A Stand–Alone Quantized State System Solver. Part II

Joaquín Fernández† and Ernesto Kofman‡

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

Abstract—In the first part of this article, we introduced a stand–alone implementation of the Quantized State System (QSS) methods for continuous and hybrid system simulation. This solver increases in more than one order of magnitude the speed of QSS methods with respect to previous discrete event implementations. However, the solver requires that the models are programmed in plain C language providing also structure information through different incidence matrices, which is very uncomfortable for an end user.

In this second part of the article we describe a Modeling Front–End that, starting from a standard model definition, automatically generates the plain C code for the model including the structure required by the QSS solver.

We also describe a Graphic User Interface that integrates the different tools.

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

1 Introduction

QSS methods [1, 2] replace the time discretization performed by classic solvers for continuous systems simulation [3, 4, 2] by the quantization of the state variables.

These numerical algorithms have advantages over classic discrete time solvers, particularly in presence of frequent discontinuities [5] and some types of stiffness [6].

For simplicity reasons, previous implementations of QSS methods were confined to discrete event simulation engines such as PowerDEVS [7], CD++ [8] and VLE [9]. These implementations of the algorithms, although simple, are inefficient, as they waste a large amount of the computational load attending the discrete event simulation mechanisms.

Motivated by the lack of efficient implementations of the QSS methods, we developed a stand–alone QSS solver that improved PowerDEVS performance in more than one order of magnitude and was described in the first part of this article [10].

The main difficulty of the QSS solver is that it requires structure information about the model, in order to know which state derivatives and zero crossing functions are affected by the changes of a given state variable or by the occurrence of a given discontinuity. This structure information must be given in the form of incidence matrices which are very uncomfortable to obtain, particularly in large and complex systems.

Another problem for end users is that the model should be programmed in plain C language, defining all the functions and structures required by the solver including the already mentioned incidence matrices.

Thus, the direct usage of the solver is restricted to skilled programmers and for simulating relatively simple models.

To overcome this difficulty, we also developed a Modeling Front–End (MFE) which takes a standard model definition in a subset of Modelica language [11] and automatically converts it into a plain C code containing the structure information and all the functions required by the solver.

Additionally, we developed a Graphical User Interface (GUI) that integrates the solver, the MFE as well as some plot and debug tools. From that GUI, an end user can easily create, edit, debug and simulate models and plot the simulation results.

In this second part of the article we describe the Modeling Front–End and the GUI, and we also illustrate the way they work with some examples.

The article is organized as follows: Section introduces some preliminary concepts regarding QSS implementation and Modelica language. Then, Section 3 introduces the Modeling Front–End and Section 4 describes the Graphic User Interface.

2 Background

2.1 Some Features of QSS Methods

Given the ODE

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

with \( x \in \mathbb{R}^n \), the first order Quantized State System method (QSS1) [1] approximates it by

\[ \dot{x}(t) = f(q(t), t) \]  

where \( q(t) \) is the vector of quantized states.

Each quantized state variable \( q_i(t) \) is related to \( x_i(t) \) by a hysteretic quantization function. Then, \( q_i(t) \) only changes when it differs from \( x_i(t) \) in a quantity \( \Delta Q_i \).
called quantum. After each change, \( q_i(t) \) becomes equal to \( x_i(t) \).

In the QSS1 method, the quantized variables \( q_i(t) \) follow piecewise constant trajectories and the different components of \( q \) change at different times.

Thus, each step of the simulation involves only a change in a single quantized variable. If the quantized variable that changes is involved in the computation of some components of \( f(q, t) \), then only those components must be reevaluated in that simulation step. The other components of \( f \) do not need to be recomputed as they will remain unchanged.

In higher order QSS methods like second order QSS (QSS2) or third order QSS (QSS3), the mechanism is similar, except that quantized trajectories are piecewise linear or parabolic. Here, every step is related to the change in a single quantized state variable and it is not necessary to evaluate the complete right hand side of Eq.(2) but only the components \( f_j(q, t) \) that are directly affected by the state that changed.

The remaining QSS algorithms like Linearly Implicit QSS (LIQSS) of order 1 to 3 (LIQSS1, LIQSS2, and LIQSS3) follow the same behavior.

In presence of discontinuities, QSS algorithms work in a similar manner. Here, we consider hybrid models of the form

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

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

\[
ZC_i(x, d, t) = 0
\]

is met. The components \( ZC_i \) form a vector of zero crossing functions \( ZC(x, d, t) \). When a zero crossing condition of Eq.(4) is verified, the state and discrete variables can change according to the corresponding event handler:

\[
(x(t), d(t)) = H_i(x(t^-), d(t^-)).
\]

Each time a discontinuity occurs, the only state derivatives that change are those that are directly affected by the variables that changed at the corresponding discontinuity handler.

The fact that each step requires only the calculation of some state derivatives makes the QSS methods very efficient to simulate large sparse systems and systems with frequent discontinuities.

However, a solver needs to know which state derivatives and which zero crossing functions are directly affected by each state variable. Also, it needs to know which state derivatives and zero crossing functions should be reevaluated after each discontinuity occurrence.

The automatic obtention of this structure information was the main motivation of the Modeling Front–End described in this work.

### 2.2 Modelica Language

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

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

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

Models in Modelica are mathematically described by differential, algebraic and discrete equations. Submodels 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 [12] and OpenModelica [13].

The Modeling Front–End developed in this work allows the user to describe models in a subset of the Modelica language.

### 3 Modeling Front-End

Compared to a model definition used by classic solvers, the stand–alone QSS solver has the disadvantage that the model must provide additional information about the system structure, so that functions are only evaluated when it is necessary. In order to do this, as we explained in the first part of this article, the stand–alone solver needs a Model instance that must be written in plain C language.

As it is mentioned above, we allow the end–user to describe models in a subset of the Modelica language, called \( \mu \)-Modelica, and the Modeling Front–End automatically generates a suitable Model instance.

To accomplish that goal, the Modeling Front–End must perform different transformations to the new model description, which are performed 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 IR module, obtains information regarding all the state, algebraic and discrete variables defined in the model equations and events.
3. Model Generator module, constructs the incidence matrices of the system and generates the Model instance.

In order to complete the modeling front-end, we developed a simple graphic user interface as a separate module, to be able to create, edit models and interact with the simulation environment. The basic interaction between the modules mentioned above is depicted in Figure 1.
3.1 The $\mu$-Modelica Parser Module

The $\mu$-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 $\mu$-Modelica, that is a subset of the Modelica language. $\mu$-Modelica was conceived so that it contains only the necessary Modelica keywords and structures to define an ODE based hybrid model.

For instance, the following code corresponds to a bouncing ball model represented in $\mu$-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);
when y < 0 then
  contact := 1;
elsewhen y > 0 then
  contact := 0;
end when;

der(y) = vy;
der(vy) = -g - (contact * F)/m;

F = k*y+b*vy;
```

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 these features 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 1:2*N loop
  u[i] := 1;
end for;

der(u[i])=(-u[i]+u[i-1])*N-mu*u[i]*(u[i]-alpha)*(u[i]-1);
```

The $\mu$-Modelica language has the following restrictions:

- The model is in a flat form, i.e. no classes are allowed.
- All variables belong to the class `Real` and there are only three types 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 also belong to class `Real`. In the Advection–Reaction Model, $alpha$ and $mu$ are parameters.
- Arrays are allowed, but indexes are restricted to expressions of the form:
  $$\alpha \cdot i + \beta$$
  where $\alpha$ and $\beta$ are integer expressions and $i$ is the iteration index.
- The equation section is composed of:
  - **State variable derivatives** definitions:
    $$\text{der}(x) = f(x(t), d, a(t), t);$$
    in explicit ODE form.
  - **Algebraic variables** definitions of the form:
    $$(a_1, \cdots, a_n) = g(x(t), d, a(t), t);$$
    where an algebraic variable can only depend 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 reinitnents are allowed. In addition, conditions on elsewhen clauses can only be the inverse relation of the zero crossing function $e_1 \oplus e_2$ defined in the preceding when clause, thus, when–elsewhen clauses have the following generic description:
  ```
  when $e_1 \oplus e_2$ then
    $h_1 = f(x(t), d, a(t), t);
  elsewhen $e_1 \oplus e_2$ then
    $h_2 = f(x(t), d, a(t), t);
  end when;
  ```
  where $h_1$ and $h_2$ describe the actions taken when the different conditions are met, $e_1$ and $e_2$ are expressions of the form $f(x(t), d, a(t), t)$ and $\oplus$ must be the inverse relation of $\oplus$.

Then, the $\mu$-Modelica Parser module produces the first model transformation, generating the AST, which is a structured representation of a given model.
3.2 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 provided by the µ-Modelica Parser module.

In order to do this, the Model IR module analyzes all the equations and statements of the model to obtain the lists of variable dependences involved on each expression.

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

- All state variables \( x_i \) involved in the computation of \( e \).
- All discrete variables \( d_i \) involved in the computation of \( e \).
- All 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 \( e \).
- \( v \) is involved in the computation of an algebraic variable which is in turn involved in the computation of expression \( e \).

Expressions inside for statements are treated generically without expansion. In this case, the lists of variables includes information about the index ranges.

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

\[
 f_1(x, d, a, t) < f_2(x, d, a, t)
\]

is transformed into the zero crossing function

\[
 zc(x, d, a, t) = f_1(x, d, a, t) - f_2(x, d, a, t)
\]

3.3 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 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 \( \dot{x}_i = f_i(x(t), d(t), a(t), t) \)
  - 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 (\( \ddot{x}_i \)). 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.

```c
void model(int _i, double **_x, double *_d, double _t, double *_dx) {
    switch(_i) {
        case 1:
            _dx[1] = _x[1][0];
            return;
        case 2:
            _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] = 1;
            return;
    }
}

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

... //incidence matrix from states to derivatives
For the case of the advection–reaction equation, part of the C code generated is shown below

```c
void model_deps(int _i, double **_x, double **_d,
                 double **_der)
{
    int _res;
    switch(_i)
    {
    case 0:
        _der[0][1] = (-_x[0][0]+1)*500-mu*_x[0][0]*(
            _x[0][0]-alpha)*(_x[0][0]-1);
        break;
    }
    _res = _i+2;
    if(_res >= 2 && _res <= 500)
    {
        _der[_res-1][1] = (-_x[_res-1][0]+_x[_res-2][0])*500-mu*_x[_res-1][0]*(
            _x[_res-1][0]-alpha)*(_x[_res-1][0]-1);
    }
    _res = _i+1;
    if(_res >= 2 && _res <= 500)
    {
        _der[_res-1][1] = (-_x[_res-1][0]+_x[_res-2][0])*500-mu*_x[_res-1][0]*(
            _x[_res-1][0]-alpha)*(_x[_res-1][0]-1);
    }
}
```

4 Graphic User Interface

The simulation framework was complemented with a simple Graphic User Interface (GUI). The goal of the GUI is to provide the end-user with a unified tool to access the simulation framework developed.

The GUI has the following features:

- It has a text editor, where models in $\mu$-Modelica can be defined.
- 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 2 shows the GUI with the advection model presented before.

The right side of the GUI in Fig.2 allows to plot simulation results, providing an interface with GnuPlot. Figure 3 shows the simulation results for the advection equation.

5 Conclusions

In the second part of this article we presented a Modeling Front–End and a GUI that allows the usage of the QSS solver in very simple and efficient manner.

The Modeling Front–End translates models written in a subset of the Modelica language into the plain C code needed by the solver, providing support for discontinuity handling and large scale models.

The GUI allows the end–users to access all the tools developed in a simple and transparent way.

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 [13] which converts models from Modelica to $\mu$–Modelica [14].

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

The QSS Solver is an open source project,
and the source code and binaries for Linux and Windows can be downloaded from http://sourceforge.net/projects/qssengine/

References


