# A stand-alone quantized state system solver for continuous system simulation

Joaquín Fernández and Ernesto Kofman

*SIMULATION* 2014 90: 782 originally published online 9 June 2014
DOI: 10.1177/0037549714536255



Simulation

*Simulation: Transactions of the Society for Modeling and Simulation International*
2014, Vol. 90(7) 782–799

DOI: 10.1177/0037549714536255
sim.sagepub.com
SAGE SCS

# A stand-alone quantized state system solver for continuous system simulation

**Joaquín Fernández and Ernesto Kofman**

**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, 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 are based on the discretization of the independent variable (which usually represents time).[1,2,3]

Quantized state system (QSS) numerical integration methods replace the time discretization of classic integration algorithms by the quantization of the state variables.[4,3] In this 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.[4,5]
- They are very efficient at simulating continuous systems with frequent discontinuities.[6]
- Due to their intrinsic capacity to exploit sparsity, they are very efficient in the simulation of large-scale discontinuous models.[7]
- They can integrate some stiff systems in a very efficient way, without performing iterations or matrix inversion.[8,9]

For these reasons, there are several applications where QSS methods simulate much faster than the most efficient discrete time algorithms, including power electronic circuits,[6,8,9] biological models,[7,10] and heating, ventilation, and air conditioning (HVAC) systems,[11,12] among other systems.

The easiest way of implementing the QSS algorithms is through the use of a DEVS (discrete event system specification) simulation engine.[13] For this reason, most implementations of 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[14,3].

The stand-alone QSS solver was implemented as a set of modules coded in the C programming language. It

CIFASIS–CONICET, FCEIA, UNR, Rosario, Argentina

**Corresponding author:**
Ernesto Kofman, CIFASIS–CONICET, FCEIA, UNR, 27 de febrero 210 bis, (S2000EZP) Rosario, Argentina.
Email: kofman@fceia.unr.edu.ar

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,[15] and automatically generates the C code of the model including the structure.

Additionally, a simple graphic user interface (GUI) that integrates the solver with the modeling front-end and some plot and debug tools were 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 QSS numerical integration algorithms and their previous implementations. We also provide a brief description of the Modelica language.

### 2.1. QSS 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 QSS method (QSS1)[4] 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 $\Delta Q_j$ 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 $\Delta Q_j$. This fact ensures stability and global error bound properties.[4,3]
- The quantum $\Delta Q_j$ of each state variable can be chosen to be proportional to the state magnitude, leading to an intrinsic relative error control.[16]
- 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.[6]

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,[5] where the quantized state follow piecewise linear trajectories, and the third-order accurate QSS3,[17] where the quantized state follow piecewise parabolic trajectories.

Regarding stiff systems, a first-order backward QSS (BQSS) method was introduced by Migoni et al.[8] 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 orders 1 to 3 was also later proposed by Migoni et al.[9] 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 equation (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 equation (2) can be thought of as the coupling of two elementary subsystems: a static one,

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

and a dynamical one

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

where $Q_j$ is the hysteretic quantization function defined by equation (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, equation (4) and equation (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 equation (4) are called *static functions* and the DEVS representations of equation (5) are called *quantized integrators*.[3]

Then, the QSS approximation equation (2) can be simulated by a DEVS model consisting in the coupling of $n$ quantized integrators with $n$ 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 equation (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,[18] 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[19] and implementations of the first-order QSS1 method can also be found in CD++ and VLE.[20,21]

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.

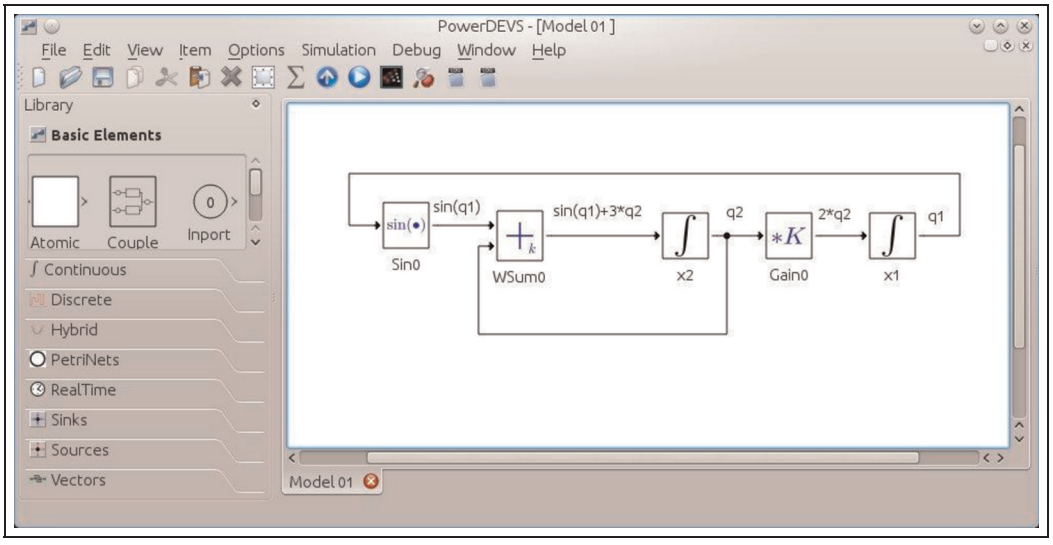


**Figure 1.** PowerDEVS model for equation (6).

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` (one function call).
3. The static functions `Wsum0` and `Gain0` execute their external transition functions where they receive $q_2$ and set their *time advance* $\sigma = 0$ (four function calls).
4. The quantized integrator `x2` executes its *Internal Transition Function* and computes the time for its next output event (two function calls).
5. The engine searches which of the five 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` (one 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 $q_1$) (two function calls).
8. The static function `Gain0` executes its *Internal Transition Function* and sets its time advance $\sigma = \infty$ (two function calls).
9. The engine searches which of the five 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` (one 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 $q_2$) (two function calls).
12. The static function `Wsum0` executes its *Internal Transition Function* and sets its time advance $\sigma = \infty$ (two function calls).
13. The engine searches which of the five 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 five 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:

(a) advance the time to the next change of $q_2$;
(b) calculate the new value for $q_2$;
(c) calculate the new derivatives $\dot{x}_1(t) = 2 \cdot q_2$ and $\dot{x}_2 = -\sin(q_1) - 3 \cdot q_2$;
(d) recompute the time of the new changes in $q_1$ and $q_2$;
(e) 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 QSS3 was implemented in the Java-based simulation tool *Open Source Physics*,[22] but that implementation was even less efficient than that of PowerDEVS and it required users to manually provide the system structure information needed by QSS methods.

## 2.3. Modelica language

Old modeling and simulation tools required the models to be directly coded with a programming language, typically

Fortran or C. Modeling in this way was very uncomfortable and it was almost impossible to code in large and complex models.

In the 1970s, some specific modeling languages started development and, at the end of the 1990s, a standard language called Modelica was defined and widely adopted by the modeling and simulation community.[15]

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 and OpenModelica.[23,24]

## 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 equation (7) stands for all algorithms can be exploited by including a common module to solve equation (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 equation (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 equation (7), the **Integrator** must evaluate function $\mathbf{f}$, which is provided by the **Model**, which constitutes a separated module of the scheme.

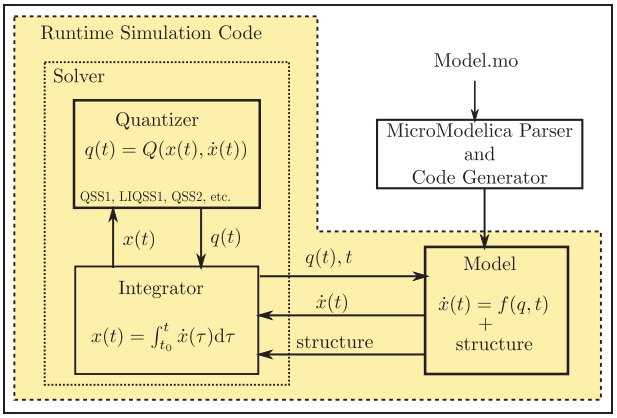


**Figure 2.** Stand-alone QSS solver—basic interaction scheme.

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.

Figure 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

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

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

equation (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. QSS 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. To accomplish that goal, it integrates equation (8), evaluating the components of **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 equation (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 equation (12);
  - evaluate $\dot{x}_j(t)$ from the **Model** to obtain the coefficients for $x_{j,k}$ with $k = 1, \ldots, n$;
  - 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 equation (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 equation (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 equation (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 equation (12);
  - evaluate $\dot{x}_j(t)$ from the **Model** to obtain the coefficients for $x_{j,k}$ with $k = 1, \ldots, n$;
  - 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 equation (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 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 equation (12). Thus, the role of the **Quantizer** can be summarized by the following functions:

- *Update Quantized State*: calculates the coefficients $q_{i,k}$ according to $x_{i,k}$;
- *Compute Next Time*: computes the time of the next change in $q_i(t)$ after a new section of $q_i(t)$ starts;
- *Recompute Next Time*: 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: $\Delta Q_{\min}$ and $\Delta Q_{\text{rel}}$, so it is possible to use logarithmic quantization. The mentioned parameters can be different for each state variable, and the quantum is computed as:

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

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

- *Update Quantized State*: sets $q_{i,0} = x_{i,0}$;
- *Compute Next Time*: computes the next time of change as
$$t_i^{q^+} = t + \frac{\Delta Q_i}{x_{i,1}}$$
- *Recompute Next Time*: 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 equations (11) and (12).

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

- *Update Quantized State*: sets $q_{i,0} = x_{i,0}$ and $q_{i,1} = x_{i,1}$;
$$q_{i,0} = x_{i,0}, \quad q_{i,1} = x_{i,1}$$
- *Compute Next Time*: computes the next time of change as
$$t_i^{q^+} = t + \sqrt{\frac{\Delta Q_i}{|x_{i,2}|}}$$
- *Recompute Next Time*: 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 equations (11) and (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*: sets $q_{i,0} = x_{i,0} + \delta q_i$, where
$$\delta q_i = \begin{cases} \Delta Q_i & \text{if } x_{i,1} > 0 \text{ and } \tilde{f}_j(x_{i,0} + \Delta Q_i) > 0 \\ -\Delta Q_i & \text{if } x_{i,1} < 0 \text{ and } \tilde{f}_i(x_{i,0} - \Delta Q_i) < 0 \\ \tilde{q}_i - x_{i,0} & \text{otherwise} \end{cases}$$
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*: computes the next time of change as
$$t_i^{q^+} = t + \frac{\Delta Q_i}{x_{i,1}}$$
- *Recompute Next Time*: 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 equations (11) and (12), and 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 equation (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 for:

- 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 four 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 for:

- Evaluating **all the state derivatives** depending on one state $x_j$ at once. In this 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 equation (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 $q_2$ (one function call).
3. The **Integrator** asks the **Model** the new values of the derivatives $\dot{x}_1$ and $\dot{x}_2$ (one function call).
4. The **Integrator** asks the **Quantizer** to recompute the time of the next change in variables $q_1$ and $q_2$ (two function calls).
5. The **Integrator** searches which of the two variables perform the next change.

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

Compared with the 17 function calls and the three searches between five 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 $\mu$-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 $\mu$-Modelica to the final plain C code is performed in different stages by the modules described below:

1. The **$\mu$-Modelica Parser** module, transforms the model described in $\mu$-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 GUI as a separate module, to be able to

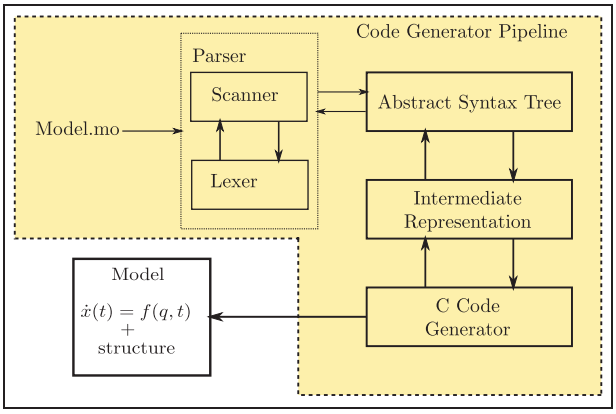


**Figure 3.** Modeling front-end basic scheme.

create and edit models and interact with the simulation environment. The basic interaction between the modules mentioned above is depicted in Figure 3.

## 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 equations (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 $\alpha$ and $\beta$ are integer expressions and $i$ is the iteration index.

- The `equation` section is composed of:
  - Definitions of **state derivatives**: $\mathtt{der}(x) = f(\mathbf{x}(t), \mathbf{d}, \mathbf{a}(t), t)$; in explicit ODE form.
  - Definitions of **algebraic variables**:

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

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 $\mu$-Modelica language can be found online.[25]

Given a model defined in this language, the **$\mu$-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 $\mu$-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. GUI

The modeling front-end was complemented with a simple 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 $\mu$-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.).

Figure 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*.

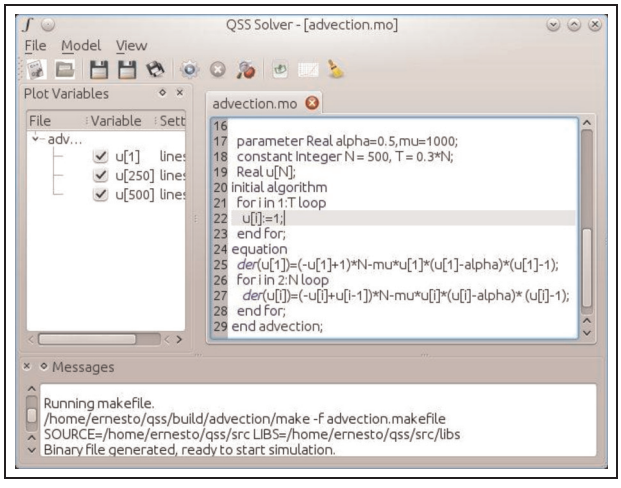


**Figure 4.** GUI.

## 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 work by Perfumo et al.,[11] is a model proposed to study the power consumption of a large population of air conditioners (ACs).

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:

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

where $R_i$ and $C_i$ are the thermal resistance and capacity of the room, respectively. $P_i$ is the power of the AC when it is in its *on* state, and $\theta_a$ is the outside temperature.

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

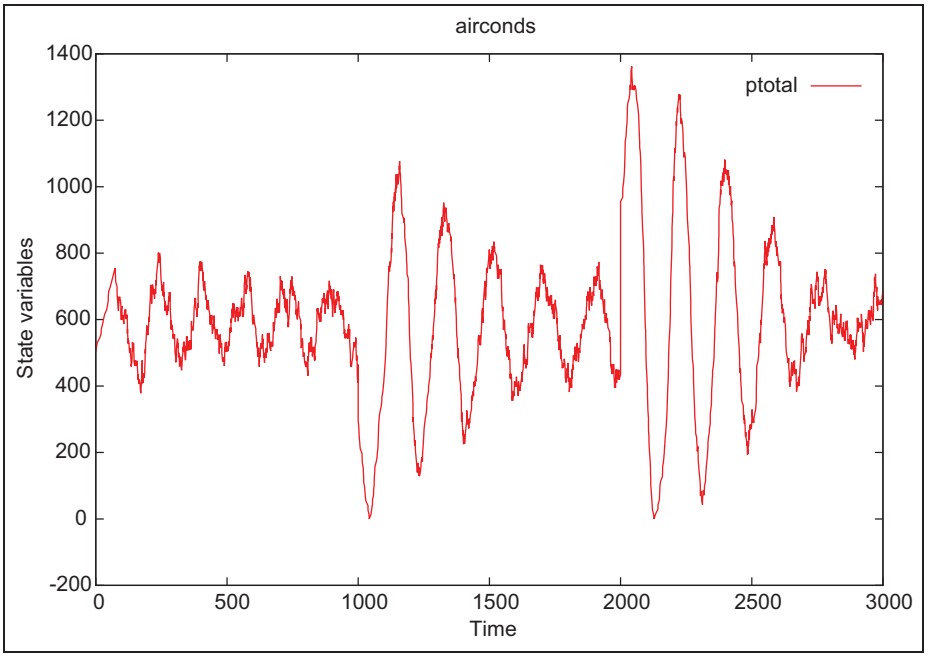


**Figure 5.** Total power consumption.

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

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.

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. However, 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.

Regarding Runge–Kutta and DASSL algorithms, the usage of different tolerance settings did not much affect 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 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.

**Table 1.** AC population results.

| | | 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 | $10^{-3}$ | 25,056 | 2.56E-02 |
| | DASSL | $10^{-7}$ | 28,280 | 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 |

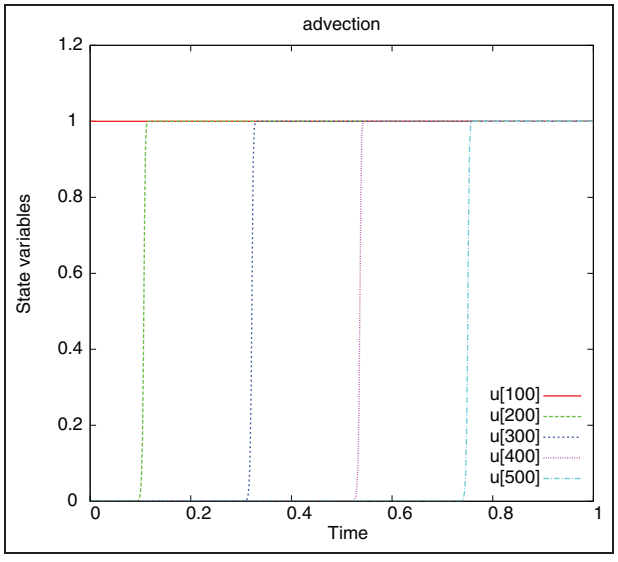


**Figure 6.** Advection–reaction trajectories.

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 Figure 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).

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 which, with low tolerances, does not experience the error reduction of the other implementations.

As 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.

**Table 2.** Advection–diffusion–reaction results.

| | | 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 | $10^{-7}$ | 17,000 | 7.30E-07 |
| | LIQSS3 | $10^{-3}$ | 950 | 8.32E-03 |
| | LIQSS3 | $10^{-7}$ | 1870 | 6.87E-07 |

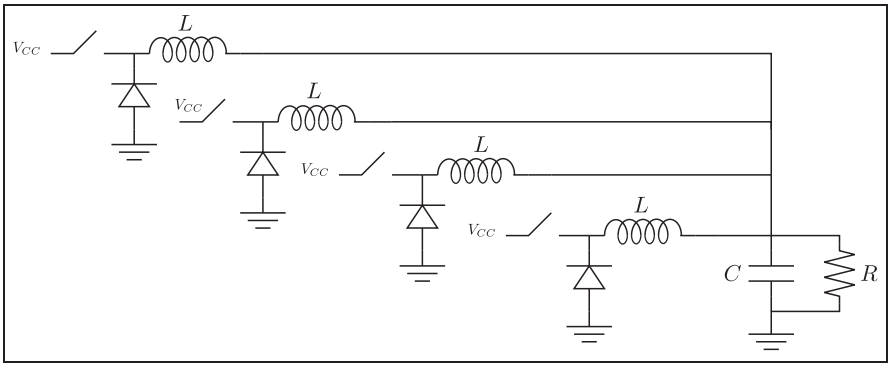


**Figure 7.** Interleaved buck converter.

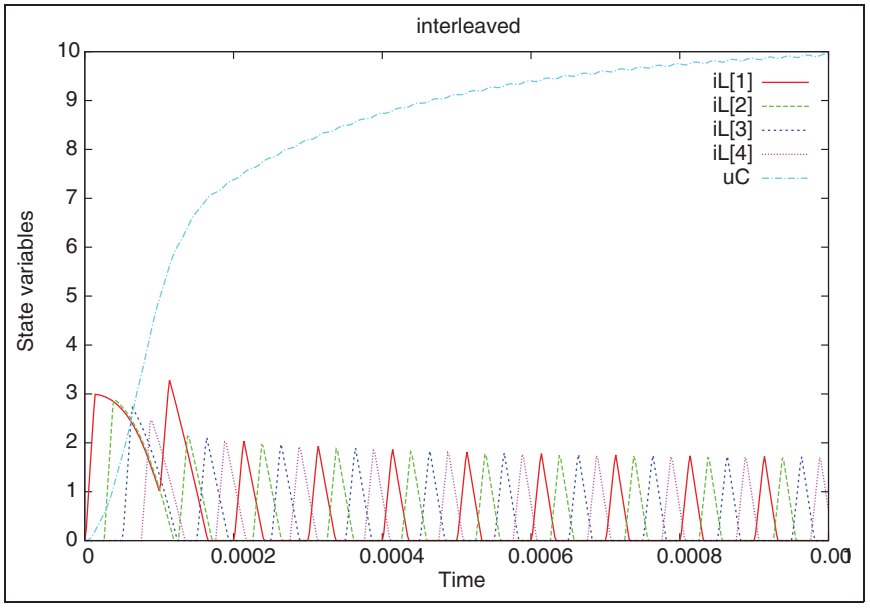


**Figure 8.** Interleaved buck converter state trajectories.

## 5.3. Interleaved buck converter

We consider here an interleaved buck converter with four branches, shown in Figure 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 consider that the switches have a period of $T = 10^{-4}$ and a duty cycle of $DC = 0.5/4$, and assume that the switch and diode have a resistance of $R_{\text{on}} = 10^{-5}$ in the *on* state and $R_{\text{off}} = 10^{5}$ in the *off* state.

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

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

## 5.4. Logical inverter chain

The following model, presented by Savcenco and Mattheij,[26] represents a chain of $m$ logical inverters

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

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

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

We used the set of parameters and initial conditions given by Savcenco and Mattheij: $\Upsilon = 100$ (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 $j$.[26] 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 $t = 17$.

**Table 3.** Interleaved buck converter results.

| | | 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 | $10^{-7}$ | 10,200 | 5.13E-07 |
| | LIQSS3 | $10^{-3}$ | 780 | 1.72E-04 |
| | LIQSS3 | $10^{-7}$ | 1640 | 5.12E-07 |

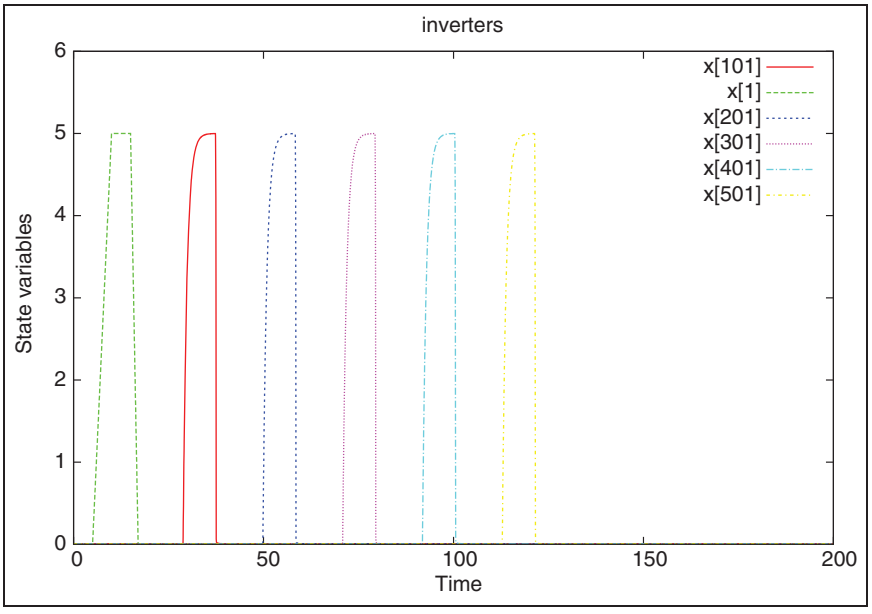


**Figure 9.** Logical inverter chain trajectories obtained with the QSS solver.

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 equation (18).

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

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

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 a different number of inverters, using LIQSS2 for the solver and PowerDEVS, and using DASSL for OpenModelica and Dymola. (Starting with 500 inverters, OpenModelica could not simulate the system, so we analyze Dymola results.) The CPU Time variation with the number of inverters is depicted in Figure 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 ms against the 7000 s of DASSL.

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

## 6. Conclusions and future research

We developed an efficient stand-alone solver for QSS algorithms. In addition, we presented a modeling front-end

**Table 4.** Logical inverter chain results.

| | | 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 | $10^{-7}$ | 10,200 | 3.98E-06 |
| | LIQSS3 | $10^{-3}$ | 780 | 6.06E-02 |
| | LIQSS3 | $10^{-7}$ | 1640 | 3.74E-06 |

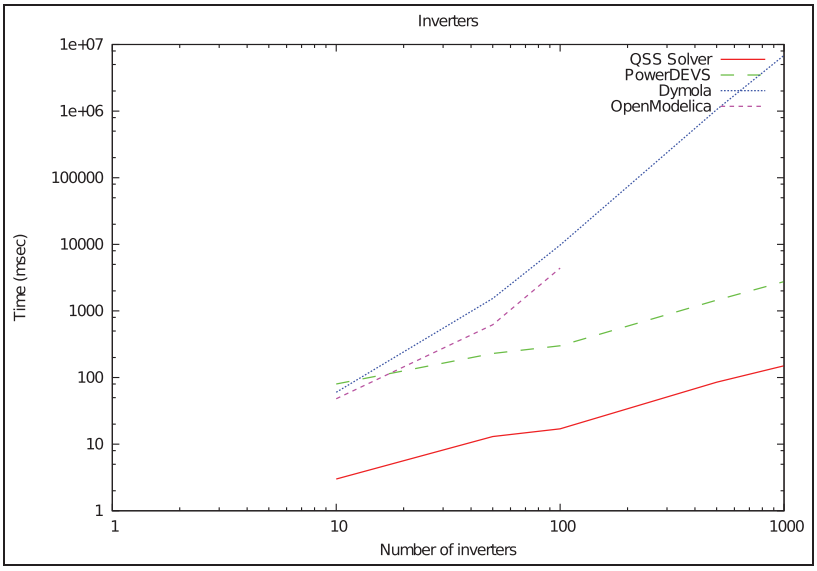


**Figure 10.** CPU time vs number of inverters.

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.[27]
- The main limitation of the modeling front-end is that it is limited to a sub-set of Modelica language ($\mu$-Modelica). However, we are developing an extension of the OpenModelica compiler[24] which converts models from Modelica to $\mu$-Modelica.[28]
- We plan to extend the modeling-front end in order to also produce 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 also contains the models simulated in this article.

## Funding

This work was supported by a CONICET grant (PIP 2012–2014 number 00216) and an ANPCYT-FONCYT grant (PICT 2012 number 0077).

## References


1. Hairer E, Nørsett S and Wanner G. *Solving Ordinary Dfferential Equations I. Nonstiff Problems*. 2nd ed. Berlin: Springer, 1993.
2. Hairer E and Wanner G. *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems*. Berlin: Springer, 1991.
3. Cellier FE and Kofman E. *Continuous System Simulation*. New York: Springer, 2006.
4. Kofman E and Junco S. Quantized state systems. A DEVS approach for continuous system simulation. *Trans SCS* 2001; 18(3): 123–132.
5. Kofman E. A second order approximation for DEVS simulation of continuous systems. *Simul—T Soc Mod Sim* 2002; 78(2): 76–89.
6. Kofman E. Discrete event simulation of hybrid systems. *SIAM J Sci Comput* 2004; 25(5): 1771–1797.
7. Grinblat G, Ahumada H and Kofman E. Quantized state simulation of spiking neural networks. *Simul—T Soc Mod Sim* 2012; 88(3): 299–313.
8. Migoni G, Kofman E and Cellier F. Quantization-based new integration methods for stiff ODEs. *Simul—T Soc Mod Sim* 2012; 88(4): 387–407.
9. Migoni G, Bortolotto M, Kofman E, et al. Linearly implicit quantization-based integration methods for stiff ordinary differential equations. *Simul Mod Prac Th* 2013; 35: 118–136.
10. Assar R and Sherman DJ. Implementing biological hybrid systems: Allowing composition and avoiding stiffness. *Appl Math Computat* 2013; 223: 167–179.
11. Perfumo C, Kofman E, Braslavsky J, et al. Load management: Model-based control of aggregate power for populations of thermostatically controlled loads. *Energy Conversion and Management* 2012; 55: 36–48.
12. Soto Frances VM, Sarabia Escriva EJ and Pinazo Ojer JM. Discrete event heat transfer simulation of a room. *Int J Therm Sci* 2014; 75: 105–115.
13. Zeigler BP, Kim TG and Praehofer H. *Theory of Modeling and Simulation*. 2nd edition. New York: Academic Press, 2000.
14. Petzold LR. Description of DASSL: A differential/algebraic system solver. *Sandia National Labs, Livermore, CA*, 1982.
15. Fritzson P. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. New York: Wiley-Interscience, 2004.
16. Kofman E. Relative error control in quantization based integration. *Lat Am Appl Res* 2009; 39(3): 231–238.
17. Kofman E. A third order discrete event simulation method for continuous system simulation. *Lat Am Appl Res* 2006; 36(2): 101–108.
18. Bergero F and Kofman E. PowerDEVS. A tool for hybrid system modeling and real time simulation. *Simul—T Soc Mod Sim* 2011; 87(1–2): 113–132.
19. Beltrame T and Cellier FE. Quantised state system simulation in Dymola/Modelica using the DEVS formalism. In: *Proceedings of the 5th international Modelica conference*, Vienna, Austria, 4–5 September 2006, Vol. 1, pp.73–82.
20. D'Abreu M and Wainer G. M/CD++ : Modeling continuous systems using Modelica and DEVS. In: *Proceedings of MASCOTS 2005*, Atlanta, GA, 27–29 September 2005, pp.229–236.
21. Quesnel G, Duboz R, Ramat E, et al. VLE: A multimodeling and simulation environment. In: *Proceedings of the 2007 summer computer simulation conference*, San Diego, CA, 15–18 July 2007, pp.367–374.
22. Esquembre F. Easy Java Simulations: A software tool to create scientific simulations in Java. *Comp Phys Comm* 2004; 156(1): 199–204.
23. Brück D, Elmqvist H, Mattsson SE, et al. Dymola for multi-engineering modeling and simulation. In: *Proceedings of the 2nd international Modelica conference*, Oberpfaffenhofen, Germany, 18–19 March 2002, pp.55.1–55.8.
24. Fritzson P, Aronsson P, Lundvall H, et al. The OpenModelica modeling, simulation, and development environment. In: *Proceedings of the 46th conference on simulation and modeling (SIMS'05)*, Trondheim, Norway, 13–14 October 2005, pp.83–90.
25. Fernández J. $\mu$–Modelica language specification. Rosario, Argentina, 2013. Available at: http://www.fceia.unr.edu.ar/control/modelica/micromodelicaspec.pdf.
26. Savcenco V and Mattheij RMM. A multirate time stepping strategy for stiff ordinary differential equations. *BIT Num Math* 2007; 47: 137–155.
27. Bergero F, Kofman E and Cellier FE. A novel parallelization technique for DEVS simulation of continuous and hybrid systems. *Simul—T Soc Mod Sim* 2013; 89(6): 663–683.
28. Bergero F, Floros X, Fernández J, et al. Simulating Modelica models with a stand-alone quantized state systems solver. In: *Proceedings of the 9th international Modelica conference*, Munich, Germany, 3–5 September 2012, pp.237–246.


## Author biographies

**Ernesto Kofman** received his BS degree in electronic engineering in 1999 and his PhD degree in Automatic Control in 2003, both from the Universidad Nacional de Rosario, Argentina, where he holds an Adjunct Professor position at the Department of Control. He also holds a research position at the French Argentine International Center for Information and Systems Sciences (CIFASIS) from the National Research Council of Argentina (CONICET). His research interests include automatic control theory and numerical simulation of hybrid systems.

**Joaquín Fernández** received his BS degree in Computer Science in 2012 from the Universidad Nacional de Rosario, Argentina. He is currently a PhD student at the French Argentine International Center for Information and Systems Sciences (CIFASIS). His research interests include hybrid system simulation, real-time and parallel simulation, simulation tools, and modeling languages.