Automated Simulation of Modelica Models with QSS Methods - The Discontinuous Case -

Xenofon Floros[1] Federico Bergero[2] François E. Cellier [1] Ernesto Kofman[2]

[1]Department of Computer Science, ETH Zurich, Switzerland
{xenofon.floros, francois.cellier}@inf.ethz.ch
[2]Laboratorio de Sistemas Dinámicos, FCEIA, Universidad Nacional de Rosario, Argentina
CIFASIS-CONICET
{fbergero, kofman}@fceia.unr.edu.ar

Abstract

This study describes the current implementation of an interface that automatically translates a discontinuous model described using the **Modelica** language into the Discrete Event System Specification (**DEVS**) formalism. More specifically, the interface enables the automatic simulation of a Modelica model with discontinuities in the **PowerDEVS** environment, where the Quantized State Systems (**QSS**) integration methods are implemented. Providing DEVS-based simulation algorithms to Modelica users should extend significantly the tools that are currently available in order to efficiently simulate several classes of large-scale real-world problems, e.g. systems with heavy discontinuities. In this work both the theoretical design and the implementation of the interface are discussed. Furthermore, simulation results are provided that demonstrate the correctness of the proposed implementation as well as the superior performance of QSS methods when simulating discontinuous systems.

Keywords: *OpenModelica, DASSL, PowerDEVS, QSS, discontinuous systems*

1 Introduction

Modelica [8, 9] is an object-oriented, equation-based language that allows the representation of continuous as well as hybrid models using a set of non-causal equations. The Modelica language enables a standardized way to model complex physical systems containing, e.g., mechanical, electrical, electronic, hydraulic, thermal, control, electric power, or process-oriented subcomponents.

Commercial environments, such as Dymola and Scicos, along with open-source implementations, such as OpenModelica [7], enable modeling and simulation of models specified in the Modelica language. All of these tools perform a series of preprocessing steps (model flattening, index reduction, sorting and optimizing the equations) and convert the model to a set of explicit Ordinary Differential Equations (ODEs) of the form $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$. Efficient C++ code is then generated to perform the simulation. Numerical ODE solvers are provided that invoke the right-hand side evaluation of the ODEs at discrete time steps t_k, in order to compute the next value of the state vector $\mathbf{x_{k+1}}$. Thus, the commonly used simulation environments make use of **time slicing**, i.e., their underlying simulation algorithms are based on time discretization rather than state quantization.

In the end of the nineties, a new class of algorithms for numerical integration based on **state quantization** and the DEVS formalism was introduced by Zeigler [17]. Improving the original approach of Zeigler, Kofman developed a first-order non-stiff Quantized State System (QSS) algorithm in 2001 [14], followed later by second- and third-order accurate non-stiff solvers, called QSS2 [10] and QSS3 [13], respectively. Finally, the family of QSS methods has been further expanded and includes now also stiff system solvers (LIQSS [15]) as well as solvers for marginally stable systems (CQSS [3]). QSS methods have been theoretically analyzed to exhibit nice stability, convergence, and error bound properties, [4, 13, 14], and in general come with several advantages over classical approaches.

Most of the classical methods that use discretization of time, need to have their variables updated in a **synchronous** way. This means that the variables

that show fast changes are driving the selection of the time steps. In a stiff system with widely-spread eigenvalues, i.e., with mixed slow and fast subsystems, the slowly changing state variables will have to be updated much more frequently than necessary, thus increasing substantially the computation time of the simulation. On the other hand, the QSS methods allow for **asynchronous** variable updates, allowing each state variable to be updated at its own pace, and specifically when an event triggers its evaluation. Furthermore, as most systems are sparse, when a state variable x_i changes its value, it suffices to evaluate only those components of $\mathbf{f}$ that depend on x_i, allowing for an additional **significant reduction of the computational costs**. In [5], comparisons have been performed between the standard DASSL solver and QSS3 on synthetically generated sparse linear models that demonstrate the superiority of QSS methods, theoretically expected, when simulating sparse systems.

Another advantage of QSS methods concerns the simulation of **discontinuous** systems with frequent switching behavior, e.g. power electronic circuits. Standard Dymola and OpenModelica software handle discontinuities by means of **zero-crossing functions** that need to be evaluated at each step. When any of them changes its sign, the solver knows that a discontinuity occurred. Then an iterative process is initiated to detect the exact time of that event. In contrast, QSS algorithms offer **dense output**, i.e., they do not need to iterate to detect the discontinuities. They rather predict them. This feature, besides improving on the overall computational performance of these solvers, enables **real-time simulation**. Since in a real-time simulation the computational load per unit of real time must be controllable, Newton iterations are usually not acceptable.

Finally, DEVS methods [12] provide a formal unified framework for the simulation of **hybrid systems**, where continuous-time, discrete-time, and discrete-event models can coexist as subcomponents of a single model.

Therefore, **QSS methods** and the principle of **state quantization** appear promising in the context of simulating certain classes of real-world problems. However, in order to simulate a system with QSS methods in the PowerDEVS environment [2], the user needs to have a thorough understanding of DEVS systems. More specifically, the model needs to be manually converted to an explicit ODE form, dependencies between subsystems need to be identified, and the corresponding DEVS structure needs to be provided. Even if a user possesses the required knowledge to do so, this approach is feasible only for very small systems.

PowerDEVS does not support object-oriented modeling, whereas Modelica does. For all these reasons, it is much more convenient for a user to formulate models in the Modelica language than in PowerDEVS.

This work aims to bridge the gap between the powerful object-oriented modeling platform of Modelica on the one hand, and the equally powerful simulation platform of PowerDEVS on the other. In [5], a first version of the interface between OpenModelica and PowerDEVS, for systems without discontinuities, has been presented and analyzed. This study extends the previously discussed interface to include discontinuous models and brings us one step closer to the final goal, enabling a modeler to formulate arbitrary models in the Modelica language, while automatically simulating them in PowerDEVS.

1.1 Relevance of Work

In [5] a first version of an interface between OpenModelica and PowerDEVS for non-stiff and non-discontinuous models has been presented. The current article extends upon [5] to enable the simulation of discontinuous models. To our knowledge there exist no other approaches that automatically translate Modelica models to the DEVS formalism. Research efforts have been reported that implemented Modelica libraries allowing DEVS models to be formulated within a Modelica environment [1, 16], but these approaches require from the users to understand the DEVS framework, as they would have to model their system in the DEVS formalism in order to make use of these libraries.

Furthermore, this is the first work offering a comparison of the run-time efficiency and simulation accuracy of various solvers (DASSL, Radau IIa, Dopri45) in Dymola and OpenModelica against QSS methods. In earlier publications describing QSS methods [10, 11, 12, 14], there can be found examples that demonstrate the superiority of the run-time efficiency of QSS methods, but the comparisons were performed after manual modeling in PowerDEVS directly, i.e. they did not make use of the same original models formulated in Modelica.

In contrast, our approach enables a Modelica user to simulate a Modelica model using QSS solvers without any explicit manual transformation. Addi-

tionally, it allows for the automatic transformations of large-scale models to the DEVS formalism, which is a difficult if not unfeasible task even for experts in DEVS modeling.

The article is organized as follows: Section 2 provides a brief introduction of the QSS methods. Section 3 describes theoretically what is needed in order to simulate a Modelica model with discontinuities employing the QSS algorithms. In Section 4, the actual implementation of the interface between OpenModelica and PowerDEVS is presented. Section 5 describes the simulation results comparing the various solvers in Dymola and the OpenModelica runtime environment with the QSS methods as implemented in PowerDEVS. Finally, Section 6 concludes this study, lists open problems, and offers directions for future work.

2 QSS Simulation

Consider a time-invariant ODE system:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t)) \tag{1}$$

where $\mathbf{x}(t) \in \mathbb{R}^n$ is the state vector. The QSS method, [14], approximates the ODE of Eq. 1 as:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t)) \tag{2}$$

where $\mathbf{q}(t)$ is a vector containing the quantized state variables, which are quantized versions of the state variables $\mathbf{x}(t)$. Each quantized state variable $q_i(t)$ follows a piecewise constant trajectory via the following quantization function with hysteresis:

$$q_i(t) = \begin{cases} x_i(t) & \text{if } |q_i(t^-) - x_i(t)| = \Delta Q_i, \\ q_i(t^-) & \text{otherwise.} \end{cases} \tag{3}$$

where the quantity ΔQ_i is called **quantum**. The quantized state $q_i(t)$ only changes when it differs from $x_i(t)$ by more than ΔQ_i. In QSS, the quantized states $\mathbf{q}(t)$ follow piecewise constant trajectories, and since the time derivatives, $\dot{\mathbf{x}}(t)$, are functions of the quantized states, they are also piecewise constant, and consequently, the states, $\mathbf{x}(t)$, are composed of piecewise linear trajectories.

Unfortunately, QSS is a first-order accurate method only, and therefore, in order to keep the simulation error small, the number of steps performed has to be large.

To circumvent this problem, higher-order methods have been proposed. In QSS2 [10], the quantized state variables evolve in a piecewise linear way with the state variables following piecewise parabolic trajectories. In the third-order accurate extension, QSS3 [13], the quantized states follow piecewise parabolic trajectories, while the states themselves exhibit piecewise cubic trajectories.

3 Simulation of Discontinuous Modelica Models with QSS Methods

In this section we shall describe a potential way to simulate a Modelica model using QSS methods. For simplicity, we shall assume that the model is described by an ODE system, but we note that the interface successfully handles DAE systems as well. Let us write Eq. 2 expanded to its individual component equations, forgetting for a while the discontinuous part:

$$\begin{aligned} \dot{x}_1 &= f_1(q_1, \ldots, q_n, t) \\ &\vdots \\ \dot{x}_n &= f_n(q_1, \ldots, q_n, t) \end{aligned} \tag{4}$$

If we consider a single component of Eq. 4, we can split it into two equations:

$$q_i = Q(x_i) = Q(\int \dot{x}_i \, dt) \tag{5}$$

$$\dot{x}_i = f_i(q_1, \ldots, q_n, t) \tag{6}$$

3.1 Accounting for Discontinuities

Discontinuities in dynamical systems are closely related to the notion of events. We can distinguish two types of events, time events and state events.

3.1.1 Time Events

Time events correspond to changes of states as a function of the built-in continuously evolving variable **time**. Such events can be scheduled in advance, since it is possible to predict the point in time when they occur. Time events in Modelica are specified basically in two ways [6]:

- With a conditional discrete-time expression that contains the variable *time* (e.g. in a when-statement) of the form:
  time >= *discrete-time expression*, e.g. $t >= t_e$

- With a periodic ***sample*** statement of the form: *sample(first, interval)* that triggers events at predefined time instants.

The first case can be taken care of by formulating a zero-crossing function of the form:

$$g(t) = t - t_e$$

When g(t) crosses through zero, an event should be produced. We shall see later, which DEVS blocks need to be defined to generate the events. The *sample()* statement can be handled easily by adding a dedicated DEVS atomic model that provokes events at the predefined time points.

3.1.2 State Events

State events are related to discrete changes in the state variables during the simulation as a function of other state variables reaching some threshold value. Therefore, they cannot be scheduled in advance. A state event can be specified by means of *when* or *if-then-else* statements involving one or more state variables. When a model is compiled by either OpenModelica or Dymola, state events are translated into **zero-crossing functions** of the form $g_i(\mathbf{x},t)$. During the execution of the simulation the zero-crossing functions are being constantly monitored and when function $g_i(\cdot)$ crosses through zero, a discontinuity is detected and handled accordingly. Therefore, we can directly exploit the zero-crossing functions generated by OpenModelica to identify state events in an identical fashion as with time events. All we need is a **Static Function** block evaluating the zero-crossing function and a **Zero-Cross Detection** block that detects when a zero-crossing takes place.

3.2 DEVS structure

The **DEVS** formalism [17] allows to describe both the continuous and discontinuous parts of the model via a coupling of simpler DEVS atomic models. More specifically, we need to define:

- A **Quantized Integrator** block (Eq. 5) that takes as input the derivative $\dot{x}_i$ and outputs q_i.
- A **Static Function** block that receives the sequence of events, $q_1,\ldots,q_n$, and calculates the sequence of state derivative values, $\dot{x}_i$ (Eq. 6). The same block can be used for the evaluation of the zero-crossing functions $g_i(\cdot)$
- A **Cross-Detection** block that receives as input the evaluated zero-crossing function and generates an output event when its input crosses throught zero.

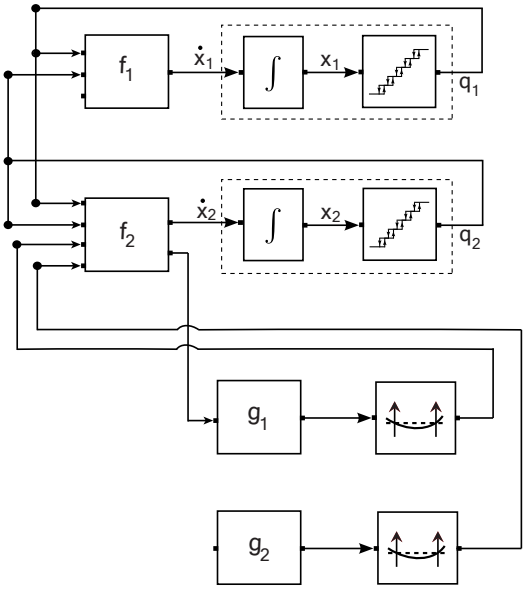

Figure 1: Coupled DEVS model for QSS simulation of a discontinuous model with 2 states and 2 zero-crossing functions $g_1(\cdot)$ and $g_2(\cdot)$.

Therefore, we can simulate a Modelica discontinuous model using a coupled DEVS model consisting of the blocks described above. A block diagram representing the final DEVS model for an example system with 2 state variables and 2 zero-crossing functions is shown in Fig. 1.

4 OpenModelica to PowerDEVS (OMPD) Interface

This section describes the work done to enable the simulation of Modelica models in PowerDEVS using QSS algorithms. The current version of the interface does not yet support **when** clauses and **sample** statements.

4.1 What is Needed by PowerDEVS

Let us first concentrate on what PowerDEVS requires in order to perform the simulation of a Modelica model. As depicted in Fig. 1, an essential component of a PowerDEVS simulation is the graphical structure. In PowerDEVS, the structure is provided in the

form of a dedicated **.pds structure file** that contains information about the blocks (nodes) of the graph as well as the connections (edges) between those blocks. More specifically, we need to add in the structure:

- A **Quantized Integrator** block for each state variable with $\dot{x}_i$ as input and q_i as output.
- A **Static Function** block for each state variable that receives as input the sequence of events, $q_1, \ldots, q_n$, and calculates $\dot{x}_i = f_i(\mathbf{q})$.
- A **Static Function** block for each one of the zero-crossing functions $g_i(\cdot)$ generated by OpenModelica that receives as inputs the dependencies of $g_i(\cdot)$ and evaluates the function in the output port.
- A **Cross-Detection Block** block after each one of the zero-crossing static functions. The cross-detection block outputs an event if a zero-crossing has been identified.
- A **connection (edge)** is added between two blocks if and only if there is a dependence between them.

Having correctly identified the DEVS structure, we need to specify what needs to be calculated inside each of the static function blocks. The different blocks need to have access to different pieces of information.

In the current implementation, a **.cpp code file** is generated that contains the code and parameters for all blocks in the structure. The generated code file contains the following information:

- For each **Quantized Integrator** block, the initial condition, error tolerance, and integration method (QSS, QSS2, QSS3).
- For each **Static Function**, the equations/expressions needed in order to calculate the derivative of each state variable in the system. Furthermore, the desired error tolerance is provided together with a listing of all input and output variables of the specific block. If the static function represents a zero-crossing then it contains the respective function $g_i(\cdot)$.

4.2 What is Provided by OpenModelica

In Section 4.1, we described what PowerDEVS expects in order to perform the simulation. Our work focuses on an automatic way to simulate Modelica models using the QSS methods in PowerDEVS. Therefore, the PowerDEVS simulation files should be automatically generated exploiting the information contained in the Modelica model supplied as input. Luckily, existing software used to compile Modelica models, such as Dymola or OpenModelica, produces simulation code that contains all information needed by PowerDEVS. Thus, we were able to make use of an existing Modelica environment by modifying the existing code generation modules at the back end of the compiler to produce the files needed by PowerDEVS.

This work is based on modifying the OpenModelica Compiler (OMC), since it is open-source and has a constantly growing contributing community. OMC takes as input a Modelica source file and translates it first to a flat model. The flattening consists of parsing, type-checking, performing all object-oriented operations such as inheritance, modifications, etc. The flat model includes a set of equation declarations and functions with all object-oriented structure removed. Then index reduction is performed on the set of model equations in order to remove algebraic dependence structures between state variables. The resulting equations are then analyzed, sorted in Block Lower Triangular (BLT) form, and optimized. Finally, the code generator at the back end of OMC produces C++ code that is then compiled. The resulting executable is used for the simulation of the model.

The information needed to be extracted from the OMC compiler is contained mainly in the DLOW structure, where the following pieces of information are defined:

- Equations: $E = \{e_1, e_2, \ldots, e_N\}$.
- Variables: $V = \{v_1, v_2, \ldots, v_N\} = V_S \bigcup V_R$ where V_S is the set of state variables with $|V_S| = N_S \leq N$ and V_R is the set of all other variables in the model.
- BLT blocks: subsets of equations $\{e_i\}$ needed to be solved together because they are part of an algebraic loop.
- Zero-Crossings: $G = \{g_1, g_2, \ldots, g_K\}$.
- Incidence matrix: An $N \times N$ adjacency matrix denoting, which variables are contained in each equation.

The OMPD interface utilizes the above information and implements the following steps:

1. **Equation splitting** : The interface identifies the equations needed in order to compute the derivative $\dot{x}_i = f_i(\mathbf{q})$ for each state variable. Then the split equations can be assigned to static function blocks according to the state derivative evaluation they are involved in.

2. **Mapping split equations to BLT blocks** : The equations are mapped back to BLT blocks of equations in order to be able to generate simulation code for solving linear/non-linear algebraic loops.

3. **Identifying zero-crossing functions** : The zero-crossing functions generated by OMC are extracted and assigned to separate static function blocks.

4. **Constructing generalized incidence matrix** : The $N \times N$ adjacency matrix has to be expanded to include also the zero-crossing functions and the variables involved in them. Thus, it has to be expanded by adding K rows corresponding to the K zero-crossing functions $g_i(\cdot)$. The result is a generalized $K \times N$ adjacency matrix.

5. **Generating DEVS structure** : In order to correctly generate the DEVS structure of the model, the dependencies between the individual DEVS blocks need to be resolved. This is accomplished by employing the generalized incidence matrix to find the corresponding inputs and outputs for each block.

6. **Generating the .pds structure file**: Having correctly produced the DEVS structure for PowerDEVS, outputting the respective .pds structure file is straightforward.

7. **Generating static blocks code** : In this step, the functionality of each static block is defined via the simulation code provided in the **.cpp code file**. Each static block needs to know its inputs and outputs, identified by the DEVS structure, as well as the BLT blocks needed to compute the corresponding state derivatives. The static blocks that are responsible for the discontinuities contain the zero-crossing functions $g_i(\cdot)$ generated by OMC. Then, the existing code generation module of OMC is employed to provide the actual simulation code for each static block, since it has already been optimized to solve linear and non-linear algebraic loops.

8. **Generating the .cpp code file**: The code for the static blocks is output in the .cpp code file along with other needed information.

5 Simulation Results

5.1 Benchmark Framework

In this section, the simulation results obtained using the OMPD interface are presented and discussed. The goal is to compare the run-time efficiency and accuracy of the QSS methods against other simulation software environments. More specifically, we want to compare QSS3 and QSS2 methods in PowerDEVS v2.0 against the DASSL, Radau IIa, and Dopri45 solvers implemented in Dymola v7.4 and the DASSL solver of OpenModelica v1.5.1.

DASSL was chosen as it represents the state-of-the-art multi-purpose stiff DAE solver used by most commercial simulation environments today. **Radau IIa** was included in the comparisons, because a single-step (Runge-Kutta) algorithm is supposed to be more efficient than a multi-step (BDF) algorithm when dealing with heavily discontinuous models, because step-size control is more expensive for the latter methods [4]. Finally, **Dopri45** was chosen, because it is an explicit Runge-Kutta method in contrast to both DASSL and Radau IIa, which are implicit algorithms that may be disadvantaged when simulating non-stiff systems.

As benchmark problems we focused on two real-world systems exhibiting heavily discontinuous behavior, namely a half-way rectifier circuit, modeled graphically with standard Modelica components as depicted in Fig. 2, and the switching power converter circuit provided in Fig. 3.

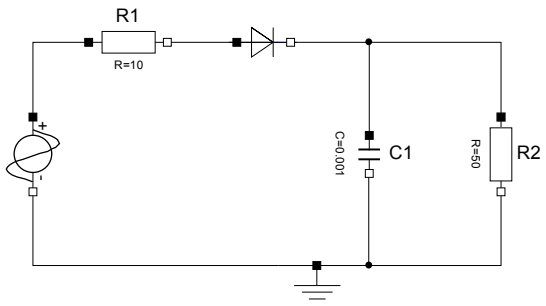

Figure 2: Graphical representation of the half-way rectifier

In order to measure the execution time for each simulation algorithm, the reported simulation time

from each environment was used. Dymola reports CPU-time for integration, OpenModelica reports timeSimulation, and PowerDEVS the elapsed simulation time. To record pure simulation time, the generation of output files was suppressed in all cases. Testing has been carried out on a Dell 32bit desktop with a quad core processor @ 2.66 GHz and 4 GB of RAM. The measured CPU time should not be considered as an absolute ground-truth since it will vary from one computer system to another, but the relative ordering of the algorithms is expected to remain the same.

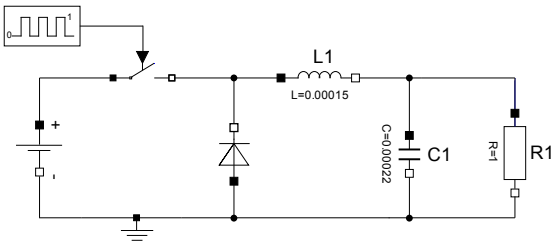

Figure 3: Graphical representation of the switching power converter

Calculating the accuracy of the simulations can only be performed approximately, since the state trajectories in the two models cannot be computed analytically. To estimate the accuracy of the simulation algorithms for a given setting, reference trajectories $(\mathbf{t}^{\mathbf{ref}}, \mathbf{y}^{\mathbf{ref}})$ have to be obtained. To this end, Dymola was employed using the default DASSL solver with a very tight tolerance of 10^{-12} and requesting 10^5 output points. Furthermore, in order to verify the accuracy of the reference solution, a second reference solution was computed using QSS3 in PowerDEVS with the tolerance set to 10^{-12}. However, we only report the simulation error against the Dymola solution since the difference between both reference solutions is on the order of 10^{-6}.

To calculate the simulation error, each one of the simulated trajectories was compared against the two reference solutions. To achieve this goal, we forced all solvers to output 10^5 equally spaced points for obtaining simulation trajectories $(\mathbf{t}^{\mathbf{ref}}, \mathbf{y}^{\mathbf{sim}})$ without changing the integration step. Then, the mean absolute error is calculated as:

$$error = \frac{1}{|t^{ref}|} \sum_{i=1}^{|t^{ref}|} |y_i^{sim} - y_i^{ref}| \qquad (7)$$

In the case of more than one state variables, we report the mean error over all state trajectories.

5.2 Half-Way Rectifier

The half-way rectifier circuit exhibits only one state variable, namely the voltage across the capacitor C1, and the model is simulated during 1 sec. In Fig. 4, the state trajectory calculated with QSS3 and a tolerance of 10^{-4} is depicted. Comparing the simulation results listed in Table 1, the following conclusions can be reached:

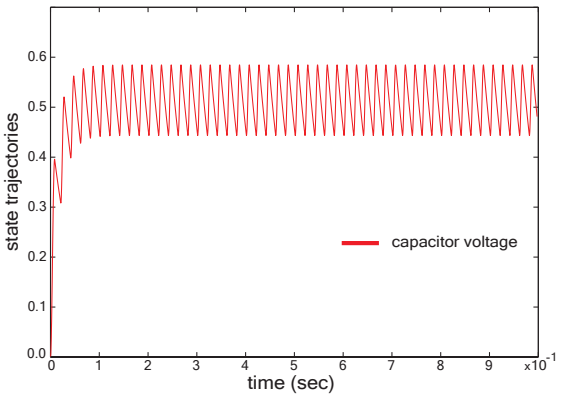

Figure 4: Simulated state trajectories with QSS3 for the half-way rectifier circuit.

There is a **substantial difference in execution efficiency between Dymola and OpenModelica** using the DASSL solver, with Dymola being around 10 times faster than OpenModelica in spite of the fact that both environments make use of the same solver software and even the same root solver (event detection) algorithms. We postulate that this difference is primarily caused by the fact that OMC does not involve **tearing**. Thereby the solution of algebraic loops becomes much less efficient, and also the integration itself suffers, because the number of iteration variables in DASSL equals the number of state variables plus the number of tearing variables. Without tearing, DASSL needs to include all variables appearing inside algebraic loops among the set of its iteration variables.

On the other hand, even though the QSS3 simulations are based on code generated by OMC, we observe that QSS3 is slightly more efficient than DASSL in Dymola. To perform the simulation for an achieved error of the order of 10^{-4}, QSS3 required 0.014 sec while DASSL 0.022 sec. Therefore, **the use of the OMPD interface and the simulation in PowerDEVS employing QSS3 speeds up the simulation by a factor of 20** compared to OpenModelica. It needs to be remarked that it is not fair to compare QSS3 with the DASSL simulation of Dy-

Table 1: This table depicts the simulation results of various algorithms for the half-way rectifier circuit for a requested simulation time of 1 sec. The comparison performed includes required CPU time (in sec) as well as the simulation accuracy relative to the reference trajectory obtained in Dymola.

			CPU time (sec)	**Simulation Error**
Dymola	**DASSL**	10^{-3}	0.019	1.45E-03
	DASSL	10^{-4}	0.022	2.35E-04
	Radau IIa	10^{-7}	0.031	2.20E-06
	Dopri45	10^{-4}	0.024	4.65E-05
PowerDEVS	**QSS3**	10^{-3}	0.014	2.59E-04
	QSS3	10^{-4}	0.026	2.23E-05
	QSS3	10^{-5}	0.041	2.30E-06
	QSS2	10^{-2}	0.242	3.02E-03
	QSS2	10^{-3}	0.891	3.04E-04
	QSS2	10^{-4}	3.063	3.00E-05
OpenModelica	**DASSL**	10^{-3}	0.265	3.80E-03
	DASSL	10^{-4}	0.281	5.40E-04

mola because of the fact analyzed earlier, namely the lack of tearing in OMC. The solution of the algebraic loops in QSS3 is based on code generated by OMC, and therefore, the inefficiencies in the compilation of the OMC are being propagated to the QSS3 simulation as well. For this reason, **we need to compare the results in PowerDEVS with the ones obtained by OpenModelica and not by Dymola**. However, it is encouraging to see that the improvement achieved over the standard OMC simulation using QSS-based solvers is such that we are able to obtain simulation results that are even more efficient than those obtained using the commercial Dymola environment. If the QSS methods were implemented in Dymola, the simulation results obtained by the QSS methods would once again be considerably faster than the simulation results that Dymola achieves currently.

Performing an internal comparison between the QSS methods, it is obvious that QSS3 is much more efficient that QSS2. This is expected, since the QSS2 solver needs to take smaller steps compared to QSS3 in order to reach the desired accuracy. **Thus, we can conclude that the third-order QSS3 algorithm should be preferred for practical applications.**

For the sake of completeness we included in the comparison two more solvers included in the Dymola environment, Radau IIa and Dopri45. **Radau IIa** is an implicit variable-step Runge-Kutta method of order 5, while **Dopri45** is an explicit step-size controlled Runge-Kutta algorithm of order 5. For this specific example, Radau IIa failed to provide correct results unless the tolerance was lowered to 10^{-7}. Radau IIa with a less tight tolerance tries to utilize larger integration steps and, apparently, misses many of the events, i.e. the event localization employed by Dymola is not robust (conservative) enough. It needs to be noted further that the problem got considerably worse between Dymola 6 and Dymola 7, i.e., whereas Radau IIa missed a few events in Dymola 6, it misses many more events in Dymola 7. This is a quite serious issue that the Dynasim company should look into. The same problem was observed for Dopri45 as well, when the tolerance was set to 10^{-3}. Due to these problems, both Runge-Kutta algorithms require CPU times comparable to that needed by the standard DASSL solver, i.e., the inherent advantages of the single-step algorithms over a multi-step technique in dealing with heavily discontinuous models could not be exploited due to the inability of their current implementation to detect events reliably.

5.3 Switching Power Converter

The switching power converter exhibits two state variables, namely the current through the inductor L1 and the voltage across the capacitor C1. From Fig. 3, we see that there is a square wave source block that, when implemented directly, would call for use of a sample block. As the sample block has not yet been implemented in the interface, we worked around this problem by replacing the square wave source by a second-order marginally stable time-invariant system described by:

```
model SquareWaveGenerator
  Real x1(start=0.0);
```

Table 2: This table depicts the simulation results of various algorithms for the switching power converter circuit for a requested simulation time of 0.01 sec. The comparison performed includes required CPU time (in sec) as well as the simulation accuracy relative to the reference trajectory obtained in Dymola.

			CPU time (sec)	**Simulation Error**
Dymola	**DASSL**	10^{-3}	0.051	1.82E-04
	DASSL	10^{-4}	0.063	7.18E-05
	Radau IIa	10^{-3}	0.064	1.11E-07
	Radau IIa	10^{-4}	0.062	1.11E-07
	Dopri45	10^{-3}	0.049	6.38E-06
	Dopri45	10^{-4}	0.047	9.76E-06
PowerDEVS	**QSS3**	10^{-3}	0.049	1.41E-03
	QSS3	10^{-4}	0.062	1.68E-05
	QSS3	10^{-5}	0.250	8.96E-06
OpenModelica	**DASSL**	10^{-3}	50.496	-
	DASSL	10^{-4}	1.035	2.62E-02

```
  Real x2(start=1.0);
  Boolean pulse(start=true);
  parameter Real freq=1e4;
equation
  der(x1)=freq*4*x2;
  der(x2)=if (x1<0) then freq*4 else -freq*4;
  pulse=(x1>0);
  idealClosingSwitch.control = pulse;
end SquareWaveGenerator;
```

This is worth noting since it adds two more states to the model ($x1, x2$) and increases the computation time since the solver also has to simulate the marginally stable system. The chosen solution is by no means unique. The desired switching behavior could have been coded in many different ways.

The model was simulated for 0.01 sec, and in Fig. 5, the state trajectories calculated using the QSS3 solver with a tolerance of 10^{-4} are plotted. The simulation results for all algorithms under comparison are presented in Table 2.

The conclusions reached in the analysis of the results of the half-way rectifier circuit also hold for the switching power converter circuit as depicted in Table 2. **The QSS3 method performs well compared to the DASSL solver in Dymola, while it outperforms DASSL in OpenModelica and the second-order QSS2**. Radau IIa and Dopri45 simulate correctly even for large tolerance values in this example, but their run-time performance is not significantly better than that of DASSL or QSS3.

For the switching power converter circuit, the simulation errors estimated for DASSL in OpenModelica are quite large. This is suspicious, as it should not be

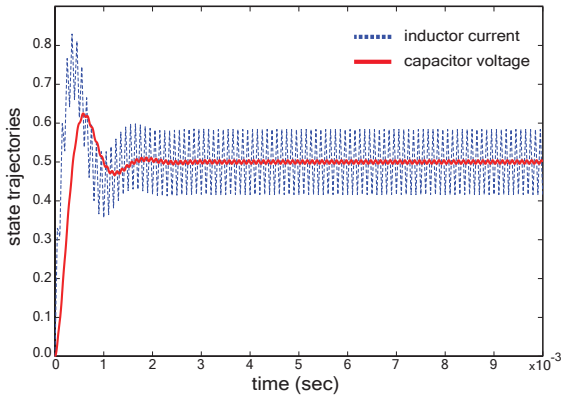

Figure 5: Simulated state trajectories with QSS3 for the switching power converter circuit.

the case. We noticed further that in OpenModelica for a relaxed tolerance of 10^{-3}, the simulation requires a substantial CPU time of 50 sec. The output files generated are also huge, around 500 MB, making it impossible to check if the simulated trajectories are correct or not. There seems to be something wrong with the compilation performed by the OMC in this example, but we cannot make any definite statements regarding this behavior yet.

6 Discussion

6.1 Conclusions

In this article, an extension of the interface between the OpenModelica environment and PowerDEVS presented in [5] is discussed and analyzed. The im-

plemented OPMD interface successfully handles discontinuities and allows to simulate real-world Modelica models with discontinuities using the PowerDEVS simulation software.

Comparisons on two example models were performed, demonstrating the increased efficiency of QSS3 over the standard DASSL solver. The proposed OMPD interface utilizes code generated by the OpenModelica compiler, therefore comparisons must be performed between QSS3 and the DASSL solver of OpenModelica, where we achieve a more than 20-fold decrease in the required CPU time.

Furthermore, comparisons show that the efficiency of QSS3 simulations using code generated by the OMC is comparable to simulations run in Dymola using the built-in DASSL solver, in spite of the fact that Dymola offers much more sophisticated model preprocessing, such as a well-tuned tearing algorithm for the efficient simulation of models involving algebraic loops. Hence we are very optimistic that there would result a significant gain in simulation efficiency if the OMPD interface were to be implemented as part of the back end of the Dymola compiler even in a single-processor implementation, i.e., without exploiting the fact that QSS-based solvers are naturally asynchronous and can therefore be much more easily and elegantly distributed over a multi-core architecture for efficient real-time simulation.

6.2 Future Work

We have shown that the implemented OMPD interface successfully allows a user to simulate Modelica models with discontinuities using PowerDEVS and QSS solvers. However, there still remain open problems that need to be addressed in the future.

As a next step full support for hybrid models needs to be incorporated. This requires the implementation of sample statements and when-clauses. The OMPD interface does not yet support a stiff-system solver. There exist already stiff QSS solvers of orders 1 to 3 [15], which, however, are not yet supported by the interface, because they have not yet been included in the general release of the PowerDEVS software. For this reason, we had to be careful to choose example systems that do not lead to stiff models.

Next, many more models will need to be tested. In particular, we shall need to run all example codes of the Modelica Standard Library that the OMC is able to handle through the interface to verify that the OMPD is capable of handling all models that are thrown its way.

Finally, we shall make use of the new platform for investigating parallel simulation on a multi-processor architecture. There are many unresolved issues here to be tackled, such as the load balancing problem, i.e., how to optimally distribute the simulation code over a multi-processor architecture. The fact that QSS-based solvers can be easily parallelized does not tell us yet how to optimally make use of that possibility.

7 Acknowledgments

We would like to acknowledge the help and support of the PELAB group at Linköping University and in particular Martin Sjölund, Per Östlund, Adrian Pop, and Prof. Peter Fritzson. Also, we would like to thank Willi Braun of Bielefeld University and Jens Frenkel of TU Dresden for their helpful comments and discussions about the OMC back end.

References

[1] Tamara Beltrame and François E. Cellier. Quantised State System Simulation in Dymola/Modelica using the DEVS Formalism. In *Modelica*, 2006.

[2] Federico Bergero and Ernesto Kofman. Powerdevs: a tool for hybrid system modeling and real-time simulation. *SIMULATION*, 2010.

[3] François Cellier, Ernesto Kofman, Gustavo Migoni, and Mario Bortolotto. Quantized State System Simulation. In *Proceedings of SummerSim 08 (2008 Summer Simulation Multiconference)*, Edinburgh, Scotland, 2008.

[4] François E. Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer-Verlag, New York, 2006.

[5] Xenofon Floros, François E. Cellier, and Ernesto Kofman. Discretizing Time or States? A Comparative Study between DASSL and QSS. In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, EOOLT, Oslo, Norway, October 3, 2010*, pages 107–115, 2010.

[6] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-Interscience, New York, 2004.

[7] Peter Fritzson, Peter Aronsson, Hakan Lundvall, Kaj Nystrom, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Development Environment. *Proceedings of the 46th Conference on Simulation and Modeling (SIMS'05)*, pages 83–90, 2005.

[8] Peter Fritzson and Peter Bunus. Modelica - A General Object-Oriented Language for Continuous and Discrete-Event System Modeling and Simulation. In *Annual Simulation Symposium*, pages 365–380, 2002.

[9] Peter Fritzson and Vadim Engelson. Modelica - a unified object-oriented language for system modeling and simulation. In Eric Jul, editor, *ECOOP '98 - Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 67–90. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0054087.

[10] Ernesto Kofman. A Second-Order Approximation for DEVS Simulation of Continuous Systems. *Simulation*, 78(2):76–89, 2002.

[11] Ernesto Kofman. Quantization-Based Simulation of Differential Algebraic Equation Systems. *Simulation*, 79(7):363–376, 2003.

[12] Ernesto Kofman. Discrete Event Simulation of Hybrid Systems. *SIAM Journal on Scientific Computing*, 25:1771–1797, 2004.

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

[14] Ernesto Kofman and Sergio Junco. Quantized-state systems: a DEVS Approach for continuous system simulation. *Trans. Soc. Comput. Simul. Int.*, 18(3):123–132, 2001.

[15] Gustavo Migoni and Ernesto Kofman. Linearly Implicit Discrete Event Methods for Stiff ODEs. *Latin American Applied Research*, 2009. In press.

[16] Victor Sanz, Alfonso Urquía, François E. Cellier, and Sebastián Dormido. System Modeling Using the Parallel DEVS Formalism and the Modelica Language. *Simulation Modeling Practice and Theory*, 18(7):998–1018, 2010.

[17] Bernard P. Zeigler and J. S. Lee. Theory of Quantized Systems: Formal Basis for DEVS/HLA Distributed Simulation Environment. *Enabling Technology for Simulation Science II*, 3369(1):49–58, 1998.