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 efficiency 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.

The 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 efficient 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,
\[ \dot{x}(t) = f(x(t), t) \] (1)
the first order Quantized State System method (QSS1) [4] approximates it by
\[ \dot{q}(t) = f(q(t), t) \] (2)
Here, \( q \) is the quantized state vector. Its entries are component-wise related with those of the state vector \( x \) by the following hysteretic quantization function:
\[ q_j(t) = \begin{cases} x_j(t) & \text{if } |x_j(t) - q_j(t^-)| \geq \Delta Q_j \\ q_j(t^-) & \text{otherwise} \end{cases} \] (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].
- 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 [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 higher 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 \( q(t) \) is a piecewise polynomial approximation of the corresponding component of the state \( x(t) \). 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 \( q \) is computed from \( x \).
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 trajectory \( q \) is known.

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

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

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 \( f \) that explicitly depend on \( q_i \).

In consequence, the models should provide the possibility of evaluating the individual components of function \( f \). 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.

\[
\dot{x}(t) = f(x,d,t) \tag{4}
\]

where \( d \) is a vector of discrete variables that can only change when the condition

\[
ZC_i(x,d,t) = 0 \tag{5}
\]

is met. The components \( ZC_i \) form a vector of zero-crossing functions \( ZC(x,d,t) \). When a zero crossing condition of Eq.(5) is verified, the state and discrete variables can change according to the corresponding event handler:

\[
(x(t),d(t)) = H_i(x(t^-),d(t^-)) \tag{6}
\]

### 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 \( x_j(t) \) of the state vector \( x(t) \):

\[
x_i(t) = \sum_{k=0}^{n-1} x_{i,k} \cdot (t-t_i^*)^k \tag{7}
\]

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

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

where \( n \) is the order of the method. For that goal, it integrates Eq.(4) 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 **QSS Integrator** proceeds as follows:

- Advance the simulation time to \( t \).
- Ask the Quantizer the new coefficients \( q_{i,k} \) and set \( t_i^* = t \).
- 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 \( t_j^* = t \).
  - Ask the Model which quantized state variables \( q_j \) other than \( q_i \) affect the expression of \( f_j \) and update the values of \( q_{j,l} \) from Eq.(8).
  - 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 \( q_i \):
In addition to this definitions, for efficiency 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 derivatives.

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

\[
\dot{\omega}_j(t) = U_{op} - \omega_j(t) - \Upsilon g(\omega_j(t), \omega_j(t))
\]

with \( j = 1, \ldots, m \) where

\[
g(u, v) = (\max(u - U_{th}, 0))^2 - (\max(u - v - U_{th}, 0))^2
\]

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 trapezoidal 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 \( \text{`max'} \) 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
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 $R_{on}=1 \cdot 10^{-5}$ in on state and $R_{off}=1 \cdot 10^{-5}$ 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.

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

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.

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 efficiency 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

<table>
<thead>
<tr>
<th>Tolerance</th>
<th>CPU time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIQSS2</td>
<td>10^{-3}</td>
</tr>
<tr>
<td>LIQSS2</td>
<td>10^{-7}</td>
</tr>
<tr>
<td>LIQSS3</td>
<td>10^{-4}</td>
</tr>
<tr>
<td>LIQSS3</td>
<td>10^{-7}</td>
</tr>
</tbody>
</table>

Table 2: Logical Inverter Chain results.

<table>
<thead>
<tr>
<th>Tolerance</th>
<th>CPU time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIQSS2</td>
<td>10^{-3}</td>
</tr>
<tr>
<td>LIQSS2</td>
<td>10^{-7}</td>
</tr>
<tr>
<td>LIQSS3</td>
<td>10^{-4}</td>
</tr>
<tr>
<td>LIQSS3</td>
<td>10^{-7}</td>
</tr>
</tbody>
</table>

Table 3: Interleaved Buck Converter results.

4.3 Advection–Diffusion–Reaction

This last example is the Method of Line discretization of an Advection–Diffusion–Reaction model, which lead to

\[
\frac{\partial^2 u_i}{\partial x^2} - \frac{\partial u_i}{\partial t} = 0
\]
and Windows can be downloaded from http://sourceforge.net/projects/qssengine/. The distribution contains also the models simulated in this article.

References