Automated Simulation of Modelica Models with QSS Methods

The Discontinuous Case

Xenofon Floros[1] Federico Bergero[2] François E. Cellier[1] Ernesto Kofman[2]

[1] Department of Computer Science, ETH Zurich, Switzerland
{xenofon.floros, francois.cellier}@inf.ethz.ch
[2] Laboratorio de Sistemas Dinámicos, FCEIA, Universidad Nacional de Rosario, Argentina
CIFASIS-CONICET
{fbergero, kofman}@fceia.unr.edu.ar

March 22nd, 2011

ITEA 2 OPENPROD Förderung KTI/CTI

CTI grant
Nr.12101.1;3 PFES-ES

Outline

Outline

Goal

OpenMODELICA

↓

PowerDEVS

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

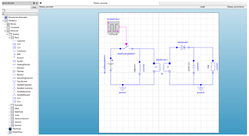
- Users can still define their models using the Modelica language or their favorite graphical interface.
- No prior knowledge of DEVS and QSS methods is needed.

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



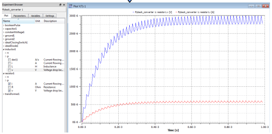
Modelica modeling environment (free or commercial)

Textual description

```
model flyback_converter
  Modelica.Electrical.Analog.Basic.Ground ground1 annotation(Placement(transformation(extent={{-70,-40},{-50,-20}})));
  Modelica.Electrical.Analog.Basic.Resistor resistor1(R=5) annotation(Placement(transformation(extent={{-10,-10},{10,10}}, rotation=270, origin={90,
  Modelica.Electrical.Analog.Basic.Inductor inductor1(L=0.0002) annotation(Placement(transformation(extent={{-10,-10},{10,10}}, rotation=270, origin
  Modelica.Electrical.Analog.Basic.Capacitor capacitor1(C=2.2e-05) annotation(Placement(transformation(extent={{-10,-10},{10,10}}, rotation=270, ori
  Modelica.Electrical.Analog.Basic.Ground ground2 annotation(Placement(transformation(extent={{60,-40},{80,-20}})));
  Modelica.Electrical.Analog.Basic.Transformer transformer1(M=-2) annotation(Placement(transformation(extent={{-8,-4},{18,22}})));
  Modelica.Electrical.Analog.Sources.ConstantVoltage constantVoltage1(V=40) annotation(Placement(transformation(extent={{-10,-10},{10,10}}, rotation
  Modelica.Electrical.Analog.Ideal.IdealClosingSwitch idealClosingSwitch1 annotation(Placement(transformation(extent={{-64,30},{-44,50}})));
  Modelica.Electrical.Analog.Ideal.IdealDiode idealDiode1 annotation(Placement(transformation(extent={{30,32},{50,52}})));
  Modelica.Blocks.Sources.BooleanPulse booleanPulse(period=1.675e-05) annotation(Placement(transformation(extent={{-86,66},{-66,86}})));
equation
  connect(idealDiode1.n,resistor1.p) annotation(Line(points={{50,42},{90,42},{90,20}}, color={0,0,255}, smooth=Smooth.None));
  connect(resistor1.n,ground2.p) annotation(Line(points={{90,0},{90,-20},{70,-20}}, color={0,0,255}, smooth=Smooth.None));
  connect(capacitor1.n,ground2.p) annotation(Line(points={{58,0},{58,-20},{70,-20}}, color={0,0,255}, smooth=Smooth.None));
  connect(idealDiode1.p,transformer1.p2) annotation(Line(points={{30,42},{18,42},{18,15.5}}, color={0,0,255}, smooth=Smooth.None));
  connect(transformer1.n2,ground2.p) annotation(Line(points={{18,2.5},{18,-20},{70,-20}}, color={0,0,255}, smooth=Smooth.None));
  connect(capacitor1.p,idealDiode1.n) annotation(Line(points={{58,20},{58,42},{50,42}}, color={0,0,255}, smooth=Smooth.None));
  connect(transformer1.n1,ground1.p) annotation(Line(points={{-8,2.5},{-8,-20},{-60,-20}}, color={0,0,255}, smooth=Smooth.None));
  connect(ground1.p,constantVoltage1.n) annotation(Line(points={{-60,-20},{-80,-20},{-80,4}}, color={0,0,255}, smooth=Smooth.None));
  connect(constantVoltage1.p,idealClosingSwitch1.p) annotation(Line(points={{-80,24},{-80,40},{-64,40}}, color={0,0,255}, smooth=Smooth.None));
  connect(idealClosingSwitch1.n,transformer1.p1) annotation(Line(points={{-44,40},{-8,40},{-8,15.5}}, color={0,0,255}, smooth=Smooth.None));
  connect(inductor1.p,idealClosingSwitch1.n) annotation(Line(points={{-32,18},{-32,40},{-44,40}}, color={0,0,255}, smooth=Smooth.None));
  connect(inductor1.n,ground1.p) annotation(Line(points={{-32,-2},{-32,-20},{-60,-20}}, color={0,0,255}, smooth=Smooth.None));
  connect(booleanPulse.y,idealClosingSwitch1.control) annotation(Line(points={{-65,76},{-54,76},{-54,47}}, color={255,0,255}, smooth=Smooth.None));
  annotation(uses(Modelica(version="3.1")), Diagram(coordinateSystem(preserveAspectRatio=true, extent={{-100,-100},{100,100}}), graphics));
end flyback_converter;
```

Free Modelica Language

Translation of Modelica models in C-Code and Simulation



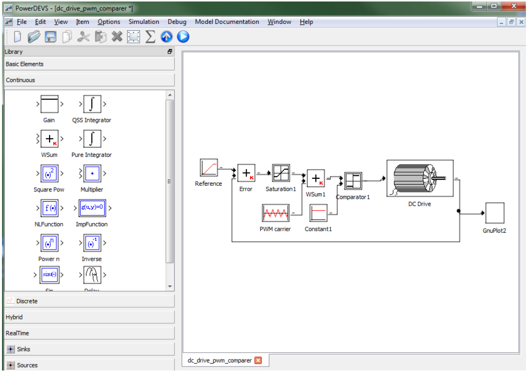
Modelica simulation environment (free or commercial)

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

Quantized State Systems Method

Definition

Given a system

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

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

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

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:

$$\dot{x}(t) = f[x(t) + \Delta x(t), t] \quad (3)$$

Notice that every component of Δx satisfies

$$|\Delta x_i(t)| = |q_i(t) - x_i(t)| \leq \Delta Q_i \quad (4)$$

where ΔQ_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 ΔQ_i is updated producing a discrete event.

Static Functions & Quantized Integrators

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

$$\begin{aligned} \dot{x}_1 &= f_1(x_1, \ldots, x_n, t) \\ &\vdots \\ \dot{x}_n &= f_n(x_1, \ldots, x_n, t) \end{aligned} \quad \overset{QSS}{\Longrightarrow} \quad \begin{aligned} \dot{x}_1 &= f_1(q_1, \ldots, q_n, t) \\ &\vdots \\ \dot{x}_n &= f_n(q_1, \ldots, q_n, t) \end{aligned} \tag{5}$$

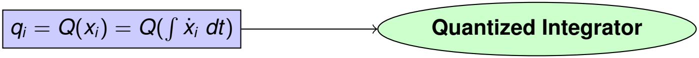
Considering a single subcomponent we can define the ”simple” DEVS models:

Static Functions & Quantized Integrators

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

$$\begin{aligned} \dot{x}_1 &= f_1(x_1, \ldots, x_n, t) \\ &\vdots \\ \dot{x}_n &= f_n(x_1, \ldots, x_n, t) \end{aligned} \quad \overset{QSS}{\Longrightarrow} \quad \begin{aligned} \dot{x}_1 &= f_1(q_1, \ldots, q_n, t) \\ &\vdots \\ \dot{x}_n &= f_n(q_1, \ldots, q_n, t) \end{aligned} \qquad (5)$$

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

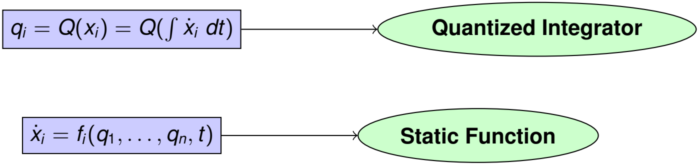


Static Functions & Quantized Integrators

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

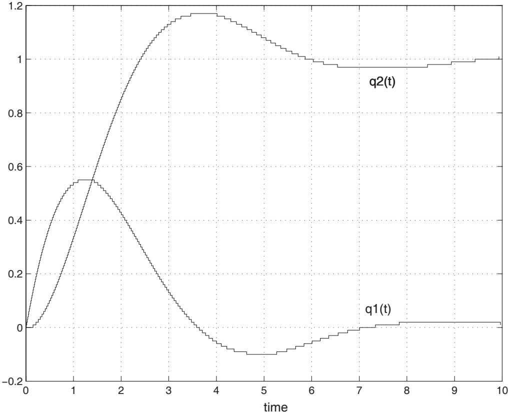
$$
\begin{aligned}
\dot{x}_1 &= f_1(x_1, \ldots, x_n, t) \\
&\vdots \\
\dot{x}_n &= f_n(x_1, \ldots, x_n, t)
\end{aligned}
\quad \overset{QSS}{\Longrightarrow} \quad
\begin{aligned}
\dot{x}_1 &= f_1(q_1, \ldots, q_n, t) \\
&\vdots \\
\dot{x}_n &= f_n(q_1, \ldots, q_n, t)
\end{aligned}
\tag{5}
$$

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



QSS – Example

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

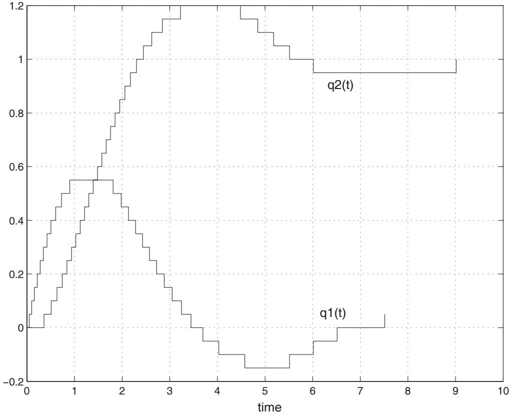


Let second order LTI system:

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

QSS – Example

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

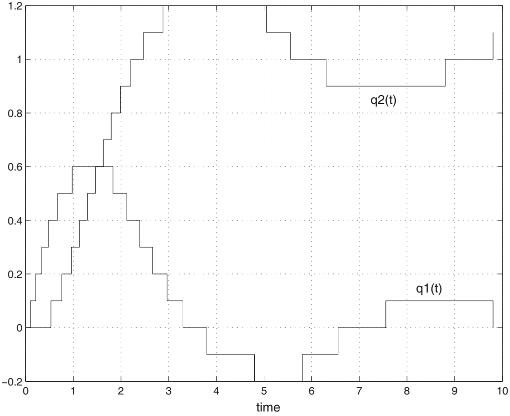


Let second order LTI system:

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

QSS – Example

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



Let second order LTI system:

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

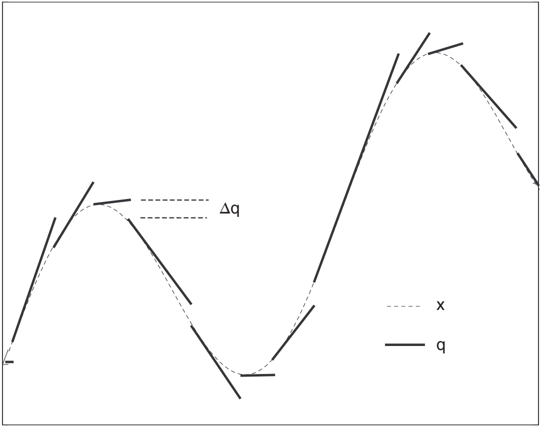
Cost vs. Accuracy in QSS

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.

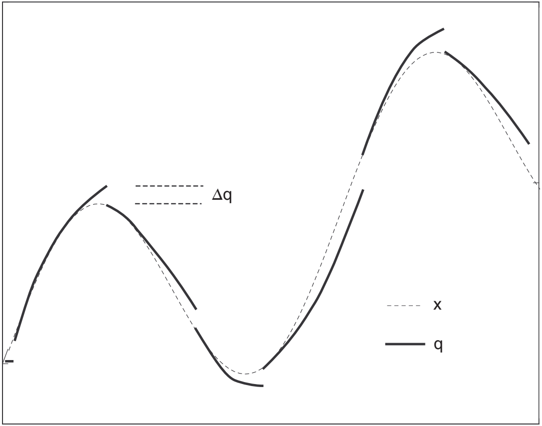
This problem is due to the fact that QSS is only first order accurate, i.e. it does not use information about the derivatives of f.

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.

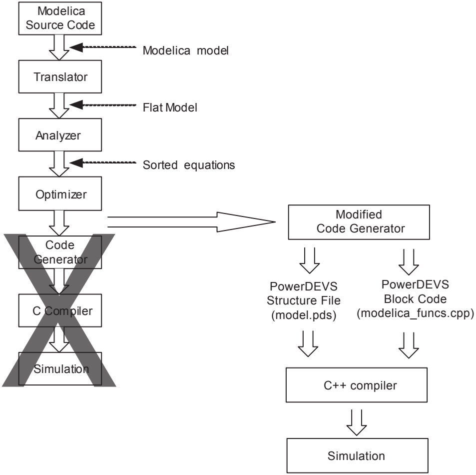
Third Order QSS (QSS3 Method)



- 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.

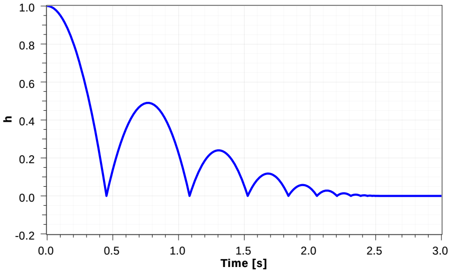
Outline

OpenModelica Compiler Modifications

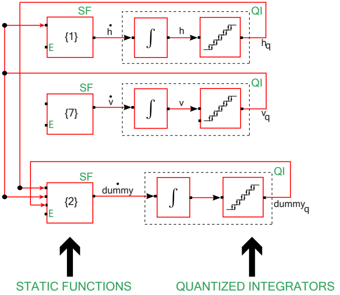


The Bouncing Ball Model

```
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;
```



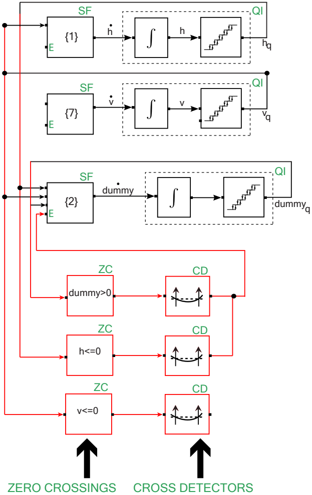
Add Static Blocks for State Variables




```
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)
```

- 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.

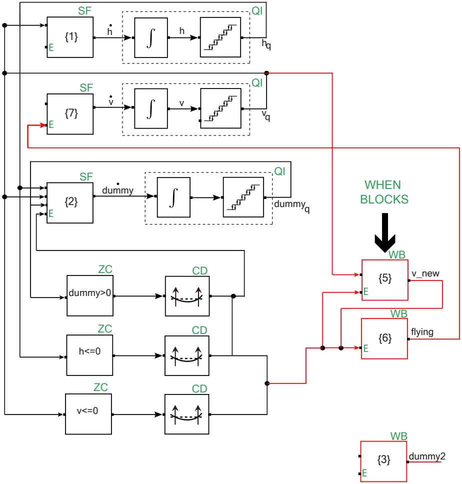
Add Zero Crossing Functions




```
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)
```

- 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.

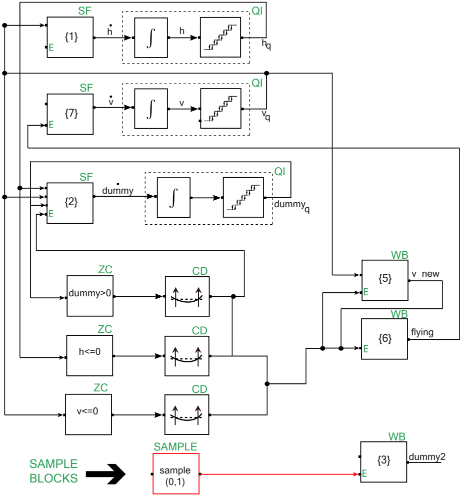
Add When Blocks




```
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} 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)
```

- 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().

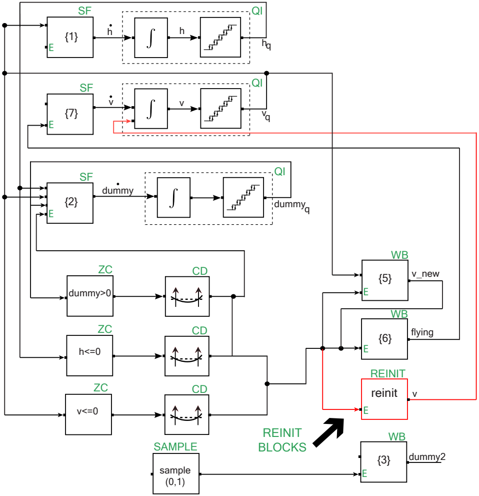
Add Sample Blocks




```
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)
```

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

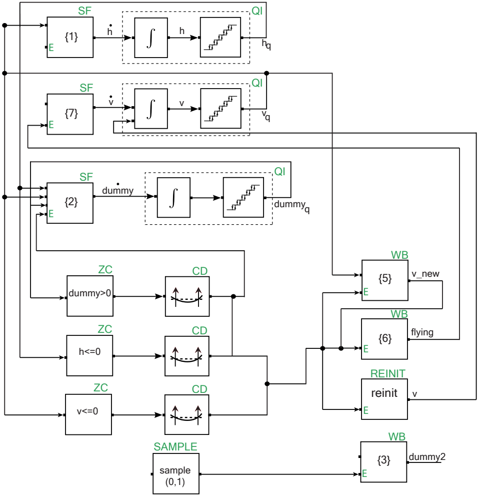
Add Reinit Blocks




```
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} 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)
```

- Add reinit blocks for the reinit statements and connect them to the corresponding integrators.

Final Structure




```
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

Compared Solvers

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

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.

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).

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).
- **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

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 $(\mathbf{t}^{\mathbf{ref}}, \mathbf{y}^{\mathbf{ref}})$.

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 $(\mathbf{t}^{\mathbf{ref}}, \mathbf{y}^{\mathbf{ref}})$.

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.

Simulation Accuracy

Simulation Accuracy

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

Simulation Accuracy

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

Simulation Accuracy

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

$$error = \frac{1}{|t^{ref}|} \sum_{i=1}^{|t^{ref}|} |y_i^{sim} - y_i^{ref}| \tag{6}$$

Half-Wave Rectifier

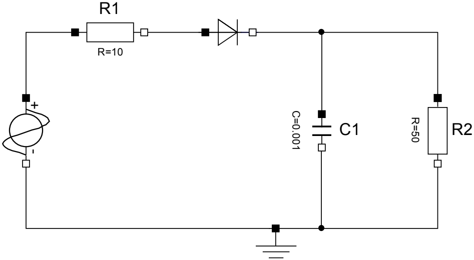
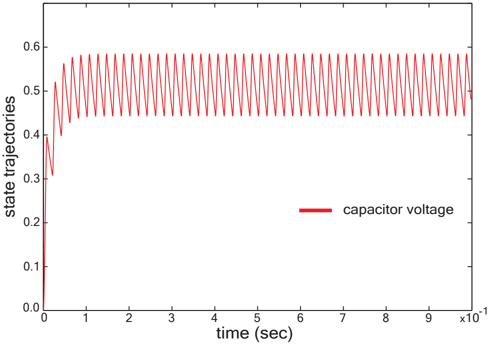


Figure: Graphical representation of the half-wave rectifier in Dymola

Simulated trajectories for the half-wave rectifier



Half-Wave Rectifier (Simulated for 1 sec)

			CPU time (sec)	Simulation Error
Dymola	**DASSL**	10^{-3}	0.019	1.45E-03
	DASSL	10^{-4}	*0.022*	*2.35E-04*
	Radau IIa	10^{-7}	0.031	2.20E-06
	Dopri45	10^{-4}	0.024	4.65E-05
PowerDEVS	**QSS3**	10^{-3}	**0.014**	**2.59E-04**
	QSS3	10^{-4}	0.026	2.23E-05
	QSS3	10^{-5}	0.041	2.30E-06
	QSS2	10^{-2}	0.242	3.02E-03
	QSS2	10^{-3}	0.891	3.04E-04
	QSS2	10^{-4}	3.063	3.00E-05
OpenModelica	**DASSL**	10^{-3}	0.265	3.80E-03
	DASSL	10^{-4}	*0.281*	*5.40E-04*

Switching Power Converter

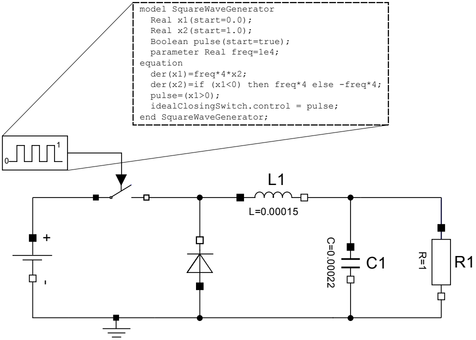
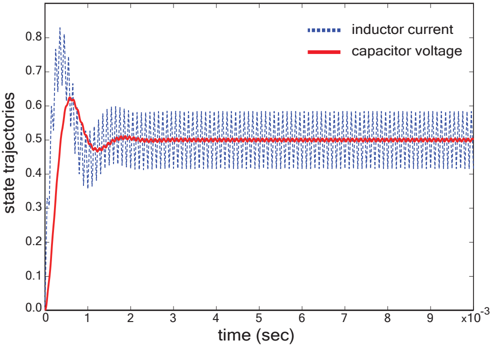


Figure: Graphical representation of the switching power converter in Dymola

Simulated state trajectories for the switching power converter



Switching Power Converter (Simulated for 0.01 sec)

			CPU time (sec)	Simulation Error
Dymola	**DASSL**	10^{-3}	0.051	1.82E-04
	DASSL	10^{-4}	*0.063*	*7.18E-05*
	Radau IIa	10^{-3}	0.064	1.11E-07
	Radau IIa	10^{-4}	0.062	1.11E-07
	Dopri45	10^{-3}	0.049	6.38E-06
	Dopri45	10^{-4}	0.047	9.76E-06
PowerDEVS	**QSS3**	10^{-3}	0.049	1.41E-03
	QSS3	10^{-4}	**0.062**	**1.68E-05**
	QSS3	10^{-5}	0.250	8.96E-06
OpenModelica	**DASSL**	10^{-3}	50.496	-
	DASSL	10^{-4}	*1.035*	*2.62E-02*

Outline

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?