Automated Simulation of Modelica Models with QSS Methods
The Discontinuous Case

Xenofon Floros\textsuperscript{1}  Federico Bertero\textsuperscript{2}  François E. Cellier\textsuperscript{1}  Ernesto Kofman\textsuperscript{2}

\textsuperscript{1}Department of Computer Science, ETH Zurich, Switzerland
{xenofon.floros, francois.cellier}@inf.ethz.ch

\textsuperscript{2}Laboratorio de Sistemas Dinámicos, FCEIA, Universidad Nacional de Rosario, Argentina
CIFASIS-CONICET
{fbergero, kofman}@fceia.unr.edu.ar

March 22nd, 2011
Outline

Introduction

QSS Methods

OMPD Interface

Simulation Results

Discussion

Interfacing OpenModelica and PowerDEVS

ETH Zurich
Outline

Introduction

QSS Methods

OMPD Interface

Simulation Results

Discussion

Interfacing OpenModelica and PowerDEVS
Goal

Design and implement an interface between OpenModelica and PowerDEVS (OMPD Interface)

Enable the simulation of Modelica models with QSS methods
Why?

Interfacing OpenModelica and PowerDEVS we take advantage of:

- The powerful modeling tools and market share offered by Modelica
  - Users can still define their models using the Modelica language or their favorite graphical interface.
  - No prior knowledge of DEVS and QSS methods is needed.

- The superior performance of quantization-based techniques in some particular problem instances
  - QSS methods allow for asynchronous variable updates, which potentially speeds up the computations for real-world sparse systems.
  - QSS methods do not need to iterate backwards to handle discontinuities, they rather predict them, enabling real-time simulation.
Modelica—The next generation modeling language

Graphical editor for Modelica users

Modelica modeling environment (free or commercial)

Textual description

Free Modelica Language

Translation of Modelica models in C-Code and Simulation

Modelica simulation environment (free or commercial)
QSS methods

Simulation of continuous systems by a digital computer requires discretization.

- Classical methods (e.g. Euler, Runge-Kutta etc.), that are implemented in Modelica environments, are based on discretization of time.
- On the other hand, the Discrete Event System Specification (DEVS) formalism, introduced by Zeigler in the 90s, enables the discretization of states.
- The Quantized-State Systems (QSS) methods, introduced by Kofman in 2001, improved the original quantized-state approach of Zeigler.
- PowerDEVS is the environment where QSS methods have been implemented for the simulation of systems described in DEVS.
PowerDEVS

- Specify system structure (using DEVS formalism)
- Block implementation hidden (C++ code)
- Integrators implement the QSS methods
- Simulation using hierarchical master-slave structure and message passing

http://sourceforge.net/projects/powerdevs/
Outline

Introduction

QSS Methods

OMPD Interface

Simulation Results

Discussion
Quantized State Systems Method

Definition

Given a system

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

with \(x \in \mathbb{R}^n\), \(t \in \mathbb{R}\) and \(f : \mathbb{R}^{n+1} \rightarrow \mathbb{R}^n\), the QSS approximation is given by

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

where \(q(t)\) and \(x(t)\) are related componentwise by hysteretic quantization functions.

Under certain assumptions, the QSS approximation (2) is shown to be equivalent to a legitimate DEVS model.
QSS Method and Perturbed Systems

Defining $\Delta x(t) \triangleq q(t) - x(t)$, the QSS approximation (2) can be rewritten as:

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

(3)

Notice that every component of $\Delta x$ satisfies

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

(4)

where $\Delta Q_i$ is the quantization width (or quantum) in the $i$–th component.

The effect of the QSS discretization can be studied as a problem of bounded perturbations over the original ODE.

At each step only one (quantized) state variable that changes more than the quantum value $\Delta Q_i$ is updated producing a discrete event.
Static Functions & Quantized Integrators

If we break (2) into the individual components we have that:

\[
\dot{x}_1 = f_1(x_1, \ldots, x_n, t) \\
\vdots \\
\dot{x}_n = f_n(x_1, \ldots, x_n, t)
\]

\[\rightarrow_{QSS}\]

\[
\dot{x}_1 = f_1(q_1, \ldots, q_n, t) \\
\vdots \\
\dot{x}_n = f_n(q_1, \ldots, q_n, t)
\]

Considering a single subcomponent we can define the "simple" DEVS models:
Static Functions & Quantized Integrators

If we break (2) into the individual components we have that:

\[ \dot{x}_1 = f_1(x_1, \ldots, x_n, t) \quad \rightarrow \quad \dot{x}_1 = f_1(q_1, \ldots, q_n, t) \]

\[ \vdots \]

\[ \dot{x}_n = f_n(x_1, \ldots, x_n, t) \quad \rightarrow \quad \dot{x}_n = f_n(q_1, \ldots, q_n, t) \]

(5)

Considering a single subcomponent we can define the ”simple” DEVS models:

\[ q_i = Q(x_i) = Q(\int \dot{x}_i \, dt) \]

Quantized Integrator
Static Functions & Quantized Integrators

If we break (2) into the individual components we have that:

\[
\begin{align*}
\dot{x}_1 &= f_1(x_1, \ldots, x_n, t) \\
\vdots \\
\dot{x}_n &= f_n(x_1, \ldots, x_n, t)
\end{align*}
\]

\[\Rightarrow\]

\[
\begin{align*}
\dot{x}_1 &= f_1(q_1, \ldots, q_n, t) \\
\vdots \\
\dot{x}_n &= f_n(q_1, \ldots, q_n, t)
\end{align*}
\]

(5)

Considering a single subcomponent we can define the "simple" DEVS models:

\[
q_i = Q(x_i) = Q(\int \dot{x}_i \, dt)
\]

Quantized Integrator

\[
\dot{x}_i = f_i(q_1, \ldots, q_n, t)
\]

Static Function
QSS – Example

Solution with $\Delta Q = 0.01$, $u(t) = 1$

Let second order LTI system:

\[
\begin{align*}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= -x_1(t) - x_2(t) + u(t)
\end{align*}
\]
QSS – Example

Solution with $\Delta Q = 0.05$, $u(t) = 1$

Let second order LTI system:

$$\dot{x}_1(t) = x_2(t)$$
$$\dot{x}_2(t) = -x_1(t) - x_2(t) + u(t)$$
QSS – Example

Solution with \( \Delta Q = 0.1, u(t) = 1 \)

Let second order LTI system:

\[
\begin{align*}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= -x_1(t) - x_2(t) + u(t)
\end{align*}
\]
Higher-Order QSS Methods

Cost vs. Accuracy in QSS

In QSS, we know that the quantum is proportional to the global error bound. Thus,

- If we want to increase the global accuracy for a factor of 100, we should divide the quantum by that factor.
- Since the number of steps is inversely proportional to the quantum, that modification would increase the number of computations by a factor of 100.

This problem is due to the fact that QSS is only first order accurate, i.e. it does not use information about the derivatives of \( f \).
Higher-Order QSS Methods

Second Order QSS (QSS2 Method)

- Same definition and properties as QSS.
- **Second order** accurate method.
- The number of steps grows with the square root of the accuracy.
- The quantized variables have piecewise linear trajectories thus the state derivatives are also piecewise linear and the state variables piecewise parabolic.
Higher-Order QSS Methods

Third Order QSS (QSS3 Method)

- Same definition and properties as QSS.
- Third order accurate method.
- The number of steps grows with the cubic root of the accuracy.
- The method of choice for simulating real-world systems.
Outline

Introduction

QSS Methods

OMPD Interface

Simulation Results

Discussion
OpenModelica Compiler Modifications

Modelica Source Code

Translator

Analyzer

Optimizer

Code Generator

C Compiler

Simulation

Modelica model

Flat Model

Sorted equations

Modified Code Generator

PowerDEVS Structure File (model.pds)

PowerDEVS Block Code (modelica_funcs.cpp)

C++ compiler

Simulation
The Bouncing Ball Model

model BouncingBall
  parameter Real e=0.7 "coefficient of restitution";
  parameter Real g=9.81 "gravity acceleration";
  Real h(start=1) "height of ball";
  Real v "velocity of ball";
  Boolean flying(start=true) "true, if ball is flying";
  Real v_new;
  Boolean impact;
  Real dummy;
  Boolean dummy2;

equation
  der(dummy) = if (dummy>0 and h<=0) then
    dummy else h*v; // Dummy part 1
  when {sample(0,1)} // Dummy part 2
    dummy2 = false;
  end when

  impact = h <= 0.0;
  der(v) = if flying then -g else 0;
  der(h) = v;

  when {h <= 0.0 and v <= 0.0,impact} then
    v_new = if edge(impact) then -e*v else 0;
    flying = v_new > 0;
    reinit(v, v_new);
  end when;

end BouncingBall;
Add Static Blocks for State Variables

- Extract equations (BLT blocks) needed to compute state derivative variables.
- Place the splitted equations in respective static function blocks.
- Resolve dependencies in the inputs/outputs.

\[
\text{der}(h) = v; \quad \text{(Eq. 1)}
\]
\[
\text{der}(\text{dummy}) = \begin{cases} 
\text{if } (\text{dummy}>0 \text{ and } h\leq0) \text{ then} \\
\text{dummy} \text{ else } h \cdot v; 
\end{cases} \quad \text{(Eq. 2)}
\]
\[
\text{dummy}_2 = \text{false}; \quad \text{(Eq. 3)}
\]
\[
\text{impact} = h \leq 0.0; \quad \text{(Eq. 4)}
\]
\[
\text{when } \{h \leq 0.0 \text{ and } v \leq 0.0, \text{impact}\} \text{ then}
\]
\[
\text{v}_\text{new} = \begin{cases} 
\text{if } \text{edge}(\text{impact}) \text{ then} \\
-e \cdot v \text{ else } 0; 
\end{cases} \quad \text{(Eq. 5)}
\]
\[
\text{flying} = v\_\text{new} > 0; \quad \text{(Eq. 6)}
\]
\[
\text{reinit}(v, v\_\text{new});
\]
\[
\text{end when};
\]
\[
\text{der}(v) = \begin{cases} 
\text{if } \text{flying} \text{ then } -g \text{ else } 0; 
\end{cases} \quad \text{(Eq. 7)}
\]
Add Zero Crossing Functions

- Add zero-crossing functions and the corresponding zero-cross detectors.
- Resolve dependencies in the inputs/outputs.
- The zero-cross detectors produce events at discontinuities and propagate them to the corresponding static blocks.

\[
\text{der}(h) = v; \quad (\text{Eq. 1})
\]
\[
\text{der}(\text{dummy}) = \begin{cases} 
\text{dummy} & \text{if (dummy} > 0 \text{ and } h \leq 0) \\
\text{else } h \cdot v & \end{cases}; \quad (\text{Eq. 2})
\]
\[
\begin{cases} 
\text{dummy2} = \text{false}; & \text{when } \{\text{sample}(0,1)\} \\
\text{impact} = h \leq 0.0; & (\text{Eq. 4}) \\
\text{when } (h \leq 0.0 \text{ and } v \leq 0.0, \text{impact}) \text{ then} \\
\text{v}._{\text{new}} = \begin{cases} 
\text{edge}(\text{impact}) & \text{then} \\
-e \cdot v & \text{else } 0; & (\text{Eq. 5}) \\
\text{flying} = v._{\text{new}} > 0; & (\text{Eq. 6}) \\
\text{reinit}(v, v._{\text{new}}); & \text{end when} \\
\text{der}(v) = \begin{cases} 
\text{if flying} & \text{then } -g \text{ else } 0; & (\text{Eq. 7}) \\
\end{cases}
\end{cases}
\]
Add When Blocks

\[
\text{der}(h) = v; \quad (\text{Eq. 1})
\]
\[
\text{der}(\text{dummy}) = \begin{cases} 
\text{dummy} & \text{if } (\text{dummy}>0 \text{ and } h \leq 0) \text{ then } \\
\text{h} \times v & \text{else}
\end{cases}; \quad (\text{Eq. 2})
\]
\[
\text{when } \{\text{sample}(0,1)\}
\quad \text{dummy2} = \text{false}; \quad (\text{Eq. 3})
\text{end when}
\]
\[
\text{impact} = h \leq 0.0; \quad (\text{Eq. 4})
\]
\[
\text{when } \{h \leq 0.0 \text{ and } v \leq 0.0, \text{impact}\} \text{ then}
\quad v_{\text{new}} = \begin{cases} 
-e \times v & \text{if } \text{edge}(\text{impact}) \text{ then } \\
0 & \text{else}
\end{cases}; \quad (\text{Eq. 5})
\quad \text{flying} = v_{\text{new}} > 0; \quad (\text{Eq. 6})
\quad \text{reinit}(v, v_{\text{new}});
\text{end when;}
\]
\[
\text{der}(v) = \begin{cases} 
-e \times v & \text{if } \text{flying} \text{ then } \\
0 & \text{else}
\end{cases}; \quad (\text{Eq. 7})
\]

- Add when-blocks for each generated when-clause and resolve dependencies.
- If a static function depends on a discrete variable calculated in a when-block (e.g. flying) an event is sent to the corresponding static block.
- When a cross detector fires, all the discrete variables are updated via calling the OMC function updateDepend().
Add Sample Blocks

- Add one sample block for each sample statement.
- Connect the sample blocks to the dependent when-clauses.

\[
der(h) = v; \quad \text{(Eq. 1)}
\]
\[
der(dummy) = \text{if (dummy}>0 \text{ and } h<=0) \text{ then dummy else } h*v; \quad \text{(Eq. 2)}
\]
\[
\text{when \{sample(0,1)\}}
\]
\[
dummy2 = \text{false}; \quad \text{(Eq. 3)}
\]
\[
\text{end when}
\]
\[
impact = h <= 0.0; \quad \text{(Eq. 4)}
\]
\[
\text{when \{ h <= 0.0 and v <= 0.0,impact \} then}
\]
\[
v_{\text{new}} = \text{if edge(impact) then } -e*v \text{ else 0}; \quad \text{(Eq. 5)}
\]
\[
\text{flying} = v_{\text{new}} > 0; \quad \text{(Eq. 6)}
\]
\[
\text{reinit(v, v_{\text{new}});}
\]
\[
\text{end when;}
\]
\[
der(v) = \text{if flying then } -g \text{ else 0}; \quad \text{(Eq. 7)}
\]
**Add Reinit Blocks**

- **der(h) = v;** (Eq. 1)
- **der(dummy) = if (dummy>0 and h<=0) then dummy else h*v;** (Eq. 2)
- **when {sample(0,1)}
  dummy2 = false;** (Eq. 3)
- **end when
impact = h <= 0.0;** (Eq. 4)
- **when {h <= 0.0 and v <= 0.0,impact}
  v_new = if edge(impact) then -e*v else 0;** (Eq. 5)
  **flying = v_new > 0;** (Eq. 6)
  **reinit(v, v_new);**
- **end when
  der(v) = if flying then -g else 0;** (Eq. 7)

▶ Add reinit blocks for the reinit statements and connect them to the corresponding integrators.
der(h) = v; (Eq. 1)
der(dummy) = if (dummy>0 and h<=0) then
dummy else h*v; (Eq. 2)
when {sample(0,1)}
dummy2 = false; (Eq. 3)
end when
impact = h <= 0.0; (Eq. 4)
when {h <= 0.0 and v <= 0.0,impact} then
v_new = if edge(impact) then
-e*v else 0; (Eq. 5)
flying = v_new > 0; (Eq. 6)
reinit(v, v_new);
end when;
der(v) = if flying then -g else 0; (Eq. 7)
Outline

Introduction

QSS Methods

OMPD Interface

Simulation Results

Discussion
Compared Solvers

The goal is to compare the **run-time efficiency** and **accuracy** of QSS methods against the following representative methods and environments:

- DASSL in OpenModelica v1.5.1 and Dymola v7.4: A state-of-the-art multi-purpose solver used by most simulation environments today.
- Radau IIa in Dymola v7.4: A single-step (Runge-Kutta) algorithm is supposed to be more efficient than a multi-step algorithm when dealing with discontinuities (due to step-size control for the latter methods).
- Dopri45 in Dymola v7.4: An explicit Runge-Kutta method which could be more efficient when simulating non-stiff systems.
The goal is to compare the run-time efficiency and accuracy of QSS methods against the following representative methods and environments:

- **DASSL** in OpenModelica v1.5.1 and Dymola v7.4
  - State-of-the-art multi-purpose solver used by most simulation environments today.
Compared Solvers

The goal is to compare the **run-time efficiency** and **accuracy** of QSS methods against the following representative methods and environments:

- **DASSL** in OpenModelica v1.5.1 and Dymola v7.4  
  ▶ State-of-the-art multi-purpose solver used by most simulation environments today.

- **Radau IIa** in Dymola v7.4  
  ▶ A single-step (Runge-Kutta) algorithm is supposed to be more efficient than a multi-step algorithm when dealing with discontinuities (due to step-size control for the latter methods).
The goal is to compare the **run-time efficiency** and **accuracy** of QSS methods against the following representative methods and environments:

- **DASSL** in OpenModelica v1.5.1 and Dymola v7.4
  - State-of-the-art multi-purpose solver used by most simulation environments today.
- **Radau IIa** in Dymola v7.4
  - A single-step (Runge-Kutta) algorithm is supposed to be more efficient than a multi-step algorithm when dealing with discontinuities (due to step-size control for the latter methods).
- **Dopri45** in Dymola v7.4
  - An explicit Runge-Kutta method which could be more efficient when simulating non-stiff systems.
Run-time Efficiency (Execution Time)

Problem

- Measuring the execution time of each simulation across different environments could be tricky, e.g. it is not enough just to run the executables and measure the CPU-time elapsed.
Run-time Efficiency (Execution Time)

Problem

- Measuring the execution time of each simulation across different environments could be tricky, e.g. it is not enough just to run the executables and measure the CPU-time elapsed.

Approach

- We resort in using the reported simulation time that each environment provides.
Run-time Efficiency (Execution Time)

Problem

- Measuring the execution time of each simulation across different environments could be tricky, e.g. it is not enough just to run the executables and measure the CPU-time elapsed.

Approach

- We resort in using the reported simulation time that each environment provides.
- The generation of output files was suppressed in all cases.
Run-time Efficiency (Execution Time)

Problem

- Measuring the execution time of each simulation across different environments could be tricky, e.g. it is not enough just to run the executables and measure the CPU-time elapsed.

Approach

- We resort in using the **reported simulation time** that each environment provides.
- The generation of output files was suppressed in all cases.

Reminder

- The measured CPU time should not be considered as an absolute ground-truth.
Run-time Efficiency (Execution Time)

Problem

- Measuring the execution time of each simulation across different environments could be tricky, e.g. it is not enough just to run the executables and measure the CPU-time elapsed.

Approach

- We resort in using the reported simulation time that each environment provides.
- The generation of output files was suppressed in all cases.

Reminder

- The measured CPU time should not be considered as an absolute ground-truth.
- But the relative ordering of the algorithms is expected to remain the same.
Simulation Accuracy

Benchmark Framework

The state trajectories in the benchmark problems cannot be computed analytically. Therefore, we can only approximate the accuracy of the simulations. To this end we need to obtain reference trajectories \((t_{\text{ref}}, y_{\text{ref}})\).

The default DASSL solver both in Dymola and OpenModelica was used with a very tight tolerance of \(10^{-12}\) and requesting \(10^5\) output points. The difference between both reference trajectories was on the order of \(10^{-6}\) therefore we report only the simulation error against the Dymola solution.
Simulation Accuracy

- The state trajectories in the benchmark problems cannot be computed analytically.
Simulation Accuracy

- The state trajectories in the benchmark problems cannot be computed analytically.
- Therefore, we can only approximate the accuracy of the simulations.
Simulation Accuracy

- The state trajectories in the benchmark problems cannot be computed analytically.
- Therefore, we can only approximate the accuracy of the simulations.
- To this end we need to obtain reference trajectories \((t^{\text{ref}}, y^{\text{ref}})\).
Simulation Accuracy

- The state trajectories in the benchmark problems cannot be computed analytically.
- Therefore, we can only approximate the accuracy of the simulations.
- To this end we need to obtain reference trajectories \((t^{\text{ref}}, y^{\text{ref}})\).

Reference Trajectories

- The default DASSL solver both in Dymola and OpenModelica was used with
  - a very tight tolerance of \(10^{-12}\) and
  - requesting \(10^5\) output points.
- The difference between both reference trajectories was on the order of \(10^{-6}\) therefore we report only the simulation error against the Dymola solution.
For each state a reference trajectory \((t_{\text{ref}}, y_{\text{ref}})\) is calculated.

Each solver is forced to output 10 equally spaced points to obtain \((t_{\text{ref}}, y_{\text{sim}})\) without changing the integration step.

Then, the mean absolute error is calculated as:

\[
\text{error} = \frac{1}{|t_{\text{ref}}|} \sum_{i=1}^{|t_{\text{ref}}|} |y_{\text{sim}} - y_{\text{ref}}| \quad (6)
\]
Simulation Accuracy

- For each state a reference trajectory ($t^{\text{ref}}$, $y^{\text{ref}}$) is calculated.
Simulation Accuracy

- For each state a reference trajectory \((t^{\text{ref}}, y^{\text{ref}})\) is calculated.
- Each solver is forced to output \(10^5\) equally spaced points to obtain \((t^{\text{ref}}, y^{\text{sim}})\) without changing the integration step.
- Then, the mean absolute error is calculated as:
Simulation Accuracy

- For each state a reference trajectory \((t^{\text{ref}}, y^{\text{ref}})\) is calculated.
- Each solver is forced to output \(10^5\) equally spaced points to obtain \((t^{\text{ref}}, y^{\text{sim}})\) without changing the integration step.
- Then, the mean absolute error is calculated as:

\[
\text{error} = \frac{1}{|t^{\text{ref}}|} \sum_{i=1}^{|t^{\text{ref}}|} |y_{i}^{\text{sim}} - y_{i}^{\text{ref}}|
\]  

(6)
Half-Wave Rectifier

Figure: Graphical representation of the half-wave rectifier in Dymola
Simulated trajectories for the half-wave rectifier

![Simulated trajectories for the half-wave rectifier](image-url)
### Simulated Discontinuous Models

#### Half-Wave Rectifier (Simulated for 1 sec)

<table>
<thead>
<tr>
<th></th>
<th>CPU time (sec)</th>
<th>Simulation Error</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dymola</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DASSL $10^{-3}$</td>
<td>0.019</td>
<td>1.45E-03</td>
</tr>
<tr>
<td>DASSL $10^{-4}$</td>
<td>0.022</td>
<td>2.35E-04</td>
</tr>
<tr>
<td>Radau Ila $10^{-7}$</td>
<td>0.031</td>
<td>2.20E-06</td>
</tr>
<tr>
<td>Dopri45 $10^{-4}$</td>
<td>0.024</td>
<td>4.65E-05</td>
</tr>
<tr>
<td><strong>PowerDEVS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QSS3 $10^{-3}$</td>
<td>0.014</td>
<td>2.59E-04</td>
</tr>
<tr>
<td>QSS3 $10^{-4}$</td>
<td>0.026</td>
<td>2.23E-05</td>
</tr>
<tr>
<td>QSS3 $10^{-5}$</td>
<td>0.041</td>
<td>2.30E-06</td>
</tr>
<tr>
<td>QSS2 $10^{-2}$</td>
<td>0.242</td>
<td>3.02E-03</td>
</tr>
<tr>
<td>QSS2 $10^{-3}$</td>
<td>0.891</td>
<td>3.04E-04</td>
</tr>
<tr>
<td>QSS2 $10^{-4}$</td>
<td>3.063</td>
<td>3.00E-05</td>
</tr>
<tr>
<td><strong>OpenModelica</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DASSL $10^{-3}$</td>
<td>0.265</td>
<td>3.80E-03</td>
</tr>
<tr>
<td>DASSL $10^{-4}$</td>
<td>0.281</td>
<td>5.40E-04</td>
</tr>
</tbody>
</table>
Simulated Discontinuous Models

Switching Power Converter

---

**Figure**: Graphical representation of the switching power converter in Dymola
Simulated state trajectories for the switching power converter

![Graph showing simulated state trajectories for the switching power converter. The graph plots inductor current and capacitor voltage over time. The x-axis represents time in seconds, scaled to $10^{-3}$, while the y-axis represents the state trajectories. The inductor current is shown as a blue dashed line, and the capacitor voltage is shown as a red line. The trajectories stabilize after an initial transient.]
## Switching Power Converter (Simulated for 0.01 sec)

<table>
<thead>
<tr>
<th>Method</th>
<th>Dymola</th>
<th>PowerDEVS</th>
<th>OpenModelica</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dymola</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DASSL</td>
<td>$10^{-3}$</td>
<td>0.051</td>
<td>1.82E-04</td>
</tr>
<tr>
<td>DASSL</td>
<td>$10^{-4}$</td>
<td>0.063</td>
<td>7.18E-05</td>
</tr>
<tr>
<td>Radau IIa</td>
<td>$10^{-3}$</td>
<td>0.064</td>
<td>1.11E-07</td>
</tr>
<tr>
<td>Radau IIa</td>
<td>$10^{-4}$</td>
<td>0.062</td>
<td>1.11E-07</td>
</tr>
<tr>
<td>Dopri45</td>
<td>$10^{-3}$</td>
<td>0.049</td>
<td>6.38E-06</td>
</tr>
<tr>
<td>Dopri45</td>
<td>$10^{-4}$</td>
<td>0.047</td>
<td>9.76E-06</td>
</tr>
<tr>
<td>PowerDEVS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QSS3</td>
<td>$10^{-3}$</td>
<td>0.049</td>
<td>1.41E-03</td>
</tr>
<tr>
<td>QSS3</td>
<td>$10^{-4}$</td>
<td>0.062</td>
<td>1.68E-05</td>
</tr>
<tr>
<td>QSS3</td>
<td>$10^{-5}$</td>
<td>0.250</td>
<td>8.96E-06</td>
</tr>
<tr>
<td>OpenModelica</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DASSL</td>
<td>$10^{-3}$</td>
<td>50.496</td>
<td>-</td>
</tr>
<tr>
<td>DASSL</td>
<td>$10^{-4}$</td>
<td>1.035</td>
<td>2.62E-02</td>
</tr>
</tbody>
</table>
Outline

Introduction

QSS Methods

OMPD Interface

Simulation Results

Discussion

Interfacing OpenModelica and PowerDEVS
Conclusions

- An interface between OpenModelica and PowerDEVS is presented and analyzed.
- The OMPD interface successfully handles discontinuities allowing the simulation of real-world Modelica models using QSS solvers.
- Comparing QSS3 and DASSL in OpenModelica, a **20-fold decrease** in the required CPU time was achieved for the example models.
- Furthermore in our discontinuous examples, QSS3 is as efficient as DASSL in Dymola, in spite of the fact that Dymola offers a much more sophisticated model preprocessing than OMC.
Future Work

- Provide support for stiff QSS solvers.
- Perform more extensive simulations of benchmark problems in order to test the correctness of the interface and the performance of QSS methods.
- Incorporate QSS solvers in future official OpenModelica releases.
- Investigate the parallel simulation capabilities of QSS methods.
Discussion

Questions?