Discretizing Time or States? A Comparative Study between DASSL and QSS - Work in Progress Paper -

Xenofon Floros[1] 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
kofman@fceia.unr.edu.ar

Abstract

In this study, a system is presented and analyzed that automatically translates a model described within the **Modelica** framework into the Discrete Event System Specification (**DEVS**) formalism.

More specifically, this work interfaces the open-source implementation of Modelica, **OpenModelica**, and one particular software tool for DEVS modeling and simulation, the **PowerDEVS** environment, which implements the Quantized State Systems (**QSS**) integration methods introduced by Kofman.

The interface enables the automatic simulation of large-scale models with both DASSL (using the OpenModelica run-time environment) and QSS (using PowerDEVS) and extracts features, such as accuracy and simulation time, that allow a quantitative comparison of these integration methods. In this way, meaningful insight can be obtained on their respective advantages and disadvantages when used for simulating real-world applications. Furthermore, the implemented interface allows any user without any knowledge of DEVS and/or QSS methods to simulate their systems in PowerDEVS by supplying a Modelica model as input only.

Keywords OpenModelica, DASSL, PowerDEVS, QSS, sparse system simulation

1. Introduction

Modelica [4, 5] is a an object-oriented, equation-based language that enables a standardized way to model complex physical systems containing, e.g., mechanical, electrical, electronic, hydraulic, thermal, control, electric power, or process-oriented subcomponents. The Modelica language allows the representation of continuous and hybrid models with a set of non-causal equations.

Modelica modeling environments, such as Dymola, Scicos, and the open-source OpenModelica software [3], after performing a series of preprocessing steps (model flattening, sorting and optimizing the equations, index reduction), convert the model to a set of explicit ODEs of the form:

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

The built-in simulation environments provide routines (solvers) that invoke the right-hand side evaluation of Eq. 1 at discrete time steps t_k, in order to compute the next value of the state vector $\mathbf{x_{k+1}}$. At least in the case of Dymola and OpenModelica, efficient C++ code is generated in order to perform the simulation. Both software environments make use of **time slicing**, i.e., their underlying simulation algorithms are based on time discretization rather than state quantization.

Recently, a new class of algorithms for the numerical integration of ODEs based on **state quantization** and the DEVS formalism introduced by Zeigler [13] was proposed. A first-order non-stiff Quantized State System (QSS1) algorithm was introduced by Kofman in 2001 [6], followed by second and third-order accurate non-stiff solvers, called QSS2 [10] and QSS3 [9], respectively. The family of QSS methods presented are implemented in PowerDEVS, [11], a DEVS-based simulation software. In the mean time, also stiff QSS solvers as well as QSS solvers for dealing with marginally stable systems were introduced.

QSS methods have been theoretically analyzed to exhibit nice stability, convergence, and error bound properties, [2, 9, 10], and in general come with the following benefits 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

3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. October, 2010, Oslo, Norway.

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 $\mathbf{f}$ in Eq. 1 that depend on x_i, allowing for a **significant reduction of the computational costs**.

- Dymola and OpenModelica handle **discontinuities** using **zero-crossing functions** that need to be evaluated at each step, and when they change their sign, the solver knows that a discontinuity occurred. Then an iterative process is initiated in order to detect the exact time of that event. In contrast, QSS methods provide dense output and do not need to iterate to detect discontinuities, but 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 admitted for use in real-time simulation.
- Another important advantage of DEVS methods arises in the context of **hybrid systems**, where continuous time, discrete time, and discrete event models can coexist as subcomponents of an overall system. DEVS methods [8] provide a unified simulation framework for hybrid models, because all of these model types can be represented as valid DEVS models.

Therefore, **state quantization** and the QSS methods appear promising in the context of simulating certain classes of real-world problems. However in order to simulate systems in PowerDEVS directly, the user will have to manually convert his or her model to an explicit ODE form. This is only feasible in the case of very small systems. PowerDEVS unfortunately is not object oriented. For this reason, it is much more convenient for a user to formulate models in the Modelica language than in PowerDEVS.

This works 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. The interface between OpenModelica and PowerDEVS, introduced in this article, allows a modeler to formulate his or her model in the Modelica language, while simulating it in PowerDEVS. The necessary compilation of the Modelica model to PowerDEVS is fully automatic, and the user does not need to know anything about either DEVS or QSS in order to take advantage of it.

1.1 Relevance of Work

The run-time efficiency of the DASSL and QSS solvers, when used to simulate Modelica models, has so far not been compared in an automated, large-scale framework. In earlier publications describing QSS methods, [6, 7, 8, 10], there can be found examples that demonstrate the superiority of the run-time efficiency of QSS methods, when simulating sparse and discontinuous systems, but the comparison has invariably been restricted to small-scale models that could be easily modeled in PowerDEVS directly.

Furthermore, there have been other approaches, [1, 12], to implement Modelica libraries that allow for DEVS models inside a Modelica environment, 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. In that context, the object orientation of continuous-time models is lost.

In contrast, our approach enables a Modelica user to simulate a Modelica model using QSS solvers without any explicit manual transformation. Furthermore, 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 without 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 DASSL solver of the OpenModelica run-time 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

Let a time invariant ODE system:

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

where $\mathbf{x}(t) \in \mathbb{R}^n$ is the state vector. The QSS1 method approximates the ODE in Eq. 2 as:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t)) \tag{3}$$

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

$$q_i(t) = \begin{cases} x_i(t) & \text{if } |q_i(t^-) - x_i(t)| = \Delta Q_i, \\ q_i(t^-) & \text{otherwise.} \end{cases} \tag{4}$$

where ΔQ_i is called quantum. Thus, the quantized state $q_i(t)$ changes if and only if it differs more than the value of the quantum from the state variable $x_i(t)$. In QSS1, the quantized states $\mathbf{q}(t)$ are following 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, $\mathbf{x}(t)$, themselves are composed of piecewise linear trajectories.

Unfortunately, QSS1 is a first-order accurate method only, and therefore, in order to keep the simulation error small, a large number of short integration steps needs to be calculated.

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 [9], the quantized states follow piecewise parabolic trajectories, while the states themselves exhibit piecewise cubic trajectories.

Eq. 3 reveals one of the key concepts and advantages of QSS methods. During simulation, steps are only performed when a quantized variable $q_i(t)$ changes substantially, i.e. when it deviates by more than a quantum from the corresponding state $x_i(t)$. Thus, QSS methods inherently focus on the part of the system that is active at a certain time point, which is particularly attractive for the simulation of sparse models of large real-world systems.

3. Modelica Models Simulation with QSS and DEVS

The Modelica language enables high-level modeling of complex systems. However, at the core of every Modelica model lies an Ordinary Differential Equation (ODE) system or, in general, a Differential Algebraic Equation (DAE) system that mathematically represents the system under consideration. We shall now show how a Modelica model can be simulated using QSS methods. For simplicity, we shall assume that the model is described by an ODE system.

Let us write again Eq. 3 expanded to its individual component equations:

$$\begin{aligned} \dot{x}_1 &= f_1(q_1, \ldots, q_n, t) \\ &\vdots \\ \dot{x}_n &= f_n(q_1, \ldots, q_n, t) \end{aligned} \tag{5}$$

If we consider a single component of Eq. 5, we can split it into two equations:

$$q_i = Q(x_i) = Q(\int \dot{x}_i \, dt) \tag{6}$$

$$\dot{x}_i = f_i(q_1, \ldots, q_n, t) \tag{7}$$

The **DEVS** formalism [13] allows to describe the above equations via a coupling of simpler DEVS models. More specifically:

- The first equation (Eq. 6) can be represented by an atomic DEVS model, called **Quantized Integrator**, with $\dot{x}_i$ as input and q_i as output.
- The second equation (Eq. 7) can also be represented as an atomic DEVS model, called **Static Function**, that receives the sequence of events, $q_1, \ldots, q_n$, and calculates the sequence of state derivative values, $\dot{x}_i$.

Thus in absence of discontinuities, we can simulate Eq. 5 using a coupled DEVS model consisting of the coupling of n Quantized Integrators and n Static Functions. A block diagram representing the final DEVS model is shown in Fig. 1.

The model depicted in Fig. 1 contains all possible connections between the state variables. However, real-world systems are sparse as each state normally depends on a small subset of other states only. Thus, the graphical DEVS structure is typically sparse for most practical applications.

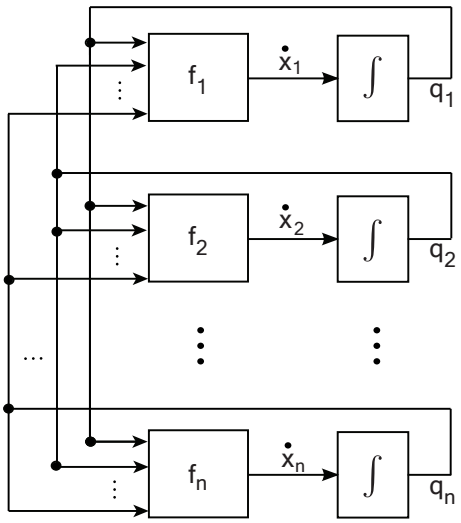

Figure 1. Coupled DEVS model for QSS simulation of Eq. 5

4. OpenModelica to PowerDEVS (OMPD) Interface

This section describes the work done to enable the simulation of Modelica models in PowerDEVS using QSS algorithms.

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 $\dot{x}_i$.
- A connection 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 (QSS1, 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.

4.2 What is Provided by OpenModelica

In Section 4.1, we described what must be contained in the files needed to perform simulations in PowerDEVS. The PowerDEVS simulation files should be generated automatically exploiting the information contained in the Modelica model supplied as input. Luckily, existing software used to simulate 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 simulation environment by modifying the existing code generation modules to produce the desired simulation files.

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 the equations are 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 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.
- 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 extracts the indices of the equations needed in order to compute the derivative of each state variable. To achieve this, it builds a dependence graph, where the nodes are the equations, and the edges represent dependences between these equations. The interface traverses the graph backwards from each state derivative until it cannot no longer reach any additional nodes.
2. **Mapping split equations to BLT blocks** : Having extracted the indices of the equations that calculate each state derivative, the equations are mapped back to BLT blocks of equations. This is needed in order to pass this information to the part of the OpenModelica compiler that is responsible for solving linear/non-linear algebraic loops.
3. **Mapping split equations to DEVS blocks** : The split equations are also mapped onto the static blocks of the DEVS structure. Since the state variables and the equations needed to compute them have been identified, they are assigned sequentially to static blocks in the DEVS structure. Each static block corresponds then to a state variable, and the lookup of equations in static blocks can be performed efficiently if needed in the future.
4. **Generating DEVS structure** : In order to correctly generate the DEVS structure of the model, the dependences between the state variables that are computed in each static block have to be resolved. This is accomplished by employing the incidence matrix and the mapping of equations to DEVS blocks from step 3 to find the corresponding inputs for each block. In Fig. 2, an example of a DEVS structure automatically generated for model M_1 is depicted.
5. **Generating the .pds structure file**: Having correctly produced the DEVS structure for PowerDEVS, outputting the respective .pds structure file is straightforward.
6. **Generating static blocks code** : In this step, the functionality of each static block is defined via the simulation code provided in the code.cpp 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, described by the mapped split equations. 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.
7. **Generating the .cpp code file**: The code for the static blocks is output in the .cpp code file along with other needed information.

4.3 Example Model M_1

Various tests have been performed in order to ensure that the implemented OMPD interface is performing the desired tasks correctly. Here are presented the results of processing the following second-order non-linear model through the implemented interface:

```
model M1
  Real x1;
  Real x2;
  Real u2;
equation
  der(x1)=x2-x1/10;
```

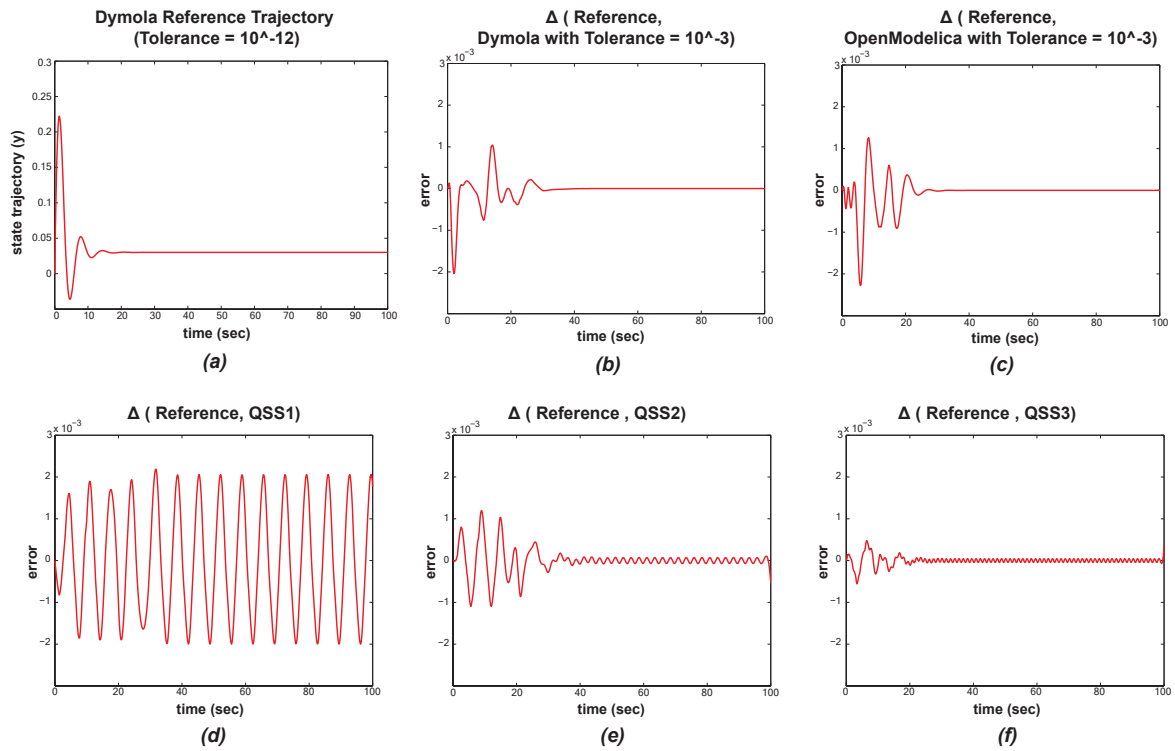

Figure 3. Simulation results for model M_1. In (a), the reference trajectory of state variable x_2 is depicted. It was computed using DASSL by setting the tolerance to 10^{-12}. In (b) and (c), the simulation errors of Dymola and OpenModelica relative to the reference solution are plotted with the tolerance value now set to 10^{-3}. The achieved accuracy is indeed in the order of 10^{-3} with the errors converging to zero in both simulations. In (d), (e), and (f), the simulation errors of QSS1, QSS2, and QSS3 are depicted. Again we observe that the desired accuracy of 10^{-3} is approximately attained. In QSS2 and QSS3, the errors decay also, but a small amplitude high-frequency oscillation around the steady-state remains. In QSS1, the errors don't converge to zero, but remain of the order of 10^{-3}, which is still within specs.

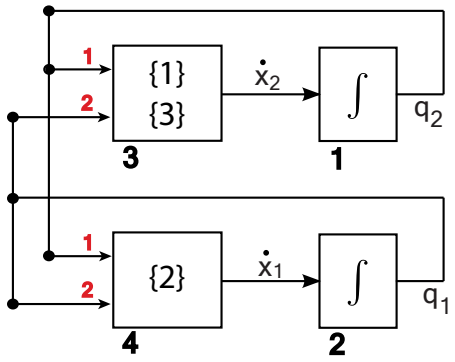

Figure 2. Automatically generated DEVS structure for M_1

```
  der(x2)=u2-x1;
  u2=(1-u2-x2)^3;
end M1;
```

The following information has been extracted from the OMC compiler:

```
Variables (3)
=========
1:  $u2:VARIABLE sys3, Real type: Real ...
2:  $x2:STATE sys3, Real type: Real  ...
3:  $x1:STATE sys3, Real type: Real ...

Equations (3)
=========
1 : $u2 = (1.0 - $u2 - $x2) ^ 3.0
2 : $DER$x1 = $x2 - $x1 / 10.0
3 : $DER$x2 = $u2 - $x1

BLT blocks
2,
1,
3,
```

In other words, OMC has identified that model M_1 has 2 state variables, x_1 and x_2, 1 algebraic variable, u_2, and 3 equations:

$$u_2 = (1 - u_2 - x_2)^3 \qquad \{1\}$$
$$\dot{x}_1 = x_2 - \frac{x_1}{10} \qquad \{2\}$$
$$\dot{x}_2 = u_2 - x_1 \qquad \{3\}$$

Each of the equations is placed in a BLT block. These are sorted according to the execution order as follows: {2}, {1}, {3}.

The automatically generated DEVS structure contained in the .pds file is illustrated in Fig. 2. Since both state variables are depending on each other, the obtained structure is full. The BLT blocks needed to calculate each state derivative are shown inside the static function blocks.

5. Simulation Results

In this section, the simulation results obtained by use of the OPMD interfaced are presented and discussed. First in order to provide a simple working example of the implemented interface, model M_1 was simulated using all three currently implemented QSS methods of PowerDEVS as well as the standard DASSL solvers of OpenModelica and Dymola. The stand-alone version of PowerDEVS contains additional QSS-based solvers for simulating stiff and marginally stable systems, but in the OPMD, only QSS1, QSS2, and QSS3 were included until now.

The article also presents preliminary results regarding the performance of QSS algorithms in comparison with DASSL when simulating sparse systems.

5.1 Simulation of a Simple Model (M_1)

In Fig. 3, the simulation results obtained for model M_1 are depicted. In Fig. 3(a), the reference trajectory of state variable x_2 is plotted. To obtain the reference trajectory, Dymola was employed using the default DASSL solver while setting the tolerance value to 10^{-12}. In the remaining panels, the simulation errors relative to the reference solution are shown that were obtained by setting the desired tolerance to 10^{-3}. Fig. 3(b) shows the simulation error obtained by Dymola using DASSL. Fig. 3(c) depicts the simulation error obtained by OpenModelica using DASSL. Figs. 3(d-f) graph the simulation error obtained by OpenModelica with PowerDEVS using QSS1, QSS2, and QSS3, respectively.

We observe that all solvers accomplish the desired task of keeping the global simulation error approximately within 10^{-3}. Using DASSL, this is not obvious, because DASSL only controls the local simulation error of a single integration step. Errors can in principle accumulate over time, but this didn't happen in this simple example. QSS-based algorithms all control the global simulation error, and consequently, are expected to perform as desired. In this simple example, the simulation errors decrease over time in all solvers except for QSS1. Yet, whereas the errors approach zero in the case of DASSL, a small-amplitude, high-frequency oscillation remains in the cases of QSS2 and QSS3.

These steady-state oscillations are bad news, because they will foce small step sizes upon us in steady state, if we decide for whatever reason to simulate the system over a longer time period. Such steady-state oscillations are observed frequently in the non-stiff QSS solvers. They will disappear when the OPMD is extended to the stiff LIQSS1, LIQSS2, and LIQSS3 solvers.

In QSS1, the errors do not decay over time in this example. After all, this is only a simple first-order non-stiff ODE solver. However, the errors remain approximately within 10^{-3}, i.e., the simulation still performs within the desired specifications.

Based on these results and a number of additional example models studied, we conclude that the OMPD interface is able to simulate arbitrary linear or non-linear Modelica models without discontinuities.

In the following experiments, only the QSS3 algorithm is compared to DASSL, since it is a third-order accurate method and is expected to be more efficient than the other implemented QSS methods.

5.2 Benchmark Framework

One of the goals of this study was to compare the performance of the standard DASSL solver of OpenModelica with the most efficient among the hitherto implemented methods of the QSS family, namely QSS3. Since the current interface implementation allows only for non-stiff models without discontinuities, we only focus on studying the effect of the sparsity of a model on the CPU performance of both algorithms. To achieve our goal, we need to be able to automatically generate models of arbitrary size and sparsity.

For the benchmark, we chose to generate linear models of the form:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} \tag{8}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the vector of state variables. Matrix $\mathbf{A}$ controls the dynamics of the generated system. Since we needed to control the eigenvalues of the system and avoid producing stiff systems, we constructed $\mathbf{A}$ as follows:

1. Generate real-valued random eigenvalues drawn from a Gaussian distribution : $eig \sim -\mathcal{N}(5, 2)$.
2. Create a diagonal matrix $\mathbf{D} = diag(eig)$.
3. Create a random orthogonal matrix $\mathbf{M}$.
4. Then matrix $\mathbf{A} = \mathbf{M} \cdot \mathbf{D} \cdot \mathbf{M}^T$ has the desired eigenvalues eig.

The constructed matrix $\mathbf{A}$ is a full matrix. **Sparsity** s is defined as the number of connections to every state variable. To achieve a certain sparsity level s, we set the $n - s$ absolute smallest elements of each row of $\mathbf{A}$ to zero. The absolute smallest elements were eliminated in order to minimize the impact on the eigenvalue locations of the resulting matrix $\tilde{\mathbf{A}}$. Having constructed a matrix $\tilde{\mathbf{A}}$ of given size n and sparsity s, it is straightforward to generate an equivalent Modelica model.

For the comparison simulations, the OpenModelica 1.5.0 environment was used with DASSL as the standard solver. The tolerance was set to 10^{-3}, and the simulation end time was set to 3 sec for both OpenModelica and PowerDEVS. Furthermore, the output file generation was disabled for both environments in order to measure the pure simulation time.

For each parameter configuration (n, s), 100 Modelica models were randomly generated and given as input to the

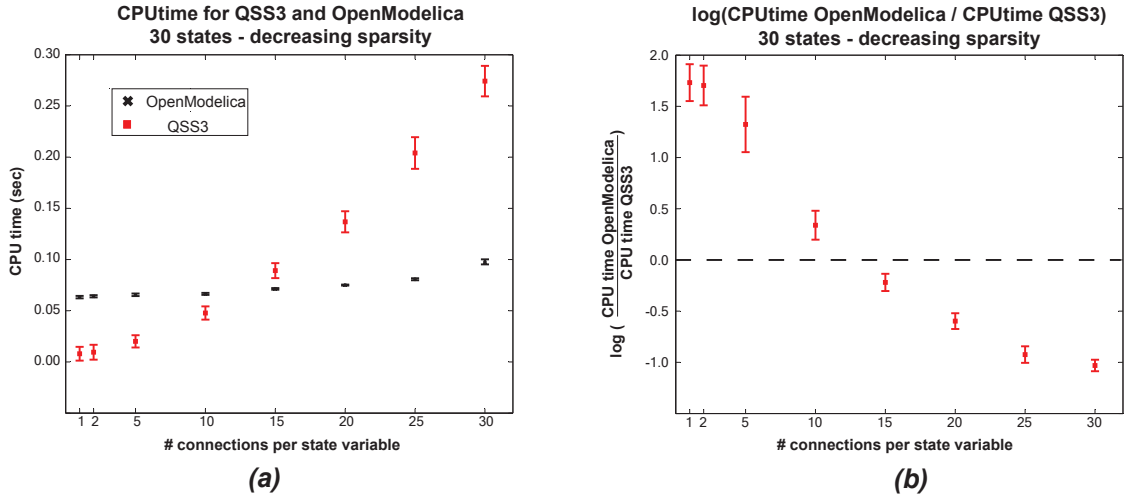

Figure 4. Simulation results for automatically generated models with $n = 30$ state variables and varying sparsity. In (a), the mean CPU time needed for simulating these systems using QSS3 and OpenModelica is plotted (red squares and black crosses, respectively).In (b), the logarithm of the ratio between the CPU time of OpenModelica and QSS3 is shown. Observing both plots, we conclude that QSS3 is more efficient than DASSL when simulating sparse systems ($s < 13$) Most real-world large-scale systems belong to that category.

standard OpenModelica compiler and via the OMPD interface to PowerDEVS. The CPU time needed for the simulation was measured for each generated executable. In order to obtain more reliable results, each simulation was repeated 10 times, and the median over all 10 repetitions was considered as the CPU time needed for each simulation. For each parameter setting (n, s) the mean of the CPU time measurements is reported along with $\pm$ 1 standard deviation.

Two types of experiments were conducted. First, the number of states n was kept constant with the sparsity s being varied, and then the reverse procedure was performed by fixing the sparsity s while varying the number of states. For each experiment, the CPU time was measured for simulations performed with DASSL and QSS3, respectively.

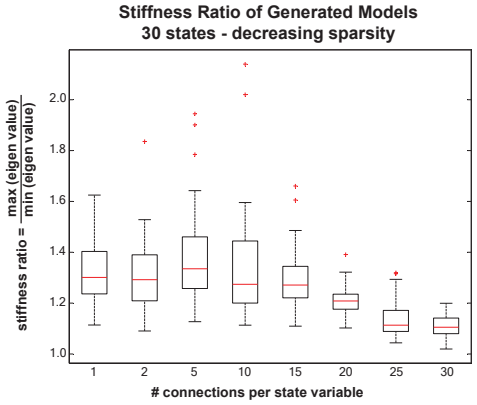

Figure 5. The stiffness ratio of the generated models in Sec. 5.3 is depicted. We observe that the resulting models exhibit approximately the same stiffness for different sparsity values.

5.3 Fixed Number of States - Varying Sparsity

In the first experiment, the number of states n in the generated models was fixed to $n = 30$ states. Then starting from a sparsity of $s = 1$, meaning that only one non-zero connection was preserved for the computation of each state derivative, the number of connections was gradually increased until we reached the full model with $s = 30$ connections. In Fig. 4(a), the obtained average CPU time is plotted against s for QSS3 (red squares) and for DASSL (black crosses). We observe that QSS3 is considerably faster than DASSL for sparse models (low values of s), whereas DASSL gets more efficient than QSS3 for approximately $s \geq 13$ connections.

This result is not overly surprising. In fact, it was expected. Whereas QSS3 benefits from sparsity as it only updates states and state derivatives asynchronously if and when there is a need for it, DASSL benefits in a fully connected model from the fact that it calls the right-hand equations only once per iteration, i.e., maybe thrice per step, whereas QSS3 updates each state derivative separately. In a non-sparse model, there is nothing that QSS3 can exploit.

However, real-world large-scale systems are always sparse. It simply doesn't happen in real life that a state derivative in a 30-th order model depends on all 30 state variable. Usually each state equation depends on two or three state variables only. Hence the case with $s = 3$ is probably the most relevant for all practical purposes.

The areas where each algorithm is superior in terms of CPU performance are more clearly visible in Fig. 4(b), where the logarithm of the ratio between the CPU times of DASSL and QSS3 is plotted against s. A positive log-ratio value means that QSS3 is more efficient than DASSL, whereas a negative value indicates the opposite.

In Fig. 5, the stiffness ratio of the generated models is plotted. The stiffness ratio is defined as the ratio of the largest to the smallest eigenvalue of matrix $\tilde{\mathbf{A}}$ and is used as an indicator of a model being stiff or not. The resulting stiffness ratio of the models used is kept smaller than 10 and does not vary much with s. Therefore, we can conclude that the differences in the measured execution times are not a result of a significant difference in the stiffness ratios of the models used.

5.4 Fixed Sparsity - Varying Number of States

Next, we studied how the simulation performance is affected when the number of connections per state variable is kept constant, whereas the number of states is modified. In Fig. 6, the sparsity s is set to 2 inputs per state, and the number of states is increased from 10 to 150. We observe that up to $n = 130$ states, QSS3 is more efficient in terms of needed CPU time. On the other hand, if we increase the sparsity s to 5 connections per state variable, the range where QSS3 is superior to DASSL is reduced to about $n = 70$ states, as depicted in Fig. 7.

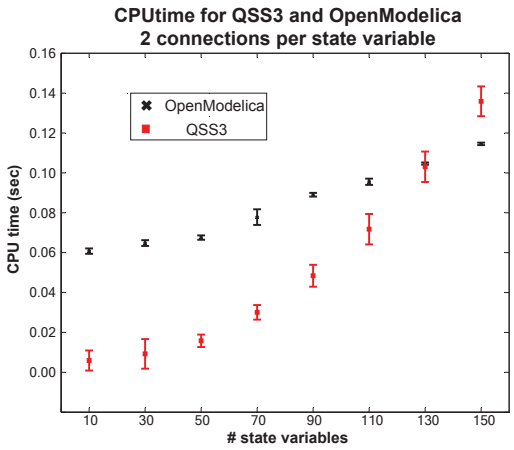

Figure 6. Sparsity s is set to 2 connections per state, and the number of states is increased from 10 to 150. We observe that up to 130 states, QSS3 is more efficient in terms of required CPU time. However, the computational cost of QSS3 increases with the model size much faster for QSS3 than for DASSL due to inefficiency in the current PowerDEVS implementation.

In both cases, the CPU time needed by QSS3 is increasing much faster than the CPU time that DASSL needs as the complexity of the model increases. This means bad news for PowerDEVS. PowerDEVS evidently does not handle large systems very well.

On the other hand, it can been shown that the computational load of the QSS3 algorithm grows only linearly with the number of states. Thus, the measurement results are in contradiction with theory.

The reason is that we measure not only the time needed by the QSS3 algorithm for integration, but also the time needed by PowerDEVS for administrating the simulation.

PowerDEVS was not designed to simulate large DEVS models. After all, the PowerDEVS users were expected to construct their models manually, which clearly limits the size of the models that users may want to simulate in PowerDEVS. For this reason, the designers of PowerDEVS did not spend much thought on an efficient implementation of the underlying simulation engine. In particular, block are implemented as a linearly linked list, and PowerDEVS traverses that list in a linear fashion when looking for the next model to be executed. This kills the overall efficiency of the implementation, and what we measured in Fig. 6 for larger numbers of states is primarily not simulation time, but rather the quadratic growth pattern of a linear search algorithm. Re-implementing the blocks as an equilibrated binary tree, a trivial programming exercise, will reduce the growth pattern of the search time to $n \cdot \log n$ and will make PowerDEVS perform considerably better for larger models.

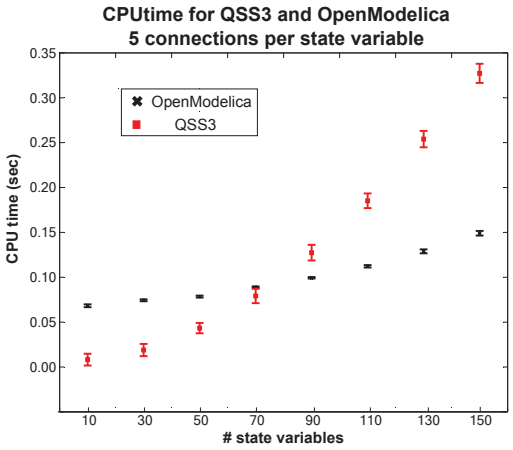

Figure 7. Sparsity s is set to 5 connections per state, and the number of states is increased from 10 to 150. Now, QSS3 is more efficient for up to 70 states.

6. Discussion

6.1 Conclusions

In this article, an interface between the OpenModelica environment and PowerDEVS is presented and analyzed. The OPMD interface implemented until now does not handle discontinuities yet, but it represents the first effort to automatically bridge the gap from the powerful Modelica modeling language standard to the also very powerful state-quantization based (QSS) simulation methods. The currently available interface allows a Modelica user to simulate arbitrarily complex non-stiff Modelica models without discontinuities using the PowerDEVS simulation software. Future extensions of the interface shall handle stiff and discontinuous models as well.

Preliminary results exhibit the superiority of QSS3 over the standard DASSL solver when simulating sparse models. More rigorous experiments will need to be performed in order to reach concrete conclusions about the performance of each algorithm. An inefficiency in the current implementation of PowerDEVS when simulating large systems was observed and will be addressed in the next distribution of PowerDEVS.

6.2 Future Work

We have shown that the implemented OMPD interface successfully allows a user to simulate an arbitrary Modelica model without discontinuities using PowerDEVS and the QSS methods. However there are open problems that need to be addressed in the future.

The OMPD interface should be extended to cover models with discontinuities. QSS methods are intrinsically well suited for simulating discontinuous models. Therefore, it is of great importance to add this functionality to the OMPD interface. Most importantly, we shall then be able to perform large-scale comparisons between DASSL and QSS algorithms for a variety of real-world models that are inherently discontinuous.

On the other hand, as discussed in Section 5.4, PowerDEVS needs to be implemented more efficiently in order to take advantage of all theoretical properties of the QSS methods. In particular, the current simulation engine is quite inefficient in the way it searches through the blocks to find the one that produces the next event in the simulation. This issue can be addressed by employing a more efficient search strategy, e.g. by organizing the atomic models in an equilibrated binary tree structure. These modifications are already being implemented, and we are looking forward to incorporating these modifications in the next PowerDEVS distribution.

7. Acknowledgements

We would like to acknowledge the help and support from the PELAB group at Linköping University and in particular Per Östlund, Adrian Pop, Martin Sjölund, and Prof. Peter Fritzson.

References

[1] Tamara Beltrame and François E. Cellier. Quantised state system simulation in dymola/modelica using the devs formalism. In *Modelica*, 2006.

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

[3] 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.

[4] 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.

[5] Peter Fritzson and Vadim Engelson. Modelica - a unified object-oriented language for system modelling and simulation. In *ECOOP*, pages 67–90, 1998.

[6] Ernesto Kofman. A second-order approximation for devs simulation of continuous systems. *Simulation*, 78(2):76–89, 2002.

[7] Ernesto Kofman. Quantization-based simulation of differential algebraic equation systems. In *Simulation, Transactions of the Society for Modeling and Simulation International*, volume 79, pages 363–376, 2003.

[8] Ernesto Kofman. Discrete event simulation of hybrid systems. *SIAM JOURNAL ON SCIENTIFIC COMPUTING*, 25:1771–1797, 2004.

[9] Ernesto Kofman. A third order discrete event simulation method for continuous system simulation. *Latin America Applied Research*, 36(2):101–108, 2006.

[10] 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.

[11] Ernesto Kofman, Marcelo Lapadula, and Esteban Pagliero. Powerdevs: A devs-based environment for hybrid system modeling and simulation. Technical Report LSD0306, Universidad Nacional de Rosario, Argentina, 2003.

[12] 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.

[13] 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.