for the solution approach polymorphism is used which can be explained roughly as follows. In C++ it is possible to let a pointer of a base class type point to an instance of a derived class. Further, if in the base class there is a method marked as `virtual`, its functionality has to be redefined in each derived class. Now, if there is a pointer `ptr` of base class type, pointing to an instance of a derived class and a method `meth()` marked as `virtual` is called: `ptr->meth()`, then the defintition of `meth()` in the derived class is used.

This concept can be utilized for the problem of making deep copies of arriving messages without knowing their exact type. At first a class `DEVSMessage` (as depicted in Listing 4.22) is defined and it is demanded that every message between two PowerDEVS blocks has to be an instance of `DEVSMessage` or of a derived class. More exactly, every message is a pointer of type `DEVSMessage*` to such an object. As `DEVSMessage` has a `virtual` method named `getCopy()`, every derived class has to define that method too. The task of `getCopy()` is to create a deep copy of its calling instance and to return a pointer of type `DEVSMessage*` pointing to it. Therefore, when there arrives an input message in form of a memory address of an instance of an arbitrary, `DEVSMessage` derived class, this address can be stored in a `DEVSMessage*` pointer. Afterwards a deep copy of the corresponding instance can be created by calling `getCopy()` and the position in memory of that copy, again, can be stored in a `DEVSMessage*` pointer. So, a deep copy of an input message can be made, without knowing its exact type.

4.2.1.3 PowerDEVS Implementation

For a better understanding, a concrete example will be elaborated in the following. First the base class `DEVSMessage` is defined in Listing 4.22. This base class has only one attribute: `retrieve`. This is because with the approach to simplify the formulation of a correct DEVS model that was worked out in section 4.1, there has to be a mechanism for retrieving already sent messages. Therefore, if `retrieve` is `true`, the only effect of the message is to reverse the sending of a former message, by restoring the backup that has been made.

The member functions of `DEVSMessage` consist of an empty constructor and of a copy constructor as well as of the mentioned `getCopy()` method. `getCopy()` returns a pointer of type `DEVSMessage*` pointing to a copy of the instance from which `getCopy()` was called

from. The keyword `virtual` forces derived classes to also define a `getCopy()` method with exactly the same signature.

```
1  class DEVSMessage {
2  public:
3   bool retrieve;
4
5   DEVSMessage() { retrieve=false; }
6   DEVSMessage(const DEVSMessage &msg) { retrieve=false; }
7   virtual bool operator==(DEVSMessage& msg) = 0;
8   virtual bool operator!=(DEVSMessage &msg){ return(!((*this)==msg)); }
9   virtual DEVSMessage* getCopy() { return(new DEVSMessage(*this)); }
10  virtual ~DEVSMessage(){}
11 };
```

Listing 4.22: The C++ base class `DEVSMessage`.

Next, in Listing 4.23 a class `Entity` is derived form `DEVSMessage`.

```
1  class Entity:public DEVSMessage {
2   public:
3     double costs;
4     double CO2;
5
6     Entity():DEVSMessage() {
7       costs = 0;
8       CO2 = 0;
9     }
10    Entity(const Entity &msg):DEVSMessage((DEVSMessage&)msg) {
11      costs = msg.costs;
12      CO2 = msg.CO2;
13    }
14    bool operator==(Entity& e) {
15     if(costs != e.costs) return false;
16     if(costs != e.CO2) return false;
17    }
18    virtual bool operator==(DEVSMessage &msg){
19      if(typeid(msg)!=typeid(*this)) return false;
20      return( (*this)==((Entity&)msg) );
21    }
22    bool operator!=(Entity &e){
23      return(!((*this)==e));
24    }
25    virtual DEVSMessage* getCopy() { return(new Entity(*this));}
26    virtual ~Entity(){}
27 };
```

Listing 4.23: The C++ base class `Entity`.

With the BaMa project in mind, each entity has the attributes `costs` and `CO2` which store an accumulated value for the costs and for the CO2 emissions that were caused by that entity so far

during the production process. The member functions of the class `Entity` are again an empty constructor, a copy constructor and the method `getCopy()`. As it can be seen, the definition of a copy constructor is everything that is needed to define the `getCopy()` method.

To also have a more concrete entity, the class `Bread` is defined (see Listing 4.24).

```
1  class Bread:public Entity {
2   public:
3     double weight;
4     double T;
5
6     Bread():Entity() {
7       weight = 0;
8     }
9     Bread(const Bread &msg):Entity((Entity&)msg) {
10      weight = msg.weight;
11    }
12    bool operator==(Bread& b) {
13     if(weight != b.weight) return false;
14     return((Entity)(*this)==(Entity&)b);
15    }
16    virtual bool operator==(DEVSMessage &msg){
17      if(typeid(msg)!=typeid(*this)) return false;
18      return( (*this)==((Bread&)msg) );
19    }
20    bool operator!=(Bread &b){
21      return(!((*this)==b));
22    }
23    virtual DEVSMessage* getCopy() { return(new Bread(*this));}
24    virtual ~Bread(){}
25 };
```

Listing 4.24: The C++ base class `Bread`.

`Bread` was chosen, because as example a simple model of a baking oven will be presented later. So, the attributes of the class `Bread` are the `weight` and the Temperature `T` of the bread and additionally the attributes `costs` and `CO2` that are inherited from `Entity`. The temperature of the bread will later be used for restricting the baking time of a loaf of bread by the reaching of a certain temperature. This will be modeled as state event in the hybrid model of the baking oven in section 5.

The example that will be looked at consists of a baking oven that expects entities of type `bread` and of a preceding queue that stores arriving bread while the oven is baking (see Figure 4.3). Since the queue shall be implemented as generic as possible, it should be able to handle all kinds of entities that are derived from the class `Entity`.

As, from now on, it is demanded that each input and output message is of type `DEVSMessage` or of a derived type, the classes `InOutput` and `InOuputVector` have to be adapted as well (see Listings 4.25 and 4.26).

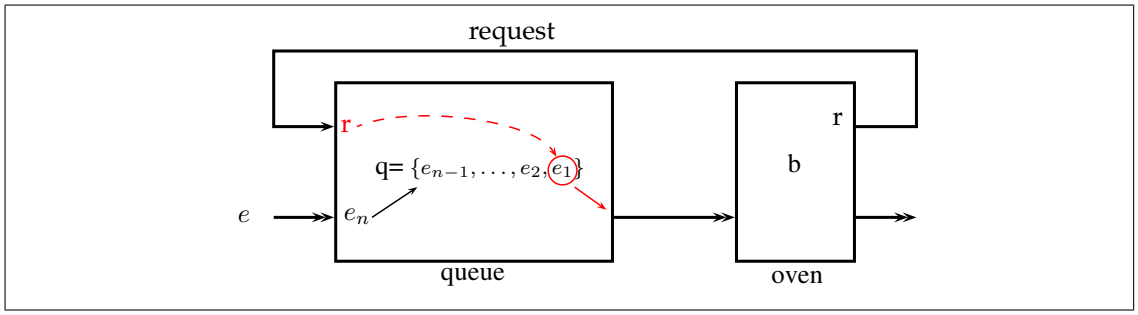


Figure 4.3: An Illustration of the queue - baking oven example. The queue works according the first in - first out (FIFO) principle. The baking oven requests new entities of bread by sending a message to the request-input port of the queue.

```
1  class InOutput {
2    public:
3      DEVSMessage *msgPtr;
4      DEVSMessage *backupPtr;
5      double last_change_time;
6      double last_change_time_bk;
7      bool already_treated;
8      bool link;
9
10     InOutput() {
11       msgPtr=NULL;
12       backupPtr=NULL;
13       last_change_time = -1;
14       last_change_time_bk = -1;
15       already_treated = true;
16       link = true;
17     }
18     void set(DEVSMessage *msg, double t) {
19       if(!link) delete msgPtr;
20       msgPtr = msg->getCopy();
21       link = false;
22       this->last_change_time = t;
23       this->already_treated = false;
24       msgPtr->retrieve=false;
25     }
26     void linkTo(DEVSMessage *msg, double t) {
27       if(!link) delete msgPtr;
28       msgPtr = msg;
29       link = true;
30       this->last_change_time = t;
31       this->already_treated = False;
32       msgPtr->retrieve=false;
33     }
34     void backup() {
35       if(NULL!=backupPtr) {delete backupPtr; backupPtr=NULL;}
36       if(NULL!=msgPtr) backupPtr = msgPtr->getCopy();
```

```
37        last_change_time_bk = last_change_time;
38      }
39      void restoreBackup() {
40        if(NULL!=backupPtr) {
41          if(false == link) {
42             delete msgPtr;
43          }
44          msgPtr = backupPtr;
45          link = false;
46          last_change_time = last_change_time_bk;
47          backupPtr=msgPtr->getCopy();
48        }
49      }
50      void setbk(DEVSMessage *msg, double t) {
51        if(last_change_time < t) {
52          backup();
53        }
54        set(msg,t);
55      }
56      DEVSMessage* treat(){
57        this->already_treated = true;
58        return(msgPtr);
59      }
60      bool isEqualTo(DEVSMessage *msg,double t) {
61        if(NULL == msgPtr) return(false);
62        if(last_change_time!=t) return(false);
63        return(msgPtr->operator==(*msg));
64      }
65      ~InOutput(){
66        if(!link) delete msgPtr;
67        if(NULL != backupPtr) delete backupPtr;
68      }
69  };
```

Listing 4.25: Extension of class `InOutput`, compared to Listing 4.14, to be able to store general input messages of type `DEVSMessag` or derived.

Instead of a value of type `double`, `InOutput` stores a pointer `msgPtr` of type `DEVSMessage*` and also the backup variable `backupPtr` is of type `DEVSMessage*` now. As described above, when for example an input message arrives a deep copy is created, using the method `getCopy()`, and `msgPtr` will be pointing to it. Therefore, when the next input message arrives, the memory that was allocated in the former call of `getCopy()` has to be freed again before letting `msgPtr` point to another location in memory. However, to obtain the possibility to hand on only pointers instead of making deep copies, the member variable `link` is introduced. If `link` is `true`, it is known, that `msgPtr` is only linked to an area in memory that has been allocated somewhere else and therefore is not allowed to be freed. This is also what the new member function `linkTo` is for. It has to be used instead of the function `set` when no deep copy is to be made.

Also the creation and restoration of backups is different now, as again the method `getCopy()` has to be used, however, the working principle stays the same.

```
class InOutputVector:public std::vector<InOutput> {
  public:
    int untreated_entry_changes;
    int current_values;

    InOutputVector() {
      untreated_entry_changes = 0;
    }
    void setAt(int c, DEVSMessage *msg, double t) {
      if(true==(*this)[c].already_treated) {
        untreated_entry_changes++;
      }
      if((*this)[c].last_change_time < t) {
        current_values++;
      }
      (*this)[c].set(msg,t);
    }
    void linkToAt(uint j, DEVSMessage *msg,double t) {
      if(False != (*this)[j].already_treated) {
        untreated_entry_changes++;
      }
      (*this)[j].linkTo(msg,t);
    }
    void setAtbk(int c, DEVSMessage *msg, double t) {
      if(true==(*this)[c].already_treated) {
        untreated_entry_changes++;
      }
      if((*this)[c].last_change_time < t) {
        current_values++;
      }
      (*this)[c].setbk(msg,t);
    }
    void restoreBackupAt(int c){
      if(false==(*this)[c].already_treated) {
        untreated_entry_changes--;
      }
      if((*this)[c].last_change_time_bk < (*this)[c].last_change_time) {
        current_values--;
      }
      (*this)[c].restoreBackup();
    }
    DEVSMessage* treatAt(int c) {
      if(false==(*this)[c].already_treated) {
        untreated_entry_changes--;
      }
      return((*this)[c].treat());
    }
    void treatAll() {
      for(int i=0; i<this->size(); i++){
        (*this)[i].already_treated = true;
      }
      untreated_entry_changes=0;
    }
```

```
54      void setUntreatedAt(int c) {
55        if(false != (*this)[c].already_treated) {
56          untreated_entry_changes++;
57          (*this)[c].already_treated=false;
58        }
59      }
60  };
```

Listing 4.26: Extension of class `InOutputVector`, compared to Listing 4.15, to be able to store general input messages of type `DEVSMessage` or derived.

The class `InOutputVector` has not change that much either. Only the type of some function arguments and return values have changed from `double` to `DEVSMessage*`.

Listings 4.27–4.32 show the source code for the PowerDEVS model of the queue depicted in Figure 4.3.

```
1  InOutputVector in_array; // input messages
2  InOutputVector out_array;
3  double sigma, sigma_n, sigma_n_bk;
4  double last_event_time;
5  char flag;          // possible values: 'i','e','c','n'
6  int outport;
7  #define INF 1e20
8  bool req, req_old;
9  std::deque<Entity*> q;
10 bool pushed;
11 bool popped;
```

Listing 4.27: The PowerDEVS definitions area for the queue block example.

New compared to the definitions area of the summation block example is the variable `last_event_time`. So far the backup of the old state of the system needed to be programmed twice, once in the external transition function and once in the output function. Where the old state really was backed up, depended on whether the first event at a concrete simulation time was an external one or an internal one. As each external event always also leads to the execution of the output function, it suffices if the backing up of the state of the system is only done in the output function. However, the condition `tl<t` does not work anymore then. This is because in the case of the first event at time `t` being an external one, in the output function `tl` already will be equal to `t` and thus no state backup would be accomplished. To solve that problem `last_event_time` is used. It stores the time of the last execution of the output function and therefore is smaller than `t` every time, the output function is called for the first time at simulation time `t` (see Listing 4.31).

As the queue is a completely passive system that only reacts to stimuli at its inputs, there are no internal events. Thus, `sigma`, the time to the next internal event, has to be infinity. This is what the definition of `INF` is for. As infinity lies out of the range of `double` variables, a very huge value (`1e20`) is used instead.

To store the queue of waiting entities the standard C++ library class `deque` (double ended queue) is used. Arriving input entities will be pushed into queue and requested output entities

will be popped from the queue according to the first in - first out (FIFO) principle. Since the queue example will work according to the principles worked out in section 4.1, the whole system's state has to be doubled. However, as it would take a lot of effort to double the whole queue, the changes that are applied onto the queue are done in such a way that they can be withdrawn if a repetition of the output loop and thus a recalculation of the queue's new state becomes necessary. This is what the booleans `pushed` and `popped` are used for.

When an entity request arrives at input port 0 that can not be responded to because the queue `q` is empty, the pending request will be remembered by setting `req` to `true`. Thus, when later there arrives an entity, it immediately will be output to satisfy the formerly unacknowledged request.

```
1  in_array.resize(2);
2  out_array.resize(1);
3  sigma = INF;
4  sigma_n = INF;
5  last_event_time = -1;
6  flag = 'n';
7  outport = -1;
8  req = false;
9  req_old = false;
10 pushed = false;
11 popped = false;
```

Listing 4.28: The PowerDEVS init function for the queue block example.

Listing 4.28 shows the init function of the queue model. The queue block has two input ports and one output port. Therefore, the input and the output buffer are sized accordingly in lines 1 and 2. As the queue shall be able to store objects of type `Entity` and of types derived from `Entity`, it is defined as container for pointers of type `Entity*`.

```
1  if(out_array.untreated_entry_changes==0 && in_array.untreated_entry_changes
       ==0) {
2    if(flag=='i') {
3      sigma_n = INF;
4    } else if(flag=='e') {
5      if(in_array[0].last_change_time==t && in_array[1].last_change_time==t) {
6        if(q.size()>0) {
7          popped=true;
8          pushed=true;
9          req=false;
10       }
11     } else if(in_array[0].last_change_time==t) {
12       if(q.size()>0) {
13         req=false;
14         popped=true;
15         pushed=false;
16       } else {
17         popped=false;
18         pushed=false;
```

```
19          req=true;
20        }
21      } else if(in_array[1].last_change_time==t) {
22        if(q.size()==0 && req_old == true) {
23          popped=false;
24          pushed=false;
25          req=false;
26        } else if(req_old == true){
27          popped=true;
28          pushed=true;
29          req=false;
30        } else {
31          popped=false;
32          pushed=true;
33          req=false;
34        }
35      }
36      sigma_n = INF;
37    } else if(flag=='c') {
38      sigma_n = INF;
39    } else if(flag=='n') {
40      req = req_old;
41      pushed = false;
42      popped = false;
43      sigma_n = INF;
44    }
45    sigma = sigma_n;
46  }
```

Listing 4.29: The PowerDEVS internal transition function for the queue block example.

Listing 4.29 shows the internal transition function of the queue model. It has the same structure as the internal transition function of the summation block (Listing 4.11). So the interesting parts are the sections of code describing the behaviours of the δ functions. Since the queue is completely passive, the only type of transition that occurs is a pure external one. Therefore, in the cases `flag=='i'` and `flag=='c'`, the only thing to be done is to set the time until the next internal event to infinity. The only time, the internal transition is entered with `flag` being `'n'` is when an event is withdrawn completely, i.e. all the input messages that caused the event have been retrieved. Therefore, in this case there are no changes to be conducted to the queue.

Solely by looking at which input messages arrived currently and at the value of `req_old`, it can be determined which outputs have been produced and whether a new entity has been pushed into the queue. These actions are stored coded into the variables `pushed` and `popped`. Therefore, and due to the fact that a possible output entity has not been popped from the queue but only read, it later will be possible to withdraw those actions if necessary. How this works, is depicted in Listing 4.31.

```
1  if(tl<t) { // if first event at current time
2    flag = 'e';
3    sigma_n = INF;
4    if(sigma != INF) {
5      sigma_n = sigma-e;
6    }
7    sigma_n_bk = sigma_n;
8    if(sigma_n == 0) {
9      flag = 'c';
10   }
11   in_array.current_values = 0;
12 }
13 DEVSMessage *input_msg = (DEVSMessage*)x.value;
14 if(input_msg->retrieve == true) {
15   in_array.restoreBackupAt(x.port);
16   if(in_array.current_values == 0) {
17     if(flag=='c') {
18       flag = 'i';
19     } else if (flag=='e'){
20       flag = 'w';
21       sigma_n = sigma_n_bk;
22     }
23     sigma=0;
24   }
25 } else {
26   if(false == in_array[x.port].isEqualTo(input_msg,t)) {
27     in_array.setAtbk(x.port,input_msg,t);
28     if(flag=='i') {
29       flag = 'c';
30     }
31     sigma = 0;
32   }
33 }
```

Listing 4.30: The PowerDEVS external transition function for the queue block example.

Listing 4.30 shows the external transition function of the queue model. Usually, when there is an external event which does not alter the time to the next internal event or which is retrieved again, the time `sigma` to the next internal event can be calculated as `sigma - e`. However, if the time to the next internal event is infinity (`INF = 1e20`), then `sigma` stays infinity no matter how big `e` is. Therefore, in such a case, the new value `sigma_n` for `sigma` cannot be calculated by `sigma-e`, but needs to set to `INF` again (see line 3).

Further, compared to the external transition function in section 4.1.3.3 (Listing 4.18), there is no backup of the old state in the external transition function anymore. As mentioned above, this is now accomplished solely in the output function with the help of `last_event_time`. The last difference to Listing 4.18 is the type of `input_msg`, which is now `DEVSMessage*` instead of `DEVSMessage`.

```
if(tl<t) { // if first event at current time
  flag = 'i';
  sigma = 0;
  in_array.current_values = 0;
}
if(last_event_time < t) { // backup old state, apply state changes
  last_event_time = t;
  if(pushed == true) {
    q.push_back((Entity*)(in_array.treatAt(1)->getCopy()));
  }
  if(popped == true) {
    delete (Entity*)q.front();
    q.erase(q.begin());
  }
  req_old = req;
}
if(out_array.untreated_entry_changes == 0 || flag=='w') {
  if(flag=='i') { // create outputs if pure internal event
    // this case should never occur
  } else if(flag=='e' || flag=='c') {
    if(in_array[0].last_change_time==t && in_array[1].last_change_time==t) {
      if(q.size()>0) {
        out_array.setAt(0,(DEVSMessage*)q.front(),t);
      } else {
        out_array.setAt(0,in_array.treatAt(1),t);
      }
    } else if(in_array[0].last_change_time==t) {
      if(q.size()>0) {
        out_array.setAt(0,(DEVSMessage*)q.front(),t);
      }
    } else if(in_array[1].last_change_time==t) {
      if(q.size()==0 && req_old == true) {
        out_array.setAt(0,in_array.treatAt(1),t);
      } else if(req_old == true){
        out_array.setAt(0,(DEVSMessage*)q.front(),t);
      }
    }
  }
  if(flag=='w') {
    flag = 'n';
  }
  for(outport=0; outport<out_array.size(); outport++) {
    if(out_array[outport].already_treated == true) {
      if(out_array[outport].last_change_time == t) {
        out_array[outport].last_change_time = t - 1;
        out_array[outport].msgPtr->retrieve=true;
        out_array.setUntreatedAt(outport);
      } else {
        out_array[outport].msgPtr->retrieve=false;
      }
    }
  }
  in_array.treatAll();
```

```
54    outport = -1;
55  }
56  while(out_array.untreated_entry_changes>0) {
57    outport++;
58    if(false == out_array[outport].already_treated) {
59      return Event(out_array.treatAt(outport),outport);
60    }
61  }
62  return Event();
```

Listing 4.31: The PowerDEVS output function for the queue block example.

Listing 4.31 shows the output function of the queue. In line 6 it is checked, whether this is the first execution of the output function at the current instant of time. If it is, the old system state is backed up and, in this case, the changes to the queue caused by the last event are conducted. This is done here, because time already advanced since the last event and therefore the changes cannot be withdrawn anymore.

The most interesting part ranges from line 17 to 38, representing λ. Here, depending on the current input messages, an entity from the queue or directly from the input is output.

In the for loop starting in line 42 the output buffer is searched through for outputs that need to be retrieved.

The while loop at the end of the code again carries out the outputting of created output messages.

```
1 for (int i=0; i<q.size(); i++) {
2   delete (Entity*)q.front();
3 }
```

Listing 4.32: The PowerDEVS exit function for the queue block example.

Listing 4.32 shows the exit function of the queue model, which is used for the first time here. It is needed, because the queue stores pointers to instances that were created using the method `getCopy()` which allocates new memory to store those instances. This memory needs to be freed again somewhere. This is done either after an entity has been popped from the queue, in the output function (line 12), or it is done in the exit function for the entities that are left in the queue at the end of the simulation.

4.2.2 Downwards Compatibility

Using the member variable `void* value` of the class `Event` (Listing 4.20) any imaginable object can be transmitted from one PowerDEVS block to another. However, in section 4.2.1 it is demanded that every message is an instance of the class `DEVSMessage` or of a derived class. All the already existing PowerDEVS library blocks though, of course do not use the class `DEVSMessage` as it did not exist when they were programmed. Now the question arises, if it is possible to reprogram the class `Event` in a way that every message is an object of a class derived from `DEVSMessage` and still, all the existing library blocks continue to work.

4.2.2.1 Problem Identification

As already mentioned in the introduction of section 4.2, most of the library blocks use `double` arrays of size ten to communicate. This is because most of the signals are QSS signals and the entries of the arrays represent the coefficients of the Taylor polynomials. Further, in the library for vectorial signals, instances of the class `vector` (see Listing 4.21) are used. They simply again consist of a `double` array of size ten and additionally of an integer giving the index of the vectorial signal the instance of vector is representing.

So in diverse library blocks the following lines of code can be found in the external transition function and in the output function:

```
1 double y[10];
2 vector v;
3 // in the outputfunction:
4 return Event(y,port);
5 // or
6 return Event(&y,port);
7 // or
8 return Event(&v,port);
9
10 // and in the extern transition function:
11 double *xy = (double*)x.value;
12 vector vec = *(vector*)x.value;
```

Listing 4.33: Usage of the `void` pointer of the class `Event` in diverse PowerDEVS library blocks.

Thereby, all combinations of lines with `return` in it with lines with `x.value` in it occur. That is, on the one hande `double[]` and `double*` are type cast to `vector*` and the result is even dereferenced, and on the other hand `vector*` is type cast to `double*`. The reason for this working at all is the simple structure of the class `vector` and, of course, the fact that its first member variable is also a `double` array of size ten. However, as soon as there are some member functions added to class `vector` the casts do not work anymore.

Moreover, the class `Event` possesses the member functions depicted in Listing 4.34.

```
1 Event() {
2   value = 0;
3   mode = NOREALTIME;
4   interrupted = 0;
5   port = 0;
6 }
7 Event( void* val, Port p) {
8   value=val;
9   mode = NOREALTIME;
10  interrupted = 0;
11  port=p;
12 }
13 void setNullEvent(){ value=0; port=0; }
14 bool isNotNull(){
```

```
15   bool state;
16   if (value==0) {state=false;} else {state=true;};
17   return state;
18 }
19 double getDouble(int i) { return ((double*)value)[i]; }
20 double getDouble() { return getDouble(0); }
21 double getInt() { return ((int*)value)[0]; }
22 void   setDouble(double &v) { value=&v; }
23 void   setInt(int &v) { value=&v; }
```

Listing 4.34: Member functions of the class `Event` that access its member variable `void *value`.

As it can be seen, the class `Event` offers methods to set `value` to a pointer of type `void*`, `int*`, and `double*` as well as the method `setNullEvent()`, where `value` is set to zero (type `int`). Further, there are the methods `getDouble(int i)` and `getInt()` which type cast `value` to `double*` and `int*` respectively. Finally, `value` is compared to zero in line 16.

So if the attribute `void* value` of `Event` was replaced by `DEVSMessage* value`, to ensure compatibility, it would have to be taken care of all these accesses to `value` still to be working. The problem is that it is not possible to control the behaviour of a pointer variable when it is type casted. However, it is possible for instances of a class.

4.2.2.2 Solution Approach

The Class ValuePointer. The idea is now to exchange the the member variable `void *value` of `Entity` for a non-pointer variable `ValuePointer value` of a custom type `ValuePointer` which is specially designed for making `Event` and all the PowerDEVS library blocks believe that `value` is still of type `void*`. The definition of the class `ValuePointer` is depicted in Listing A.2 which is, due its length, located in the appendix along with some other classes that will be explained in this section.

The class `ValuePointer` possesses a member variable `DEVSMessage *msgPtr` which is used for storing the actual message. Further, the type cast operators `(void*)`, `(int*)`, and `(double*)` are overloaded for `ValuePointer` objects as well as the comparison operator `==` for comparisons with `int`. The latter one actually is only defined for comparisons with zero. Finally also the assignment operator = is overloaded for assignments with `void*`, `int*`, `double*`, and `int` variables and, of course, for assignments with `ValuePointer` objects, as instances of Event are copied during the process of handing on messages from one block to another.

Furthermore, also the casts of `ValuePointer` objects to objects of type `DEVSMessage*` are generically overloaded to return the according type cast applied onto the member variable `msgPtr` instead. Due to this, in the external transition function `x.value` still can be treated as it were of type `DEVSMessage*`.

However, there is one type cast that needs to be handled separately, the cast to `vector*` when `ValuePointer value` formerly was assigned with a `double` array. In such a case, the solution is to create a new instance of `vector`, assign its `double` array with the array that

was formerly assigned to `ValuePointer value` and return the address to this newly created instance of `vector`. For this purpose, the class `ValuePointer` needs a second member variable to store that instance of type `vector` which is created inside the type cast overload member function.

Additionally there is one special case to be considered concerning the assignment operators. The assignment of `double*` variables has been mentioned already, but there are also the assignments of the form shown in line 6 in Listing 4.33, where the address `&y` of an array `double y[10]` is assigned to `ValuePointer value`. In this case, the type of `&y` is not `double*` but even depends on the size of the array. The concrete handling of this case can be seen in the source code (Listing A.2), although it is kind of not elegant. With the use of the C++ library `type_traits` a more elegant solution would be possible, however, this requires the compiler flag `-c++` to be set, which is not done in the make-file PowerDEVS uses.

The Class DEVSMessage. In the preceding paragraph it is mentioned that a set of type cast and assignment operators for objects of type `ValuePointer` need to be overloaded, but it is not mentioned how to overload them. The idea concerning this matter is to let the class `DEVSMessage` and its derived classes define how to handle the type casts and the assignments. Therefore, the interface of the class `DEVSMessage`, consisting of methods defined as `virtual` is extended to the set of methods depicted in Listing 4.35.

```
1  // used for the type-casts
2  virtual void* getVoidPtr() { return value;}
3  virtual double* getDoublePtr() { return((double*)value);}
4  virtual int* getIntPtr() { return((int*)value);}
5
6  // used for the assignment operators
7  virtual void set(void *ptr) { value=ptr; }
8  virtual void set(double *ptr) { value=(double*)ptr; }
9  virtual void set(int *ptr) { value=(int*)ptr; }
10
11 // necessary for the method 'isEqualTo' of the class 'InOutput'
12  virtual bool operator==(DEVSMessage& msg) {
13    if(this->index!=msg.index) return(false);
14    return(this->value==msg.value);
15  }
16  virtual bool operator!=(DEVSMessage &msg){
17    return(!((*this)==msg));
18  }
```

Listing 4.35: As `virtual` marked interface of the class `DEVSMessage`.

The whole definition of the class `DEVSMessage` can be seen in Listing A.3 in the appendix. The `get...`-methods are called by `ValuePointer` in the corresponding type-cast overloads. The `set...`-methods are called by `ValuePointer` in the corresponding assignment overloads. These methods access the member variable `value` of `DEVSMessage` that has not been defined so far. However, in Listing A.3 it is defined and its purpose is exactly to predefine the behaviour of the type cast and assignment operators. Further, `DEVSMessage` now still

can be used to transmit addresses of arbitrary objects by using its member variable `value` of type `void*`. The third attribute that `DEVSMessage` is equipped with now, is `int index`. `index` simply is used for giving the index when the regarded `DEVSMessage` instance is part of a vectorial signal, the same way it has been done in the class `vector` so far.

The Class QSSDoubleArray. As most of the PowerDEVS library blocks work with `double` arrays, a class derived from `DEVSMessage` is needed to store and manage those `double` arrays. For this purpose the class `QSSDoubleArray` is defined (see Listing A.4). It uses the `void*` pointer `value` of its base class to store a `double` array whose size may differ from ten and therefore is stored in the member variable `uint size`. As the memory to store the array is allocated dynamically, the member variable `bool allocated_memory` is needed to know whether there is memory to be freed before the array is reassigned and in the destructor of the class.

Moreover, `QSSDoubleArray` implements a bunch of methods that are intended to facilitate the handling of the `double` array, particularly in the case when it describes a QSS signal. Among those methods are an overload of the `[.]`-operator to be able to access the array entries directly as well as overloads for the operators $+$, $-$, $*$, $/$. Thereby, these operations are conducted as operations on polynomials with the array entries representing the coefficients of polynomials. Furthermore, a member function called `advance_time` is implemented. As explained in section 2.2, QSS signals in form of polynomials are Taylor polynomials of the described signal with a concrete expansion point in time. In PowerDEVS, when such a polynomial is received, it is always assumed that the expansion point is the time of reception. Therefore, from time to time in PowerDEVS it is necessary to shift the expansion point of a polynomial from a former point in time to the current one. This is, what `advance_time` can be used for. As argument it expects the distance in time from the old expansion point of the polynomial currently stored in the instance of `QSSDoubleArray` to the new expansion point (mostly the current time). For more details, it is referred to the source code (Listing A.4).

The Class vector. As the original class `vector` (see Listing 4.21) is not derived from `DEVSMessage` it has to be redefined as well (see Listing A.5). Since `vector` is supposed to store only two variables, a `double` array `double value[10]` and an integer `int index` for the index of the vector entry that is represented, actually the class `QSSDoubleArray` already fulfils everything demanded. However, in PowerDEVS library blocks entries of the double array are accessed in the way it is depicted in Listing 4.36.

```
1 vector v= *(vector*)x.value;
2 v.value[0] = 0;
```

Listing 4.36: The way entries of the `double` array of instances of the class `vector` are accessed.

However, the member variable `value` of `vector`(inherited from `DEVSMessage`) is of type `void*` and therefore the `[.]`-operator would not return the right thing when applied to `value` without casting it to `double` before.

That is why the member variable `value` is redefined in the class `vector`. If it simply would be redefined as `double*` pointer all the methods of the base class `QSSDoubleArray` would become worthless for `vector` as they would alter the attribute `value` of the base class `DEVSMessage`. To solve that problem, the same principle is applied that was used to make the class `Event` believe that its member variable `value` still is of type `void*`. A member variable named `value` of type `ValueDuplicate` is defined. `ValueDuplicate` simply stores a pointer to an instance of `QSSDoubleArray` and overloads the cast operations `(void*)`, `(int*)`, and `(double*)` as well as the operator `[.]`. As the class `vector` is derived from the class `QSSDoubleArray`, an instance of `vector` also represents an instance of `QSSDoubleArray`. Thus, for each instance of type `vector` the `QSSDoubleArray*` pointer of its attribute `value` points to the `QSSDoubleArray` instance of the `vector` instance itself. For more details it is referred to the source code (Listing A.5) again.

4.3 Atomic PDEVS Block

In this section a source code template, named *Atomic PDEVS*, for programming an atomic PowerDEVS block with the features developed in in the sections 4.1 and 4.2 will be presented. So far, all given examples worked with scalar input and output ports. In general though also vectorial ports should be allowed. To achieve that, the class `InOutputVector` that was used to describe the whole input and output interface of a block is now used to describe only a single port. For the description of a whole interface a new class `InOutputArray` is introduced (Listing 4.37) which is derived from `std:vector<InOutputVector>`. That is, the same way an instance of `InOutputVector` is an array of instances of `InOutput`, `InOutputArray` is an array of instances of `InOutputVector`.

```
1  class InOutputArray:public std::vector<InOutputVector> {
2   public:
3     uint untreated_entry_changes;
4     int current_values;
5
6   public:
7     InOutputArray() {untreated_entry_changes=0;}
8
9     void setAt(int r,int c,DEVSMessage *msg,double t) {
10      if(true==(*this)[r][c].already_treated) {
11        untreated_entry_changes++;
12      }
13      if((*this)[r][c].last_change_time<t) {
14        current_values++;
15      }
16      (*this)[r].setAt(c,msg,t);
17      (*this)[r][c].index = c;
18    }
19    void linkToAt(uint r, uint c, DEVSMessage *msg,double t) {
20      if(false != (*this)[r][c].already_treated) {
21        untreated_entry_changes++;
22      }
```

```
23        (*this)[r].linkToAt(c,msg,t);
24        (*this)[r][c].index = c;
25      }
26      void setAtbk(int r,int c, DEVSMessage *msg, double t) {
27        if(true==(*this)[r][c].already_treated) {
28          untreated_entry_changes++;
29        }
30        if((*this)[r][c].last_change_time < t) {
31          current_values++;
32        }
33        (*this)[r].setAtbk(c,msg,t);
34        (*this)[r][c].index = c;
35      }
36      void restoreBackupAt(int r, int c){
37        if(false==(*this)[r][c].already_treated) {
38          untreated_entry_changes--;
39        }
40        if((*this)[r][c].last_change_time_bk < (*this)[r][c].last_change_time)
              {
41          current_values--;
42        }
43        (*this)[r].restoreBackupAt(c);
44      }
45      DEVSMessage* treatAt(int r, int c) {
46        if(false==(*this)[r][c].already_treated) {
47          untreated_entry_changes--;
48        }
49        return((*this)[r].treatAt(c));
50      }
51      void treatAll() {
52        for(int i=0; i<this->size(); i++){
53          (*this)[i].treatAll();
54        }
55        untreated_entry_changes=0;
56      }
57      void setUntreatedAt(int r, int c) {
58        if(false != (*this)[r][c].already_treated) {
59          untreated_entry_changes++;
60        }
61        (*this)[r].setUntreatedAt(c);
62      }
63  };
```

Listing 4.37: The Defintion of the class `InOutputArray` which is used to represent the input and output interface of a block with vectorial input and output ports.

The working principle of `InOutputArray` is exactly the same as the one of `InOutputVector` (see Listing 4.26). Therefore, all member variables and member functions are already known and will not be explained here again.

Figure 4.4 shows the Atomic PDEVS block in the PowerDEVS model editor with its Parameters dialogue opened. The parameters to be entered are the number of input and output ports as

well as their dimensions and three example parameters demonstrating the user how to read their values in the init function.

```
1  InOutputArray in_array; // input messages
2  InOutputArray out_array;
3  double sigma, sigma_n, sigma_n_bk;
4  double last_event_time;
5  char flag;          // possible values: 'i','e','c','n','w'
6  int outport, outindex;
7  #define INF 1e20
8  // modeller's input area:
9    // define internal state s of the DEVS
10   // define backup s_old for the internal state s
11   // define model parameters and auxiliary variables, e.g.:
12   double Par1;
13   int Par2;
14   std::string Aux1;
```

Listing 4.38: The PowerDEVS definitions area for the generic Atomic PDEVS block.

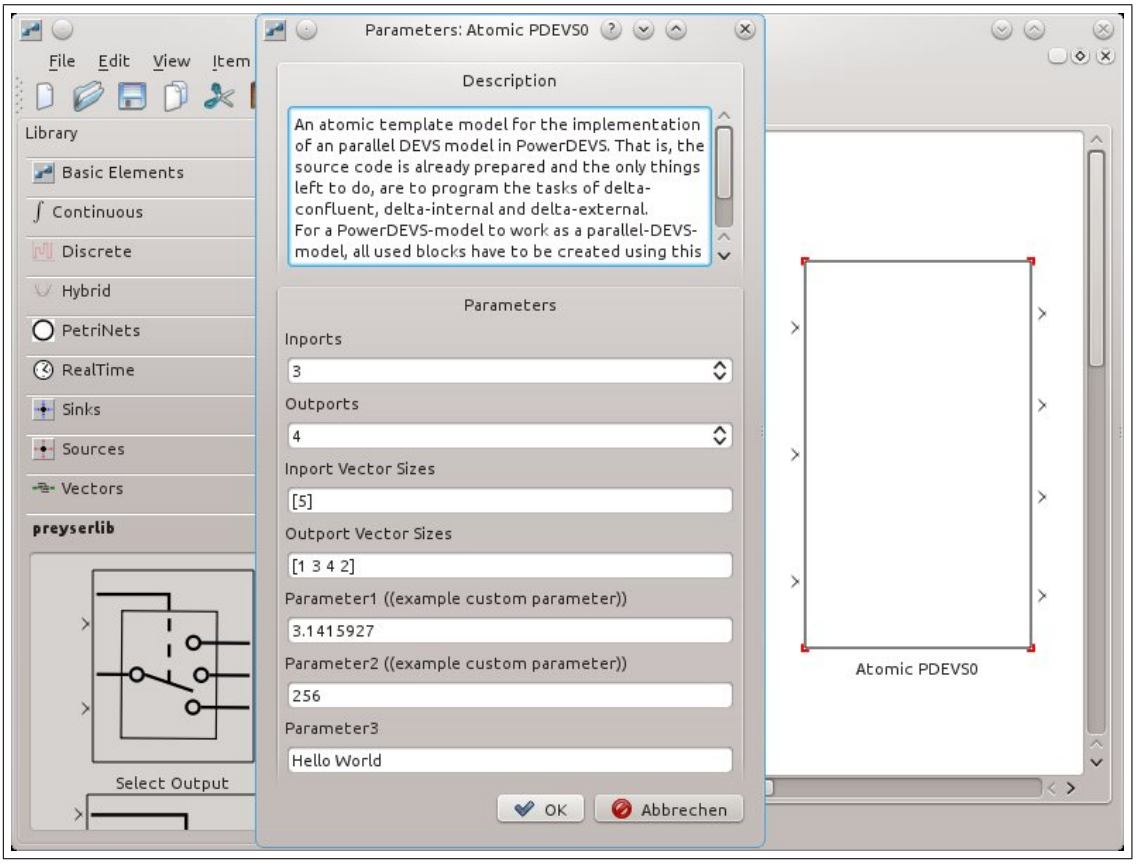


Figure 4.4: The Parameters dialogue of the Atomic PDEVS PowerDEVS block.

Listing 4.38 shows the defintions area of the Atomic PDEVS block. As it can be seen, input and output buffer are now variables of type `InOutputArray` instead of `InOutputVector`. Further, there is the variable `int outindex` which is also new compared to definitions area of the queue example in section 4.2.1 (Listing 4.27). Since the output ports are vectorial, in the while loop at the end of the output function (Listing 4.42) not only a port counter `outport`, but also an index counter `outindex` is needed.

The last three variables `Par1`, `Par2`, and `Aux1` are used to store the values of example block parameter 'Parameter1', 'Parameter2', and 'Parameter3' that can be seen in Figure 4.4. In their place the definition of model specific state variables, of model parameters, and of auxiliary variables should be entered by the modeller who uses this template to design a custom PowerDEVS block.

```
1  int n_inports = (int)va_arg(parameters,double);
2  int n_outports = (int)va_arg(parameters,double);
3  in_array.resize(n_inports);
4  out_array.resize(n_outports);
5  std::string par_var = std::string((char*)va_arg(parameters,char*));
6  std::string length_str ("length(" + par_var + ")");
7  int length = (int)getScilabVar(&length_str[0]);
8  double *vec_size_d = (double*)malloc(length*sizeof(double));
9  getScilabVector(&par_var[0], &length, vec_size_d);
10 int k=0;
11 for(int i=0; i<n_inports;i++) {
12   if(k==length) k=0;
13   in_array[i].resize((int)vec_size_d[k]);
14   k++;
15 }
16 free(vec_size_d);
17 par_var = std::string((char*)va_arg(parameters,char*));
18 length_str = std::string("length(" + par_var + ")");
19 length = (int)getScilabVar(&length_str[0]);
20 vec_size_d = (double*)malloc(length*sizeof(double));
21 getScilabVector(&par_var[0], &length, vec_size_d);
22 k=0;
23 for(int i=0; i<n_outports;i++) {
24   if(k==length) k=0;
25   out_array[i].resize((int)vec_size_d[k]);
26   k++;
27 }
28 free(vec_size_d);
29 sigma = INF;
30 sigma_n = INF;
31 last_event_time = -1;
32 flag = 'n';
33 outport = -1;
34 outindex = 0;
35 // modeller's input area
36   // initialize state s
37   // read custom parameters, for example:
```

```
38 char *fvar = va_arg(parameters,char*);
39 Par1 = (double)getScilabVar(fvar);
40 fvar = va_arg(parameters,char*);
41 Par2 = (int)getScilabVar(fvar);
42 Aux1 = std::string(va_arg(parameters,char*));
```

Listing 4.39: The PowerDEVS init function for the generic Atomic PDEVS block.

Listing 4.39 shows the init function of the Atomic PDEVS block. In the first two lines the number of input and output ports are read from the list `parameters` which contains all the parameter values that have been entered in the Parameters dialogue. Then the input and output buffers are resized accordingly. The next parameter to be read from `parameters` is a vector of the form $[d_1, d_2, \ldots, d_n]$ stored as c-string, where n denotes the number of input ports and d_i denotes the dimension of the i-th input port. To transform this c-string into a `double` array, it is sent to Scilab in order to be interpreted there. For this purpose the function `getScilabVector` in line 9 is called. However, this function needs to know the dimension of the vector in advance. Therefore, in line 6 a string of the form `length(`$[d_1, d_2, \ldots, d_n]$`)` is created which is then also interpreted by Scilab using the command `getScilabVar` which returns the dimension n of the vector. Having the vector's dimension, it can be transformed into a `double` array and stored in `vec_size_d`. Notice that Scilab does not only transform the vector from its c-string form into a `double` array but also interprets any arithmetic expressions that appear in the c-string vector.

After reading the dimensions of input ports from the `parameters` list, the corresponding entries of `in_array`, which are instances of `InOutputVector`, are resized accordingly. If the size of the vector entered as parameter is smaller than the number of input ports, the vector simply is repeated periodically until all input port dimensions are configured. Thus, when all input ports have the same dimension d, it suffices to enter $[d]$ in the Parameters dialogue.

After the sizes of the input ports are read and assigned, exactly the same is done for the output ports in the lines 17 to 28.

The last thing that is new compared to former presented init functions are the last five lines of code. They simply perform the reading of the example parameter values out of `parameters`, interpret them in Scilab and store the results in the corresponding variables. This again is the place, where the modeller using the Atomic PDEVS template is supposed to enter state and parameter initialisations.

```
1 if(out_array.untreated_entry_changes==0 && in_array.untreated_entry_changes
    ==0) {
2   if(flag=='i') {
3   // modeller's input area:
4     // s = delta_int(s_old)
5     // sigma_n = ...;
6   } else if(flag=='e') {
7   // modeller's input area:
8     // s = delta_ext(s_old,in_array)
9     // sigma_n = ...;
10  } else if(flag=='c') {
11  // modeller's input area:
```

```
12      // s = delta_conf(s_old,in_array)
13      // sigma_n = ...;
14    } if(flag=='n') {
15    // modeller's input area:
16      // restore old state: s = s_old;
17      sigma_n = sigma_n_bk;
18    }
19    sigma = sigma_n;
20  }
```

Listing 4.40: The PowerDEVS internal transition function for the generic Atomic PDEVS block.

Listing 4.40 shows the internal transition function. The working principle is exactly the same as it was in the queue example in section 4.2.1. For each case `flag==i`, `flag==e` and `flag==c` the modeller has to fill in the source code describing the corresponding δ function and the state changes it calculates. To check whether has been received at a specific input port `port` with a specific index `index` the statement `if(in_array[port][index].last_change_time==t)` can be used. To access the received message which for example is of type `QSSDoubleArray` a statement of the form `QSSDoubleArray da=(QSSDoubleArray*)in_array.treatAt(port,index);` can be used. `da` then stores the arrived `double` array whose `i`-th entry can simply be accessed by `da[i]`.

In any case, somewhere in each δ function also the time to the next internal transition has to be calculated and stored in `simga_n`. In case of δ_{ext} though, if `simga_n` is not assigned with any new value, the time of the next internal event will simply stay the same that it was before the external transition occurred.

```
1  if(tl<t) { // if first event at current time
2    flag = 'e';
3    sigma_n = INF;
4    if(sigma != INF) {
5      sigma_n = sigma-e;
6    }
7    sigma_n_bk = sigma_n;
8    if(sigma_n == 0) {
9      flag = 'c';
10   }
11   in_array.current_values = 0;
12 }
13 DEVSMessage *input_msg = (DEVSMessage*)x.value;
14 if(input_msg->retrieve == true) {
15   in_array.restoreBackupAt(x.port,input_msg->index);
16   if(in_array.current_values == 0) {
17     if(flag=='c') {
18       flag = 'i';
19     } else if (flag=='e'){
20       flag = 'w';
21     }
22     sigma=0;
```

```
23    }
24  } else {
25    if(false == in_array[x.port][input_msg->index].isEqualTo(input_msg,t)) {
26      in_array.setAtbk(x.port,input_msg->index,input_msg,t);
27      if(flag=='i') {
28        flag = 'c';
29      }
30      sigma = 0;
31    }
32  }
```

Listing 4.41: The PowerDEVS external transition function for the generic Atomic PDEVS block.

Listing 4.41 shows the external transition function of the Atomic PDEVS block. Apart from the fact that beside the input port, now also the index at which an input message arrived has to be considered, there is nothing new here.

For the modeller using the Atomic PDEVS block as template, there is also nothing to be changed in the external transition function.

```
1  if(tl<t) { // if first event at current time
2    flag = 'i';
3    sigma = 0;
4    in_array.current_values = 0;
5  }
6  if(last_event_time < t) { // backup old state, apply state changes
7    last_event_time = t;
8    // modeller's input area
9    // backup state here: s_old = s;
10 }
11 if(out_array.untreated_entry_changes == 0 || flag=='w') {
12   if(flag=='i') {
13     // modeller's input area:
14     // create outputs in case of a pure internal event
15       // out_array = lambda_int(s_old)
16       // use method out_array.setAt(int port,int index,DEVSMessage *value,
             double time);
17   } else if(flag=='e') {
18     // modeller's input area:
19     // create outputs in case of a pure external event
20       // out_array = lambda_ext(s_old, in_array)
21       // use method out_array.setAt(int port,int index,DEVSMessage *value,
             double time);
22   } else if(flag=='c') {
23     // modeller's input area:
24     // create outputs in case of a confluent event
25       // out_array = lambda_conf(s_old, in_array)
26       // use method out_array.setAt(int port,int index,DEVSMessage *value,
             double time);
27   }
28   if(flag=='w') {
```

```
29      flag = 'n';
30    }
31    for(outport=0; outport<out_array.size(); outport++) {
32      for(outindex=0; outindex<out_array[outport].size(); outindex++) {
33        if(out_array[outport][outindex].already_treated == true) {
34          if(out_array[outport][outindex].last_change_time == t) {
35            out_array[outport][outindex].last_change_time = t - 1;
36            out_array[outport][outindex].msgPtr->retrieve = true;
37            out_array.setUntreatedAt(outport,outindex);
38          } else {
39            out_array[outport][outindex].msgPtr->retrieve = false;
40          }
41        }
42      }
43    }
44    in_array.treatAll();
45    outport = -1;
46    outindex = -1;
47  }
48  while(out_array.untreated_entry_changes>0) {
49    outport++;
50    while(out_array[outport].untreated_entry_changes>0) {
51    outindex++;
52      if(false == out_array[outport][outindex].already_treated) {
53        return Event(out_array.treatAt(outport,outindex),outport);
54      }
55    }
56    outindex=-1;
57  }
58  return Event();
```

Listing 4.42: The PowerDEVS output function for the generic Atomic PDEVS block.

Listing 4.42 shows the output function of the Atomic PDEVS block. Its structure is again the same as it has already been in the queue example in section 4.2.1. The only difference is that the for loop (line 31) being responsible for marking retrieve outputs, as well as the while loop (line 48) being responsible for outputting of pending output messages, are each realised as double loops. This is because now not only the output ports have to be iterated but also their indices.

There are two areas where the modeller has to insert code. The first is the place where all defined state variables have to be backed up at the very beginning of every event treatment in line 9. The second area consists of three subareas for the three cases `flag=='i'` (pure internal event), `flag=='e'` (pure external event), and `flag=='c'` (confluent event) for which λ has to be calculated (see lines 12 to 27).

It is very important to use the backup values of the state, made in line 9, for the calculation of the output messages and also for the calculation of the new state in the internal transition function. Otherwise, if due to feedback a recalculation becomes necessary, a state inconsistency would be the consequence. To give an example, Listing 4.43 shows how to cause an output of the `double` value `3.1415` with index `3` at output port `1`.

```
1 QSSDoubleArray da (3.1415);
2 out_array.setAt(1,3,&da,t);
```

Listing 4.43: An example of how to create an output message with index 3 at output port 1 of the Atomic PDEVS block.

CHAPTER 5

Implementation of DEV&DESS Models

5.1 Motivation

In the BaMa project a formalism is needed to describe hybrid system models simulator independently. For this purpose DEV&DESS is perfectly suited. For a better understanding a particular example of a baking oven depicted in Figure 5.1 is given. The goal is to model the following

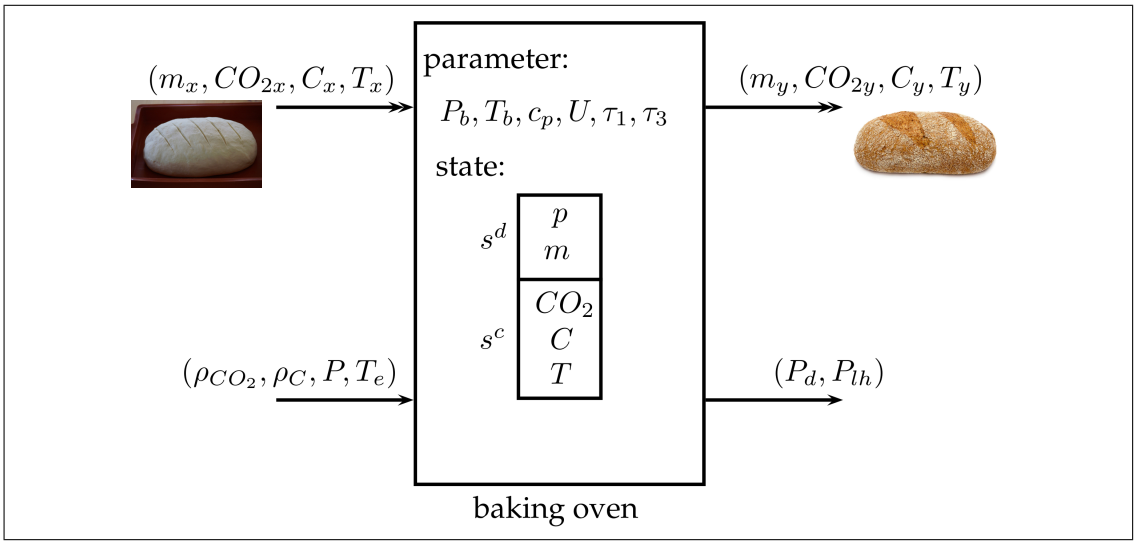


Figure 5.1: Illustration of the hybrid baking oven example. Pastes enter the oven, start a sequence of processes in it, and finally leave the oven as bread.

behaviour of the oven:

p=0: When there is currently no paste in the oven, the process phase p is zero.

p=1: As soon as a paste entity arrives at the first input port, the oven switches into phase $p = 1$, where it remains for the time τ_1. Each arriving paste posseses, among others, the attribute temperature T_x. While the baking oven resides in phase one, there is a heat exchange taking place between the paste with initial temperature T_x and the environment with temperature T_e.

p=2: After the period τ_1 has elapsed, the baking phase $p = 2$ is started. During the baking phase a constant amount of power P to heat the oven is consumed. This power is afflicted with a certain amount of CO_2 per Watt and second ρ_{CO_2} (CO_2 density) which has been released into the atmosphere while generating it. Similarly, the power consumption also produces costs C per Watt and second ρ_C. During the baking phase, the amount of produced CO_2 and costs C is calculated by integrating the two terms $\rho_{CO_2} \cdot P$ and $\rho_C \cdot P$ over time. Further, it is assumed that all the power being consumed directly flows into the heating of the paste and thus leads to a rise of the paste's temperature T. The baking phase is finished as soon as the temperature T reaches a certain threshold T_b.

p=3: After the baking phase is over, the phase $p = 3$ is entered and kept for the duration τ_3. During that phase the bread exchanges heat with the environment again.

p=4: After τ_3 the bread entity with increased values of CO_2 and costs C ,and with a changed value of its temperature T is output and the oven switches back into phase zero.

The heat exchange between the paste/bread and the environment in process phases one and three is described by the following differential equation:

$$\dot{T}(t) = \frac{d}{dt}T(t) = \frac{U \cdot (T(t) - T_e)}{m \cdot c_p} \tag{5.1}$$

where U (thermal transmittance) and c_p (heat capacity) denote thermal parameters.

As in the BaMa project the energy consumption of a whole factory is of interest, it is important to take the heat loss of an oven to its environment, which is possibly air-conditioned, into account. This is what the output P_{lh} is for. Further, the CO_2 density of the delivered energy may depend on the overall energy consumption of the factory, for example if a part of it can be covered by a photovoltaic facility. Therefore, each power consuming device has to report its current power demands. For this purpose the output P_d of the baking oven is used. When everything works fine, the power P delivered should be equal to the power P_d demanded.

A DEV&DESS formulation of the baking oven model looks like the following:

$$\begin{aligned}
X^{discr} &= Y^{discr} = \mathbb{R}_0^+ \times \mathbb{R} \times \mathbb{R}_0^+ \times \mathbb{R}_0^+ \\
X^{cont} &= \mathbb{R}_0^+ \times \mathbb{R}_0^+ \times \mathbb{R}_0^+ \times \mathbb{R} \\
Y^{cont} &= \mathbb{R}_0^+ \times \mathbb{R}_0^+ \times \mathbb{R}_0^+ \times \mathbb{R} \\
S^{discr} &= \{0, 1, 2, 3\} \times \mathbb{R}_0^+ \\
S^{cont} &= \mathbb{R}_0^+ \times \mathbb{R}_0^+ \times \mathbb{R}
\end{aligned}$$

In the following, arriving entities $[m_x, T_x, CO_{2x}, C_x]$ will be referred to as x^d and leaving entities $[m_y, T_y, CO_{2y}, C_y]$ will be referred to as y^d. The continuous input signals ρ_{CO_2}, ρ_C, P, and T_e will be combined to x^c and the continuous output signals P_d and P_{lh} will be combined to y^c. Further, the discrete state variables p and m will be combined to s^d as well as the continuous state variables CO_2, C, and T will be combined to s^c.

$$[s_d, s_c] = \delta_{ext}(s^d, s^c, e, x^c, x^d) = \begin{cases} [1, m_x, CO_{2x}, C_x, T_x] & \text{if } p = 0 \\ [s^d, s^c] & \text{if } p \neq 0 \end{cases}$$

$$C_{int}(s^d, s^c, e, x^c) = \begin{cases} e > \tau_1 & \text{if } p = 1 \\ T \geq T_b & \text{if } p = 2 \\ e > \tau_3 & \text{if } p = 3 \\ false & \text{else} \end{cases}$$

$$[y^d] = \lambda^{discr}(s^d, s^c, x^c) = \begin{cases} [m, CO_2, C, T] & \text{if } p = 3 \\ \emptyset & \text{else} \end{cases}$$

$$[s_d, s_c] = \delta_{int}(s^d, s^c, x^c) = \begin{cases} [2, m, CO_2, C, T] & \text{if } p = 1 \\ [3, m, CO_2, C, T] & \text{if } p = 2 \\ [0, 0, 0, 0, 0] & \text{if } p = 3 \\ [s^d, s^c] & \text{else} \end{cases}$$

$$[y^c] = \lambda^{cont}(s^d, s^c, e, x^c) = \begin{cases} [0, U \cdot (T - T_e)] & \text{if } p = 1 \ \vee \ p = 3 \\ [P_b, 0] & \text{if } p = 2 \\ [0, 0] & \text{else} \end{cases}$$

$$[\dot{CO}_2, \dot{C}, \dot{T}] = f(s^d, s^c, e, x^c) = \begin{cases} [0, 0, 0] & \text{if } p = 0 \\ [\rho_{CO_2} \cdot P, \rho_C \cdot P, \frac{U \cdot (T_e - T)}{m \cdot c_p}] & \text{if } p = 1 \ \vee \ p = 3 \\ [\rho_{CO_2} \cdot P, \rho_C \cdot P, \frac{P}{m \cdot c_p}] & \text{if } p = 2 \end{cases}$$

In the BaMa project one of the first major goals is to define a method for formally describing hybrid systems in production processes. Although it is not asked for explicitly, investigation about the implementability and simulatability of models described with that method are necessary.

Models formulated with DEV&DESS consist of a DEVS and DESS part. As PowerDEVS naturally can simulate DEVS models and additionally is specialised to also simulate continuous models more or less accurately approximated by their quantised versions, it seems to be suitable for implementing combinations of DEVS and DESS models. However, so far there is nothing like an atomic DEV&DESS library block in PowerDEVS to directly implement a model formulated as DEV&DESS. As PowerDEVS actually can only simulate pure DEVS, a general method is needed to embed the DEV&DESS into DEVS. Zeigler proposed such a method in [**?**] which serves as basis for the embedding of DEV&DESS in PowerDEVS presented in the following.

5.2 Concept

PowerDEVS library already provides basic hybrid blocks such as switches and zero-crossing-detection blocks. The first can be used to implement case distinctions as they appear in the definition of f and λ^{cont} in a block diagram. The latter ones produce an output whenever the input signal crosses a specified value and thus can be used to implement state event localisation.

So the idea is to divide a DEV&DESS into a continuous (DESS) part that can be implemented graphically as block diagram and into a discrete part that can be implemented as atomic DEVS block. For this purpose the function C_{int} is also divided into two parts: one for detecting state events and one for scheduling time events. The first one is a component of the continuous part and the second one is included in the atomic DEVS definition. Figure 5.2 shows a graphical illustration of this separation. The continuous part consists of λ^{cont}, f, an integrator, and of the

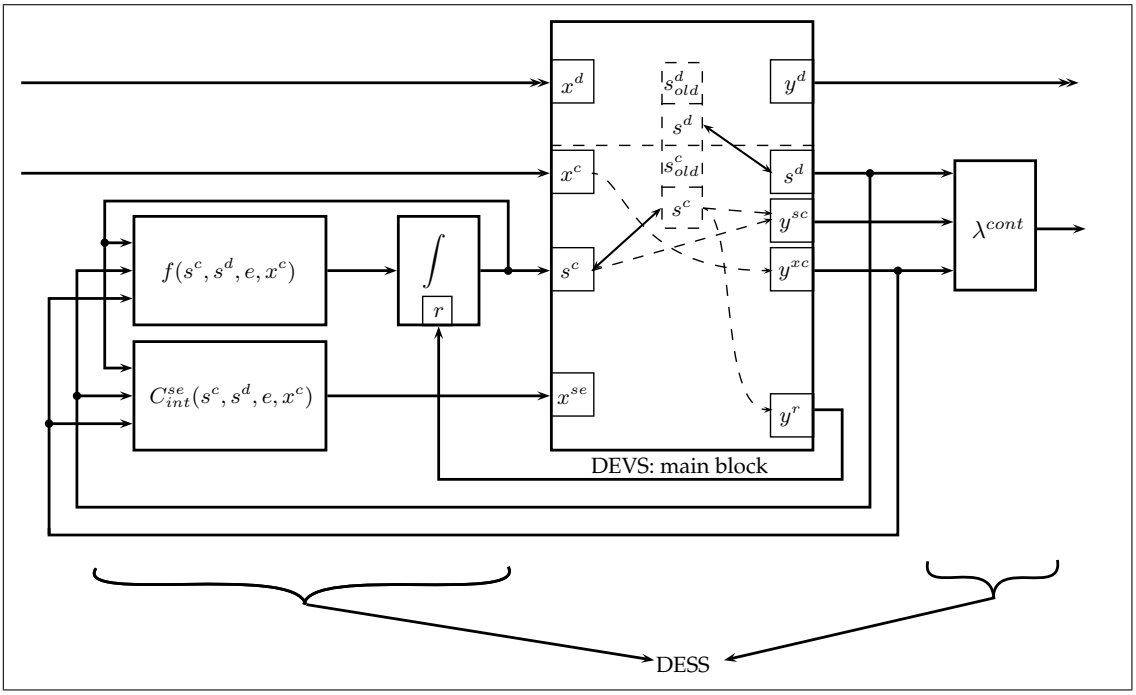


Figure 5.2: The concept of how to embed DEV&DESS in DEVS.

state event part C^{se}_{int} of C_{int}. The discrete part, responsible for the implementation of δ_{ext}, δ_{int}, λ^{discr}, and the time event part C^{te}_{int} of C_{int} is realized as one single DEVS, in the following referred to as *main block*. As it calculates δ_{ext} and δ_{int} it has to administer the entire state of the system consisting of s^d and s^c.

For the calculation of δ_{ext}, the discrete input x^d is needed, and thus, it is connected to the first input port of the main block. Further, both δ_{ext} and δ_{int} as well as λ^{discr} depend on the current value of the continuous input x^c, hence, it is coupled to the second input of the main block. Although s^c is calculated by the main block in each call of one of the δ functions, due to its continuous nature, its value may also change between two such calls, where it is calculated by the integrator block. Thus, the output of the integrator is coupled to the third input port s^c of

the main block. The reason why this input port has exactly the same name as the discrete state itself is that its input buffer is used for storing the input messages and for storing the continuous state at the same time as they always have the same value anyway. The fourth input port of the main block is coupled to the output of the C_{int}^{se} block. Therefore, state events in the original DEV&DESS become external events at the fourth input port of the main block.

The output ports of the main block have the following tasks:
The first output port is directly coupled to the discrete output y^d of the whole DEV&DESS. Therefore, the results of the calculation of λ^{discr} are output there. The second output port s^d transmits the discrete state of the system to the continuous output function block as λ^c depends on it. The reason why it is named exactly like the discrete state itself is the same why the fourth input port is named s^c. The third output port y^{sc} supplies λ^c with the continuous part of the state. The fourth output port y^{xc} actually represents a more or less direct feed through of the input x^c. However, if x^c has changed several times before the main block calculates λ, only its last value is output avoiding unnecessary input events at the λ^c block. The last output port y^r is used to reset the integrator. This is done every time a new state has been calculated by one of the δ functions as well as every time when those state changes need to be withdrawn due to a repetition of the output loop. Fortunately in the PowerDEVS library there already exists such an integrator block with a reset input port.

The main block is designed on basis of the Atomic PDEVS block and thus first gathers all input messages originating at blocks with higher priority and then treats them all at once in one call of the appropriate δ function after all output messages have been calculated by λ and output. Thus, it is important that the blocks for the calculation of f and for the calculation of C_{int} as well as the integrator block have higher priority than the main block. Otherwise the main block would already calculate its new state and produce output messages before it has received current inputs from the continuous part and thus would have to repeat the calculations unnecessarily. The block responsible for calculating λ^{cont} though, should have lower priority than the main block because thus, it will also have received all current inputs before it calculates its output messages.

5.3 Problem Identification and Solution Approaches

5.3.1 Non - PDEVS Blocks Used in the Continuous Part

The DESS part is quantised, i.e. all continuous signals are QSS signals. Therefore, from outside, the coupled system depicted in Figure 5.2 again behaves like an atomic DEVS. As elaborated in the preceding sections, there occur some problems when creating coupled DEVS models. To counter these problems the Atomic PDEVS block has been developed in section 4.1. Thus, it is intended to design the embedded DEV&DESS block in way that, from outside, it looks like an Atomic PDEVS block. This goal can only be achieved if all blocks used in the continuous part are based on the Atomic PDEVS block as they need to be able to handle retrieve messages and to recalculate their inner state in a consistent way after messages originating from feedbacks have arrived. However, the existing PowerDEVS library does not support message retrieving and consistent state recalculation and that is why none of the continuous blocks in Figure 5.2

is directly connected to the DEV&DESS input x^c. They all obtain the continuous input signal from the output port y^{xc} of the main block. The main block does not send any retrieve messages at other output ports than the first one. Nevertheless, it is necessary to treat output messages that have been sent at one of the other output ports in a former repetition of the output loop but they are not scheduled to be resent. The solution is to resend the corresponding backup that has been made for the according output port instead of a retrieve message. In the case of y^{xc}, a backup at the output port itself is made when it is changed for the first time at the current simulation time. In the case of the state output ports s^d and y^{sc} the backup states s^d_{old} and s^c_{old} can be used. At the reset output port y^r also s^c is sent thus the backup s^c_{old} can be used there as well.

As y^{sx} and y^{sc} are QSS signals though, it may be necessary to advance the expansion point of the corresponding Taylor polynomial to the current point in time before sending the backup value. This is because a QSS signal of order higher than one corresponds to a Taylor polynomial with a particular expansion point in time. Thus, when the same QSS signal is resent later in time, it is not valid anymore.

The method of resending updated signals instead of retrieve messages though only works with continuous signals as they exist in any point in time anyway. With discrete signals, i.e. with messages, it is not the same to resend an old message instead of sending no message at all. As long as the continues systems that are fed are pure functional blocks, i.e. mealy type blocks with no internal state, everything works fine, but as soon as they have internal states inconsistencies may be created.

One block of the continuous part of DEV&DESS that has an internal state is the integrator. However, as it is reset by the main block, each time s^c changes discontinuously, it does not cause any troubles. The block for calculating f and λ^c can be generally described as functional block with case distinction. If the case distinction is programmed in a way that all output signals are resent every time the state has been changed, there should not be a problem with f and λ^c either.

The block C^{se}_{int} though may cause problems as it may have an internal state. The sending of messages that would need to be retrieved by C^{se}_{int} does not make much troubles as the only receiver is the main block that can be programmed to be able to deal with that. However, C^{se}_{int} should be able to recalculate its output in a consistent way, i.e. on basis of its initial inner state of the current simulation time and not on basis of a state value that has already been calculated at current simulation time. This is because whenever a repetition of the output loop of the main block is necessary also a recalculation of C^{se}_{int} needs to be done. So the simple solution would be to construct C^{se}_{int} using only pure functional blocks and blocks that are based on the Atomic PDEVS block.

5.3.2 State Events

At a particular instant of time, during the calculation of the new state, not only the state consisting of s^d and s^c may change several times but also the value of the continuous input signal x^c. Thus, also the output of the C^{se}_{int} block may change several times. So the question arises, at which time the value C^{se}_{int} is valid and able to trigger an internal event.

One cause for temporal inconsistencies are vectorial signals which all signals in Figure 5.2 are supposed to be. The reason for this is that if several indices need to be changed, they are changed one after the other. So from the point in time after the first new index value has been

sent to the point in time before the last new index value is sent, the current vectorial signal on the coupling line is wrong. Moreover, due to the fact that the continuous blocks left from the main block have higher priority than the main block, each single index change of a vectorial signal conducted by the main block is immediately interpreted by them.

The way to handle this problem is to give the block C_{int}^{se} the lowest priority among the continuous blocks on the left of the main block and to discard every input at the state event input port x^{se} as soon as there arrives a new input message at x^c or at s^c and each time before an output is sent at s^d. Due to the low priority of C_{int}^{se}, if the internal event still needs to be triggered with the new value of x^c, s^c, and s^d, also a new value at x^{se} will arrive after the arrival of x^c and s^c and after the output at output port s^d.

The second cause for temporal inconsistencies of course is a repetition of output loop in the main block and the recalculation of its output connected to it.

Therefore, every time a new repetition of the output loop is started, before the new output and state is calculated the state values that are delivered to the continuous blocks at the output ports s^d and y^r need to be reset to the initial state of the current simulation time backed up in s_{old}^d and s_{old}^c. Additionally, the output y^{xc} has to be updated to the current value of x^c. This assures that the value of C_{int}^{se} is calculated from s_{old}^d, s_{old}^c, and from the current value of x^c and not from a temporal invalid state and input value.

As value for the backup s_{old}^c of s^c, of course, the value at the input port s^c is used that has been stored there, when the first external transition is triggered in the main block. However, when considering the situation at which the integrator produces an output because the difference between the current value of its internal polynomial and the current value of its output reaches the quantum, the question arises whether this output should be used as s_{old}^c, or the former value.

To decide that, a special situation is looked at. It is assumed that at a particular instant of time there is no other input message at the input ports x^d and x^c but a change of the integrator's output. Further, it is assumed that C_{int}^{se} triggers a state event caused by the change in the output value of the integrator and thus in s^c. As consequence $\delta_{int}(s_{old}^d, s_{old}^c, x^c)$ is calculated. If for s_{old}^c, the value of s^c before the output change of the integrator is used, δ_{int} is calculated using a value s_{old}^c with which actually no state event should have been triggered at all. Thus, if there arrives an input at the continuous state input port s^c of the main block before the main block has produced any output messages at the current instant of time, this input changes s_{old}^c. Since the integrator has higher priority than the main block, a change in the integrator's output due to an internal event in the integrator block is always produced before the main block is allowed to execute its output function for the first time.

Another special situation that needs to be considered concerning state events occurs when after an execution of one of the δ functions the new state value immediately triggers a state event. This state event would be discarded, as it has not been calculated using the old state values s_{old}^d and s_{old}^c. Therefore, it is prohibited for the δ functions to return a state that immediately leads to a state event. As a δ function has been called anyway, during this call the state event that would be caused could have been treated there as well and thus, it could lead to the state that would be the result of an additional internal transition due to the state event right away.

5.3.3 Execution Order in the Main Block

The usual execution order in a DEVS or in a P-DEVS in case of an internal or of a confluent event is to first call λ and then call δ_{int} or δ_{conf}. The task of λ is to calculate and output all output messages that are to be produced due to the current event. The task of the δ function is to calculate the new inner state. With DEV&DESS though, the new state is already needed to calculate the output signals, as it is part of the output. Therefore, in the main block the order of execution of λ and δ needs to be reversed.

To achieve such a behaviour with a formally correct DEVS, it has to fulfil the following: Each non-transitional state of the DEVS as well as each state resulting from an external transition produces no output. Therefore at each event in the first call of λ no output is produced. Afterwards a transitional state is calculated by the corresponding δ function for which λ produces the desired output. Finally the δ function calculates a new non-transitional state.

However, when programming this behaviour in PowerDEVS, the calculation of the δ function is simply shifted into the output function.

5.4 Implementation

5.4.1 Continuous Part

Concerning the implementation of the DEV&DESS in PowerDEVS, Figure 5.2 together with section 5.3.1 tells already everything that is needed for a modeller to construct the continuous part. The specific internal structure of the blocks for f, C_{int}^{se} and λ^c is fully depending on the particular model that is to be implemented. An example is presented in section 5.6.

However, there is very useful mechanism in PowerDEVS to hand on block parameters of a coupled block to its child blocks. When defining custom block parameters in the block's 'Edit' dialogue, each parameter has to be assigned a name. This name in combination with a preceding '%' can then be entered into a parameter value field of any child block.

A DEV&DESS as depicted in Figure 5.2 corresponds to a coupled model in PowerDEVS which, one hierarchical level above, is represented by a single coupled block with two input and two output ports. Therefore, block parameters can be defined for this coupled block that then are accessible in every sub block, i.e. in the main block and in the diverse continuous blocks. In Figure 5.5 it is depicted how the parameter 'in_vec_sizes' of the coupled block is entered as '%in_vec_sizes' in the Parameters dialogue of the block 'calc sizes and store in Workspace'.

5.4.2 The Main Block

The working principle of the main block is based on the Atomic PDEVS block (see section 4.3). However, only the first input port and the first output port show the behaviour of an Atomic PDEVS block. All other ports need special treatment.

In an Atomic PDEVS block, every new input message leads to an execution of one of the δ functions and thus to a calculation of a new state of the block. In the main block of a DEV&DESS though, only the discrete input port x^d and, as explained in section 5.3.2 in some cases also the state event input port x^{se} are capable of that. The input ports x^c and s^c describe

continuous signals that, theoretically, may change all the time. However, as they are quantised, they only change when their polynomial description deviates more than the quantum from the real value or from a more exact approximation of their real value. Although such signal update changes do not cause a state recalculation in the main block, they have to call the output function, as their changes need to be forwarded to the continuous output function λ^c and thus to the outputs y^{sc} and y^{xc}. In the following, an event in the main block which causes an execution of λ and of one of the δ functions will be referred to as *discrete event*, whereas an event due to changes in one of the two continuous inputs x^c and s^c will be referred to as *continuous event*. Of course, it is also possible that there occur changes in the continuous inputs simultaneously with a discrete event. The three sources for a discrete event are an input at the discrete input port, an input at the state event input port (a state event) and a time event. The sources for a continuous event are changes at the input ports x^c and s^c.

Further there is a third kind of event that occurs, when either a discrete or a continuous event has been triggered by an input message which afterwards is retrieved again. In such a case, the changes that have been applied to the output and to the state need to be withdrawn. This kind of event will be referred to as *withdraw event*.

Another difference of the main block to the Atomic PDEVS block is that there is no retrieve message treatment necessary at the third and fourth input ports. Whereas at the output ports two to five a special retrieve message treatment has to be implemented, as described in section 5.3.1.

The internal state of the main block consists of the discrete part s^d and of the continuous part s^c. Since, between two discrete events, the continuous state is calculated outside the main block and therefore, has to be fed back at an input port, the corresponding input port buffer can be used to store the inputs and at the same time serves as storage for the continuous state itself. The discrete part of the state is needed as output signal, as all the continuous function blocks depend on it. Thus the output buffer for the output port s^d is also used both for storing output messages and for storing the discrete state itself. The backups of s^d and s^c though, have to be stored separately.

The processes in the main block that are carried out in case of an event can be described by six phases: gathering `'g'`, checking `'c'`, resetting `'r'`, calculation of C_{int}^{se} `'C'`, calculation of λ and δ `'l'`, and outputting `'o'`.

'g'|'c': a) *external transition function:*
If this is the first event at that time, set `phase='g'` and backup state.
Gather input messages produced by blocks with higher priority.
Determine the value for `flag` (`'i'`, `'e'` or `'c'`).

b) *output function:*
If this is the first event at that time, set `flag='i'`, $\sigma = 0$ and backup state.
If `phase='g'` change it to `'C'`.
Set all entries of input port s^c to be treated.
If the state s is not equal to s_{old}, mark necessary reset output messages at output ports s^d and y^r and go to `phase 'r'`.
If there are new input messages at x^c, update output port y^{xc} and go to `phase 'r'`.
Otherwise go to `phase 'C'` .

'r': *output function:*
If there are untreated outputs left, x^{se} to be treated and output the next pending output.
Otherwise go to `phase 'C'`.

'C': *output function:*
If there is an untreated change at input port x^{se} recalculate `flag` as follows:
`'n'`$\mapsto$`'I'` and `'e'`$\mapsto$`'C'`
In any case go to `phase 'l'`.

'l': *output function:*
Calculate δ and λ according the value of `flag` and thereby calculate the new state s^d, s^c and the the outputs at x^d.
Change `flag` as follows: `'I'`$\mapsto$`'n'` and `'C'`$\mapsto$`'e'`.
Mark retrieve messages at output port x^d.
Mark state changes to be output at s^d and at y^r.
Set all input messages at x^d and at x^c to be treated.
Go to `phase 'o'`.

'o': a) *output function:*
As long as there are pending outputs, send the next pending output message.
After all messages at output port y^r have been sent, mark all untreated entries of s^c to be sent at y^{sc}.

b) *internal transition function:*
If all outputs are sent and there are no untreated input messages at input ports x^d and x^c, set the new non-zero value for σ and `phase` to `'c'`.

New concerning the range of values of `flag` are the two cases `flag='I'` and `flag='C'`. They also indicate an internal and an confluent transition, respectively but in their case, the cause for the corresponding event is not a time event, but a state event. State events have to be marked separately, as a change in the continuous input x^c may withdraw a state event which is not possible with time events. Although this description is already quite detailed, there are even some more details to be considered. However, for a complete description it is referred to the source code in section 5.5.

5.5 Atomic DEV&DESS Block

The Atomic DEV&DESS block is supposed to be a template for implementing arbitrary DEV&DESS in PowerDEVS. The modeller intended to use the block has to accomplish three tasks, the definition of the continuous part, the definition of the discrete part and the definition of the dimensions of the state space and of the vectorial input and output ports.

The third part is accomplished by entering the corresponding values into the Parameters dialogue of the coupled DEV&DESS block. Figure 5.3 shows this dialogue. The first four text fields serve for the input of the initial discrete state s^d, the initial continuous state s^c, the dimensions of the discrete and the continuous input port x^d and x^c and of the dimensions of

the discrete and continuous output ports y^d and y^c. The other text fields are examples of model specific parameters that can be added by the modeller on demand. Figure 5.4 shows the interior

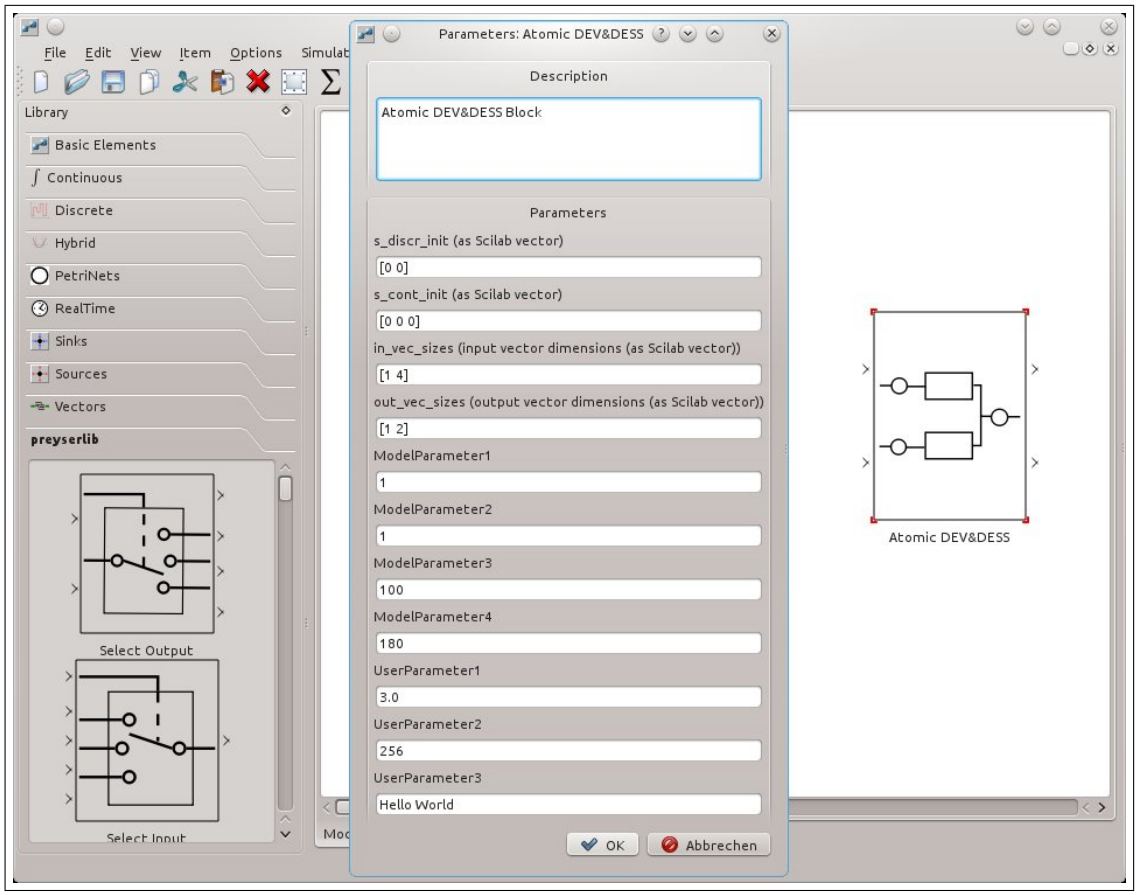


Figure 5.3: The Parameters dialogue of the Atomic DEV&DESS block

of the coupled DEV&DESS block, as well as the priority list bottom right. The only difference to the schematic in Figure 5.2 is the block named ‘calc sizes and store in Workspace’ top left. This block simply is used to extract the single values for state dimensions and input output signal dimensions out of the vectorial values that have been entered as parameter values of the Atomic DEV&DESS block. Figure 5.5 shows the Parameters dialogue of that block and how a parameter of the Atomic DEV&DESS block can be handed on to it. The block stores the single dimension values as variables in the Scilab workspace. This variables, in turn, can be used in all other blocks as inputs for their block parameters. Therefore, the dimension of all the coupling lines is configured automatically using the inputs in the Parameters dialogue of the Atomic DEV&DESS block.

The second task for the modeller, the creation of the continuous part, is accomplished by graphically building the interior of the coupled sub blocks for f, C_{int}^{se}, and λ^c with the help of the PowerDEVS library. Figure 5.6 shows the interior of the block ‘f(q,x_cont)’, representing the right hand side of the ODE that describes the dynamics of the continuous part. As it can be seen,

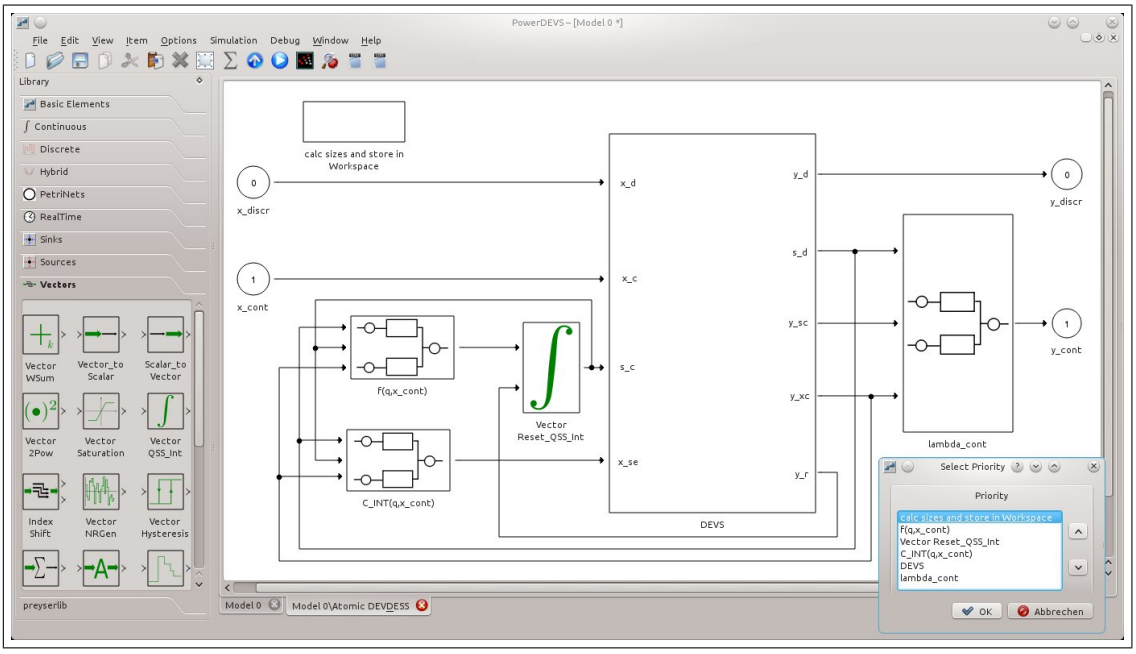


Figure 5.4: The coupled PowerDEVS model representing an Atomic DEV&DESS.

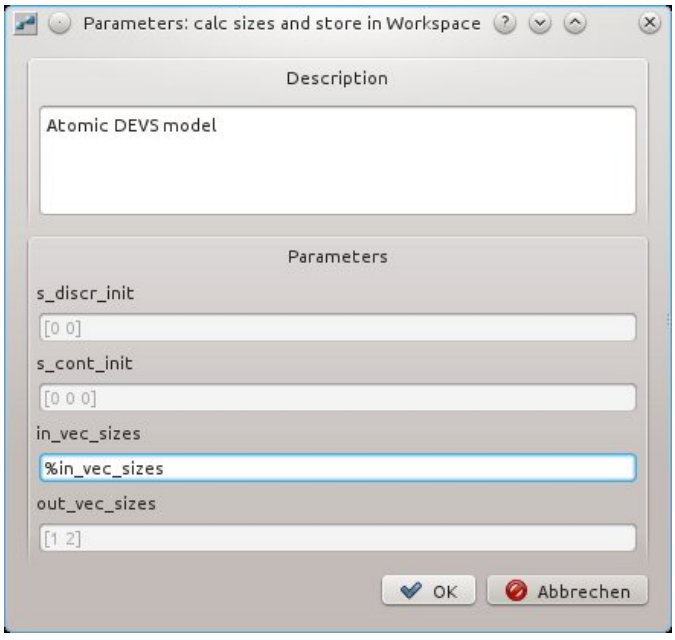


Figure 5.5: The Paramters dialogue of a sub block of the Atomic DEV&DESS PowerDEVS block.

there are three cases in the model, ‘f0’, ‘f1’ and ‘f2’ which again are coupled models. Since the model in the figure belongs to the DEV&DESS of the baking oven introduced at the beginning

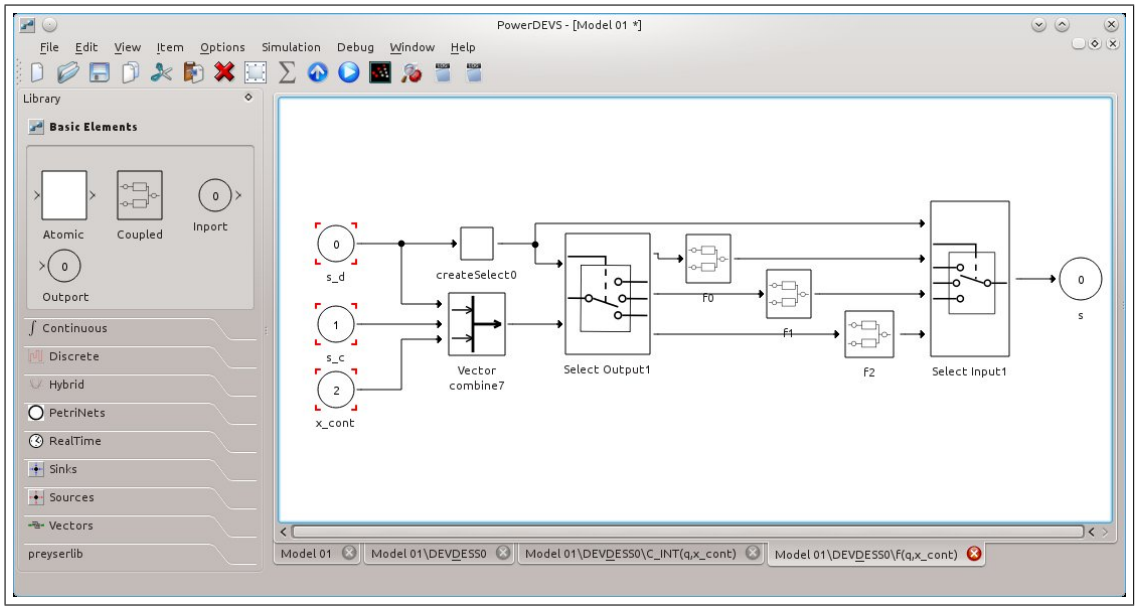


Figure 5.6: The coupled model representing the left side of the Differential equation $f(s^d, s^c, x^c)$.

of this chapter, the three cases correspond to the three cases in the definition of f there. Figure 5.7 shows the implementation of 'f2'. The 'Switch' and the 'check boolean expression' block around the 'Inverse' block make sure that the output of the inverse block is not forwarded if its input is zero. Figure 5.8 shows the interior of the block implementing C^{se}_{int}, again for the baking oven example. In this example, whether a state event is possible or not depends on the value of p which is part of the discrete state s^d. The discrete state enters the coupled model at input port 0. As it is vectorial, the specific index at which p is transported is selected in the first block after the input port. Then a select signal is created of it which controls which other input of the 'Select Input' block is forwarded to its single output port. If the select signal is zero, a constant zero signal is forwarded. If the select signal is one, the output of the 'Cross detect' block is forwarded. As the 'Select Input' block always immediately outputs the value it received last at the selected port whenever the selected input port changes, it is necessary to immediately set the input signal at the second input port back to zero after a state event has been reported. Otherwise the state event wrongly would be reported again after the third input port of the 'Select Input' block is reselected. This is what the block 'output on trigger' is for.

After the continuous part has been built, the last task is to fill out the modeller's input areas in the source code of the main block. Due to the fact that the calculation of λ and of δ need to be done simultaneously for the main block (see section 5.3.3), there are only two places in the whole source code to be filled out by the modeller. The first is the part for reading custom block parameter values which is located in the init function and the second part is the definition of λ and of δ located in the output function. Nevertheless, to complete the picture, the entire source code is presented.

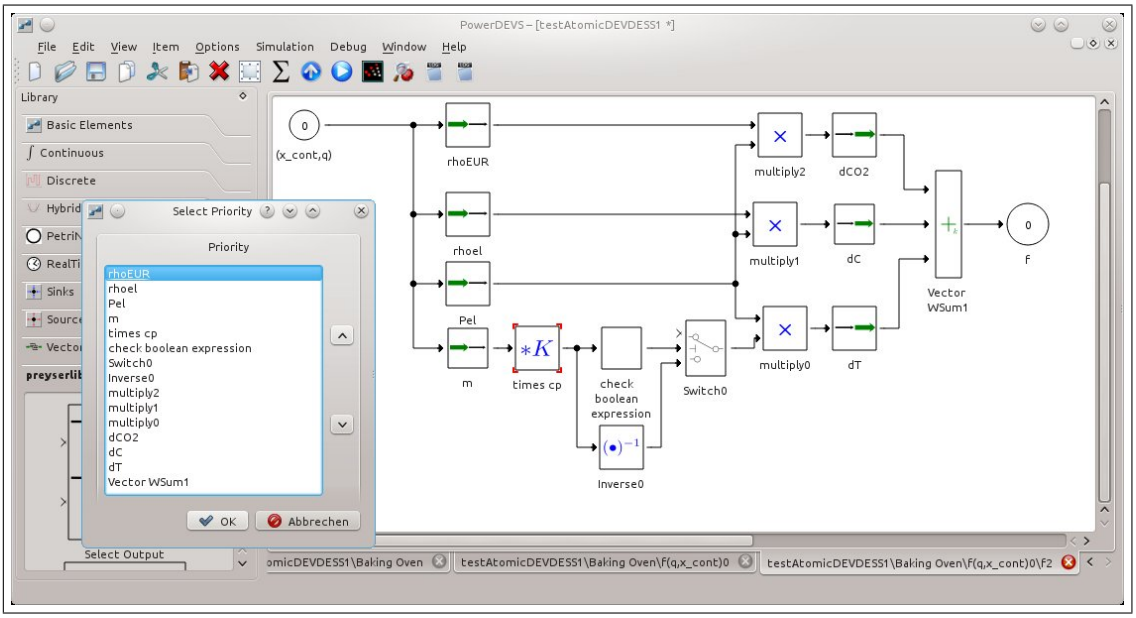


Figure 5.7: The coupled model representing the interior of the block 'f2' of the model depicted in Figure 5.6.

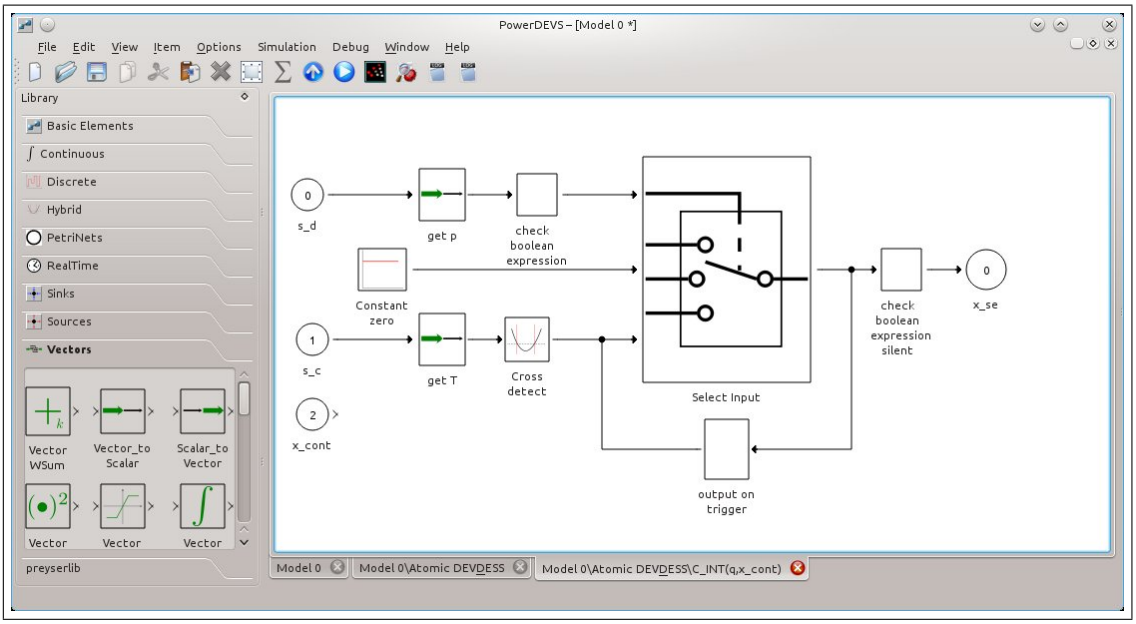


Figure 5.8: The coupled model representing the state event function $C^{se}_{int}(s^d, s^c, x^c)$.

```
InOutputArray in_array; // input messages
InOutputArray out_array;
InOutputVector *s_discr;
InOutputVector *s_cont;
InOutputVector s_discr_old;
InOutputVector s_cont_old;
double sigma, sigma_n, sigma_n_bk;
double e_old;
double last_event_time;
char flag;          // possible values: 'i','e','c','n'
char phase;         // possible values: 'g','C','c','l','o'
int outport, outindex;
#ifndef INF
  #define INF 1e20
#endif
#ifndef DEVDESS_DISCR_INPORT
  #define DEVDESS_DISCR_INPORT 0
  #define DEVDESS_CONT_INPORT 1
  #define DEVDESS_SC_INPORT 2
  #define DEVDESS_SE_INPORT 3
  #define DEVDESS_DISCR_OUTPORT 0
  #define DEVDESS_SD_OUTPORT 1
  #define DEVDESS_SC_OUTPORT 2
  #define DEVDESS_XC_OUTPORT 3
  #define DEVDESS_RESET_OUTPORT 4
#endif
// modeller's input area:
  // define model parameters and auxiliary variables, e.g.:
  double Par1;
  int Par2;
  std::string Aux1;
```

Listing 5.1: The PowerDEVS definitions area for the main block of the generic Atomic DEV&DESS block.

Listing 5.1 shows the definitions area for the main block. The only difference to the definitions area of the Atomic DEVS block are the variables `s_discr`, `s_cont`, `s_discr_old`, and `s_cont_old`. As the name suggests they store the state and its backup. `s_discr` and `s_cont` are pointers of type `InOutputVector` and point to their corresponding output and input port, respectively. The backups in contrast are full instances of `InOutputVector` as they need to store a deep copy of the state.

```
std::string par_var ((char*)va_arg(parameters,char*));
std::string length_str = ("length(" + par_var + ")");
int s_discr_size = (int)getScilabVar(&length_str[0]);
s_discr_old.resize(s_discr_size);
double *s_discr_init = (double*)malloc(s_discr_size*sizeof(double));
getScilabVector(&par_var[0], &s_discr_size, s_discr_init);
// read size of continuous state:
par_var = std::string("_s_cont_size");
int s_cont_size = (int)getScilabVar(&par_var[0]);
```

```
10 s_cont_old.resize(s_cont_size);
11 // read vector-dimensions of inputs (x_discr, x_cont, x_se, s_cont)
12 par_var = std::string((char*)va_arg(parameters, char*));
13 int length = 2;
14 double input_sizes[2];
15 getScilabVector(&par_var[0], &length, input_sizes);
16 in_array.resize(4);
17 in_array[DEVDESS_DISCR_INPORT].resize((int)input_sizes[0]);
18 in_array[DEVDESS_CONT_INPORT].resize((int)input_sizes[1]);
19 in_array[DEVDESS_SC_INPORT].resize(s_cont_size);
20 in_array[DEVDESS_SE_INPORT].resize(1);
21 // initialize output-array
22 par_var = std::string("_y_discr_size");
23 int y_discr_size = (int)getScilabVar(&par_var[0]);
24 out_array.resize(5);
25 out_array[DEVDESS_DISCR_OUTPORT].resize(y_discr_size);
26 out_array[DEVDESS_XC_OUTPORT].resize((int)input_sizes[1]);
27 out_array[DEVDESS_SD_OUTPORT].resize(s_discr_size);
28 out_array[DEVDESS_SC_OUTPORT].resize(s_cont_size);
29 out_array[DEVDESS_RESET_OUTPORT].resize(s_cont_size); // integrator-reset-
     output
30 for(int i=0; i<s_discr_size; i++) { // set initial value for s_d
31   QSSDoubleArray qda (s_discr_init[i]);
32   qda.index = i;
33   out_array.setAt(DEVDESS_SD_OUTPORT,i,&qda,t);
34   s_discr_old.setAt(i,&qda,t);
35 }
36 free(s_discr_init);
37 s_cont = &in_array[DEVDESS_SC_INPORT];
38 s_discr = &out_array[DEVDESS_SD_OUTPORT];
39 sigma = INF;
40 sigma_n = INF;
41 last_event_time = -1;
42 flag = 'n';
43 phase = 'g';
44 outport = -1;
45 outindex = 0;
46 // modeller's input area
47   // read custom parameters, for example:
48 char *fvar = va_arg(parameters,char*);
49 Par1 = (double)getScilabVar(fvar);
50 fvar = va_arg(parameters,char*);
51 Par2 = (int)getScilabVar(fvar);
52 Aux1 = std::string(va_arg(parameters,char*));
```

Listing 5.2: The PowerDEVS init function for the main block of the generic Atomic DEV&DESS block.

Listing 5.2 shows the init function of the main block. In the upper part, the block's parameter values are read and interpreted in Scilab using the same methods as explained at the Atomic PDEVS block in section 4.3. From the parameter values then the sizes of the input and output buffers are determined. Further, the discrete state is initialised. At the end of the init function

the modeller can read custom block parameters and store them in parameter variables defined earlier in the definitions area.

```
if(out_array.untreated_entry_changes==0) {
  if(phase=='o') {
    if(in_array[DEVDESS_DISCR_INPORT].untreated_entry_changes==0 && in_array
        [DEVDESS_CONT_INPORT].untreated_entry_changes==0) {
      sigma = sigma_n;
    }
    phase = 'c';
  }
}
```

Listing 5.3: The PowerDEVS internal transition function for the main block of the generic Atomic DEV&DESS block.

Listing 5.3 shows the internal transition function of the main block. As it can be seen, the only task of it is to resume to a non transitional state by setting `sigma` to the new value `sigma_n` which needs to be calculated by the modeller in the output function. It only resumes though, if all outputs have been produced and in the meanwhile no new input message has arrived. The phase `phase` is set back to `'c'`.

```
if(tl<t) { // if first event at current time
  e_old = e;
  flag = 'n';
  sigma_n = INF;
  if(sigma != INF) {
    sigma_n = sigma-e;
  }
  sigma_n_bk = sigma_n;
  phase='g';
  if(t+sigma_n == t) {
    flag = 'i';
    sigma = 0;
  }
  for(int i=0; i<s_cont_old.size(); i++) {
    s_cont_old.setAt(i,(*s_cont)[i].msgPtr,(*s_cont)[i].last_change_time);
  }
  in_array.treatAt(DEVDESS_SE_INPORT,0);
  in_array[DEVDESS_DISCR_INPORT].current_values = 0;
  in_array[DEVDESS_CONT_INPORT].current_values = 0;
}
DEVSMessage *in_msg = (DEVSMessage*)x.value;
int index = in_msg->index;
if(x.port == DEVDESS_DISCR_INPORT) {
  if(in_msg->retrieve == true) {
    in_array.restoreBackupAt(x.port,index);
    if(in_array[DEVDESS_DISCR_INPORT].current_values == 0) {
      if(flag=='c') {
        flag = 'i';
      } else if (flag=='e'){
```

```
30          sigma_n = sigma_n_bk;
31          flag = 'n';
32        }
33        sigma=0;
34      }
35    } else {
36      if(false == in_array[x.port][index].isEqualTo(in_msg,t)) {
37        in_array.setAtbk(x.port,index,in_msg,t);
38        if(flag=='i') {
39          flag = 'c';
40        } else if(flag!='c') {
41          flag = 'e';
42        }
43        sigma = 0;
44      }
45    }
46  } else if (x.port == DEVDESS_CONT_INPORT) {
47    if(in_msg->retrieve == true) {
48      in_array.treatAt(DEVDESS_SE_INPORT,0);
49      in_array.restoreBackupAt(x.port,index);
50      if(true==out_array[DEVDESS_XC_OUTPORT][index].already_treated) {
51        out_array.restoreBackupAt(DEVDESS_XC_OUTPORT,index);
52        QSSDoubleArray *da = (QSSDoubleArray*)out_array[DEVDESS_XC_OUTPORT][
              index].msgPtr;
53        da->advance_time(t-out_array[DEVDESS_XC_OUTPORT][index].
              last_change_time);
54        out_array[DEVDESS_XC_OUTPORT][index].last_change_time = t;
55        out_array.setUntreatedAt(DEVDESS_XC_OUTPORT,index);
56      } else {
57        out_array.restoreBackupAt(DEVDESS_XC_OUTPORT,index);
58      }
59      sigma=0;
60    } else {
61      if(false == in_array[x.port][index].isEqualTo(in_msg,t)) {
62        in_array.treatAt(DEVDESS_SE_INPORT,0);
63        in_array.setAtbk(x.port,index,in_msg,t);
64        out_array.setAtbk(DEVDESS_XC_OUTPORT,index,in_msg,t);
65        sigma = 0;
66      }
67    }
68  } else if (x.port == DEVDESS_SE_INPORT) {
69      in_array.setAt(x.port,index,in_msg,t);
70      if(phase=='g') {
71        sigma=0;
72      }
73  } if (x.port == DEVDESS_SC_INPORT) {
74    if(false == in_array[x.port][index].isEqualTo(in_msg,t)) {
75      in_array.treatAt(DEVDESS_SE_INPORT,0);
76      if(phase == 'g') {
77        s_cont_old.setAt(index,in_msg,t);
78        out_array.setAt(DEVDESS_SC_OUTPORT,index,in_msg,t);
79      } else if(phase == 'r') {
80        s_cont_old.setAt(index,in_msg,t);
```

```
81        }
82        in_array.setAt(x.port,index,in_msg,t);
83        sigma=0;
84      }
85    }
```

Listing 5.4: The PowerDEVS external transition function for the main block of the generic Atomic DEV&DESS block.

Listing 5.4 shows the external transition function of the main block. The code from line 1 to line 20 is only executed if the call of the external transition function is the first event at the current simulation time. There `flag` and `phase` as well as `sigma_n` are initialised. Further, the continuous state is backed up and x^{se} is set treated to not wrongly trigger a state event due to an input at x^{se} that has been produced after the calculation of λ and δ at the previous event. Finally, the number of current input values is initialised with zero.

The following `if-else` construct beginning in line 23 is responsible for treating each input message according to the input port of its arrival. In the first block starting with line 23 discrete input messages are treated. In the second block starting with line 46 continuous input messages are treated. The special thing about that is the retrieve section. It can be seen, that retrieve messages not only are replaced by their backup, but the backup is advanced in time as it is a QSS signal, and afterwards it is marked for being output at y^{xc} output port.

The third block starting with line 68 treats input messages that origin at the block C_{int}^{se} and thus signal state events. However, such input messages can only lead directly to an internal event, if they arrive before the state has been altered by δ for the first time at the current simulation time. In all other cases the value of x^{se} is checked after every reset phase and if there has been an input after the reset, `flag` is altered accordingly. This is because right after a reset the continuous blocks have just received the backup state $s_o^d ld$ and $s_o^c ld$ and the current input x^c and thus, the value at the input port x^{se} is equal to $C_{int}^{se}(s^d, s^c, x^c)$.

The last block starting with line 73 treats changes at the output of the integrator. If such changes appear before the output function has been executed for the first time (`phase='g'`) or during the reset phase, the backup s_{old}^c of s^c is updated as well. The update during the reset phase is conducted because when resetting a PowerDEVS QSS integrator block which works with QSS of higher order than one, the integrator block does not output exactly the reset value but the reset value and possibly already values for its derivatives. Thus, although reset with s_{old}^c, the value of s^c immediately after the reset may not to agree completely with s_{old}^c. Moreover, changes in `phase='g'` are directly forwarded to the output port y^{sc}.

```
1  if(tl<t) { // if first event at current time
2    flag = 'i';
3    e_old = e;
4    sigma = 0;
5    in_array[DEVDESS_DISCR_INPORT].current_values = 0;
6    in_array[DEVDESS_CONT_INPORT].current_values = 0;
7    for(int i=0; i<s_cont_old.size(); i++) {
8      s_cont_old.setAt(i,(*s_cont)[i].msgPtr,(*s_cont)[i].last_change_time);
9    }
```

```
10 }
11 if(last_event_time < t) { // backup old state
12   last_event_time = t;
13   for(int i=0; i<s_discr_old.size(); i++) {
14     s_discr_old.setAt(i,(*s_discr)[i].msgPtr,(*s_discr)[i].last_change_time)
         ;
15   }
16 }
17 if(phase=='g') {
18   phase = 'c';
19 }
20 if(phase=='c') {
21   s_cont->treatAll();
22   bool state_changed=false;
23   for(int i=0; i<s_discr_old.size();i++) {
24     if((*s_discr)[i].msgPtr->operator!=(*s_discr_old[i].msgPtr)) {
25       state_changed = true;
26       out_array.setAt(DEVDESS_SD_OUTPORT,i,s_discr_old[i].msgPtr,s_cont_old[
           i].last_change_time);
27     }
28   }
29   for(int i=0; i<s_cont_old.size();i++) {
30     if(((*s_cont)[i].msgPtr->getDoublePtr())[0]!= (s_cont_old[i].msgPtr->
         getDoublePtr())[0] || s_cont_old[i].last_change_time!=(*s_cont)[i].
         last_change_time) {
31       state_changed = true;
32       if(s_cont_old[i].last_change_time<t) {
33         ((QSSDoubleArray*)s_cont_old[i].msgPtr)->advance_time(t-s_cont_old[i
             ].last_change_time);
34         s_cont_old[i].last_change_time=t;
35       }
36       out_array.setAt(DEVDESS_RESET_OUTPORT,i,s_cont_old.treatAt(i),t);
37     }
38   }
39   if(out_array[DEVDESS_XC_OUTPORT].untreated_entry_changes>0) {
40     state_changed = true;
41   }
42   if(state_changed==false) {
43     phase = 'C';
44   } else {
45     in_array.treatAt(DEVDESS_SE_INPORT,0);
46     phase = 'r';
47     outport = DEVDESS_RESET_OUTPORT;
48     outindex = -1;
49   }
50 }
51 if(phase=='r') {
52   if(out_array.untreated_entry_changes>0) {
53     in_array.treatAt(DEVDESS_SE_INPORT,0);
54   } else {
55     phase = 'C';
56   }
57 }
```

```
58 if(phase=='C') {
59   if(false==in_array[DEVDESS_SE_INPORT][0].already_treated) {
60     if(flag=='n') flag='I';
61     else if(flag=='e') flag='C';
62   }
63   phase = 'l';
64 }
65 if(phase=='l') {
66   if(flag=='i' || flag=='I') {
67     // modeller's input area:
68     // create outputs in case of a pure internal event
69       // [*s_discr,*s_cont] = delta_int(s_discr_old,s_cont_old,, in_array[
           DEVDESS_CONT_INPORT])
70       // out_array = lambda_int(s_discr_old,s_cont_old, in_array[
           DEVDESS_CONT_INPORT])
71       // use method out_array.setAt(int port,int index,DEVSMessage *value,
           double time);
72     if(flag=='I') flag='i';
73   } else if(flag=='e') {
74     // modeller's input area:
75     // create outputs in case of a pure external event
76       // [*s_discr,*s_cont] = delta_ext(s_discr_old,s_cont_old,e_old,
           in_array[DEVDESS_CONT_INPORT],in_array[DEVDESS_DISCR_INPORT])
77       // out_array = lambda_int(s_discr_old,s_cont_old,in_array[
           DEVDESS_CONT_INPORT],in_array[DEVDESS_DISCR_INPORT])
78       // use method out_array.setAt(int port,int index,DEVSMessage *value,
           double time);
79   } else if(flag=='c' || flag=='C') {
80     // modeller's input area:
81     // create outputs in case of a confluent event
82       // [*s_discr,*s_cont] = delta_ext(s_discr_old,s_cont_old,in_array[
           DEVDESS_CONT_INPORT],in_array[DEVDESS_DISCR_INPORT])
83       // out_array = lambda_int(s_discr_old,s_cont_old,in_array[
           DEVDESS_CONT_INPORT],in_array[DEVDESS_DISCR_INPORT])
84       // use method out_array.setAt(int port,int index,DEVSMessage *value,
           double time);
85     if(flag=='C') flag='c';
86   }
87   for(int i=0; i<out_array[DEVDESS_DISCR_OUTPORT].size(); i++) {
88     if(out_array[DEVDESS_DISCR_OUTPORT][i].already_treated == true) {
89       if(out_array[DEVDESS_DISCR_OUTPORT][i].last_change_time == t) {
90         out_array[DEVDESS_DISCR_OUTPORT][i].last_change_time = t - 1;
91         out_array[DEVDESS_DISCR_OUTPORT][i].msgPtr->retrieve = true;
92         out_array.setUntreatedAt(DEVDESS_DISCR_OUTPORT,i);
93       }
94     }
95   }
96   for(int i=0; i<s_discr_old.size();i++) {
97     if(false==(*s_discr)[i].already_treated) {
98        out_array.untreated_entry_changes++;
99     }
100  }
101  for(int i=0; i<s_cont_old.size();i++) {
```

```
102     if(false == (*s_cont)[i].already_treated) {
103       out_array.setAt(DEVDESS_RESET_OUTPORT,i,(*s_cont)[i].msgPtr,t);
104     }
105   }
106   in_array[DEVDESS_DISCR_INPORT].treatAll();
107   in_array[DEVDESS_CONT_INPORT].treatAll();
108   outport = DEVDESS_RESET_OUTPORT;
109   outindex = -1;
110   phase='o';
111 }
112 if(phase=='o'||phase=='r') {
113   while(out_array.untreated_entry_changes>0) {
114     while(out_array[outport].untreated_entry_changes>0) {
115     outindex++;
116       if(false == out_array[outport][outindex].already_treated) {
117         return Event(out_array.treatAt(outport,outindex),outport);
118       }
119     }
120     outport++;
121     outindex=-1;
122     if(outport > DEVDESS_RESET_OUTPORT) {
123       outport = 0;
124       for(int i=0; i<s_cont_old.size();i++) {
125         if(false == (*s_cont)[i].already_treated) {
126           out_array.setAt(DEVDESS_SC_OUTPORT,i,s_cont->treatAt(i),t);
127         }
128       }
129     }
130   }
131 }
132 return Event();
```

Listing 5.5: The PowerDEVS output function for the main block of the generic Atomic DEV&DESS block.

Listing 5.5 shows the core part, the output function of the main block. In lines 1 to 16, again the initialisations that need to be done including the state backup are accomplished. Then there follows a row of `if` blocks, each corresponding to a particular phase. The first is the gathering phase `'g'`. Its tasks are accomplished in the external transition function and thus, here there is nothing to do but to change `phase` to `'c'`. The checking phase starting in line 20 checks whether the current state in the continuous blocks still is unchanged and whether the actual value of the continuous input is delivered to them. If any of those two points is not satisfied, a reset is conducted and thus the `phase` is changed to `'r'`. If no reset needs to be conducted it directly can be continued with the calculation of C_{int}^{se}, i.e. with the checking of the state event input. For this purpose, `phase` is changed to `'C'` (line 58). There, the value of `flag` is updated if a state event occurred and afterwards it is proceeded with the execution of phase `'l'` in line 65. In phase `'l'` first of all λ and the corresponding δ function is calculated. This is the place where the modeller needs to insert the model specific code. After the calculation of λ and δ, in the `for` loop it is checked whether there are retrieve messages to be sent at the discrete output port y^d. Then, the state values are scanned for changes applied by the formerly executed δ function.

Theses changes are then marked to be output at the reset output y^r and at the corresponding outputs s^d and y^{sc}. Finally all changes at the input ports x^d and x^c are marked to be fully treated before it is changed to the output phase (line 112).

The code for the output phase is used for the reset phase as well, as in both phases the marked output messages need to be generated. The only difference lies in the execution of the internal transition function. In the internal transition function `sigma` can be changed only if `phase='o'`. In the `for` loop at the end of the code of the output phase, the changes applied to s^c are also applied to y^{sc}. This is done here because it has to be waited until all output messages at the reset port y^r are output before s^c is forwarded to y^{sc} as otherwise the reset outputs are changing s^c again.

5.6 Implementation Example: Baking Oven

In this section, it is demonstrated how to implement the baking oven DEV&DESS presented at the beginning of this chapter. Since the baking oven model does not generate any simulations results without any stimulus, it has to be fed with some input signals. In Figure 5.9 the DEV&DESS baking oven block is depicted, coupled to other blocks that generate the desired input signals. On the left in the figure, the Parameters dialogue of the block 'Baking Oven' is shown. It can be seen, that the custom example parameters of the Atomic DEV&DESS template block have been replaced by the parameters defined in the DEV&DESS of the oven. The specific parameter values are chosen to get state trajectories of the single state variables which are approximately in the same range and thus can be plotted in one diagram. They do not have any practical relevance. The first input port of the block 'Baking Oven' is its discrete input port. There the entities of paste are delivered. Two special blocks, 'Entity Source' and 'Entity Sink' have been programmed to create and destroy entities which are instances of the class `Entity` (see Listing A.6 in the appendix). The vectorial continuous input consists of the components ρ_{CO_2}, ρ_C, P and T_e. As it can be seen, ρ_{CO_2} and ρ_C are generated by use of PowerDEVS library blocks that produce periodic and constant signals. As power supply P simply the power demand P_d output by the oven is used. Thus, the demanded power is always equal to the delivered Power. However, this coupling causes a zero time feedback. The environmental temperature T_e is assumed to be constant.

Figure 5.10 shows the interior of the DEV&DESS block 'Baking Oven'. As it can be seen in comparison to Figure 5.4, it cannot be distinguished from the interior of the Atomic DEV&DESS indicating that this structure stays the same for different DEV&DESS models. The two additional blocks right bottom are used to plot the state trajectory during simulation. The Parameters dialogue on the left belongs to the main block. There, the values for the parameters τ_1 and τ_3 are entered as `%tau1` and `%tau3`, since `tau1` and `tau3` are parameters of the coupled block 'Baking Oven'.

5.6.1 Continuous Part

The continuous part of the DEV&DESS consists of f, C_{int}^{se} and λ^{cont}. f is implemented as coupled block 'f(q,x_cont)'. Its interior is depicted in Figure 5.6 in section 5.5. As there are

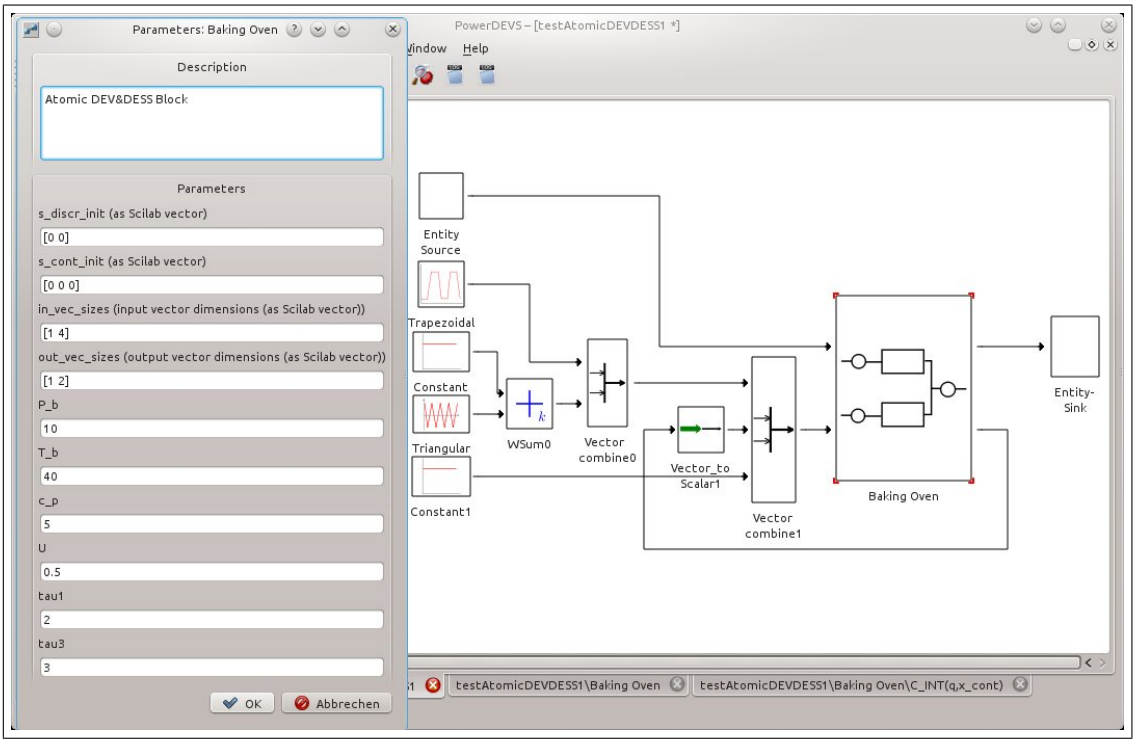


Figure 5.9: The PowerDEVS block for the DEV&DESS baking oven with the blocks generating test input signals.

three cases in which f is calculated differently, in the graphical model three corresponding sub blocks 'f0','f1', and 'f2' exist. They again are coupled models. The interior of block 'f0' is depicted in Figure 5.11. It simply produces a signal which is constantly zero. The interior of block 'f1' is depicted in Figure 5.12. There, only the derivatives of CO_2 and C are zero. The derivative of the temperature T is calculated according the formula $\dot{T} = \frac{U \cdot (T_e - T)}{m \cdot c_p}$. However, to avoid division by zero, the block 'Inverse' is deactivated if its input signal representing $m \cdot c_p$ is zero. The interior of 'f2' is depicted in Figure 5.7 in section 5.5. The content of the coupled block 'C_INT(q,x_cont)' representing C_{int}^{se} is depicted in Figure 5.7 in section 5.5. Finally, there is the block 'lambda_cont' corresponding to the function λ^{cont} in the DEV&DESS. Its internal structure is completely analogue to the structure of the block 'f(q,x_cont)' (see Figure 5.13). Again, there are three sub blocks that correspond to the three cases in the definition of λ^{cont}. Their interior is depicted in the Figures 5.14 – 5.16.

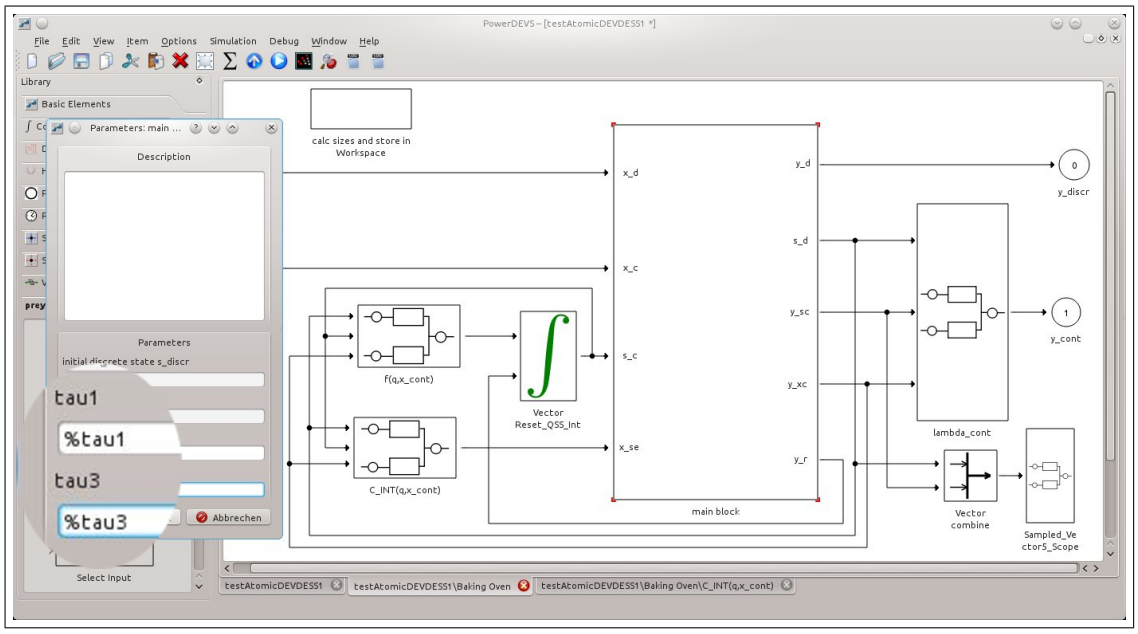


Figure 5.10: The interior of the PowerDEVS block for the DEV&DESS baking oven with the Parameters dialogue of the main block opened.

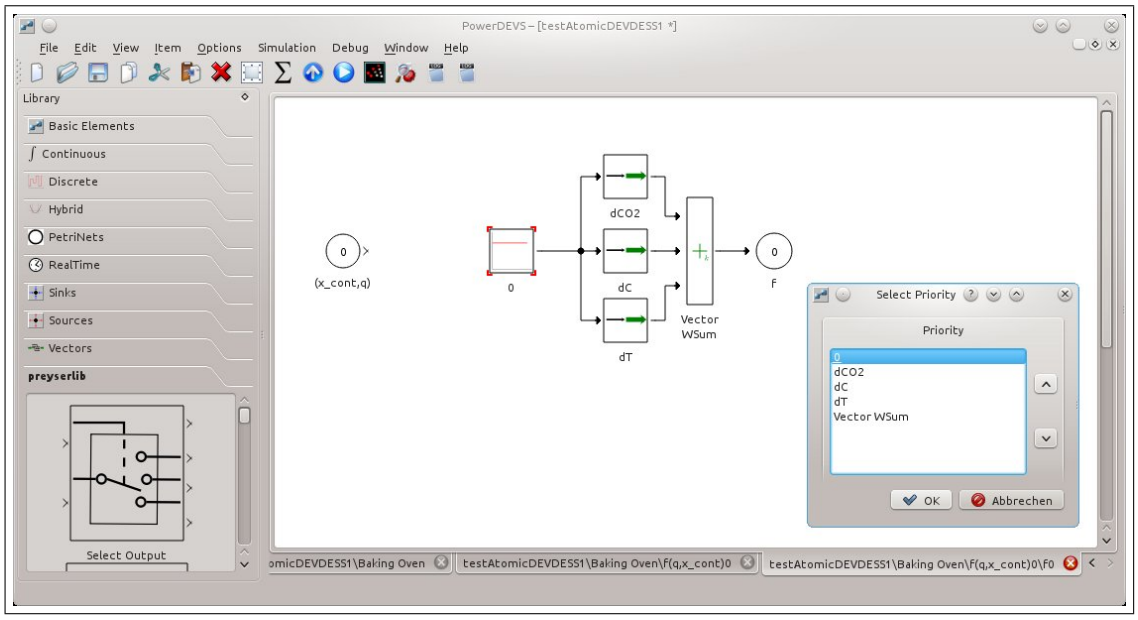


Figure 5.11: The interior of the coupled block ’f0’ of the baking oven example.

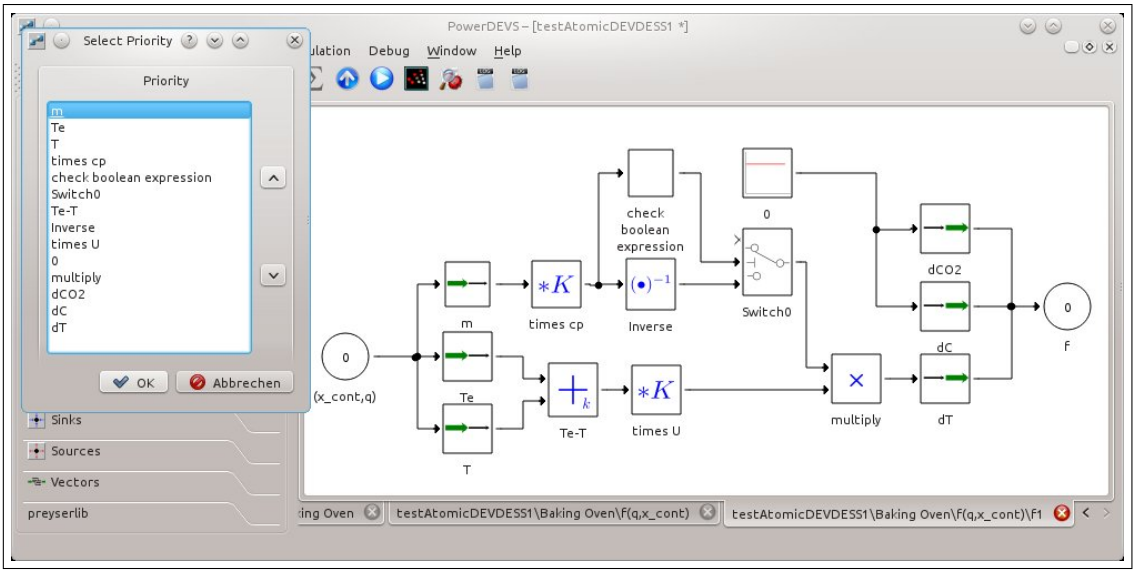


Figure 5.12: The interior of the coupled block 'f1'of the baking oven example.

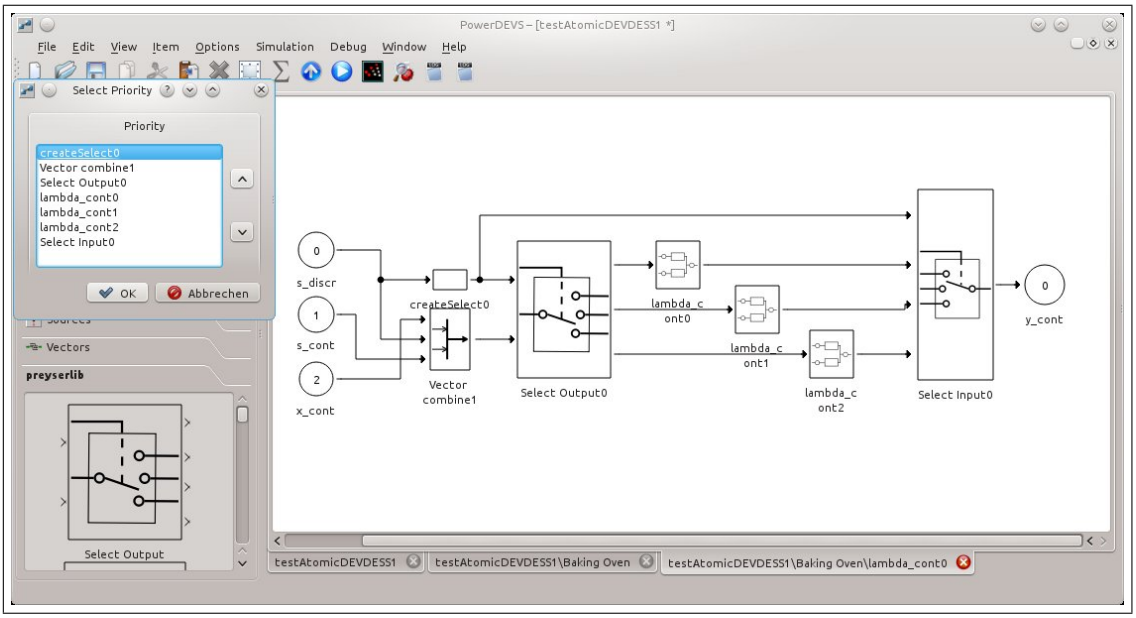


Figure 5.13: The interior of the coupled block 'lambda_cont' of the baking oven example.

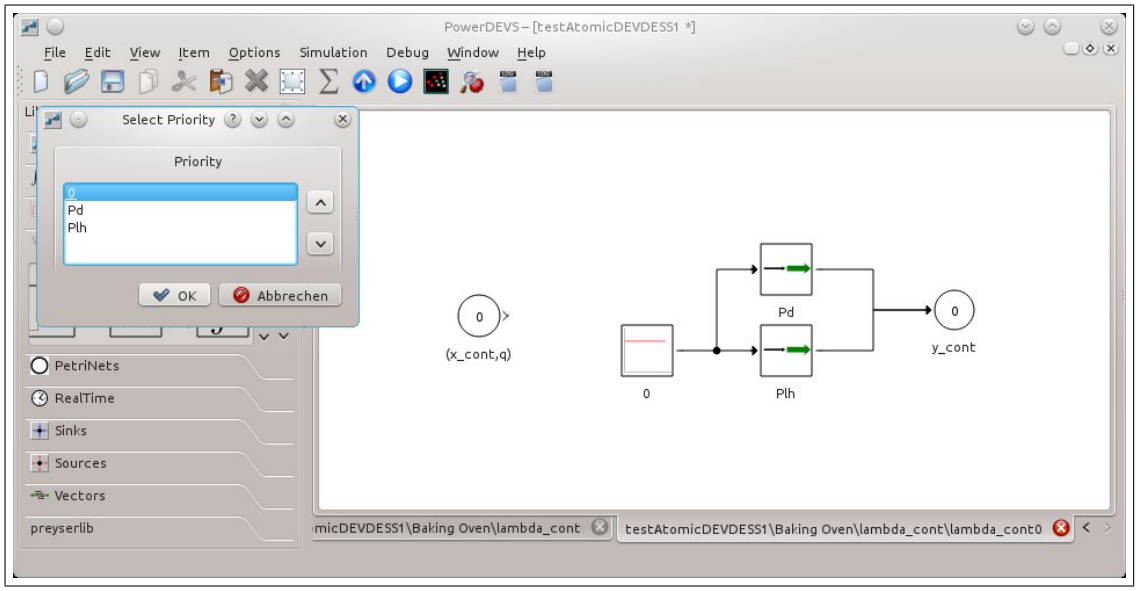


Figure 5.14: The interior of the coupled block 'lambda_cont0' of the baking oven example.

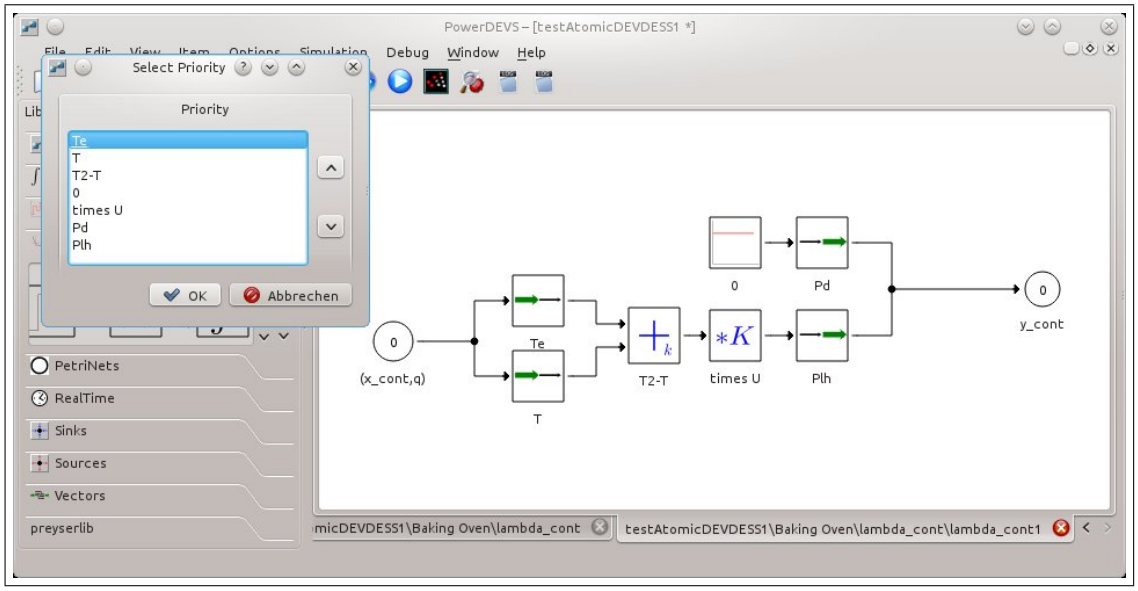


Figure 5.15: The interior of the coupled block 'lambda_cont1' of the baking oven example.

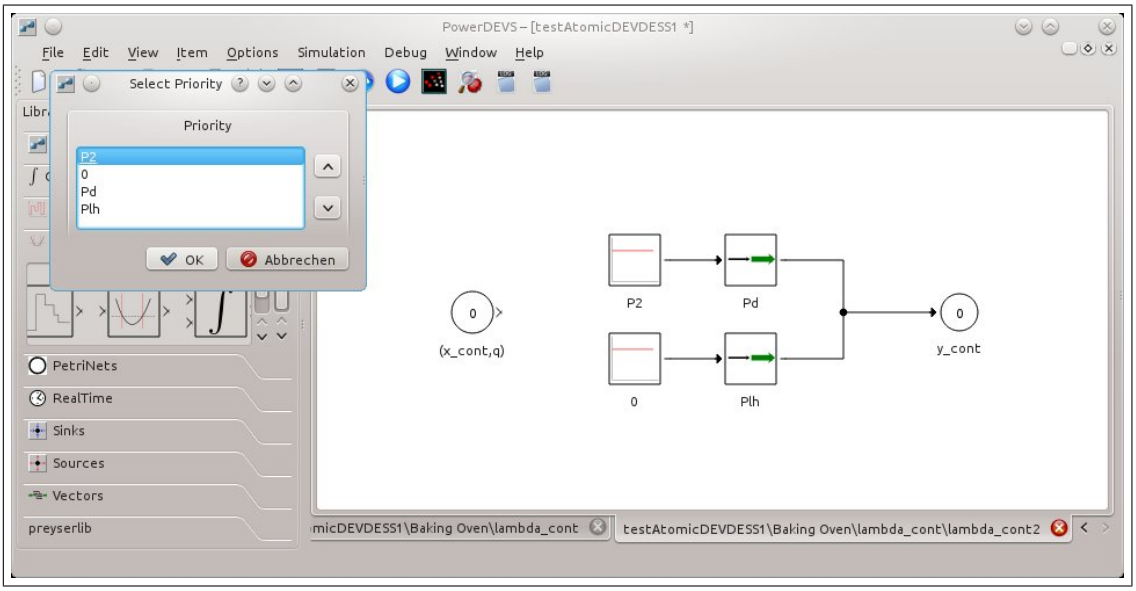


Figure 5.16: The interior of the coupled block 'lambda_cont2' of the baking oven example.

5.6.2 Discrete Part

The discrete part of the DEV&DESS model of the baking oven needs to be programmed in the source code of the main block. At first variables for the parameters τ_1 and τ_2 have to be defined. The right place for this is the definitions area after the comment `\\ modeller's input area` (see Listings 5.6 and 5.1). Further an instance of the class `Entity` is defined which is used later to store and modify received entities and entities to be sent.

```
1 // modeller's input area:
2   double tau1;
3   double tau2;
4   Entity entity;
```

Listing 5.6: Code snippet of the PowerDEVS definitions area for the main block of the DEV&DESS baking oven.

Next, the parameter values have to be read from the block's Parameters dialogue and stored in `tau1` and `tau2` as showed in Listing 5.7. Again, the designated place for this is marked by the comment `// modeller's input area` in the init function of the main block (see Listing 5.2).

The last and most interesting modeller's input area is the one in the output function of the main block. There, the functionality of λ and δ depending on the value of `flag` has to be programmed. There are the three cases, internal transiton `flag='i'`, external transition `flag='e'`, and confluent transition `flag='c'` to be distinguished. Listing 5.8 shows the implementation for the case `flag='i'`.

```
// modeller's input area
// read custom parameters, for example:
char *fvar = va_arg(parameters,char*);
tau1 = (double)getScilabVar(fvar);
tau3 = (double)getScilabVar(fvar);
```

Listing 5.7: Code snippet of the PowerDEVS init function for the main block of the DEV&DESS baking oven.

```
if(flag=='i' || flag=='I') {
  // modeller's input area:
  QSSDoubleArray qda = *(QSSDoubleArray*)s_discr_old.treatAt(1);
  switch((int)qda[0]) {
     case 1: qda = 2.0;
             s_discr->setAt(1,&qda,t);
             sigma_n = INF;
             break;
     case 2: qda = 3.0;
             s_discr->setAt(1,&qda,t);
             sigma_n = tau3;
             break;
     case 3: qda = *(QSSDoubleArray*)s_discr_old.treatAt(0);
             entity.set_attribute("m",qda[0]);
             qda = *(QSSDoubleArray*)s_cont_old.treatAt(0);
             entity.set_attribute("CO2",qda[0]);
             qda = *(QSSDoubleArray*)s_cont_old.treatAt(1);
             entity.set_attribute("C",qda[0]);
             qda = *(QSSDoubleArray*)s_cont_old.treatAt(2);
             entity.set_attribute("T",qda[0]);
             out_array.setAt(DEVDESS_DISCR_OUTPORT,0,&entity,t);
             qda = 0.0;
             for(int j=0; j < s_discr->size(); j++){
               s_discr->setAt(j,&qda,t);
             }
             for(int j=0; j < s_cont->size(); j++) {
               s_cont->setAt(j,&qda,t);
             }
             sigma_n = INF;
             break;
     default:
             printLog("t=%G, %s: illegal state-event\n",t,name);
             break;
  }
  if(flag=='I') flag='i';
```

Listing 5.8: The source code responsible for treating pure internal events in the main block of the DEV&DESS baking oven model.

A pure internal transition is only allowed to occur when the oven is one of the process phases $p \in \{1, 2, 3\}$. Process phase $p = 0$ means the oven waits for the next entity to arrive and therefore,

it is inactive. The process phase p is part of the discrete state of the system. To be exact, it is the second entry (index 1) of `s_discr_old` and `s_discr` respectively. Since both of them are of type `InOutputVector`, which is not quite handy to write and read double values from, first of all, in line 3 an instance `qda` of `QSSDoubleArray` is created and the process phase p is extracted out of `s_discr_old` and stored in `qda`. Then, a `switch - case` construct is used to treat the different situations in which the internal event has to be handled differently. The individual treatments depend solely on the process phase p stored in `qda`.

If the oven is in process phase one when an internal event occurs, the time τ_1 has elapsed since the last paste arrived and thus, the phase is increased to two and the time to the next time event is set to infinity (actually it is set (`1e20`, see section 4.2.1.3 for an explanation). This is because the next internal event has to be a state event, triggered by the external block 'C_INT(q,x_cont)'.

If the oven is in process phase two when an internal event occurs, it means that the baking phase is over and the paste has reached the desired temperature. According to the DEV&DESS model, the next step is to keep the bread in the oven for the duration τ_3 and to output it afterwards. Thus, the process phase is increased to three and `sigma_n` is set to τ_3.

Finally, if the oven is in process phase three when an event occurs, the bread has to be output. For this purpose, the attributes of the bread need to be read out of the systems state, stored in an instance of `Entity` and output at the discrete output port (line 21). Afterwards the process phase as well as all other state entries are set to zero (`for` loops starting at line 23) and the time to the next event `sigma_n` is set to infinity (`1e20` actually). The oven will become active again when the next paste arrives at its discrete input port.

```
36 else if(flag=='e') {
37   // modeller's input area:
38   QSSDoubleArray qda = *(QSSDoubleArray*)s_discr_old.treatAt(1);
39   if( 0 == qda[0]) {
40     entity = *(Entity*)in_array.treatAt(DEVDESS_DISCR_INPORT,0);
41     bool found;
42     qda = entity.get_attribute("m",&found);
43     s_discr->setAt(0,&qda,t);
44     // output new s_cont
45     qda = entity.get_attribute("CO2",&found);
46     s_cont->setAt(0,&qda ,t);
47     qda = entity.get_attribute("C",&found);
48     s_cont->setAt(1,&qda ,t);
49     qda = entity.get_attribute("T",&found);
50     s_cont->setAt(2,&qda ,t);
51     qda = 1;
52     s_discr->setAt(1,&qda,t);
53     sigma_n = tau1;
54   }
55 }
```

Listing 5.9: The source code responsible for treating pure external events in the main block of the DEV&DESS baking oven model.

Listing 5.9 shows the treatment of a pure external event. Again, at first the process phase is read out of the vectorial discrete state and stored in the `QSSDoubleArray` variable `qda`. An external event means a new entity arrived at the discrete input port. The oven though, is only capably of baking one paste at a time. Therefore pastes arriving when the oven is not in process phase zero are simply discarded. That means, the oven possesses no input queue. However, if the oven is in process phase zero, the entity is read out of the output buffer (line 40) and its attributes are stored in the system's state. The member method `get_attribute` of the class `Entity` reports with its second argument, whether an attribute with the name given in its first attribute exists or not. Thus, for example if the value of the boolean `found` is `false` after a call of `get_attribute("CO2",&found)`, it is known that the entity does not have any attribute named 'CO2'. Anyway, this report mechanism is not used here as it is known that all attributes that are asked for exist. Finally the process phase is set to one and the duration to the next internal event is set to τ_1.

```
56 else if(flag=='c' || flag=='C') {
57   // modeller's input area:
58   QSSDoubleArray qda = *(QSSDoubleArray*)s_discr_old.treatAt(1);
59   switch((int)qda[0]) {
60     case 1: qda = 2.0;
61             s_discr->setAt(1,&qda,t);
62             sigma_n = INF;
63             break;
64     case 2: qda = 3.0;
65             s_discr->setAt(1,&qda,t);
66             sigma_n = tau3;
67             break;
68     case 3: qda = *(QSSDoubleArray*)s_discr_old.treatAt(0);
69             entity.set_attribute("m",qda[0]);
70             qda = *(QSSDoubleArray*)s_cont_old.treatAt(0);
71             entity.set_attribute("CO2",qda[0]);
72             qda = *(QSSDoubleArray*)s_cont_old.treatAt(1);
73             entity.set_attribute("C",qda[0]);
74             qda = *(QSSDoubleArray*)s_cont_old.treatAt(2);
75             entity.set_attribute("T",qda[0]);
76             out_array.setAt(DEVDESS_DISCR_OUTPORT,0,&entity,t);
77             entity = *(Entity*)in_array.treatAt(DEVDESS_DISCR_INPORT,0);
78             bool found;
79             qda = entity.get_attribute("m",&found);
80             s_discr->setAt(0,&qda,t);
81             // output new s_cont
82             qda = entity.get_attribute("CO2",&found);
83             s_cont->setAt(0,&qda ,t);
84             qda = entity.get_attribute("K",&found);
85             s_cont->setAt(1,&qda ,t);
86             qda = entity.get_attribute("T",&found);
87             s_cont->setAt(2,&qda ,t);
88             qda = 1;
89             s_discr->setAt(1,&qda,t);
90             sigma_n = tau1;
91             break;
```

```
92          default:
93              printLog("t=%G, %s: illegal state-event\n",t,name);
94              break;
95      }
96      if(flag=='C') flag='c';
97  }
```

Listing 5.10: The source code responsible for treating confluent events in the main block of the DEV&DESS baking oven model.

The treatment of a confluent transition is nothing else but a combination of the treatments of an internal and an external event. Since the only case, when an external event can modify the systems state is when it is in process phase zero, a concurrent internal event is only influenced when its result phase is zero. For the oven this means, if there arrives a new paste at the same time a bread is output, the paste will be accepted and not discarded.

Looking at the source code in Listing 5.10, it can be recognized that there is exactly the same `switch - case` construct as in Listing 5.8 with the only differences in the case in which process phase equals three (line 68). There, until line 76, where the bread entity is ouput, the behaviour is the same as in the case of a pure internal event. However, then instead of resetting the systems state the currently arrived paste entity is read in line 77 and its attribute values are stored in the system's state. Therefore, the behaviour from line 77 on is exactly the same as in case of a pure external event (see Listing 5.9).

5.6.3 Simulation Result

Figure 5.17 shows simulations results. The diagram was generated by the 'GnuPlot' PowerDEVS library block. It shows the state trajectory of the baking oven during a simulation run of 30 seconds. There are two paste entities processed during these 30 seconds. The small, green trajectory with the shape of stairs corresponds to the current process phase of the oven. The straight line that ends in a sharp point belongs to the temperature T. The value of the summit is exactly $40\ ^{\circ}C$ as this is point where the state event defined as $T > 40\ ^{\circ}C$ is triggered. Since all signals are actually discrete event signals, i.e. they only exist at specific points in time, they have to be interpolated between those points. For this purpose the library block 'Sample and Hold' has been used.

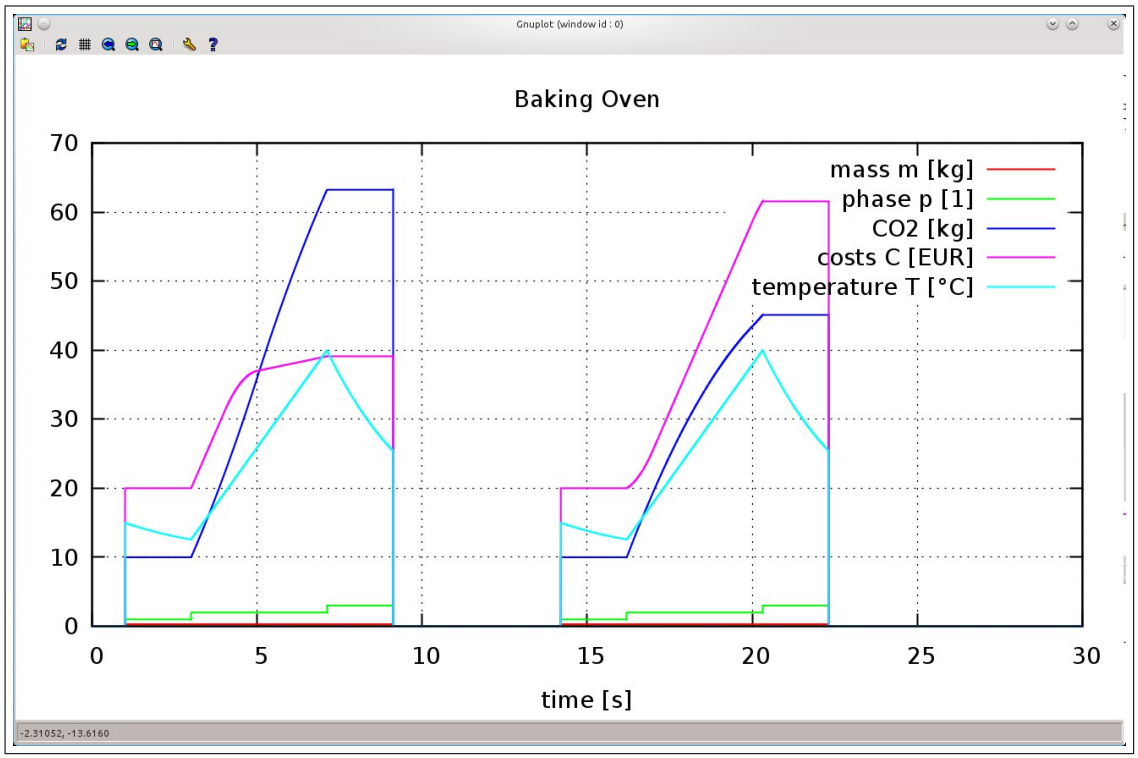


Figure 5.17: Simulation results of the baking oven example. Depicted are the state trajectories.

CHAPTER 6

Conclusion and Outlook

With the ever growing computational power of modern computers also the possibilities to simulate more and more extensive and complex models increase. However, when including more and more details into a model, very soon a point is reached where pure discrete or pure continuous models do not suffice anymore or they become unnecessarily expensive in terms of computational power. Production processes are providing a good example for this trend, as it is demanded to extend the formerly pure discrete models by the additional detail energy consumption which leads directly to hybrid models.

As lined out, most established simulators are specified either in the discrete or in the continuous domain and thus lack real effectiveness when hybrid models are simulated, if they support the implementation of hybrid models at all. Crucial issues are state event handling and the resolution of concurrent events.

The approach to come from the continuous side and to somehow insert discontinuities appears to be quite ineffective, whereas with QSS and its variations a quite promising approach seems to be found. The most important strength of QSS in this context is that discontinuities are part of the method and not something unwanted. Further, due to the discrete event nature of QSS, every state event actually is transformed into a time event as its point in time is calculated in advance. So QSS shows a way how to discretise continuous systems in a very controlled manner.

Nevertheless complex discrete event models are not easy to handle either. Thus, a well reasoned formalism is needed which on the one hand does not impose too much restrictions on the class of describable systems and on the other hand does not lead to models whose exact behaviour cannot be forecast anymore.

The DEVS formalism is a quite powerful one as the range of system behaviour that can be described by it is very wide. However, as it has been pointed out in chapter 4, it is very hard to define a DEVS that really behaves as intended in every possible situation, particularly if DEVS are coupled. The reason for that can be found on the one hand in the complexity of concurrency resolution itself and on the other hand in the property of a DEVS to exactly define its behaviour

in situations with concurrent events without forcing the modeller to explicitly formulate this behaviour. Parallel DEVS (P-DEVS) is a DEVS extension that exactly addresses this problem.

For the BaMa project a simulator was searched for that is capable of reasonably simulating hybrid models. Among the variety of DEVS simulators that can be found throughout the internet, PowerDEVS is one of the open source products that seems to be quite promising for this task as it combines the DEVS formalism with QSS. A drawback though is that it works with ordinary DEVS and not with P-DEVS.

To counter that problem, the generic Atomic PDEVS block has been developed and presented in section 4.3. This block is intended to facilitate the definition of an atomic DEVS model that behaves as intended also when coupled with other blocks.

Additionally to the pure DEVS formalism there also exists its modification DEV&DESS which is particularly designed for describing hybrid models. Although in PowerDEVS hybrid system models can easily be implemented so far there is no possibility to directly implement a DEV&DESS model. Thus, also a generic Atomic DEV&DESS block based on the Atomic PDEVS block has been developed and presented in chapter 5. This block is intended to serve as template for implementing arbitrary DEV&DESS in PowerDEVS.

However, profound theoretical investigations on this topic are desirable. In particularly a formal proof of the developed template blocks behaving as promised is work that still needs to be done.

APPENDIX A

Appendix

```
1/*************************************************************************
2**
3**   Copyright (C) 2009 Facultad de Ciencia Exactas Ingeniera y Agrimensura
4**            Universidad Nacional de Rosario - Argentina.
5**   Contact: PowerDEVS Information (kofman@fceia.unr.edu.ar, fbergero@fceia.
      unr.edu.ar)
6**
7**   This file is part of PowerDEVS.
8**
9**   PowerDEVS is free software: you can redistribute it and/or modify
10**  it under the terms of the GNU General Public License as published by
11**  the Free Software Foundation, either version 3 of the License, or
12**  (at your option) any later version.
13**
14**  PowerDEVS is distributed in the hope that it will be useful,
15**  but WITHOUT ANY WARRANTY; without even the implied warranty of
16**  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17**  GNU General Public License for more details.
18**
19**  You should have received a copy of the GNU General Public License
20**  along with PowerDEVS.  If not, see <http://www.gnu.org/licenses/>.
21**
22*************************************************************************/
23
24#ifndef EVENT_H
25#define EVENT_H
26
27#include <typeinfo>    // operator typeid
28#include<iostream>
29#include "ValuePointer.h"
30
31/*! En enumeratd type telling what kind of synchronization must be used for
      an event */
```

```
32typedef enum { NOREALTIME, REALTIME, PRECISEREALTIME } RealTimeMode;
33
34/*! \brief This class represents an output event, result of evaluating the
     lambda function of a DEVS model*/
35class Event
36{
37  /*! Tells which synchronization should be used for this event */
38  RealTimeMode mode;
39public:
40  /*! The value carried by an event is defined to be a void pointer for
       flexibility.
41  *   Most of the blocks in the library (by convention) use this value
       pointing to a double array.
42  *   Therefore an input value can be retrieved with:
43  * double *v = (double*) event.value;
44  *
45  * v[0] is the value
46  * v[1] is the first derivate
47  */
48  ValuePointer value;
49  //void *value;
50  /*!  The port value is used to know from which port the event came from */
51  Port port;
52  /*!  This bool value is used in the RTAI distribution. When set, indicates
       that an external input event
53  *    ocurred while synchronzing this event, then, it should not be
       propagated. */
54  bool interrupted;
55  Event();
56  template<class T>
57  Event( T* val, Port p){
58    value=val;
59    mode = NOREALTIME;
60    interrupted = 0;
61    port=p;
62  }
63  virtual ~Event();
64
65  /*! A null event is defined as one with value=NULL. */
66  void setNullEvent();
67  bool isNotNull();
68
69  void setRealTimeMode(RealTimeMode m) { mode=m; };
70  RealTimeMode getRealTimeMode() { return mode; };
71  /*! Retrieves the i'th double value of this event */
72  double getDouble(int i) { return ((double*)value)[i]; };
73  /*! Retrieves the first double value of this event */
74  double getDouble() { return getDouble(0); };
75  /*! Retrieves the first int value of this event */
76  double getInt()   { return ((int*)value)[0]; };
77  void   setDouble(double &v) { value=&v; };
78  void   setInt(int &v) { value=&v; };
```

```
79  /*!  This is used internally by the simulation engine to notify an external
        input event */
80  void setInterrupted() { interrupted=true; }
81  bool isInterrupted() { return interrupted; }
82};
83
84#endif
```

Listing A.1: The class `Event`.

```
1#ifndef VALUEPOINTER_H
2#define VALUEPOINTER_H
3
4#include <typeinfo>   // operator typeid
5#include "types.h"
6#include "../atomics/vector/vector.h"
7
8class ValuePointer {
9private:
10  DEVSMessage *msgPtr;
11  vector *vecPtr;
12public:
13  bool allocated_memory;
14
15public:
16  // contstructors:
17  ValuePointer() {
18    msgPtr = NULL;
19    vecPtr = NULL;
20    allocated_memory=false;
21  }
22  ValuePointer(const ValuePointer &vp) {
23    allocated_memory = false;
24    if(vp.allocated_memory) {
25      msgPtr = vp.msgPtr->getCopy();
26      allocated_memory=true;
27    } else
28      msgPtr = vp.msgPtr;
29    vecPtr = NULL;
30  }
31
32  // casts:
33  template<class T>
34  inline operator T*() {
35  if(typeid(T)==typeid(void)) {
36      return((T*)(msgPtr->getVoidPtr()));
37    } else if(typeid(T)==typeid(int)) {
38      return((T*)msgPtr->getIntPtr());
39    } else if(typeid(T)==typeid(double)) {
40      return((T*)msgPtr->getDoublePtr());
41    } else if(typeid(T)==typeid(DEVSMessage)) {
42      return((T*)msgPtr);
```

```
43    } else if( (typeid(T)==typeid(*msgPtr)) || typeid(T).before(typeid(*
          msgPtr))) {
44      return((T*)msgPtr);
45    } else if( typeid(T)==typeid(vector) ){
46    // if a double array is casted into a vector instance (which
47    // happens in some of the PowerDEVS library blocks)
48      if(NULL != vecPtr) delete vecPtr;
49          vecPtr = new vector(msgPtr);
50        return((T*)vecPtr);
51    }
52    return(NULL);
53  }
54
55  // assignment-operators:
56  ValuePointer& operator = (int n) {
57    if(0==n) {
58      if(allocated_memory) delete msgPtr;
59      allocated_memory=false;
60      msgPtr = NULL;
61    }
62    return(*this);
63  }
64  ValuePointer& operator = (const ValuePointer &ptr) {
65    if(allocated_memory) delete msgPtr;
66    allocated_memory = false;
67    if(ptr.allocated_memory) {
68      msgPtr = ((ValuePointer&)ptr).getCopy();
69      allocated_memory=true;
70    } else
71      msgPtr = ptr.msgPtr;
72
73    return(*this);
74  }
75  ValuePointer& operator = (void *ptr) {
76    if(NULL == msgPtr) {
77      msgPtr = new DEVSMessage;
78      allocated_memory=true;
79    }
80    msgPtr->set(ptr);
81    return(*this);
82  }
83  ValuePointer& operator = (double *ptr) {
84   if(NULL == msgPtr) {
85      msgPtr = new QSSDoubleArray(0.0,(uint)0);
86      allocated_memory=true;
87    }
88    msgPtr->set(ptr);
89    return(*this);
90  }
91  ValuePointer& operator = (int *ptr) {
92    if(NULL == msgPtr) {
93      msgPtr = new DEVSMessage();
94      allocated_memory=true;
```

```
95    }
96    msgPtr->set(ptr);
97    return(*this);
98  }
99  ValuePointer& operator = (DEVSMessage *msg) {
100   if(allocated_memory) delete msgPtr;
101   allocated_memory=false;
102   msgPtr = msg;
103   return(*this);
104 }
105 template<class T>
106 ValuePointer& operator = (T *ptr) {
107   // Problem: the qss-solvers return &y (where y: double y[10]) and
108   // the typeid of any class is before the typeid of every basic type (as &
          y),
109   // but bigarray& is before every array of basic-type with less than
110   // 10000 elements
111   // better would be the use of include: <type_traits> - but that would
112   // need the compilerflag -c++11
113   typedef char bigarray[10000];
114   if( (!typeid(bigarray&).before(typeid(T))) && typeid(DEVSMessage).before(
          typeid(T)) ) {
115     if(allocated_memory) delete msgPtr;
116     allocated_memory=false;
117     msgPtr = (DEVSMessage*)ptr;
118   } else {
119     (*this)=(void*)ptr;
120   }
121   return(*this);
122 }
123
124 // comparison-operators:
125 template<class T>
126 bool operator == (T *ptr) {
127   return(((T*)msgPtr)==ptr);
128 }
129 bool operator == (DEVSMessage *ptr) {
130   return(msgPtr->operator==(*ptr));
131 }
132 template<class T>
133 bool operator != (T *ptr) {
134   return( !((*this))==ptr) );
135 }
136 bool operator == (int n) {
137   if(0==n)
138     return(((void*)msgPtr)==NULL);
139   return(false);
140 }
141 bool operator != (int n) {
142   return( !((*this)==n) );
143 }
144
145 // getCopy
```

```
146   DEVSMessage* getCopy() {
147     return(msgPtr->getCopy());
148   }
149   ~ValuePointer() {
150     if(allocated_memory) delete msgPtr;
151     if(NULL != vecPtr) delete vecPtr;
152   }
153 };
154
155 #endif
```

Listing A.2: The class `ValuePointer`.

```
1 #ifndef DEVSMESSAGE_H
2 #define DEVSMESSAGE_H
3
4 class DEVSMessage {
5 public:
6   void *value;
7   int index;
8   bool retrieve;
9
10  // constructors:
11  DEVSMessage() { value=NULL; index=0; retrieve=false;}
12  DEVSMessage(void* ptr) { value=ptr; index=0; retrieve=false;}
13  DEVSMessage(const DEVSMessage &msg) { this->value = msg.value; this->index
        = msg.index; retrieve=false;}
14
15  virtual void* getVoidPtr() { return value;}
16  virtual double* getDoublePtr() { return((double*)value);}
17  virtual int* getIntPtr() { return((int*)value);}
18
19  virtual void set(void *ptr) { value=ptr; }
20  virtual void set(double *ptr) { value=(double*)ptr; }
21  virtual void set(int *ptr) { value=(int*)ptr; }
22
23  virtual bool operator==(DEVSMessage& msg) {
24    if(this->index!=msg.index) return(false);
25    return(this->value==msg.value);
26  }
27  virtual bool operator!=(DEVSMessage &msg){
28    return(!((*this)==msg));
29  }
30  virtual DEVSMessage* getCopy() { return(new DEVSMessage(*this));}
31  virtual ~DEVSMessage(){}
32 };
33
34 #endif
```

Listing A.3: The class `DEVSMessage`.

```
#ifndef QSS_SIGNAL_H
#define QSS_SIGNAL_H

#include<string.h>
#include<stdlib.h>
#include<math.h>
#include<typeinfo>
#include <stdexcept>
#include"DEVSMessage.h"

#ifndef QSS_DOUBLEARRAY_SIZE
  #define QSS_DOUBLEARRAY_SIZE 10
#endif

#ifndef INF
  #define INF 1e20
#endif

class QSSDoubleArray:public DEVSMessage {

  public:
    uint size;
    bool allocated_memory;

    // Constructors:
    QSSDoubleArray(){
      this->size = (uint)QSS_DOUBLEARRAY_SIZE;
      value = (void*)calloc(QSS_DOUBLEARRAY_SIZE, sizeof(double));
      allocated_memory = true;
      this->index=0;
    }
    QSSDoubleArray(DEVSMessage *ptr){
      this->size = (uint)QSS_DOUBLEARRAY_SIZE;
      value = (void*)calloc(QSS_DOUBLEARRAY_SIZE, sizeof(double));
      allocated_memory = true;
      this->index=0;
      for(uint i=0; i<QSS_DOUBLEARRAY_SIZE; i++)
        ((double*)value)[i] = (ptr->getDoublePtr())[i];
    }
    QSSDoubleArray(const QSSDoubleArray &instance){
      this->size = instance.size;
      this->value = (void*)malloc(size*sizeof(double));
      allocated_memory = true;
      memcpy(this->value, instance.value, instance.size*sizeof(double));
      this->index=instance.index;
    }
    QSSDoubleArray(const double value[]) {
      this->size = QSS_DOUBLEARRAY_SIZE;
      this->value = (void*)malloc(QSS_DOUBLEARRAY_SIZE*sizeof(double));
      allocated_memory = true;
      memcpy(this->value, value, QSS_DOUBLEARRAY_SIZE*sizeof(double));
      this->index=0;
    }
```

```
54    QSSDoubleArray(const double value[], const uint size) {
55      this->size = size;
56      this->value = (void*)malloc(size*sizeof(double));
57      allocated_memory = true;
58      memcpy(this->value, value, size*sizeof(double));
59      this->index=0;
60    }
61    QSSDoubleArray(const double val) {
62      this->size = QSS_DOUBLEARRAY_SIZE;
63      this->value = (void*)calloc(size,sizeof(double));
64      allocated_memory = true;
65      ((double*)value)[0]=val;
66      this->index=0;
67    }
68    QSSDoubleArray(const double val, uint size) {
69      this->size = size;
70      this->index=0;
71      if(size>0) {
72        this->value = (void*)calloc(size,sizeof(double));
73        allocated_memory = true;
74        ((double*)value)[0]=val;
75      } else {
76        this->value = NULL;
77        allocated_memory = false;
78      }
79    }
80
81    // operators:
82    inline double& operator[] (uint n) {
83      if(n<size)
84        return(((double*)value)[n]);
85      throw std::out_of_range("index in QSSDoubleArray []-Operator is out of
          range!");
86      return(((double*)value)[size-1]);
87    }
88
89    inline operator double() {
90      if(NULL != value) return(((double*)value)[0] );
91      else return(0);
92    }
93
94    QSSDoubleArray& operator= (const QSSDoubleArray &a) {
95      if(size < a.size){
96        if( (NULL != this->value) && allocated_memory) free(this->value);
97        this->value = (void*)malloc(a.size*sizeof(double));
98        allocated_memory = true;
99      }
100     this->size = a.size;
101     memcpy(this->value, a.value, size*sizeof(double));
102     this->index=a.index;
103     return(*this);
104   }
105   QSSDoubleArray& operator= (const double val[]) {
```

```
106       if(size < QSS_DOUBLEARRAY_SIZE){
107         if( (NULL != this->value) && allocated_memory) free(this->value);
108         this->value = (void*)malloc(QSS_DOUBLEARRAY_SIZE*sizeof(double));
109         allocated_memory = true;
110       }
111       this->size = QSS_DOUBLEARRAY_SIZE;
112       memcpy(this->value, val, size*sizeof(double));
113       return(*this);
114     }
115     QSSDoubleArray& operator= (const double val) {
116       ((double*)value)[0]=val;
117       for(uint i=1; i<size; i++)
118         ((double*)value)[i]=0;
119       return(*this);
120     }
121
122     QSSDoubleArray invert() {
123       QSSDoubleArray result (1.0, size);
124       if( (*this)[0]==0) {
125         result = INF;
126         return(INF);
127       }
128       for(uint i=0; i<size; i++){
129         for(uint j=1; j<=i; j++) {
130           result[i] -= result[i-j]*(*this)[j];
131         }
132         result[i] = result[i]/(*this)[0];
133       }
134       return(result);
135     }
136     QSSDoubleArray operator+(const QSSDoubleArray &da) {
137       uint biggest_size = this->size;
138       if(da.size > biggest_size) biggest_size = da.size;
139       QSSDoubleArray result (0.0, biggest_size);
140       for(uint i=0; i<biggest_size; i++){
141         if(i>=this->size)
142           result[i] = ((QSSDoubleArray)da)[i];
143         else if(i>=da.size)
144           result[i] = (*this)[i];
145         else
146           result[i] = (*this)[i] + ((QSSDoubleArray)da)[i];
147       }
148       return(result);
149     }
150     QSSDoubleArray operator*(const QSSDoubleArray &da) {
151       uint biggest_size = this->size;
152       if(da.size > biggest_size) biggest_size = da.size;
153       QSSDoubleArray result (0.0, biggest_size);
154       for(uint i=0; i<biggest_size; i++) {
155         for(uint j=0; j<=i; j++) {
156           if( (j<this->size)&&((i-j)<da.size) )
157             result[i] += (*this)[j]*((QSSDoubleArray)da)[i-j];
158           else
```

```
159           result[i] += 0;
160         }
161       }
162       return(result);
163     }
164     QSSDoubleArray operator/(const QSSDoubleArray &da) {
165       return((*this)*((QSSDoubleArray)da).invert());
166     }
167
168     bool operator==(QSSDoubleArray &a){
169       if(this->size != a.size) return(false);
170       for(uint i=0; i<size; i++) {
171         if( (*this)[i]!=a[i])
172           return false;
173       }
174       return true;
175     }
176     virtual bool operator==(DEVSMessage &msg){
177       if(typeid(msg)!=typeid(*this)) return false;
178       return( (*this)==((QSSDoubleArray&)msg) );
179     }
180     bool operator!=(QSSDoubleArray &a){
181       return(!((*this)==a));
182     }
183
184     double advance_time(double delta_t){
185       QSSDoubleArray derivatives(*this);
186       (*this)=0.0;
187       for(uint i=0; i<size; i++) {
188         // evaluate derivatives at t=t_last_change + delta_t
189         for(uint j=i; j<size; j++)
190           (*this)[i] += derivatives[j]*pow(delta_t,(double)j-i);
191         // derive 'derivatives'
192         for(uint j=i; j<size;j++)
193           derivatives[j] = ((double)(j-i))*derivatives[j];
194       }
195       return((*this)[0]);
196     }
197
198     // DEVSMessage - Interface:
199     virtual void set(void *ptr) {
200       if( (NULL!=value) && allocated_memory) free(value);
201       allocated_memory = false;
202       value = ptr;
203       size = QSS_DOUBLEARRAY_SIZE;
204     }
205     virtual void set(double *ptr) {
206       if( (NULL!=value) && allocated_memory) free(value);
207       allocated_memory = false;
208       value = (void*)ptr;
209       size = QSS_DOUBLEARRAY_SIZE;
210     }
211     virtual void set(int *ptr) {
```

```
212       if( (NULL!=value) && allocated_memory) free(value);
213       allocated_memory = false;
214       value = (void*)ptr;
215       size = 1;
216     }
217
218     virtual DEVSMessage* getCopy() {
219       return((DEVSMessage*)(new QSSDoubleArray((*this))) );
220     }
221
222     virtual ~QSSDoubleArray(){
223       if((NULL!=value) && allocated_memory) free(value);
224     }
225 };
226
227 #endif
```

Listing A.4: The class `QSSDoubleArray`.

```
1 #ifndef VECTOR_H
2 #define VECTOR_H
3
4 #include "string.h"
5 #include "../preyserlib/QSSDoubleArray.h"
6
7 /*  this class is needed, because library blocks use the following code:
8  *  vector v; v.value[i]; however, value is an attribute of the base class
9  *  DEVSMessage and does not have an operator [] defined on it.
10 *  Therefore, vector gets a new attribute value that covers the value
       attribute
11 *  of DEVSMessage. Therefore v.value[i] will also work.
12 */
13 class ValueDuplicate {
14 private:
15 //  void **ptr;
16   QSSDoubleArray *ptr;
17
18 public:
19
20   ValueDuplicate() { ptr=NULL;}
21   ValueDuplicate(QSSDoubleArray *ptr) { this->ptr=ptr;}
22
23   void set(QSSDoubleArray* ptr) { this->ptr=ptr; }
24
25   operator void*() {
26     return(ptr->getVoidPtr());
27   }
28   operator double*() {
29     return(ptr->getDoublePtr());
30   }
31   operator int*() {
32     return(ptr->getIntPtr());
```

```
33   }
34
35   inline double& operator[] (uint n){ return( (*ptr)[n] ); }
36
37 };
38
39 class vector:public QSSDoubleArray
40 {
41 public:
42   ValueDuplicate value;
43
44   // constructors:
45   vector():QSSDoubleArray() {
46     this->index=0;
47     value.set(this);
48   }
49   vector(const vector &v):QSSDoubleArray((QSSDoubleArray&)v) {
50     this->value.set(this);
51     this->index = v.index;
52   }
53   vector(const double value[], const size_t size):QSSDoubleArray(value, size)
         {
54     this->value.set(this);
55     this->index = 0;
56   }
57   vector(const QSSDoubleArray value):QSSDoubleArray(value) {
58     this->value.set(this);
59     this->index = 0;
60   }
61   vector(const double value[], const int index):QSSDoubleArray(value) {
62     this->value.set(this);
63     this->index = index;
64   }
65   vector(const QSSDoubleArray value, const int index):QSSDoubleArray(value) {
66     this->value.set(this);
67     this->index = index;
68   }
69   vector(const double val):QSSDoubleArray(val) {
70     this->index = 0;
71     this->value.set(this);
72   }
73   vector(const double val, int index):QSSDoubleArray(val) {
74     this->index = index;
75     this->value.set(this);
76   }
77   vector(const double val, uint size, int index):QSSDoubleArray(val, size) {
78     this->index = index;
79     this->value.set(this);
80   }
81   vector(DEVSMessage *msg):QSSDoubleArray(0.0,(uint)0) {
82     this->set(msg->getVoidPtr());
83
84     if(typeid(QSSDoubleArray)==typeid(*msg))
```

```
85        this->size = ((QSSDoubleArray*)msg)->size;
86      else this->size=QSS_DOUBLEARRAY_SIZE;
87
88      this->index = msg->index;
89      this->value.set(this);
90    }
91
92    // operators:
93    vector& operator=(const vector &v){
94      this->index = v.index;
95      (QSSDoubleArray&)(*this) = (QSSDoubleArray&)v;
96      return(*this);
97    }
98    vector& operator=(const double value[]){
99      this->index = 0;
100     (QSSDoubleArray&)(*this) = value;
101     return(*this);
102   }
103   vector& operator=(const double val){
104     ((QSSDoubleArray*)this)->operator=(val);
105     return(*this);
106   }
107
108   bool operator==(vector &v){
109     if(index != v.index) return(false);
110     return( ((QSSDoubleArray&)(*this))==((QSSDoubleArray&)v)  );
111   }
112   virtual bool operator==(DEVSMessage &msg){
113     if(typeid(msg)!=typeid(*this)) return false;
114     return( (*this)==((vector&)msg) );
115   }
116   bool operator!=(vector &v){
117     return(!((*this)==v));
118   }
119
120   virtual DEVSMessage* getCopy(){
121     return ((DEVSMessage*)(new vector(*this)));
122   }
123   virtual ~vector() {}
124 };
125
126 #endif
```

Listing A.5: The class `vector`.

```
1 #ifndef ENTITY_H
2 #define ENTITY_H
3
4 #include<string>
5 #include<map>
6 #include<stdio.h>
7 #include<sstream>
```

```
8#include "DEVSMessage.h"
9
10class Entity:public DEVSMessage
11{
12public:
13  std::string log;
14  std::string log_file_path;
15  std::map<std::string,double> attribute_map;
16
17  Entity():DEVSMessage(){}
18  Entity(const Entity &e):DEVSMessage((DEVSMessage&)e){
19    this->log = e.log;
20    this->log_file_path = e.log_file_path;
21    this->attribute_map = e.attribute_map;
22  }
23  Entity(std::string &log_file_path, std::string &log, std::map<std::string,
      double> &attribute_map):DEVSMessage(){
24    this->log = log;
25    this->log_file_path = log_file_path;
26    this->attribute_map = attribute_map;
27  }
28  Entity(const char log_file_path[], const char log[], std::map<std::string,
      double> &attribute_map):DEVSMessage(){
29    this->log = std::string(log);
30    this->log_file_path = std::string(log_file_path);
31    this->attribute_map = attribute_map;
32  }
33  Entity(std::string &log_file_path, std::string &log, std::map<std::string,
      double> &attribute_map, void* val):DEVSMessage(val){
34    this->log = log;
35    this->log_file_path = log_file_path;
36    this->attribute_map = attribute_map;
37  }
38  Entity(const char log_file_path[], const char log[], std::map<std::string,
      double> &attribute_map, void* val):DEVSMessage(val){
39    this->log = std::string(log);
40    this->log_file_path = std::string(log_file_path);
41    this->attribute_map = attribute_map;
42  }
43
44  Entity& operator=(const Entity &e){
45    ((DEVSMessage&)*this) = (DEVSMessage&)e;
46    this->log = e.log;
47    this->log_file_path = e.log_file_path;
48    attribute_map = e.attribute_map;
49    this->index=e.index;
50
51    return(*this);
52  }
53
54  bool operator==(Entity& e) {
55   if(this->log.compare(e.log)) return false;
56   if(this->log_file_path.compare(e.log_file_path)) return false;
```

```
57   for (std::map<std::string,double>::iterator it=attribute_map.begin(); it!=
         attribute_map.end(); ++it) {
58     if(e.attribute_map[it->first]!=it->second)
59       return false;
60   }
61   if(this->index != e.index) return(false);
62   return((DEVSMessage)(*this)==(DEVSMessage&)e);
63  }
64  virtual bool operator==(DEVSMessage &msg){
65    if(typeid(msg)!=typeid(*this)) return false;
66    return( (*this)==((Entity&)msg) );
67  }
68  bool operator!=(Entity &e){
69    return(!((*this)==e));
70  }
71
72  void set_log_file_path(std::string path, bool reset){
73    log_file_path=path;
74    if(reset){
75      FILE *fp = fopen(&path[0],"w");
76      fclose(fp);
77    }
78  }
79
80  void add_log_entry(const char log_entry[], double t) {
81    std::string log_e (log_entry);
82    this->add_log_entry(log_e,t);
83  }
84  void add_log_entry(std::string &log_entry, double t) {
85    std::stringstream sstr;
86    sstr<<"log entry at t="<< t << ": "<< log_entry << "\n";
87    sstr.flush();
88    this->log += sstr.str();
89  }
90  void write_log_to_file() {
91    FILE *fp = fopen(&log_file_path[0],"a");
92    fprintf(fp,"%s",&log[0]);
93    fclose(fp);
94  }
95
96  void set_attribute(std::string &att_name, double val){
97    attribute_map[att_name] = val;
98  }
99  void set_attribute(const char att_name[], double val){
100   std::string name_str (att_name);
101   set_attribute(name_str,val);
102 }
103
104 double get_attribute(std::string name, bool *ptr) {
105     std::map<std::string, double>::iterator it = attribute_map.find(name);
106     if(it == attribute_map.end()) {
107 (*ptr)=false;
108 return(0);
```

```
109         }
110         (*ptr)=true;
111         return(it->second);
112     }
113     double get_attribute(std::string name) {
114         std::map<std::string, double>::iterator it = attribute_map.find(name);
115         if(it == attribute_map.end()) {
116     return(0);
117         }
118         return(it->second);
119     }
120     bool delete_attribute(std::string name) {
121       return((bool)attribute_map.erase(name));
122     }
123
124     DEVSMessage* getCopy() {
125       return ((DEVSMessage*)(new Entity(*this)));
126     }
127     virtual ~Entity() {}
128 };
129
130 #endif
```

Listing A.6: The class `Entity`.

List of Figures

Bibliography

[BEBG93] F. Breitenecker, H. Ecker, and I. Bausch-Gall. *Simulation mit ACSL,Eine Einfhrung in die Modellbildung, numerischen Methoden und Simulation*. Springer Fachmedien Wiesbaden, 1993.

[BF06] T. Beltrame and Cellier F.E. Quantised state system simulation in dymola/modelica using the devs formalism. *Simulation News Europe*, 16(3-4):3 – 12, 2006.

[BK11] F. Bergero and E. Kofman. PowerDEVS. A Tool for Hybrid System Modeling and Real Time Simulation. *Simulation: Transactions of the Society for Modeling and Simulation International*, 87(1–2):113–132, 2011.

[CK06] F.E. Cellier and E. Kofman. *Continuous System Simulation*. Springer, New York, 2006.

[CKC11] R. Castro, E. Kofman, and F. Cellier. Quantization Based Integration Methods for Delay Differential Equations. *Simulation Modelling Practice and Theory*, 19(1):314–336, 2011.

[KJ01] E. Kofman and S. Junco. Quantized State Systems. A DEVS Approach for Continuous System Simulation. *Transactions of SCS*, 18(3):123–132, 2001.

[Kof02] E. Kofman. A Second Order Approximation for DEVS Simulation of Continuous Systems. *Simulation: Transactions of the Society for Modeling and Simulation International*, 78(2):76–89, 2002.

[Kof03] E. Kofman. Quantized-State Control. A Method for Discrete Event Control of Continuous Systems. *Latin American Applied Research*, 33(4):399–406, 2003.

[Kof04] E. Kofman. Discrete Event Simulation of Hybrid Systems. *SIAM Journal on Scientific Computing*, 25(5):1771–1797, 2004.

[Kof06] E. Kofman. A Third Order Discrete Event Simulation Method for Continuous System Simulation. *Latin American Applied Research*, 36(2):101–108, 2006.

[Kof09] E. Kofman. Relative Error Control in Quantization Based Integration. *Latin American Applied Research*, 39(3):231–238, 2009.

[MBKC13] G. Migoni, M. Bortolotto, E. Kofman, and F. Cellier. Linearly Implicit Quantization-Based Integration Methods for Stiff Ordinary Differential Equations. *Simulation Modelling Practice and Theory*, 35:118–136, 2013.

[MK09] G. Migoni and E. Kofman. Linearly Implicit Discrete Event Methods for Stiff ODEs. *Latin American Applied Research*, 39(3):245–254, 2009.

[MKC12] G. Migoni, E. Kofman, and F. Cellier. Quantization-Based New Integration Methods for Stiff ODEs. *Simulation: Transactions of the Society for Modeling and Simulation International*, 88(4):387–407, 2012.

[PHR+14] N. Popper, I. Hafner, M. Rssler, F. Preyser, B. Heinzl, P. Smolek, and I. Leobner. A General Concept for Description of Production Plants with a Concept of Cubes. *Simulation Notes Europe*, 24(2):105–114, 2014.

[Pra91] H. Prahofer. Systems theoretic formalisms for combined discrete-continuous system simulationm. *International Journal of General Systems*, 19(3):219 – 240, 1991.

[SP14] A. Schmidt and T. Pawletta. Hybride modellierung fertigungstechnischer prozessketten mit energieaspekten in einer ereignisdiskreten simulationsumgebung. In *ASIM 2014 – 22. Symposium Simulationstechnik, Berlin, 03.-05.09.2014*, pages 109–116. ARGESIM/ASIM, 2014.

[Zei] B.P. Zeigler. Embedding dev&dess in devs. In *DEVS Integrative M&S Symposium (DEVS'06)*.

[ZPK00] B.P. Zeigler, H. Praehofer, and T.G. Kim. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, 2000.