XV Reunión de Trabajo en Procesamiento de la Información y Control, 16 al 20 de septiembre de 2013

A Stand–Alone Quantized State System Solver. Part I

Joaquín Fernández[†] and Ernesto Kofman[†‡]

† *CIFASIS-CONICET*
‡ *Facultad de Cs. Exactas, Ingeniería y Agrim., UNR, Argentina*
fernandez@cifasis-conicet.gov.ar - kofman@fceia.unr.edu.ar

***Abstract*— This article introduces a stand–alone implementation of the Quantized State System (QSS) methods for continuous and hybrid system simulation. QSS methods replace the time discretization of classic numerical integration by the quantization of the state variables, leading to discrete event approximations that show some advantages over classic numerical integration schemes.**

For simplicity reasons, previous implementations of QSS methods were confined to discrete event simulation engines, which deteriorated the efficience of the algorithms wasting most of the computational load in internal synchronization mechanisms. The QSS solver presented here overcomes this problem, improving in more than one order of magnitude the computation times of previous implementations.

Thr first part of this work describes the solver structure and functionality and compares the performance of the new solver with a discrete event implementation, and also with different classic solvers in three benchmark problems.

***Keywords*— ODE Solvers, Discontinuity Handling, Quantized State Systems Methods**

1 Introduction

Solving ordinary differential equations (ODEs), requires the use of numerical integration methods. Classic integration algorithms [1, 2, 3] are based on the discretization of the independent variable (which usually represents time).

The QSS methods [4, 3] replace the time discretization 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, resulting particularly efficent in the simulation of systems with frequent discontinuities [5], large scale discontinuous models [6], and in some stiff systems, where they do not need to perform iterations or matrix inversions [7, 8].

The easiest way of implementing QSS algorithms is through the use of a DEVS (Discrete EVent System Specification) [9] simulation engine. For this reason, most implementations of the QSS methods are limited to DEVS simulation tools.

However, these implementations are inefficient, as they waste much of the computational effort in synchronization and event transmission mechanisms of the DEVS engine itself.

This drawback motivated the development of a stand–alone QSS solver, following the idea of classic numerical integration solvers such as DASSL [10, 3] or Matlab solvers (ode15s, ode23s, ode45, etc.) [11].

The stand–alone QSS solver was implemented as a set of modules coded in plain C 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 [12], 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 solver and we study and compare the performance of the new tool with that of DEVS implementations of QSS methods and with some implementations of classic solvers (DASSL). The Modeling front–end is described in the companion article.

The work is organized as follows: Section 2 presents the family of QSS methods and their implementations. Then, Section 3 describes the structure, the components and the functionality of the QSS Solver. The solver performance is analyzed in Section 4, comparing simulation results with previous implementations of QSS methods and with other classic solvers.

2 Background

2.1 Quantization State System Methods

Quantized State System (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) [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 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 [4].
- 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 [13].
- 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 [5].

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 higer order QSS methods like QSS2 [14], where the quantized state follow piecewise linear trajectories, and QSS3 [15] where the quantized state follow piecewise parabolic trajectories.

Regarding stiff systems, a first order backward QSS method (BQSS) was introduced in [7]. This method, in spite of being backward, is explicit. While BQSS cannot be extended to higher order approximations, a family of linearly implicit QSS methods (LIQSS) of order 1 to 3 was also proposed in [8]. LIQSS methods, like BQSS, are also explicit algorithms.

LIQSS methods have the same advantages of QSS methods, and they are able to efficiently handle 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

It was shown that the behavior of the QSS approximation of Eq.(2) can be described by a discrete event system in terms of the DEVS formalism [9]. Thus, the easiest way of implementing these algorithms is through their equivalents on a DEVS simulation engine.

The whole family of QSS methods were implemented in PowerDEVS [16], 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 [17] and implementations of the first–order QSS methods can also be found in CD++ [18] and VLE [19].

DEVS–based implementations of QSS methods are simple but they are not efficient. The problem is that the DEVS simulation engines waste a large amount of the computational load attending the DEVS simulation mechanism. This fact motivated the development of stand–alone QSS solvers like the one described in this work.

A first approach to a stand–alone version of QSS1 to 3 was implemented in the Java–based simulation tool *Open Source Physics* [20], but that implementation was not more efficient than that of PowerDEVS and it required the user to provide the system structure information needed by QSS methods.

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

QSS integration methods solve the equation (2) 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.(2) stands for all algorithms can be exploited by including a common module to solve it independently on 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 (2) assuming that the piecewise polynomial quantized state trayectory $\mathbf{q}$ is known.
2. The **Quantizer** that given $\mathbf{x}(t)$, effectively calulates $\mathbf{q}(t)$ using the corresponding method. There is a different **Quantizer** for each QSS method.

In order to integrate Eq.(2), the **Integrator** must evaluate function $\mathbf{f}(\cdot,\cdot)$, 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 $\mathbf{f}(\mathbf{x},t)$.

In QSS, 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 explicitely 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 a separated module that automatically generates the structure information from a standard model definition.

Figure 1 shows the basic interaction scheme between the four modules mentioned above.

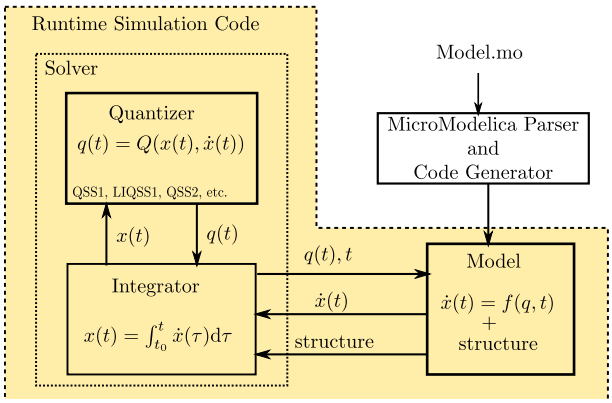

Figure 1: Stand Alone QSS Solver – Basic Interaction Scheme

This scheme was symplified 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

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}, \mathbf{d}, t) \tag{4}$$

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

$$ZC_i(\mathbf{x}, \mathbf{d}, t) = 0 \tag{5}$$

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.(5) is verified, the state and discrete variables can change according to the corresponding event handler:

$$(\mathbf{x}(t), \mathbf{d}(t)) = H_i(\mathbf{x}(t^-), \mathbf{d}(t^-)). \tag{6}$$

3.2 QSS Integrator Module

The QSS 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 $\mathbf{x}(t)$:

$$x_i(t) = \sum_{k=0}^{n} x_{i,k} \cdot (t - t_i^x)^k \tag{7}$$

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

$$q_i(t) = \sum_{k=0}^{n-1} q_{i,k} \cdot (t - t_i^q)^k \tag{8}$$

where n is the order of the method. For that goal, it integrates Eq.(4) 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 **QSS 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.(7), 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.(8).
  - 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.(8).
  - 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 **QSS Integrator** proceeds in a similar manner.

3.3 QSS Quantizer Module

The QSS 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.(8). 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 [13].

3.4 QSS Model Module

The **Integrator** module solves Eq.(4), obtaining $\mathbf{q}(t)$ from the **Quantizer** and evaluating $\mathbf{f}(\cdot,\cdot)$ at the **Model**. Besides evaluating this function, the model should also provided the already described structure information.

The main functions of the **Model** module can be summarized as follows:

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

In addition to this definitions, for effiency reasons, it provides functions for:

- Evaluate **all the state derivatives** depending on one state x_j in a single call.
- Evaluate state variables and zero crossings **higher order derivates**.

In the second part of this article we describe a Modeling Front–End that automatically generates the plain C code for this module from a standard model definition [21].

3.5 Simulation

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

4 Results

This section studies the performance of the new solver on three examples, comparing results with the same algorithms in PowerDEVS and also with DASSL solver in OpenModelica and Dymola, two of the most efficient tools for simulation of continuous and hybrid systems.

4.1 Logical Inverter Chain

The following model, presented in [22], represents a chain of m logical inverters

$$\dot{\omega}_j(t) = U_{op} - \omega_j(t) - \Upsilon g\left(\omega_{j-1}(t), \omega_j(t)\right) \quad (9)$$

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

$$g(u, v) = (\max(u - U_{th}, 0))^2 - (\max(u - v - U_{th}, 0))^2 \quad (10)$$

We used the set of parameters and initial conditions given in [22]: $\Upsilon = 100$ (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.(10).

Table 1 summarizes the simulation time and errors for the different solvers analyzed and the different tolerance settings.

For the *standard* tolerance setting of 10^{-3}, the QSS solver with LIQSS2 shows the best performance, simulating about 17 times faster than PowerDEVS using the same algorithm and accuracy. For these tolerance settings, LIQSS2 is more than 200 times faster than OpenModelica's DASSL.

We repeated the simulation with 500 inverters, obtaining a similar difference between the QSS Solver and PowerDEVS. The QSS solver simulates it in 110 milliseconds. This time, DASSL in Dymola takes about 20

		Tolerance	CPU time (msec)
QSS Solver	**LIQSS2**	10^{-3}	25
	LIQSS2	10^{-7}	807
	LIQSS3	10^{-3}	44
	LIQSS3	10^{-7}	210
OpenModelica	**DASSL**	10^{-3}	4405
	DASSL	10^{-7}	8734
Dymola	**DASSL**	10^{-3}	9950
	DASSL	10^{-7}	17600
PowerDEVS	**LIQSS2**	10^{-3}	300
	LIQSS2	10^{-7}	10200
	LIQSS3	10^{-3}	780
	LIQSS3	10^{-7}	1640

Table 1: Logical Inverter Chain results.

minutes to finish the simulation, so the QSS solver with LIQSS2 is about $10,000$ times faster.

For the last case, the usage of specialized multirate algorithms reported a simulation time of about 6 seconds [22]. Thus, the QSS Solver is performing more than 50 times faster than those special purpose methods.

4.2 Interleaved Buck Converter

We consider here an interleaved buck converter with 4 branches, with parameters $C = 1 \cdot 10^{-4}$ for the capacitor, $L = 1 \cdot 10^{-4}$ for the inductance, $R = 10$ for the load resistance, $U = 24$ for the input voltage. Also, we considered a switching period $T = 1 \cdot 10^{-4}$, a duty cycle $DC = 0.5$, and we assumed that the switch and diode have a resistance $R_{\text{On}} = 1 \cdot 10^{-5}$ in *on* state and $R_{\text{Off}} = 1 \cdot 10^{-5}$ in *off* state. This is the same model used in [8].

The results, summarized in Table 2, 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 implementations of Dymola and OpenModelica show also a huge speed up.

		Tolerance	CPU time (msec)
QSS Solver	**LIQSS2**	10^{-3}	7
	LIQSS2	10^{-7}	186
	LIQSS3	10^{-3}	25
	LIQSS3	10^{-7}	59
OpenModelica	**DASSL**	10^{-3}	157
	DASSL	10^{-7}	264
Dymola	**DASSL**	10^{-3}	170
	DASSL	10^{-7}	320
PowerDEVS	**LIQSS2**	10^{-3}	300
	LIQSS2	10^{-7}	10200
	LIQSS3	10^{-3}	780
	LIQSS3	10^{-7}	1640

Table 2: Interleaved Buck Converter results.

4.3 Advection–Diffusion–Reaction

This las example is the Method of Line discretization of an Advection–Diffusion–Reaction model, which lead to the sets 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. We simulated the model for $N = 500$, obtaining the CPU time described in Table 3.

		Tolerance	CPU time (msec)
QSS Solver	**LIQSS2**	10^{-3}	8
	LIQSS2	10^{-7}	600
	LIQSS3	10^{-3}	64
	LIQSS3	10^{-7}	217
OpenModelica	**DASSL**	10^{-3}	789
	DASSL	10^{-7}	3260
Dymola	**DASSL**	10^{-3}	660
	DASSL	10^{-7}	2870
PowerDEVS	**LIQSS2**	10^{-3}	250
	LIQSS2	10^{-7}	17000
	LIQSS3	10^{-3}	950
	LIQSS3	10^{-7}	1870

Table 3: Advection–Diffusion–Reaction results.

Again, the results show 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.

5 Conclusions and Future Research

We developed a stand–alone solver for QSS algorithms. The work described here with the addition of the modeling front–end described in the companion paper constitute a complete modeling and simulation environment based on QSS integration methods.

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 efficence of QSS methods on these systems makes also this tool up to two orders of magnitude faster than other solvers.

In the companion paper [21] we describe the modeling front–end that automatically produces the plain C code for the models needed by this solver.

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 [23].

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.

References

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

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

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

[4] E. Kofman and S. Junco, "Quantized State Systems. A DEVS Approach for Continuous System Simulation," *Transactions of SCS*, vol. 18, no. 3, pp. 123–132, 2001.

[5] E. Kofman, "Discrete Event Simulation of Hybrid Systems," *SIAM Journal on Scientific Computing*, vol. 25, no. 5, pp. 1771–1797, 2004.

[6] G. Grinblat, H. Ahumada, and E. Kofman, "Quantized State Simulation of Spiking Neural Networks," *Simulation: Transactions of the Society for Modeling and Simulation International*, vol. 88, no. 3, pp. 299–313, 2012.

[7] 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*, vol. 88, no. 4, pp. 387–407, 2012.

[8] 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*, 2013, in Press.

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

[10] L. R. Petzold, "Description of dassl: A differential/algebraic system solver," Sandia National Labs., Livermore, CA (USA), Tech. Rep., 1982.

[11] L. Shampine and M. Reichelt, "The matlab ODE suite." *SIAM Journal on Scientific Computing*, vol. 18, no. 1, pp. 1–22, 1997.

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

[13] E. Kofman, "Relative Error Control in Quantization Based Integration," *Latin American Applied Research*, vol. 39, no. 3, pp. 231–238, 2009.

[14] ——, "A Second Order Approximation for DEVS Simulation of Continuous Systems," *Simulation: Transactions of the Society for Modeling and Simulation International*, vol. 78, no. 2, pp. 76–89, 2002.

[15] ——, "A Third Order Discrete Event Simulation Method for Continuous System Simulation," *Latin American Applied Research*, vol. 36, no. 2, pp. 101–108, 2006.

[16] 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*, vol. 87, no. 1–2, pp. 113–132, 2011.

[17] T. Beltrame and F. Cellier, "Quantised state system simulation in Dymola/Modelica using the DEVS formalism," in *Proceedings of the Fifth International Modelica Conference*, vol. 1, Vienna, Austria, 2006, pp. 73–82.

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

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

[20] F. Esquembre, "Easy Java Simulations: a software tool to create scientific simulations in Java," *Computer Physics Communications*, vol. 156, no. 1, pp. 199–204, 2004.

[21] J. Fernandez and E. Kofman, "A Stand–Alone Quantized State System Solver. Part II," CIFASIS – CONICET, Rosario, Argentina, Tech. Rep., 2013, submitted to RPIC 2013. [Online]. Available: http://www.fceia.unr.edu.ar/∼kofman/files/FK13b.pdf

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

[23] F. Bergero, E. Kofman, and F. E. Cellier, "A Novel Parallelization Technique for DEVS Simulation of Continuous and Hybrid Systems." *Simulation: Transactions of the Society for Modeling and Simulation International*, 2012, in Press.