A Stand–Alone Quantized State System Solver for Continuous System Simulation

Joaquín Fernández and Ernesto Kofman*

CIFASIS–CONICET. FCEIA - UNR.
27 de febrero 210 bis - (S2000EZP) Rosario, Argentina
Phone: +54 (341) 4237248 Ext. 336
fernandez@cifasis-conicet.gov.ar kofman@fceia.unr.edu.ar
∗ Corresponding author

Abstract

This article introduces a stand–alone implementation of the Quantized State System (QSS) integration methods for continuous and hybrid system simulation. QSS methods replace the time discretization of classic numerical integration by the quantization of the state variables. These algorithms lead to discrete event approximations of the original continuous systems and show some advantages over classic numerical integration schemes.

For simplicity reasons, most implementations of QSS methods were confined to discrete event simulation engines. The problem is that they were not fully efficient as they wasted much of the computational load in the discrete event simulation mechanism. The Stand–Alone QSS solver presented here overcomes this problem, improving in more than one order of magnitude the computation times of the previous discrete event implementations.

Besides describing the solver structure and functionality, the article analyzes four different models and compares the performance of the new solver with that of the discrete event implementation, and with that of different classic solvers.

Keywords: ODE Solvers, Discontinuity Handling, Quantized State Systems Methods

1 Introduction

Solving ordinary differential equations, requires the use of numerical integration methods. Classic integration algorithms [14, 15, 7] are based on the discretization of the independent variable (which usually represents time).

The QSS (Quantized State System) numerical integration methods [20, 7], replace the time discretization of classic integration algorithms by the quantization of the state variables. That way, these methods lead to discrete event approximations of the originally continuous systems and have some advantages over their classic counterparts:

- They satisfy strong stability and error bound theoretical properties [20, 16].
- They are very efficient to simulate continuous systems with frequent discontinuities [17].
- Due to their intrinsic capacity to exploit sparsity, they are very efficient in the simulation of large scale discontinuous models [13].
- They can integrate some stiff systems in a very efficient way, without performing iterations or matrix inversion [22, 21].

For these reasons, there are several applications where QSS methods simulate much faster than the most efficient discrete time algorithms, including power electronic circuits [17, 22, 21], biological models [13, 1], Heating, Ventilation, and Air Conditioning (HVAC) systems [23, 27], among other systems that combine different features under which QSS methods are efficient.

The easiest way of implementing the QSS algorithms is through the use of a DEVS (Discrete EVent System Specification) [28] simulation engine. For this reason, most implementations of the QSS methods are limited to DEVS simulation tools.

These implementations, although simple, are inefficient, as they waste much of the computational effort in synchronization and event transmission mechanisms of the DEVS engine itself. Additionally, the models must be defined as block diagrams, which can be inconvenient.

These drawbacks motivate the development of a Stand–Alone QSS solver, following the idea of classic numerical integration solvers such as DASSL [24, 7].

The Stand–Alone QSS solver was implemented as a set of modules coded in the C programming language. It implements the whole family of QSS methods and the models can contain time and state discontinuities.

A difficulty imposed by the QSS methods is that it makes use of structural information of the model. Each step in a QSS method involves a change in a single state variable and in the state derivatives that depend on it. Thus, the model must provide not only the expression to compute the state derivatives (as in classic ODE solvers) but also an incidence matrix so the solver knows which state derivatives are changed after each step.

Since it would be very uncomfortable for a user to provide this structure information, the solver has also a Modeling Front–End that automatically obtains the incidence matrices from a standard model definition. This front–end allows the user to describe

the models using a sub–set of the standard Modelica language [11], and automatically generates the C code of the model including the structure.

Additionally, a simple Graphic User Interface that integrates the solver with the Modeling Front–End and some plot and debug tools was developed.

In this article, we describe the Stand–Alone QSS solver with the mentioned additional tools. We also study and compare the performance of the new tool with that of a DEVS implementation of QSS methods and with DASSL and Runge–Kutta solvers.

The article is organized as follows: Section 2 presents the family of QSS methods, their implementations, and a brief introduction to the Modelica language. Then, Section 3 describes the structure, the components and the functionality of the QSS Solver. Similarly, Section 4 describes the modeling front–end. Section 5 analyzes the performance of the solver on three benchmark problems, comparing the results with DEVS implementations of QSS methods and with Runge–Kutta and DASSL solvers.

2 Background

In this section we introduce the Quantized State System numerical integration algorithms and their previous implementations. We also provide a brief description of the Modelica language.

2.1 Quantization State System Methods

QSS methods replace the time discretization of classic numerical integration algorithms by the quantization of the state variables.

Given the ODE

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

the first order Quantized State System method (QSS1) [20] approximates it by

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

Here, $\mathbf{q}$ is the *quantized state vector*. Its entries are component-wise related with those of the state vector $\mathbf{x}$ by the following *hysteretic quantization function*:

$$q_j(t) = \begin{cases} x_j(t) \text{ if } |x_j(t) - q_j(t^-)| \geq \Delta Q_j \\ q_j(t^-) \text{ otherwise} \end{cases} \tag{3}$$

where ΔQ_j is called *quantum*.

It can be easily seen that $q_j(t)$ follows a piecewise constant trajectory that only changes when the difference between $q_j(t)$ and $x_j(t)$ becomes equal to the quantum. After each change in the quantized variable, it results that $q_j(t) = x_j(t)$.

The QSS1 method has the following features:

- The quantized states $q_j(t)$ follow piecewise constant trajectories, and the state variables $x_j(t)$ follow piecewise linear trajectories.
- The state and quantized variables never differ more than the quantum ΔQ_j. This fact ensures stability and global error bound properties [20, 7].
- The quantum ΔQ_j of each state variable can be chosen to be proportional to the state magnitude, leading to an intrinsic relative error control [19].
- Each step is local to a state variable x_j (the one which reaches the quantum change), and it only provokes evaluations of the state derivatives that explicitly depend on it.
- The fact that the state variables follow piecewise linear trajectories makes very easy to detect discontinuities. Moreover, after a discontinuity is detected, its effects are not different to those of a normal step. Thus, QSS1 is very efficient to simulate discontinuous systems [17].

However, QSS1 has some limitations as it only performs a first order approximation, and it is not suitable to simulate stiff systems.

The first limitation was solved with the introduction of higher order QSS methods like the second order accurate QSS2 [16], where the quantized state follow piecewise linear trajectories, and the third order accurate QSS3 [18], where the quantized state follow piecewise parabolic trajectories.

Regarding stiff systems, a first order Backward QSS (BQSS) method was introduced in [22]. This method, in spite of being backward, is explicit. While BQSS cannot be extended to higher order approximations, a family of Linearly Implicit QSS (LIQSS) methods of order 1 to 3 was also proposed in [21]. LIQSS methods, like BQSS, are also explicit algorithms.

LIQSS methods have the same advantages of QSS methods, and they are able to efficiently integrate many stiff systems, provided that the stiffness is due to the presence of large entries in the main diagonal of the Jacobian matrix.

All QSS and LIQSS methods share the representation of Eq.(2). They only differ in the way that q_i is computed from x_i.

2.2 Implementation of QSS Methods

Each component of the QSS1 approximation given by Eq.(2) can be thought of as the coupling of two elementary subsystems: a static one,

$$\dot{x}_j(t) = f_j(q_1, \cdots, q_n, t), \tag{4}$$

and a dynamical one

$$q_j(t) = Q_j(x_j(\cdot)) = Q_j(\int \dot{x}_j(\tau)d\tau) \tag{5}$$

where Q_j is the hysteretic quantization function defined by Eq.(3) (notice that it is not a function of the instantaneous value $x_j(t)$, but a functional of the trajectory $x_j(\cdot)$).

Taking into account that the quantized variables $q_j(t)$ follow piecewise constant trajectories, and assuming that $f_j(\cdot)$ depends on t through a piecewise constant approximation, it results that both Subsystems, Eq.(4) and Eq.(5), receive piecewise constant input trajectories and compute piecewise constant output trajectories. These piecewise constant trajectories can be represented by sequences of events in a straightforward manner.

The relation between the input and output sequences of events of these subsystems can be expressed by simple DEVS models. The DEVS representations of Eq.(4) are called *static functions* and the DEVS representations of Eq.(5) are called *quantized integrators* [7].

Then, the QSS approximation Eq.(2) can be simulated by a DEVS model consisting in the coupling of n quantized integrators, n static functions (with the eventual addition of signal sources). The resulting coupled DEVS model looks identical to the block diagram representation of the original system of Eq.(1).

Higher order QSS methods are implemented in the same way. In this case, the events represent the changes in piecewise linear or piecewise parabolic trajectories and the static functions and quantized integrators take into account not only the values but also the slopes and second derivatives of the trajectories they receive and send.

Based on these ideas, the whole family of QSS methods were implemented in PowerDEVS [3], a DEVS–based simulation platform specially designed for and adapted to simulating hybrid systems based on QSS methods. In addition, the explicit QSS methods of orders 1 to 3 were also implemented in a DEVS library of Modelica [2] and implementations of the first–order QSS1 method can also be found in CD++ [8] and VLE [25].

DEVS–based implementations of QSS methods are simple but they are not efficient. The following example illustrates this fact.

Consider the second order ODE

$$\begin{aligned} \dot{x}_1(t) &= 2 \cdot x_2 \\ \dot{x}_2(t) &= -\sin(x_1) - 3 \cdot x_2 \end{aligned}$$

and its QSS approximation

$$\begin{aligned} \dot{x}_1(t) &= 2 \cdot q_2 \\ \dot{x}_2(t) &= -\sin(q_1) - 3 \cdot q_2 \end{aligned} \tag{6}$$

This approximation can be simulated by the PowerDEVS model depicted in Figure 1.

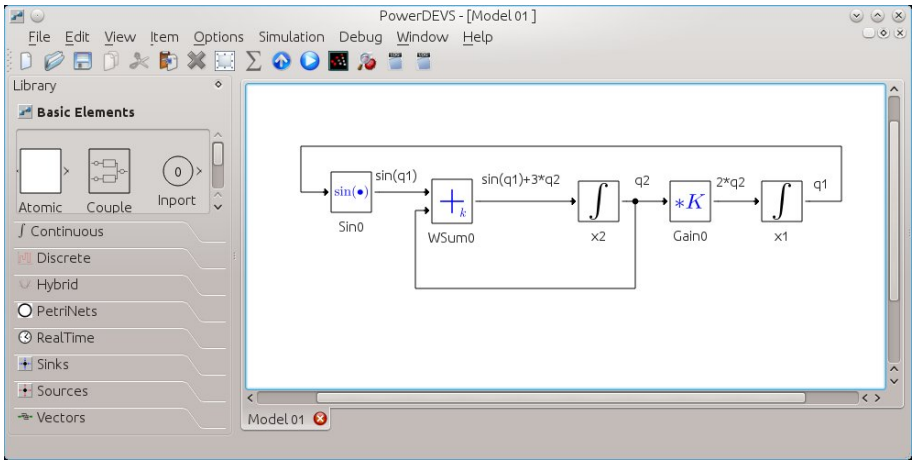

Figure 1: PowerDEVS model for Eq.(6)

Let us assume that the first step of this simulation corresponds to a change in variable q_2. This case corresponds to an internal transition on the quantized integrator `x2` in the DEVS model.

Then, the DEVS simulation engine proceeds as follows

1. The simulation engine advances the time to the next event time (i.e., the time of the change in q_2).

2. The quantized integrator `x2` computes the new value of q_2 and sends the corresponding output event to blocks `WSum0` and `Gain0` (1 function call).

3. The static functions `Wsum0` and `Gain0` execute their external transition functions where they receive q_2 and set their *time advance* $\sigma = 0$ (4 function calls).

4. The quantized integrator `x2` executes its *Internal Transition Function* and computes the time for its next output event (2 function calls).

5. The engine searches which of the 5 block performs the next event. It finds that blocks `WSum0` and `Gain0` should perform an event immediately and chooses the one with highest priority. Let us assume that it chooses `Gain0`

6. The static function `Gain0` computes $2 \cdot q_2$ and sends the corresponding output event to block `x1` (1 function call).

7. The quantized integrator `x1` executes its external transition function where it receives $2 \cdot q_2$, recomputes x_1 and the time to its next output event (i.e., the time for the next change in q_1) (2 function calls).

8. The static function `Gain0` executes its *Internal Transition Function* and sets its time advance $\sigma = \infty$ (2 function calls).

9. The engine searches which of the 5 block performs the next event. It finds that block `Wsum0` should perform an event immediately.

10. The static function `Wsum0` computes $\sin(q_1) + 3 \cdot q_2$ and sends the corresponding output event to block `x2` (1 function call).

11. The quantized integrator `x2` executes its external transition function where it receives $\sin(q_1) + 3 \cdot x_2$, recomputes x_2 and the time for its next output event (i.e., the time for the next change in q_2) (2 function calls).

12. The static function `Wsum0` executes its *Internal Transition Function* and sets its time advance $\sigma = \infty$ (2 function calls).

13. The engine searches which of the 5 block performs the next event. It should be a quantized integrator (`x1` or `x2`).

Thus, each change in a quantized variable triggers a lot of actions that are performed by the different blocks and by the DEVS simulation engine.

During the step, a minimum of 17 function calls are performed. Here, we take into account calls to external, internal, output and time advance functions of the DEVS blocks. Also, the engine performs three searches of the minimum time between 5 blocks.

Notice that these actions are independent on the DEVS simulation platform. It is just due to the DEVS simulation mechanism.

However, during a change in q_2 the only necessary actions are

1. Advance the time to the next change of q_2.

2. Calculate the new value for q_2.

3. Calculate the new derivatives $\dot{x}_1(t) = 2 \cdot q_2$ and $\dot{x}_2 = -\sin(q_1) - 3 \cdot q_2$.

4. Recompute the time of the new changes in q_1 and q_2.

5. Search which of the two variables performs the next change.

From this analysis, it is apparent that splitting the model into quantized integrators and static functions to build an equivalent DEVS model is inefficient. In order to perform a single simulation step, the DEVS simulation mechanism computes and propagates several events.

A more efficient DEVS implementation would consist of only two atomic *super–blocks*: the first one computing q_1 out of q_2 and the second one computing q_2 out of q_1. However, the behavior of those *super–blocks* would be far more complex than that of the quantized integrators and static functions. These models would depend on the functions $f_i(\mathbf{q}, t)$ and the users would need to code a different DEVS atomic block for each state, which is impractical or even impossible for large and complex models.

Also, that implementation would still perform some unnecessary steps, such as transmitting the values of q_i between blocks. It is more efficient to share the variables in a common array.

These facts motivated the development of stand alone QSS solvers like the one described in this work. Although the concept of *stand–alone* implies that the simulations are not carried out by a DEVS simulation engine, we shall see that the algorithms involved contain routines that can be thought as a sort of *ad–hoc* DEVS simulator.

A first approach to a stand–alone version of QSS1 to 3 was implemented in the Java–based simulation tool *Open Source Physics* [9], but that implementation was even less efficient than that of PowerDEVS and it required the users to provide manually the system structure information needed by QSS methods.

2.3 Modelica Language

Old modeling and simulation tools required the models to be directly coded on a programming language, typically Fortran or C. Modeling in this way was very uncomfortable and it was almost impossible to code in that way large and complex models.

In the 1970's, some specific modeling languages started developing and, at the end of the 1990's, a standard language called Modelica [11] was defined and widely adopted by the Modeling and Simulation community.

Modelica is a free high level, object-oriented language for modeling of large, complex, and heterogeneous systems.

Models in Modelica are mathematically described by differential, algebraic and discrete equations. Sub–models can be inter–connected to create more complex models and there are several software tools to compose Modelica models in a graphical way.

There are several compilers that convert Modelica models into simulation code. Among the most popular Modelica–based simulation tools we can mention Dymola [6] and OpenModelica [12].

3 The Stand–Alone QSS Solver

In this section, we describe the structure and the components of the new stand alone QSS solver.

3.1 Solver Structure

As we explained above, QSS integration methods solve the equation

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{q}, t) \tag{7}$$

where each component of $\mathbf{q}(t)$ is a piecewise polynomial approximation of the corresponding component of the state $\mathbf{x}(t)$. Different QSS methods are characterized by the way they perform this approximation.

The fact that Eq.(7) stands for all algorithms can be exploited by including a common module to solve Eq.(7) independently of the way in which $\mathbf{q}$ is computed from $\mathbf{x}$.

Taking this remark into account, the core of the solver is composed by two modules:

1. The **Integrator** that integrates Eq.(7) assuming that the piecewise polynomial quantized state trajectory $\mathbf{q}$ is known.

2. The **Quantizer** that given $\mathbf{x}(t)$, effectively calculates $\mathbf{q}(t)$ using the corresponding method. There is a different **Quantizer** for each QSS method.

In order to integrate Eq.(7), the **Integrator** must evaluate function $\mathbf{f}$, which is provided by the **Model**, which constitutes a separated module of the scheme.

Classic solvers evaluate the complete right hand side at every step. Consequently, the models only contain the code to calculate $\mathbf{f}(\mathbf{x}, t)$.

A distinctive feature of QSS methods is that different state variables are updated at different times. Thus, the QSS solver needs to know about the system structure so that after a change in a given quantized state q_i, it only evaluates those components of $\mathbf{f}$ that explicitly depend on q_i.

In consequence, the models should provide the possibility of evaluating the individual components of function $\mathbf{f}$. Moreover, the QSS solver must also know which components must be evaluated after a change in a quantized variable. This structure information is also provided by the **Model** through incidence matrices.

From an end–user point of view, it is very uncomfortable to provide a model with these features. Thus, the QSS solver was complemented with another separated module that automatically generates the structure information from a standard model definition.

Fig. 2 shows the basic interaction scheme between the four modules mentioned above.

This scheme was simplified for the purely continuous case. In presence of discontinuities, the model also contains *zero-crossing* functions and *event handlers*, providing the corresponding structure information.

For the general case, in presence of discontinuities, we consider a system of the form

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}, \mathbf{d}, t) \tag{8}$$

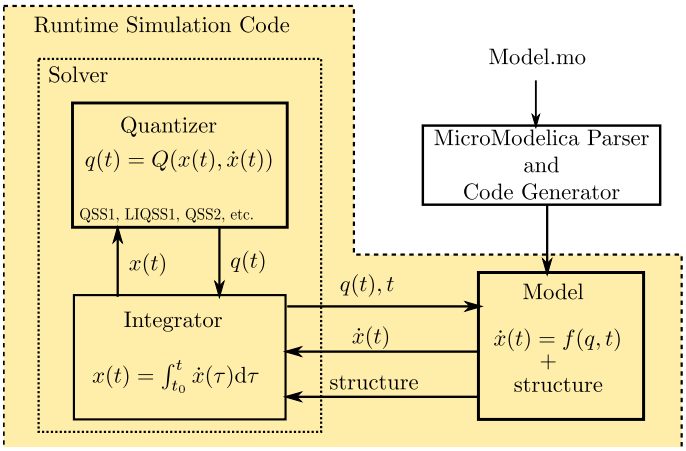

Figure 2: Stand–Alone QSS Solver – Basic Interaction Scheme

where $\mathbf{d}$ is a vector of discrete variables that can only change when a condition

$$ZC_i(\mathbf{x}, \mathbf{d}, t) = 0 \tag{9}$$

is met. The components ZC_i form a vector of zero crossing functions $\mathbf{ZC}(\mathbf{x}, \mathbf{d}, t)$. When a zero crossing condition of Eq.(9) is verified, the state and discrete variables can change according to the corresponding event handler:

$$(\mathbf{x}(t), \mathbf{d}(t)) = H_i(\mathbf{x}(t^-), \mathbf{d}(t^-)) \tag{10}$$

3.2 Integrator Module

The **Integrator** module is in charge of advancing the simulation time and computing the polynomial representation of the components $x_i(t)$ of the state vector $\mathbf{x}(t)$:

$$x_i(t) = \sum_{k=0}^{n} x_{i,k} \cdot (t - t_i^x)^k \tag{11}$$

using a known approximation of the state given by the components $q_i(t)$ of the quantized state vector $\mathbf{q}(t)$:

$$q_i(t) = \sum_{k=0}^{n-1} q_{i,k} \cdot (t - t_i^q)^k \tag{12}$$

where n is the order of the method. For that goal, it integrates Eq.(8) evaluating the components of $\mathbf{f}$ and, in presence of discontinuities, the zero crossing functions.

Each simulation step may correspond to a change in a quantized variable q_i or an event triggered by a zero-crossing function ZC_i.

When the next step corresponds to a change in a quantized variable q_i at time t, the **Integrator** proceeds as follows:

- Advance the simulation time to t.
- Ask the **Quantizer** the new coefficients $q_{i,k}$ and set $t_i^q = t$.
- Ask the **Quantizer** the next time of change in q_i.
- Ask the **Model** which state derivatives $\dot{x}_j = f_j$ depend on q_i.
- For each j so that f_j depends on q_i:
  - Obtain $x_{j,0} = x_j(t)$ from Eq.(11), and set $t_j^x = t$.
  - Ask the **Model** which quantized state variables q_l other than q_i affect the expression of f_j and update the values of $q_l(t)$ from Eq.(12).
  - Evaluate $\dot{x}_j(t)$ from the **Model** to obtain the coefficients for $x_{j,k}$ with $k = 1, \cdots, n$.
  - Ask the **Quantizer** to recompute the next time of change for q_j.
- For each j so that ZC_j depends on q_i:
  - Ask the **Model** which quantized state variables q_l other than q_i affect the expression of ZC_j and update the values of $q_l(t)$ from Eq.(12).
  - Evaluate $ZC_j(t)$ from the **Model** and estimate the next event time, at which $ZC_j(t) = 0$.
- Select the next step time as the minimum time of change in all quantized states and zero crossing functions.

Otherwise, when the next step corresponds to an event triggered by the condition $ZC_i(t) = 0$, the **Integrator** proceeds as follows:

- Advance the simulation time to t.
- Ask the **Model** which quantized state variables q_j affect the right hand side of the expressions inside the event handler H_i, and update $q_j(t)$ according to Eq.(12).
- Tell the **Model** to execute the event handler H_i.

- Ask the **Model** which state derivatives $\dot{x}_j = f_j$ depend on discrete variables d_l changed at the Handler H_i.
- For each j so that f_j depends on some d_l:
  - Obtain $x_{j,0} = x_j(t)$ from Eq.(11), and set $t_j^x = t$.
  - Ask the **Model** which quantized state variables q_m affect the expression of f_j and update the values of $q_m(t)$ from Eq.(12).
  - Evaluate $\dot{x}_j(t)$ from the **Model** to obtain the coefficients for $x_{j,k}$ with $k = 1, \cdots, n$.
  - Ask the **Quantizer** to recompute the next time of change for q_j.
- For each j so that ZC_j depends on discrete variables d_l changed at the Handler H_i:
  - Ask the **Model** which quantized state variables q_m affect the expression of ZC_j and update the values of $q_m(t)$ from Eq.(12).
  - Evaluate $ZC_j(t)$ from the **Model** and estimate the next event time, at which $ZC_j(t) = 0$.
- For each j so that x_j is changed at the event handler H_i, proceed as if a change had occurred in q_j at time t.
- Select the next step time as the minimum time of change in all quantized states and zero crossing functions.

3.3 QSS Quantizer Module

As it was described above, the **Integrator** module invokes the **Quantizer** in order to obtain the quantized state trajectories $q_i(t)$ as a function of the state trajectories $x_i(t)$. The **Quantizer** then computes the quantized state according to the QSS method specified (QSS1, QSS2, QSS3, LIQSS1, LIQSS2, etc) and the tolerance selected.

The quantized state trajectories are characterized by the polynomial coefficients ($q_{i,k}$) and the instants of change (t_i^q), as it is expressed in Eq.(12). Thus, the role of the **Quantizer** can be summarized by the following functions:

- *Update Quantized State*: It calculates the coefficients $q_{i,k}$ according to $x_{i,k}$.
- *Compute Next Time*: It computes the time of the next change in $q_i(t)$ after a new section of $q_i(t)$ starts.

- *Recompute Next Time*: It recomputes the time of the next change in $q_j(t)$ after the derivative $\dot{x}_j(t)$ changes.

These three functions depend on the QSS method in use and the selected tolerance. The tolerance is characterized by two parameters: ΔQ_{min} and ΔQ_{rel}, so it is possible to use logarithmic quantization. The mentioned parameters can be different for each state variable, and the quantum is computed as:

$$\Delta Q_i = \max(\Delta Q_{i,rel} \cdot |x_{i,0}|, \Delta Q_{i,min})$$

For instance, in the first order accurate QSS1 method the functions of the **Quantizer** calculate the quantized state as follows:

- *Update Quantized State*: It sets
$$q_{i,0} = x_{i,0}.$$
- *Compute Next Time*: It computes the next time of change as
$$t_i^{q^+} = t + \frac{\Delta Q_i}{x_{i,1}}$$
- *Recompute Next Time*: It recomputes the time of the next change as the minimum t at which
$$|x_i(t) - q_i(t)| = \Delta Q_i$$
where $x_i(t)$ and $q_i(t)$ are obtained from Eqs.(11)–(12).

In the second order accurate QSS2 method, in turn, the functions calculate as follows:

- *Update Quantized State*: It sets
$$q_{i,0} = x_{i,0}, \quad q_{i,1} = x_{i,1}$$
- *Compute Next Time*: It computes the next time of change as
$$t_i^{q^+} = t + \sqrt{\frac{\Delta Q_i}{|x_{i,2}|}}$$
- *Recompute Next Time*: It recomputes the time of the next change as the minimum $t^* > t$ at which
$$|x_i(t^*) - q_i(t^*)| = \Delta Q_i$$
where $x_i(t^*)$ and $q_i(t^*)$ are obtained from Eqs.(11)–(12). For computing t^*, the **Quantizer** finds the roots of two second order polynomials.

The **Quantizer** function for QSS3 is similar.

For the case of LIQSS1, the quantizer functions work as follows:

- *Update Quantized State*: It sets

$$q_{i,0} = x_{i,0} + \delta q_i.$$

where

$$\delta q_i = \begin{cases} \Delta Q_i & \text{if } x_{i,1} > 0 \text{ and } \tilde{f}_i(x_{i,0} + \Delta Q_i) > 0 \\ -\Delta Q_i & \text{if } x_{i,1} < 0 \text{ and } \tilde{f}_i(x_{i,0} - \Delta Q_i) < 0 \\ \tilde{q}_i - x_{i,0} & \text{otherwise} \end{cases}$$

where $\tilde{f}_i(q_i) = a_i \cdot q_i + u_i$ is a linear estimate of the state derivative $\dot{x}_i$ and $\tilde{q}_i = -u_i/a_i$ is the value at which the linear estimate is zero.

- *Compute Next Time*: It computes the next time of change as

$$t_i^{q^+} = t + \frac{\Delta Q_i}{x_{i,1}}$$

- *Recompute Next Time*: It recomputes the time of the next change as the minimum t at which

$$|x_i(t) - q_i(t) - \delta q_i| = \Delta Q_i$$

where $x_i(t)$ and $q_i(t)$ are obtained from Eqs.(11)–(12).

It also updates the parameter u_i of the linear estimate as:

$$u_i = x_{i,1} - a_i \cdot q_{i,0}$$

If the function is invoked to recompute the time of change in q_i due to a change in q_i, the function first updates the parameter a_i of the linear estimate as:

$$a_i = \frac{x_{i,1} - \text{old}(x_{i,1})}{q_{i,0} - \text{old}(q_{i,0})}$$

where $\text{old}(x_{i,1})$ is the previous value of the state derivative $x_{i,1}$ and $\text{old}(q_{i,0})$ is the previous value of the quantized state $q_{i,0}$.

LIQSS2 and LIQSS3 quantizers combine this implementation with those of QSS2 and QSS3.

3.4 QSS Model Module

The **Integrator** module solves Eq.(8), obtaining $\mathbf{q}(t)$ from the **Quantizer** and evaluating the state derivatives $\dot{x}_i = f_i(\mathbf{q}, \mathbf{d}, t)$ and the zero crossing functions $ZC_i(\mathbf{q}, \mathbf{d}, t)$ at the **Model** instance. Besides evaluating these functions, the model should also provide the already mentioned structure information.

The main functions of a **Model** must allow:

- Evaluating a **single state derivative** $\dot{x}_i = f_i(\mathbf{q}, \mathbf{d}, t)$.
- Evaluating a **zero crossing function** $ZC_i(\mathbf{q}, \mathbf{d}, t)$.
- Executing a **handler** $H_i(\mathbf{q}, \mathbf{d}, t)$.
- Evaluating 4 incidence matrices expressing the direct influence from state variables and handlers to state derivatives and zero crossing functions.

The main incidence matrix SD contains the information about the influence from state variables to state derivatives. In the implementation it is treated as a sparse matrix.

Thus, the entry $SD(j, k) = l$ tells that the k–th state derivative which is influenced by x_j is $\dot{x}_l$. This means that x_j appears explicitly on the right hand side of $\dot{x}_l = f_l(\mathbf{x}, t)$.

Similarly, there is an incidence matrix SZ that contains the information about the influence from state variables to zero–crossing functions.

Also, an incidence matrix HZ contains the information about the influence from event handlers to zero crossing functions. The entry $HZ(j, k) = l$ tells that the k–th zero crossing function influenced by the j–th event handler is ZC_l. This means that the execution of handler H_j modifies the value of ZC_l.

Similarly, the incidence matrix HD contains the information about the influence from event handlers to state derivatives.

In addition, for efficiency reasons, the **Model** module provides routines that allow:

- Evaluating **all the state derivatives** depending on one state x_j at once. That way, the **Integrator** can re–evaluate all the state derivatives that change after a step in a single function call.
- Evaluating **higher order derivatives** of state variables and zero crossing functions. These higher order derivatives are required by high order QSS methods. When they are not available, they are numerically computed which involves more function calls and computations.

As we shall see below, the incidence matrices and the routines that compute higher order derivatives as well as the functions that evaluate several derivatives in a single call are automatically obtained by the Modeling Front End based on a standard model description given by the user.

3.5 Simulation

After defining a **Model** instance, it is compiled together with the **Integrator** (which acts as the *Simulation Engine*) and the **Quantizer** modules to obtain the runtime simulation code. The three modules are written in plain C language.

3.6 Efficiency analysis

In Section 2.2 we explained why the DEVS implementations of QSS methods were inefficient, analyzing a simple simulation example given by Eq.(6). There, we saw that during a single change in $q_2(t)$, the DEVS engine took several simulation steps.

Let us analyze what the QSS solver does with the same system under the same change in variable q_2:

1. The **Integrator** advances the time to the next change in q_2.
2. The **Integrator** asks the **Quantizer** the new value of q_2 (1 function call).
3. The **Integrator** asks the **Model** the new values of the derivatives $\dot{x}_1$ and $\dot{x}_2$ (1 function call).
4. The **Integrator** asks the **Quantizer** to recompute the time of the next change in variables q_1 and q_2 (2 function calls).
5. The **Integrator** searches which of the two variables perform the next change.

During this process, the solver performs only 4 function calls and 1 search for the minimum between two values.

Compared with the 17 function calls and the 3 searches between 5 values performed by the DEVS mechanism, we can expect the implementations of the Stand–Alone QSS solver to be more efficient than those based on DEVS.

This analysis performed over a simple example can be easily extended to general systems with similar conclusions, as the model analyzed can be thought of as a part of a larger system.

4 Modeling Front-End

Compared to classic solvers, the Stand–Alone QSS solver has the disadvantage that each model must provide additional information about the system structure, so that the state derivatives $\dot{x}_i = f_i(\mathbf{q}, \mathbf{d}, t)$ and zero crossing functions $ZC_i(\mathbf{q}, \mathbf{d}, t)$ are only evaluated

when it is necessary. This structure information should be given in the form of incidence matrices.

To overcome this difficulty, a Modeling Front–End was developed that allows the user to describe models in a standard way, using a subset of the Modelica language called μ–Modelica, and then it automatically generates the corresponding plain C code with the structure matrices required by the solver.

The transformation from the original model described in μ–Modelica to the final plain C code is performed in different stages by the modules described below:

1. The **μ-Modelica Parser** module, transforms the model described in μ-Modelica to get a new structured representation.

2. The **Model Intermediate Representation (IR)** module, obtains information regarding all the state, algebraic and discrete variables defined in the model equations and events.

3. The **Model Generator** module, constructs the incidence matrices of the system and generates the **Model** instance.

In order to complete the Modeling Front–End, we developed a simple Graphic User Interface (GUI) as a separate module, to be able to create, edit models and interact with the simulation environment. The basic interaction between the modules mentioned above is depicted in Fig. 3.

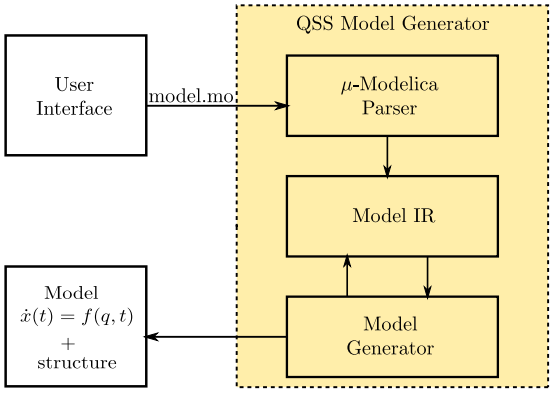

Figure 3: Modeling Front–End basic scheme.

4.1 The μ–Modelica Parser Module

The **μ-Modelica Parser** module transforms a model described using a high level modeling language into a structured representation, the AST (*Abstract Syntax Tree*), that is used by the subsequent layers of the front-end.

To achieve this goal, we defined a language called μ-Modelica consisting of a subset of the Modelica language. μ-Modelica was conceived so that it contains only the necessary Modelica keywords and structures to define an ODE based hybrid model like that of Eq. (8)-(10).

For instance, the following code corresponds to a bouncing ball model represented in μ-Modelica:

```
model bball
  Real y(start = 10),vy(start = 0), F;
  parameter Real m = 1, b = 30, g = 9.8, k = 1e6;
  discrete Real contact(start = 0);
equation
  F = k*y+b*vy;
  der(y) = vy;
  der(vy) = -g - (contact * F)/m;
algorithm
  when y < 0 then
    contact := 1;
  elsewhen y > 0 then
    contact := 0;
  end when;
end bball;
```

The QSS Solver was conceived to support the simulation of large scale models. Thus, arrays and `for` statements are allowed and efficiently handled. The following example shows this feature of the language on the model of an Advection–Reaction model.

```
model advection
  parameter Real alpha=0.5,mu=1000;
  constant Integer N = 500, T = 0.3*N;
  Real u[N];
initial algorithm
  for i in 1:T loop
    u[i]:=1;
  end for;
equation
  der(u[1])=(-u[1]+1)*N-mu*u[1]*(u[1]-alpha)*(u[1]-1);
  for i in 2:N loop
    der(u[i])=(-u[i]+u[i-1])*N-mu*u[i]*(u[i]-alpha)*(u[i]-1);
  end for;
end advection;
```

The μ-Modelica language has the following restrictions with respect to Modelica:

- The model is in flat form, i.e. no classes are allowed.

- All variables belong to the predefined type `Real` and there are only three categories of variables: **continuous states**, **discrete states** and **algebraic variables**. For instance, in the bouncing ball model, y and vy are continuous states, F is an algebraic variable and *contact* is a discrete state.

- Parameters are also of type `Real`. In the Advection–Reaction Model, *alpha* and *mu* are parameters.

- Arrays are allowed. Indexes in arrays inside `for` clauses are restricted to expressions of the form:

$$\alpha \cdot i + \beta \tag{13}$$

  where α and β are integer expressions and i is the iteration index.

- The `equation` section is composed of:

  - Definitions of **state derivatives**: $\texttt{der}(x) = f(\mathbf{x}(t), \mathbf{d}, \mathbf{a}(t), t)$; in explicit ODE form.
  - Definitions of **algebraic variables**:

$$(a_1, \cdots, a_n) = \mathbf{g}(\mathbf{x}(t), \mathbf{d}, \mathbf{a}(t), t); \tag{14}$$

    with the restriction that each algebraic variable can only depend on states and on previously defined algebraic variables.

- Discontinuities are expressed only by `when` and `elsewhen` clauses inside the `algorithm` section. Conditions on both clauses can only be relations ($<, \leq, >, \geq$) and, inside the clauses, only assignment of discrete variables and `reinit` of continuous states are allowed.

A complete specification of the μ–Modelica language can be found in [10].

Given a model defined in this language, the **μ-Modelica** parser produces the first transformation, generating the structured representation of the AST.

4.2 The Model IR Module

The next transformation is performed by the **Model IR** module. The goal of this transformation is to extract structure information from the AST obtained before.

Additionally, in presence of events, this stage is in charge of building the zero crossing functions from the zero crossing conditions. A zero crossing condition

$$f_1(\mathbf{x}, \mathbf{d}, \mathbf{a}, t) < f_2(\mathbf{x}, \mathbf{d}, \mathbf{a}, t)$$

is transformed into the zero crossing function

$$zc(\mathbf{x}, \mathbf{d}, \mathbf{a}, t) = f_1(\mathbf{x}, \mathbf{d}, \mathbf{a}, t) - f_2(\mathbf{x}, \mathbf{d}, \mathbf{a}, t)$$

The **Model IR** module analyzes all the equations and statements of the model to obtain the lists of variable dependences involved in each expression.

Thus, given an expression of the form $e = f(\mathbf{x}(t), \mathbf{d}(t), \mathbf{a}(t), t)$, the **Model IR** module must first construct a list of:

- The state variables x_i involved in the computation of e.
- The discrete variables d_i involved in the computation of e.
- The algebraic variables a_i involved in the computation of e.

Here, we say that a variable v is involved in the computation of an expression e if:

- v appears in e, or
- v is involved in the computation of an algebraic variable which is in turn involved in the computation of expression e.

This information is used to build the incidence matrices SD (from states to state derivatives), SZ (from states to zero crossing functions), HZ (from event handlers to zero crossing functions) and HD (from event handlers to state derivatives).

For instance, when we have the equations

```
a1=f1(x2,x3);
der(x1)=f2(a1,x4);
```

a simple algorithm finds that variables `x2`, `x3` and `x4` are involved in the calculation of `der(x1)` Then, this information is used as follows to build the incidence matrix SD:

- The number of influenced derivatives of `x2` is increased as $NSD_2 = NSD_2 + 1$.
- The incidence matrix entry $SD_{2,NSD_2} = 1$ is added, saying that `x2` is involved in the computation of `der(x1)`.
- The number of influenced derivatives of `x3` is increased as $NSD_3 = NSD_3 + 1$.
- The incidence matrix entry $SD_{3,NSD_3} = 1$ is added, saying that `x3` is involved in the computation of `der(x1)`.
- Idem for variable `x4`.

The module also finds that algebraic variable `a1` is involved in the calculation of `der(x1)`. This information is used by the **Model Generator** module in the next stage.

Now, let us suppose that we have the piece of μ-Modelica code:

```
equation
 der(x1)=d1;
algorithm
 when x1>2 then
    d1=-1;
 end when;
```

Here, the **Model IR** module first constructs the zero crossing function $ZC_1 = x_1 - 2$. Then it finds that the event handler H_1 (corresponding to the zero–crossing function ZC_1) influences the calculation of `der(x1)` (through the discrete variable `d1`) and that the state variable `x1` influences the zero crossing function ZC_1.

With this information, the corresponding incidence matrices are built as follows:

- The number of influenced derivatives of H_1 is increased as $NHD_1 = NHD_1 + 1$.
- The incidence matrix entry $HD_{1,NHD_1} = 1$ is added, saying that the execution of handler H_1 modifies a variable involved in the calculation of `der(x1)`.
- The number of influenced zero crossing functions of `x1` is increased as $NSZ_1 = NSZ_1 + 1$.
- The incidence matrix entry $SZ_{1,NSZ_1} = 1$ is added, saying that `x1` is involved in the computation of the zero crossing function ZC_1 .

In order to efficiently handle large scale models, expressions inside `for` statements are treated generically without expansion. In this case, the lists of variables associated to each expression include information about the index ranges.

For instance, given piece of code

```
for i in 2:100 loop
   der(x[i])=x[i-1]+x[i];
 end for;
```

the module finds that

- variable `x[i]` influences `der(x[i])` for the range $2 \leq i \leq 100$.
- variable `x[i]` influences `der(x[i+1])` for the range $1 \leq i \leq 99$.

This information is used to build matrix SD in the following way:

- For $i \in [2, 100]$ we increase $NSD_i = NSD_i + 1$ and we set $SD_{i,NSD_i} = i$.
- For $i \in [1, 99]$ we increase $NSD_i = NSD_i + 1$ and we set $SD_{i,NSD_i} = i + 1$.

This way of treatment of `for` statements without expansions allows to generate a significantly shorter code for the construction of the incidence matrices. In large scale models, this saves a huge amount of compilation time.

4.3 The Model Generator Module

This module is in charge of generating the plain C code of a **Model** instance suitable for the Stand–Alone QSS solver. Based on the **Model IR** described above, the **Model Generator** module writes the following functions:

- Initialization code, which performs the following actions:
  - Initialization of the model variables and parameters.
  - Initialization and computation of structure matrices.
  - Initialization of the model events.
  - Obtention of the initial step time for each state variable and event defined in the model.
- Code for state derivative computations, which involves
  - Calculation of individual state derivatives $\dot{x}_i = f_i(\mathbf{x}(t), \mathbf{d}(t), \mathbf{a}(t), t)$
  - Calculation in a single call of the set of state derivatives depending on a given state variable.
  - In both cases, it may include the code for the calculation of higher order derivatives ($\ddot{x}_i$, $\dddot{x}_i$). The corresponding expressions are obtained making use of symbolic differentiation with the GNU library *libmatheval*.
- Code for evaluation of zero crossing functions zc_i (it may also generate the code to compute higher-order derivatives of zc_i).
- Code for event handler routines.

For instance, the following C code shows part of the **Model** instance generated by the **Model Generator** module for the bouncing ball example introduced above.

```
void model(int _i, double **_x, double *_d, double _t, double *_dx)
{
 switch(_i)
 {
  case 0:
   _dx[1] = _x[1][0];
   return;
  case 1:
   _algvars[0][0] = k*_x[0][0]+b*_x[1][0];
   _dx[1] = -g-(_d[0]*(_algvars[0][0]))/m;
   return;
 }
}

void model_zero_crossing(int _i, double **_x, double *_d, double _t, double *_zc)
{
 switch(_i)
 {
  case 0:
   _zc[0] = _x[0][0]-(0);
   return;
 }
}

void model_handler_pos(int _i, double **_x, double *_d, double _t)
{
 switch(_i)
 {
  case 0:
   _d[0] = 0;
   return;
 }
}

void model_handler_neg(int _i, double **_x, double *_d, double _t)
{
 switch(_i)
 {
  case 0:
   _d[0] = 1;
   return;
 }
}

void MD_initializeDataStructs(SD_init _init)
{
 ...
 //incidence matrix from states to derivatives
 _localData->I[1][0] = 0;
 _localData->I[1][1] = 1;
 _localData->I[0][0] = 1;
 //incidence matrix from states to zero--crossings
 _localData->IE[0][0] = 0;
...
 //incidence matrix from handlers to derivatives
 _localData->events[0].deps[0] = 1;
...
}
```

4.4 Graphic User Interface

The Modeling Front–End was complemented with a simple Graphic User Interface (GUI) that simplifies and unifies the usage of the different components of the solver.

The GUI has the following features:

- It has a text editor, where models in μ-Modelica can be defined and modified.
- It invokes the corresponding tools to compile and run simulations.
- It provides debug information in case of errors during the model generation.
- It invokes *GNUPlot* to plot the simulation output trajectories.
- It shows statistics about simulations (number of steps, simulation time, etc.).

Fig. 4 shows the GUI with the Advection–Reaction model presented before. The left side of the GUI allows plotting the simulation results, providing an interface with *GNUPlot*.

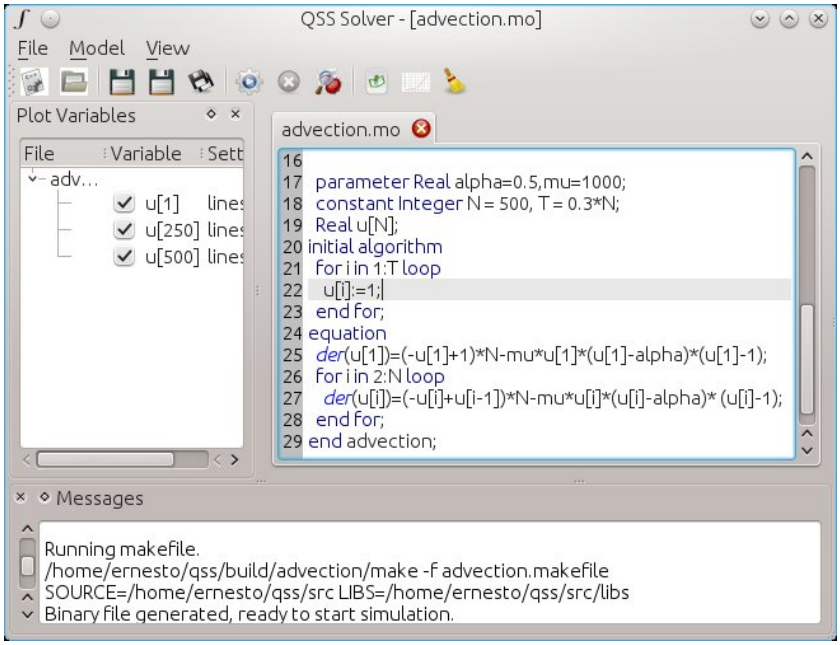

Figure 4: Graphic User Interface

5 Results

This section studies the performance of the new solver on four examples, comparing results with the same algorithms in PowerDEVS and also with DASSL and Runge Kutta solvers in OpenModelica, an efficient tool for simulation of continuous and hybrid systems.

The examples analyzed cover different features of systems where QSS methods are efficient. The first example simulates the power consumption of a large population of air conditioners, resulting in a non–stiff, large–scale model with frequent discontinuities. The second example corresponds to a large scale sparse system with certain stiffness resulting from the Method of Lines discretization of an Advection–Reaction 1D equation. The third example is a stiff model with frequent discontinuities corresponding to a power electronic converter. The last example combines all the features: it is a large scale stiff and discontinuous system representing a logic inverter chain.

In all the cases, we used QSS methods for different tolerance settings. In each model, the errors reported correspond to the mean squared error, computed against a reference solution obtained using DASSL with a tolerance of 10^{-10}.

All the simulations were run on the same computer, with an Intel i7 Processor running under Ubuntu OS. Results using Dymola with DASSL solver were also obtained, but they are not reported in most examples since they do not differ much from those of OpenModelica and they required the usage of a different OS (Windows XP). They are only analyzed in one case where OpenModelica simulations failed.

5.1 Power Consumption of an Air Conditioning population

The first example, taken from [23], is a model proposed to study the power consumption of a large population of Air Conditioners (AC).

Each AC keeps the room temperature close to a common temperature reference $\theta_{\text{ref}}(t)$, turning on and off the cooling system.

The evolution of the i–th room temperature $\theta_i(t)$ is described by a differential equation:

$$\frac{d\theta_i(t)}{dt} = -\frac{1}{C_i \cdot R_i}[\theta_i(t) - \theta_a + R_i \cdot P_i \cdot m_i(t)], \tag{15}$$

where R_i and C_i are the thermal resistance and capacity of the room, respectively. P_i is the power of the air conditioner when it is in its *on* state, and θ_a is the outside temperature.

The term $m_i(t)$ represents the on–off control of the AC, that follows the law:

$$m_i(t) = \begin{cases} 1 & \text{if } m_i(t^-) = 0 \text{ and } \theta_i(t) > \theta_{\text{ref}}(t) + 0.5 \\ 0 & \text{if } m_i(t^-) = 1 \text{ and } \theta_i(t) < \theta_{\text{ref}}(t) - 0.5 \\ m_i(t^-) & \text{otherwise} \end{cases} \tag{16}$$

We simulated this system for $N = 100$ rooms, considering a pulse in the reference temperature:

$$\theta_{\text{ref}}(t) = \begin{cases} 20 & \text{if } t < 1000 \text{ or } t > 2000 \\ 20.5 & \text{otherwise} \end{cases}$$

We used QSS2 and QSS3 methods with PowerDEVS and with the new Stand–Alone QSS solver. We also simulated it using DASSL and Runge–Kutta algorithms in OpenModelica. Figure 5 plots the total power consumption and Table 1 summarizes the simulation times and errors.

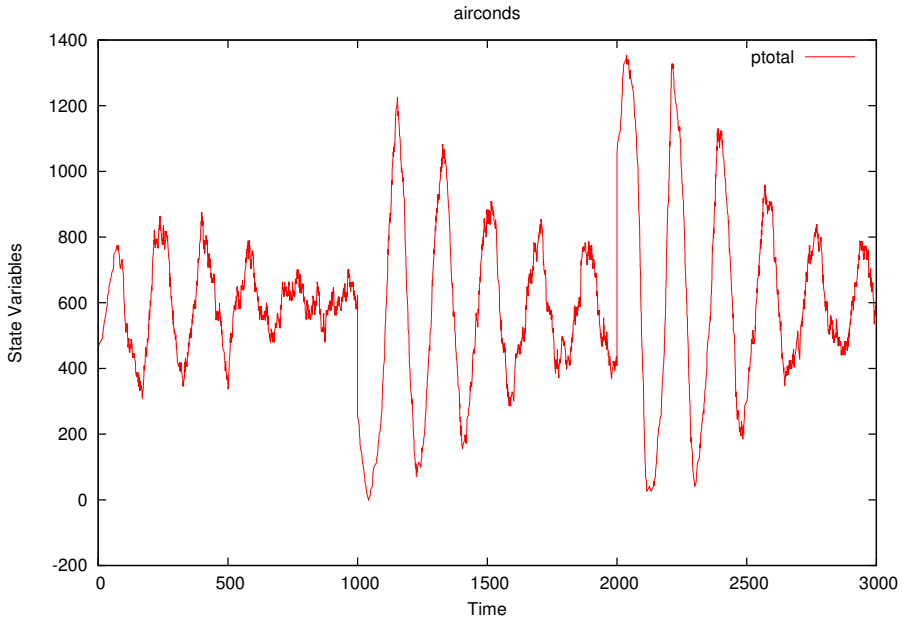

Figure 5: Total Power Consumption.

The results show that stand–alone QSS methods are from 5 to 10 times faster than the same algorithms implemented in PowerDEVS under identical tolerance settings. However, the errors obtained by the QSS solver are better than those of PowerDEVS, particularly for low tolerance settings. This is due to the fact that the solver takes into account the global tolerance settings in the event detection routines.

QSS results are also about two orders of magnitude faster than Runge–Kutta and more than three orders of magnitude faster than the stiff stable DASSL algorithm. Anyway, as this is a non stiff problem, the usage of DASSL is not actually necessary.

For low accuracy settings, the second order accurate QSS2 is faster than the third order accurate QSS3 algorithm. For more restrictive tolerances, however, QSS3 becomes more efficient.

		Tolerance	CPU time (msec)	Simulation Error
QSS Solver	**QSS2**	10^{-3}	5	4.16E-03
	QSS2	10^{-7}	123	7.01E-07
	QSS3	10^{-3}	11	2.84E-03
	QSS3	10^{-7}	28	4.29E-07
	LIQSS3	10^{-3}	12	2.48E-02
	LIQSS3	10^{-7}	30	2.12E-06
OpenModelica	**RUNGE KUTTA**	10^{-3}	1260	1.13E-02
	RUNGE KUTTA	10^{-7}	1290	1.40E-02
	DASSL	10^{-3}	25056	2.56E-02
	DASSL	10^{-7}	28280	1.42E-02
PowerDEVS	**QSS2**	10^{-3}	50	4.64E-03
	QSS2	10^{-7}	1180	1.04E-06
	QSS3	10^{-3}	60	1.26E-02
	QSS3	10^{-7}	140	3.88E-04

Table 1: Air Conditioner Population results.

Regarding Runge–Kutta and DASSL algorithms, the usage of different tolerance settings did not affect much the simulation times. Discontinuities are so frequent in this system that the step size cannot be increased even when the tolerance is low. For this same reason, the error does not change much with the tolerance, as it depends more on the accuracy of the event detection.

In this case, we also simulated with the stiff–stable LIQSS3 method, that showed an identical speed as the non–stiff QSS3 solver. This tells that LIQSS algorithms can be used as default algorithms to cover stiff and non-stiff cases without paying an extra computational cost. In classic algorithms this is not possible, as it can be seen comparing the simulation times of DASSL and Runge–Kutta.

5.2 Advection–Reaction Equation

This example is the Method of Line discretization of an Advection–Reaction model, which leads to the set of ODEs:

$$\dot{u}_i = (-u_i + u_{i-1}) \cdot N - \mu \cdot u_i \cdot (u_i - \alpha) \cdot (u_i - 1);$$

for $i = 1, \ldots, N$. We used parameters $u_0 = 1$, $\alpha = 0.5$ and $\mu = 1000$ with initial conditions $u_i(0) = 1$ for $i < 0.3 \cdot N$ and $u_i(0) = 0$ otherwise.

This is a large scale sparse system, which is also stiff due to the presence of the reaction term $\mu \cdot u_i \cdot (u_i - \alpha) \cdot (u_i - 1)$.

We simulated the model for $N = 500$, obtaining the trajectories shown in Fig. 6 and the CPU times and errors reported in Table 2. Taking into account the stiffness of the system, it was only simulated with stiff solvers (LIQSS and DASSL).

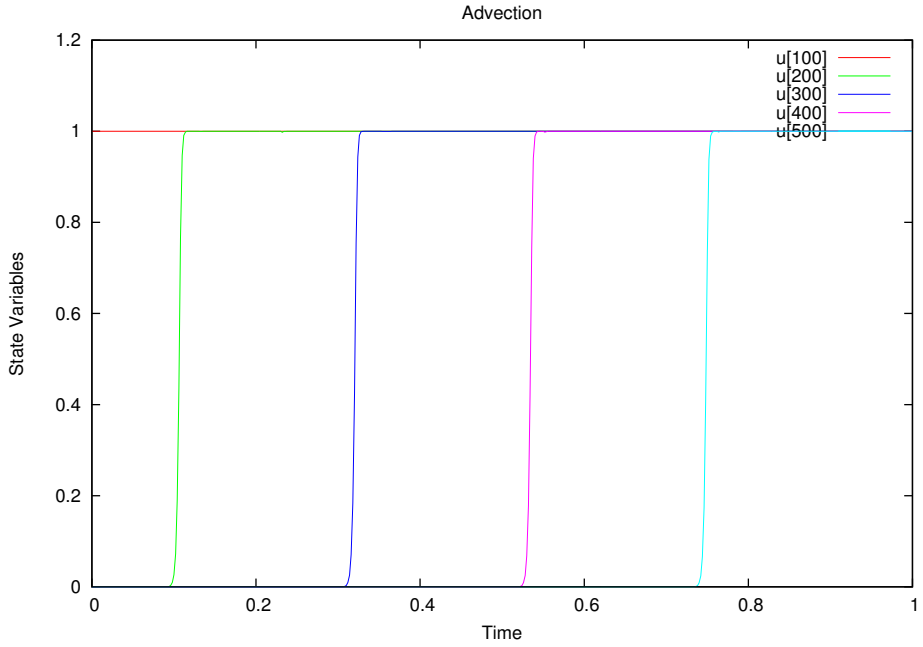

Figure 6: Advection–Reaction trajectories.

The results show now a speed up of more than one order of magnitude with respect to PowerDEVS and a speed–up of almost two orders of magnitude compared with DASSL.

Regarding the errors, they are similar in all methods for the same tolerance settings, except for PowerDEVS that with low tolerances does not experience the error reduction of the other implementations.

As it occurred with QSS2 and QSS3 in the previous example, the second order accurate LIQSS2 method is more efficient for low tolerance settings, while the third order accurate LIQSS3 method is faster for higher accuracy goals.

		Tolerance	CPU time (msec)	Simulation Error
QSS Solver	**LIQSS2**	10^{-3}	8	1.59E-03
	LIQSS2	10^{-7}	600	2.60E-11
	LIQSS3	10^{-3}	64	1.04E-03
	LIQSS3	10^{-7}	217	4.21E-12
OpenModelica	**DASSL**	10^{-3}	789	4.01E-03
	DASSL	10^{-7}	3260	3.45E-11
PowerDEVS	**LIQSS2**	10^{-3}	250	3.40E-03
	LIQSS2	10^{-7}	17000	7.30E-07
	LIQSS3	10^{-3}	950	8.32E-03
	LIQSS3	10^{-7}	1870	6.87E-07

Table 2: Advection–Diffusion–Reaction results.

5.3 Interleaved Buck Converter

We consider here an interleaved buck converter with 4 branches, shown in Fig.7, with parameters $C = 10^{-4}$ for the capacitor, $L = 10^{-4}$ for the inductance, $R = 10$ for the load resistance, $V_{cc} = 24$ for the input voltage. Also, we considered that the switches have a period of $T = 10^{-4}$, a duty cycle $DC = 0.5/4$, and we assumed that the switch and diode have a resistance $R_{\text{on}} = 10^{-5}$ in *on* state and $R_{\text{off}} = 10^{5}$ in *off* state.

This is a stiff model with frequent discontinuities. Due to stiffness, it was only simulated with LIQSS and DASSL solvers. Fig. 8 shows the state trajectories of the model (inductance currents and capacitor voltages).

The results, summarized in Table 3, show a similar relation between the QSS Solver and PowerDEVS than in the previous example (i.e., the new solver is more than 10 times faster than PowerDEVS). Comparisons with DASSL show a similar speed up.

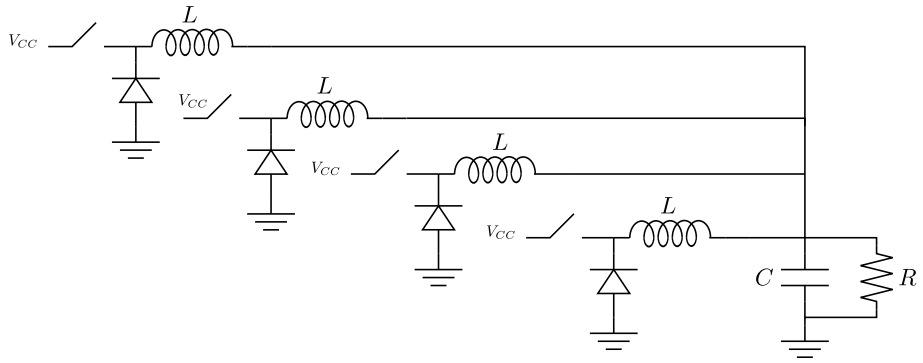

Figure 7: Interleaved Buck Converter

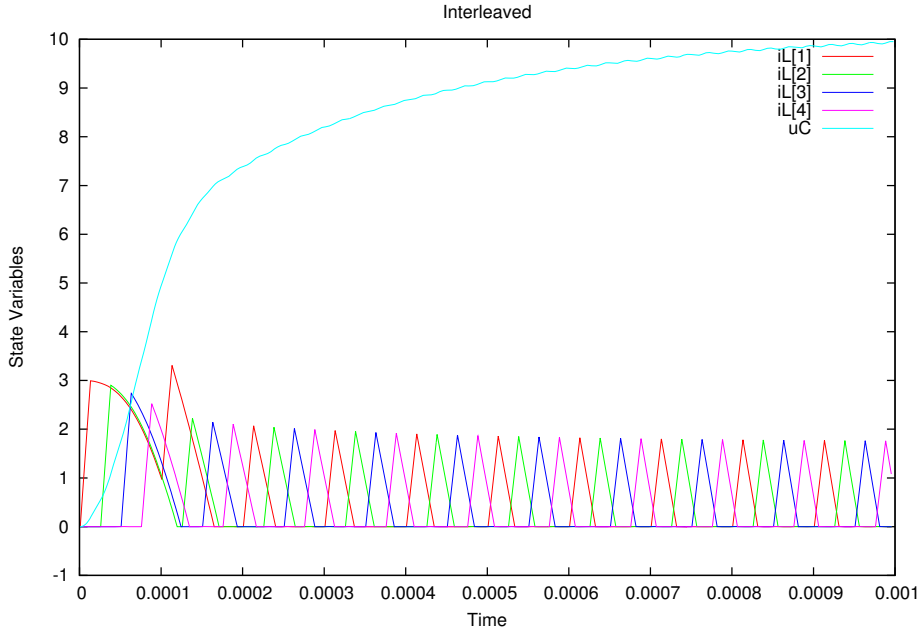

Figure 8: Interleaved Buck Converter state trajectories.

		Tolerance	CPU time (msec)	Simulation Error
QSS Solver	**LIQSS2**	10^{-3}	7	1.50E-04
	LIQSS2	10^{-7}	186	7.13E-10
	LIQSS3	10^{-3}	25	1.67E-04
	LIQSS3	10^{-7}	59	7.29E-10
OpenModelica	**DASSL**	10^{-3}	157	3.16E-04
	DASSL	10^{-7}	264	6.76E-10
PowerDEVS	**LIQSS2**	10^{-3}	300	1.55E-06
	LIQSS2	10^{-7}	10200	5.13E-07
	LIQSS3	10^{-3}	780	1.72E-04
	LIQSS3	10^{-7}	1640	5.12E-07

Table 3: Interleaved Buck Converter results.

5.4 Logical Inverter Chain

The following model, presented in [26], represents a chain of m logical inverters

$$\dot{\omega}_j(t) = U_{op} - \omega_j(t) - \Upsilon g\left(\omega_{j-1}(t), \omega_j(t)\right) \tag{17}$$

with $j = 1, \ldots, m$ where

$$g(u, v) = \left(\max(u - U_{th}, 0)\right)^2 - \left(\max\left(u - v - U_{th}, 0\right)\right)^2 \tag{18}$$

We used the set of parameters and initial conditions given in [26]: $\Upsilon = 100$ (which results in a very stiff system), $U_{th} = 1$ and $U_{op} = 5$, $\omega_j(0) = 6.247 \cdot 10^{-3}$ for odd values of j and $\omega_j = 5$ for even values of j. The input u_0 follows a trapezoid signal, that rises from 0 to 5 from time 5 to time 10 and then stays at that level, falling back to 0 from $t = 15$ to $t = 17$.

We consider a system of $m = 100$ inverters, so we have a set of 100 differential equations with 200 discontinuity conditions due to the 'max' functions in Eq.(18).

Fig. 9 plots some of the state trajectories obtained using LIQSS2 on this system.

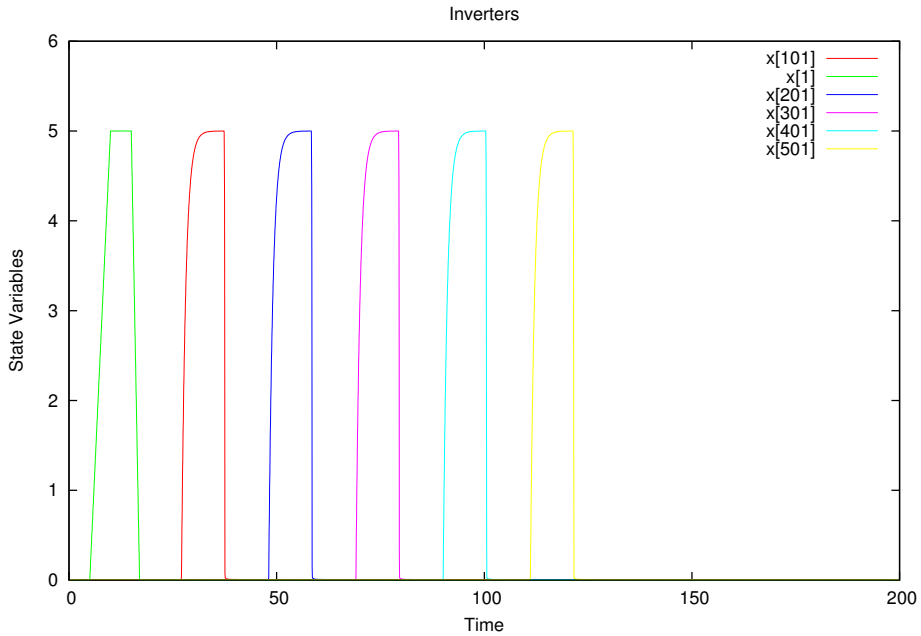

Figure 9: Logical Inverter Chain trajectories obtained with the QSS solver.

Table 4 summarizes the simulation time and errors for different solvers and tolerance settings.

		Tolerance	CPU time (msec)	Simulation Error
QSS Solver	**LIQSS2**	10^{-3}	28	3.90E-03
	LIQSS2	10^{-7}	996	6.38E-09
	LIQSS3	10^{-3}	44	1.30E-06
	LIQSS3	10^{-7}	210	4.16E-11
OpenModelica	**DASSL**	10^{-3}	4405	8.43E-03
	DASSL	10^{-7}	8734	9.01E-08
PowerDEVS	**LIQSS2**	10^{-3}	300	2.66E-05
	LIQSS2	10^{-7}	10200	3.98E-06
	LIQSS3	10^{-3}	780	6.06E-02
	LIQSS3	10^{-7}	1640	3.74E-06

Table 4: Logical Inverter Chain results.

Like in the previous examples, the Stand–Alone QSS solver performed consistently faster than PowerDEVS showing also a huge speed up with respect to DASSL.

In this last example, we fixed the tolerance at 10^{-3} and repeated the simulations for different number of inverters, using LIQSS2 for the solver and PowerDEVS, and using DASSL for OpenModelica and Dymola[1]. The CPU Time variation with the number of inverters is depicted in Fig. 10.

We can see that the simulation time of LIQSS grows about linearly with the size and the new solver implementation keeps a constant advantage of more than one order of magnitude over PowerDEVS. However, DASSL times grow almost cubically. Consequently, for 1000 inverters, LIQSS2 takes less than 200 milliseconds against 7000 seconds of DASSL.

In the simulation of 500 inverters, LIQSS2 takes about 110 milliseconds. In this case, the usage of specialized multi–rate algorithms reported a simulation time of about 6 seconds [26]. Thus, the QSS Solver is performing more than 50 times faster than those special purpose methods.

[1]Starting with 500 inverters, OpenModelica could not simulate the system, so we analyze Dymola results.

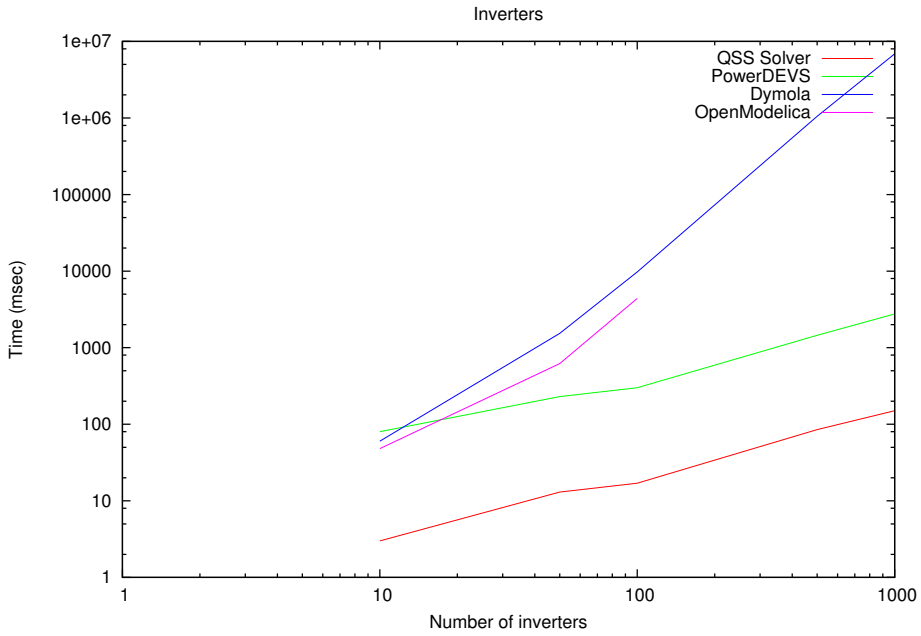

Figure 10: CPU Time vs. Number of Inverters.

6 Conclusions and Future Research

We developed an efficient stand–alone solver for QSS algorithms. In addition, we presented a Modeling Front–End that translates models written in a subset of the Modelica language into the plain C code required by the solver, providing support for discontinuity handling and large scale models.

In all the examples analyzed, the new tool is more than one order of magnitude faster than PowerDEVS using the same QSS algorithm. Moreover, the efficiency of QSS methods on these systems makes also this tool about two orders of magnitude faster than other solvers.

Regarding future work, we are considering the following issues:

- We have plans to specialize versions of our solver for some large–scale problems including **Spiking neural networks** and **MOL approximations of advection equations**.

- Another goal is to implement in the solver some recently developed **parallel simulation** techniques for QSS methods [5].

- The main limitation of the Modeling Front–End is that it is limited to a sub-set of Modelica language (μ–Modelica). However, we are developing an extension of the

OpenModelica compiler [12] which converts models from Modelica to μ–Modelica [4].

- We plan to extend the Modeling–Front End in order to produce also code for conventional solvers like DASSL, DOPRI45, etc. in order to be able to integrate in the same tool QSS and conventional methods.

The QSS Solver is an open source project, and the source code and binaries for Linux and Windows can be downloaded from *http://sourceforge.net/projects/qssengine/*. The distribution contains also the models simulated in this article.

Acknowledgment

The current work was partially supported by a CONICET grant PIP 2012–2014 Nr. 00216 and ANPCYT-FONCYT grant PICT 2012 Nr. 0077.

References

[1] Rodrigo Assar and David J Sherman. Implementing biological hybrid systems: Allowing composition and avoiding stiffness. *Applied Mathematics and Computation*, 223:167–179, 2013.

[2] T. Beltrame and F.E. Cellier. Quantised state system simulation in Dymola/Modelica using the DEVS formalism. In *Proceedings of the Fifth International Modelica Conference*, volume 1, pages 73–82, Vienna, Austria, 2006.

[3] 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.

[4] Federico Bergero, Xenofon Floros, Joaquín Fernández, Ernesto Kofman, and François E. Cellier. Simulating Modelica models with a Stand–Alone Quantized State Systems Solver. In *9th International Modelica Conference*, Munich, Germany, 2012.

[5] Federico Bergero, Ernesto Kofman, and François E. Cellier. A Novel Parallelization Technique for DEVS Simulation of Continuous and Hybrid Systems. *Simulation: Transactions of the Society for Modeling and Simulation International*, 89(6):663–683, 2013.

[6] D. Brück, H. Elmqvist, S.E. Mattsson, and H. Olsson. Dymola for multi-engineering modeling and simulation. In *Proceedings of the Second International Modelica Conference*, pages 55.1–55.8, 2002.

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

[8] M. D'Abreu and G. Wainer. M/CD++: Modeling continuous systems using Modelica and DEVS. In *Proceedings of MASCOTS 2005*, pages 229 – 236, Atlanta, GA, 2005.

[9] F. Esquembre. Easy Java Simulations: a software tool to create scientific simulations in Java. *Computer Physics Communications*, 156(1):199–204, 2004.

[10] Joaquín Fernández. *μ–Modelica Language Specification*. CIFASIS–CONICET, Rosario-Argentina, December 2013. http://www.fceia.unr.edu.ar/control/modelica/micromodelicaspec.pdf.

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

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

[13] G. Grinblat, H. Ahumada, and E. Kofman. Quantized State Simulation of Spiking Neural Networks. *Simulation: Transactions of the Society for Modeling and Simulation International*, 88(3):299–313, 2012.

[14] E. Hairer, S. Nørsett, and G. Wanner. *Solving Ordinary Dfferential Equations I. Nonstiff Problems*. Springer, Berlin, 2nd edition, 1993.

[15] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems*. Springer, Berlin, 1991.

[16] 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.

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

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

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

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

[21] 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.

[22] 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.

[23] C. Perfumo, E. Kofman, J. Braslavsky, and J.K. Ward. Load Management: Model-Based Control of Aggregate Power for Populations of Thermostatically Controlled Loads. *Energy Conversion and Management*, 55:36–48, 2012.

[24] Linda R. Petzold. Description of dassl: A differential/algebraic system solver. Technical report, Sandia National Labs., Livermore, CA (USA), 1982.

[25] G. Quesnel, R. Duboz, E. Ramat, and M. Traoré. Vle: a multimodeling and simulation environment. In *Proceedings of the 2007 Summer Computer Simulation Conference*, pages 367–374, San Diego, California, 2007.

[26] V. Savcenco and R.M.M. Mattheij. A multirate time stepping strategy for stiff ordinary differential equations. *BIT Numerical Mathematics*, 47:137–155, 2007.

[27] Victor Manuel Soto Frances, Emilio Jose Sarabia Escriva, and Jose Manuel Pinazo Ojer. Discrete event heat transfer simulation of a room. *International Journal of Thermal Sciences*, 75:105–115, 2014.

[28] B.P. Zeigler, T.G. Kim, and H. Praehofer. *Theory of Modeling and Simulation. Second edition.* Academic Press, New York, 2000.