A Solution to ARGESIM Benchmark C21 'State Events and Structural-dynamic Systems' based on Modelica Components

Jan-Philipp Disselkamp, Peter Junglas*, Alexander Niehüser, Phillip Schönfelder

Department of Engineering "Dr. Jürgen Ulderup", PHWT Vechta/Diepholz, Schlesierstr. 13a, 49356 Diepholz, Germany
*peter@peter-junglas.de

SNE 28(2), 2018, 39-48, DOI: 10.11128/sne.28.bn21.10411
Received: March 10, 2018 (Draft); Revised May 15, 2018 (Final)
Accepted: May 20, 2018
SNE - Simulation Notes Europe, ARGESIM Publisher Vienna,
ISSN Print 2305-9974, Online 2306-0271, www.sne-journal.org

Abstract. The ARGESIM C21 benchmark 'State Events and Structural-dynamic Systems' adresses difficulties that appear in the modelling and simulation of discrete systems with state and structure-changing events. The solution presented here uses Modelica and a component based approach. It shows that even though Modelica may have conceptual problems modelling such systems, it is capable to deal with all the tasks of the benchmark in a straightforward way.

Introduction

The ARGESIM C21 benchmark [1] deals with systems showing state events or even structural-dynamic behaviour. It requires to investigate three different examples: a bouncing ball, an RLC circuit with a diode and a rotating pendulum with a free flight phase. The solution shown in the following applies a component based modelling approach using the Modelica language [2] and its standard library MSL.

Since the benchmark is quite complex and consists of several subtasks, we concentrate here on the concrete tasks defined in the benchmark itself. The definition of the studied example systems in full detail can be found in [1]. Different approaches all based on Modelica components have been compared in [3], together with a discussion of underlying conceptions and encountered problems.

With Modelica one has a choice between several simulation programs. The results presented here have been obtained using MapleSim 2017-3 from Maplesoft under Kubuntu 16.04. Using Dymola from Dassault Systemes leads to identical results in most cases. Some implementation problems that showed up in one or both systems as well as minor numerical deviations are described in [3].

The models and scripts necessary to reproduce all results presented here are available from [4].

1 Case Study Bouncing Ball

The Bouncing Ball example is a model for a falling mass with or without air resistance that is reflected when hitting the ground. The reflection is either described as a simple timeless event or as a continuous process using a spring-damper model for the deformation of the ball.

1.1 Event contact model

Description of model implementation. The 'bouncing ball' model uses concepts and components of the `Mechanics.Translational` and `Blocks` parts of the MSL. One creates a component for each force acting on the falling mass, including a `Hardstop` component that is responsible for the bounce (cf. Figure 1). Except for the hardstop all components are standard or easily implemented and produce the continuous equations of the system.

For the implementation of the hardstop two different versions have been studied: A simple one, based on [5, p. 57], is defined by the following Modelica code:

```
model Hardstop1d
    parameter Real mu = 0.9;
    Position s;
```

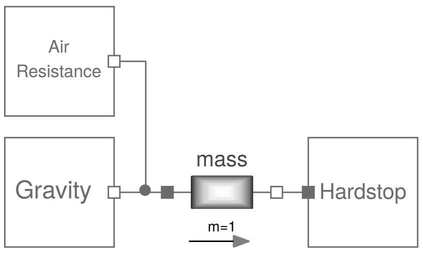

Figure 1: Bouncing ball with event contact

```
    Velocity v;
    Flange_a flange_a;
equation
    s = flange_a.s;
    v = der(s);
    flange_a.f = 0;
    when s <= 0 then
        reinit(v, -mu * pre(v));
    end when;
end Hardstop1d;
```

The `when` construction is the basic method in Modelica to create an event, the `reinit` restarts the solver with new initial values. This implementation leads to the notorious fall through problem near the Zenon point.

To cope with this one has to define another event `flying` (like in [2, pp. 96f]) and add a counter force at the hardstop

```
flying = not (s <= 0 and v <= 0);
flange_a.f = if flying then 0 else -m*g;
```

The resulting `Hardstop1dA` components works properly, the mass comes to rest after a large (but finite) number of bounces.

Simulation until last bounce – scattering prevention. With the given parameters for the free fall case one computes from [1, eq (16)] the bouncing time limit $t_{B,\infty} = 27.1290$ s. Using `Hardstop1dA` and standard solver parameters the simulation gives $t_{B,\infty} = 27.1287$ s. Adding air resistance results in $t_{B,\infty} = 25.5894$ s.

For the defective case of `Hardstop1d` the benchmark suggests adding a maximal height event and stopping the bouncing accordingly. This can be implemented in Modelica in the following way:

```
isFalling = v < 0;
isAtTop = edge(isFalling);
when isAtTop and s <= stopHeight then
  reinit(s, 0);
  reinit(v, 0);
  flange_a.f = -m*g;
elsewhen s <= 0 then
  reinit(v, -mu * pre(v));
  flange_a.f = 0;
end when;
```

The resulting limit time of course depends on the value of the parameter `stopHeight`. Interestingly, the simulation works even for the value `stopHeight` = 0 and reproduces the former result with air resistance, while in the free fall case the bounces stop earlier at $t_{B,\infty} = 27.1166$ s.

Testing accuracy of event handling. To determine the bounce times the `Hardstop1dB` component contains variables for the number and time of the last bounce that are updated at the bounce event. Figure 2 shows the difference between the theoretical values and the simulation results for a model without air resistance.

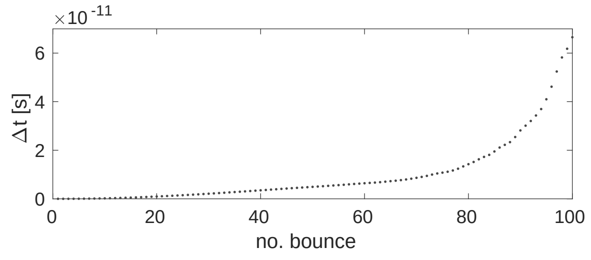

Figure 2: Accuracy of bounce times

Compensation of linear model deviation. This task asks to compensate for the later bouncing of the linear model (i. e. the one without air resistance) by introducing an initial velocity v_0. Doing so for the linear model one cannot reach identical final bounce times, because a simple calculation shows that it is bounded below by

$$t_{B,\infty}^{min} = \frac{2}{1-\mu}\sqrt{\frac{2x_0\mu}{g}} = 27.09\,\mathrm{s}$$

for the given values.

Therefore one has to introduce an initial velocity into the nonlinear model. To compute it, one can solve its ODE analytically (which is easily possible outside the bounces), use a small Matlab script to add up the bounces and compute the final bounce time as function of v_0. Finally an application of fzero gives the requested value

$$v_0 = 4.39563\,\mathrm{m/s}$$

Figure 3 shows the solutions of the original linear and the shifted nonlinear model.

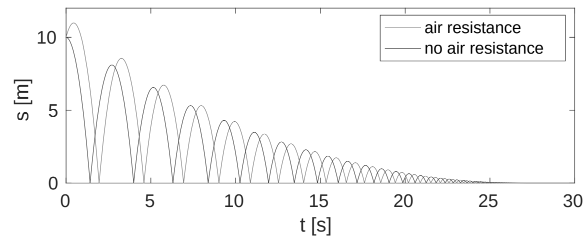

Figure 3: Compensation of linear deviation.

1.2 Model with continuous contact

Description of model implementation. The implementation of the bouncing ball model with continuous contact is very similar to the event based model, only the `Hardstop` component has been exchanged by an `ElastoGap` component (cf. Figure 4).

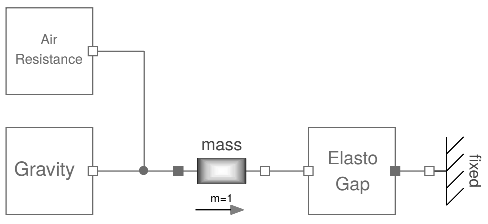

Figure 4: Bouncing Ball with continuous contact.

The MSL library already contains an `ElastoGap`, but it is more complicated in order to cope with unphysical situations. It is easy to adapt it to the benchmark requirements by using the following equations:

```
equation
  s_rel = flange_b.s - flange_a.s;
  v_rel = der(s_rel);
  y = s_rel + w;
  hasContact = (y <= 0);
  fc = if hasContact then
     -c*s_rel - d*v_rel else 0;
  flyRestarted = (fc <= 0);
  der(w) =
    if (hasContact and not flyRestarted)
      then -v_rel else -(c/d)*w;
  flange_a.f = fc;
  flange_b.f = -fc;
```

The events are defined implicitely through the `if` expressions and the corresponding logical variables. As always in Modelica the complete set of equations of the model is collected from all components and connections, then preprocessed and simplified. Only after these transformations the simulation program chooses appropriate state variables. This procedure makes it difficult to compare it with the general approaches defined in [1]. But since the total number of variables and equations is fixed in a Modelica model, one would probably describe it best as a 'maximal state space approach'.

Dependency of results from algorithms. MapleSim offers the choice of three variable-step solvers, which are all well known: the Runge-Kutta-Fehlberg solver RKF45, a Cash-Karp solver CK45 and a Rosenbrock solver ROS of third-fourth order. They have all been adapted by Maplesoft to cope with DAE systems. The model has been simulated with all three solvers and the following parameters: ε_{abs} = 1e-6, ε_{rel} = 1e-6, N_{plot} = 30001. Reference values with higher accuracy have been created using RKF45 and the parameters ε_{abs} = 1e-12, ε_{rel} = 1e-12. Using a different solver for the reference values leads to almost identical results. Creation of additional output points at events has been switched off to get output values at fixed times.

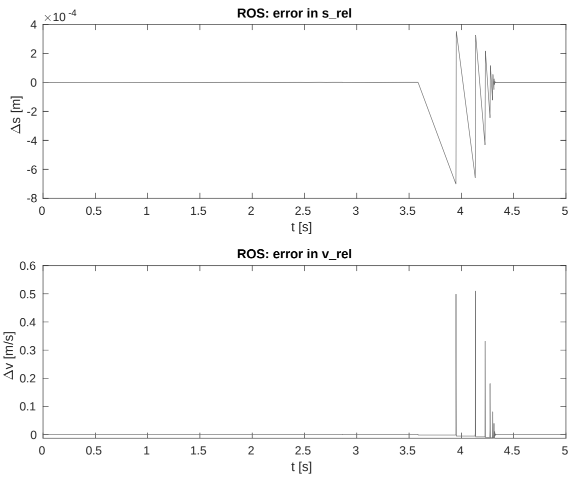

Figure 5: Errors for solver ROS.

All error plots are very similar, a typical result is shown in Figure 5. The much higher error of the velocity is due to its large slope together with deviations in the event times (cf. Figure 6).

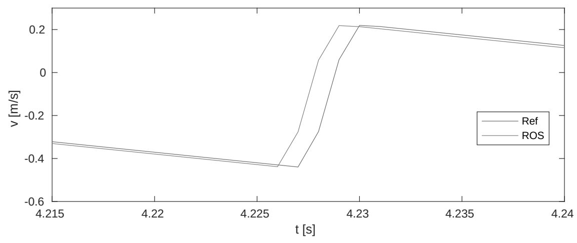

Figure 6: Velocity results near a bounce.

The maximal errors against the reference solution are given in Table 1. The standard solver CK45 is the worst here, one should use the ROS solver instead, which is generally recommended for stiff problems, and gives much better results than the other two. This is no surprise, since DAEs generally require stiff solvers.

	RKF45	CK45	ROS
s [1e-3 m]	3.7940	11.8984	0.7023
v [m/s]	2.3443	2.6007	0.5104

Table 1: Absolute errors for different solvers.

Investigation of contact phase. In order to output values of the maximal height h_{max} und maximal depression w_{max} the component `ElastoGapA` has been extended to create additional output events. For a closer look at the contact phase the model is simulated with higher accuracy (ε_{abs} = 1e-10, ε_{rel} = 1e-10, N_{plot} = 100000) using the Rosenbrock solver.

The results of the state and output variables (including the contact force) can be seen in Figure 7 and Figure 8 for the first and second contact phases and in Figure 9 for the second flight phase.

Table 2 shows the values for the maximal height and maximal depression. After the first 10 bounces the ball doesn't reach another flight phase, but oscillates while remaining in contact phase.

Parameter studies. Increasing k by a factor 100 leads to much more bounces since the contact time and the energy loss per bounce are small. Decreasing k by 100 leads to a sticking behaviour, the energy loss is too high for a second flight phase. A decrease or increase of d by a factor 10 leads to similar results for the same reasons (cf. Figure 10).

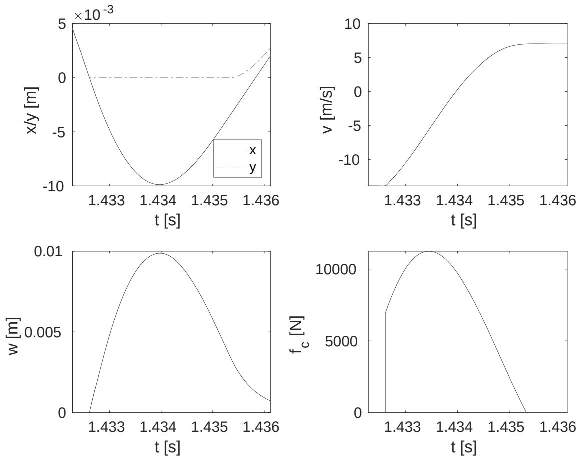

Figure 7: First contact phase.

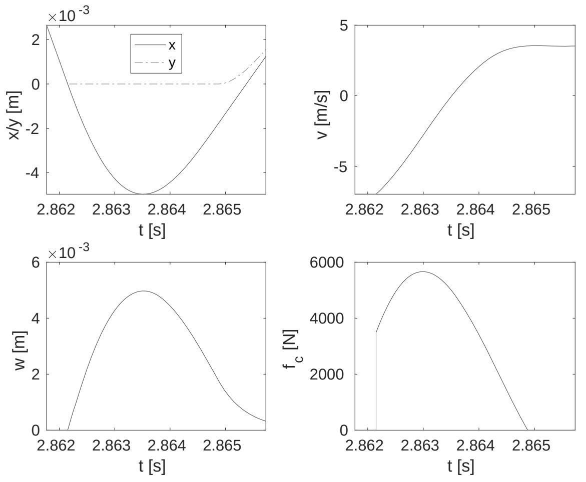

Figure 8: Second contact phase.

The results for changing d by a factor F and changing k by a factor $1/F^2$ are almost identical (cf. Figure 11). This can be explained easily by solving the simple contact equation [1, eq. 21] analytically, which shows that the percentage of energy loss per bounce depends on k/d^2 [6].

Bouncing Ball on Mars. Due to the lower gravity the bouncing ball behaviour on Mars is stretched in time, but otherwise similar (cf. Figure 12). The small differences are due to the different air resistances.

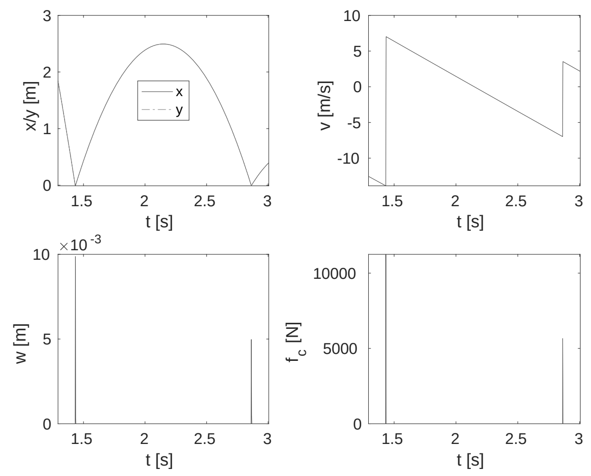

Figure 9: Second flight phase.

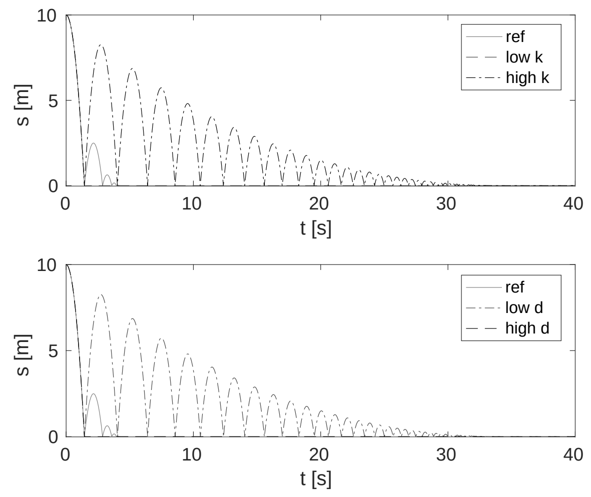

Figure 10: Variation of k and d.

n	h_{max} [m]	w_{max} [mm]
1	10.00000000000	9.87387827
2	2.49466100656	4.97183608
3	0.63276017545	2.51178619
4	0.16058207486	1.26910129
5	0.04046397059	0.64031534
6	0.01002858933	0.32201325
7	0.00239917584	0.16085257
8	0.00053012002	0.07922586
9	0.00009490838	0.03783789
10	0.00000588204	0.01701446
11	-0.00000660874	0.01123246
12	-0.00000917794	0.01009085
13	-0.00000968520	0.00986545
14	-0.00000978536	0.00982095
15	-0.00000980527	0.00981176

Table 2: Maximal heights and depressions.

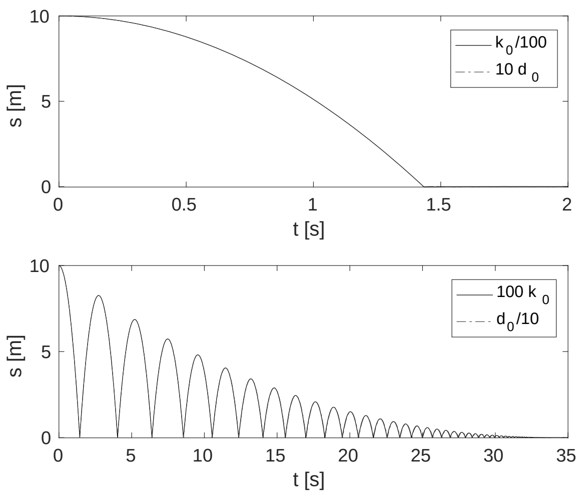

Figure 11: Comparison of k and d changes.

2 Case Study RLC Circuit with Diode

The second example is a simple RLC circuit with a diode, where different diode models are to be investigated. It is constructed easily with standard components from the `Electrical.Analog` part of the Modelica Standard Library (MSL) (cf. Figure 13), which even contains two simple diode components. This model is used throughout this section, only the implementation of the diode component is changed.

Description of model implementations. To simplify the construction of electrical components, the MSL contains a partial model `OnePort` that defines external connection points and the internal variables i und v. One creates a concrete model by inheriting from `OnePort` and adding the equation that defines the connection between i and v. This mechanism will be used in the following for all diode models.

The shortcut diode can be implemented using the `IdealDiode` from MSL. A simpler version can be

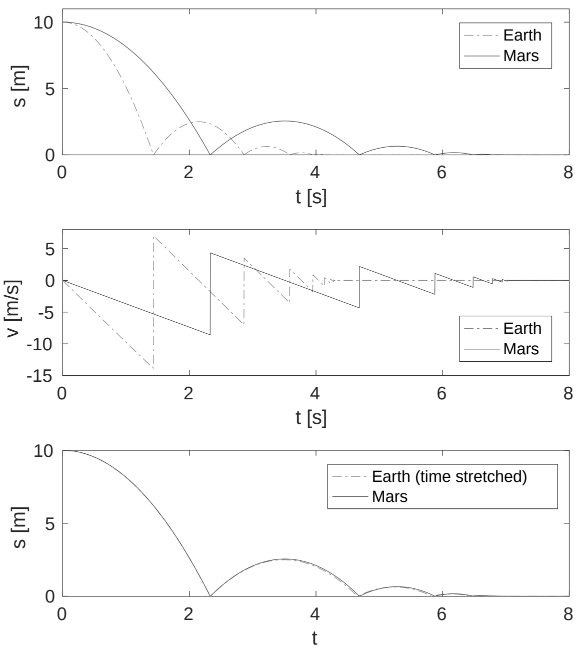

Figure 12: Bouncing ball on Earth and on Mars.

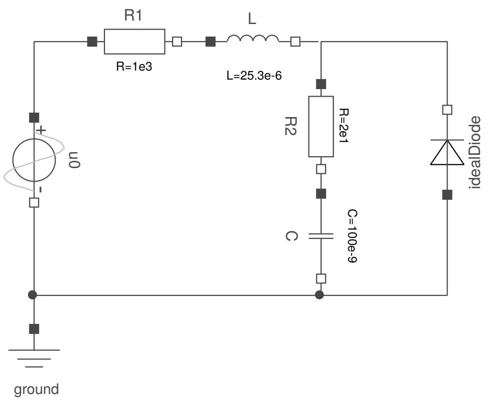

Figure 13: RLC circuit with diode.

found in [5, p.56], which uses a standard trick to cope with the non-functional relation between *i* and *v*:

```
model DIsc1 "short-cut diode"
  extends OnePort;
  Real s;
  Boolean off;
equation
  off = s < 0;
  v = if off then s else 0;
  i = if off then 0 else s;
end DIsc1;
```

The Shockley diode `DIshu1` is even simpler, it defines parameters I_S and U_T and the equation

```
i = if v < 0 then 0 else IS*(exp(v/UT)-1);
```

For the implementation of the approximated Shockley diode one writes a simple linear interpolation function and has

```
model DIas1 "approximated Shockley diode"
  extends OnePort;
  parameter Real IS = 1e-8;
  parameter Real UT = 26e-3;
  parameter Integer N = 10;
  parameter Voltage uMax = 3.84e-2;
  Real up[N] = linspace(0, uMax, N);
  Real ip[N] = IS*(exp(up/UT) - ones(N));
  Boolean off;
equation
  off = v < 0;
  i = if off then 0
           else linInterp(v, up, ip);
end DIas1;
```

The final model is the 'explicit Shockley diode', which is defined by differentiating the algebraic equation of the complete RLC model. In the context of a component based environment used here, one can only differentiate the *i*-*v* relation to get the explicit component `DIesu1` defined by the equations

```
  off = v < 0;
  if off then
    i = 0;
  else
    der(i) = (Is/UT)*exp(v/UT)*der(v);
  end if;
```

The events are again defined implicitely by the if-expressions. According to the terminology of [1], one may call this a 'switching model parts' approach. The only noteworthy detail is in the explicit diode, where the variable *i* is a state variable (differentiated) in one branch, and a simple algebraic variable in the other. Such a situation often presents problems for the simulation environment, but MapleSim works nicely here. The well-known Dymola program can't cope with this component and stops the simulation, when the first locking phase appears, claiming to hit upon a singular linear system of equations. The obvious workaround – substitute `i = 0` by `der(i) = 0` – makes *i* a state variable always and saves the day of the Dymola user.

Dependency of results from algorithms. The general procedure (using reference values of high accuracy) and the general solver parameters are the same as in Section 1.2.

For the interesting variables i_L, u_C, i_D and u_D relative errors have been computed by comparing to the reference solution and scaling by maximal absolute values of the variable. The results for the different solvers are displayed in Table 3 for the shortcut diode and in Table 4 for the Shockley diode.

	RKF45	CK45	ROS
ε_{iL}	425.90	440.30	305.94
ε_{uC}	4.65	3.48	4.13
ε_{iD}	549.64	328.32	243.72
ε_{uD}	254.71	163.66	19.29

Table 3: Shortcut diode: Relative errors [in 1e-6].

	RKF45	CK45	ROS
ε_{iL}	521.08	494.31	42.67
ε_{uC}	5.59	5.97	5.82
ε_{iD}	215.95	199.58	18.82
ε_{uD}	283.15	302.26	27.23

Table 4: Shockley diode: Relative errors [in 1e-6].

Again the Rosenbrock solver gives the highest accuracy, but in the shortcut model the difference to the other solvers is only marginal (except for u_D), whereas it is an order of magnitude in the Shockley model.

The behaviour of the errors over time is similar for all variables and both models. An example for the shortcut diode and variable i_D is shown in Figure 14.

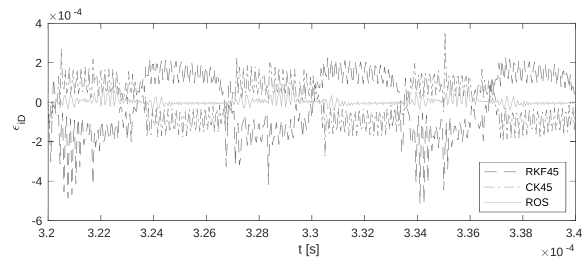

Figure 14: Shortcut diode: Relative errors for i_D.

Comparison of shortcut and Shockley diode model. Figure 15 shows the behaviour of the relevant variables for models with shortcut resp. Shockley diode over two switching periods starting at 0.3 ms to get rid of initial effects. The very small values of the diode current i_D for the Shockley diode are due to its rather large value of U_T leading to a high resistance in conducting phase.

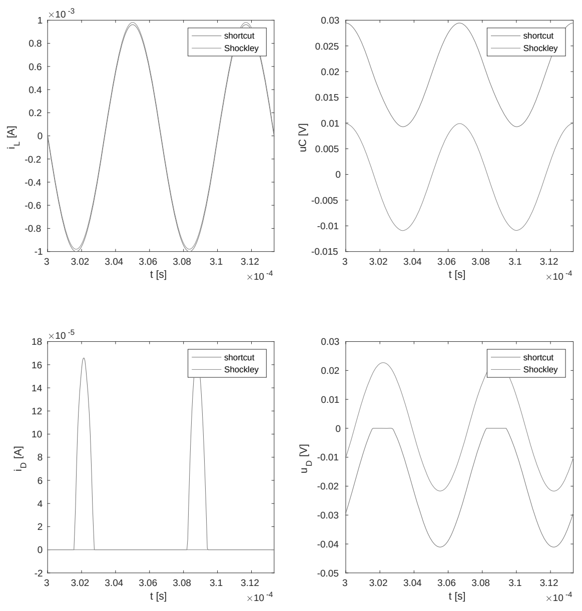

Figure 15: Comparison of shortcut and Shockley diode.

Simulation times have been obtained by performing seven runs each and computing mean values of the last five, thereby minimizing initial loading time effects. MapleSim outputs timing values for different stages of the computation, which shows that the largest part here is not the integration itself, but a task described as ‘preparing for integration’. This part is done much faster for the Shockley model than for the shortcut model, reducing the total computation time by 44%.

Approximation of Shockley diode model. The model using the approximated Shockley diode with different numbers of interpolation points almost reproduces the results of the Shockley diode, the only notable difference being the diode current i_D (cf. Figure 16). The corresponding plot of the absolute errors nicely shows the approximation points. Relative errors for the other variables are small, a typical behaviour is displayed in the lower plot.

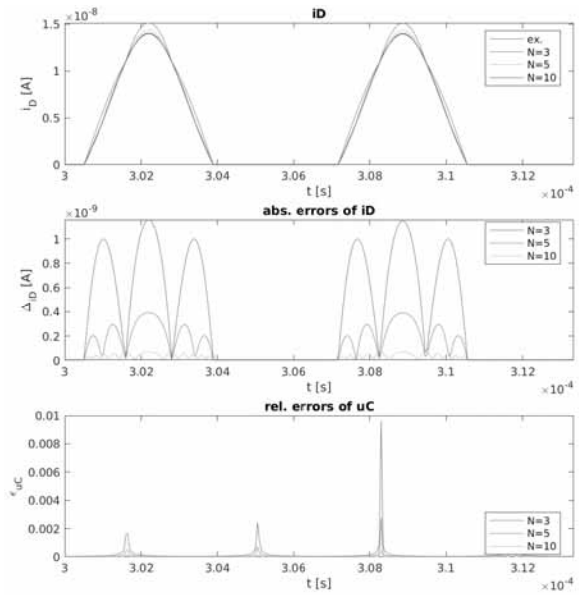

Figure 16: Comparison of Shockley and approx. Shockley diode.

Relevance of choice of algebraic state. It's easy to use the inverse relation $v(i)$ for the Shockley diode writing

```
if v < 0 then
  i = 0;
else
  v = UT * log(i / IS + 1);
end if;
```

This leads to identical simulation results. But interestingly, the computing times are different drastically: The new variant is 25 times slower in MapleSim. In Dymola the simulation times do not differ at all.

Investigation for real-time simulation. MapleSim provides a few fixed-step solvers, in the following the RK4 solver is used with a step size of 1e-8. Furthermore the constraint projection has been switched off and the number of event iterations set to 1. Compared to the standard solver parameters this leads to identical results for the shortcut and shockley diodes, whereas the model using the "explicit Shockley" diode `DIesu1` differs in the diode current i_D: It drifts to negative values during the locking phases (cf. Figure 17).

As described above the equation used inside the diode component is

```
der(i) = (Is/UT)*exp(v/UT)*der(v);
```

The alternative component `DIesi1` uses the derivative of the inverse relation $v(i)$ and leads to slightly different deviations. Figure 17 compares both explicit versions with the correct Shockley diode.

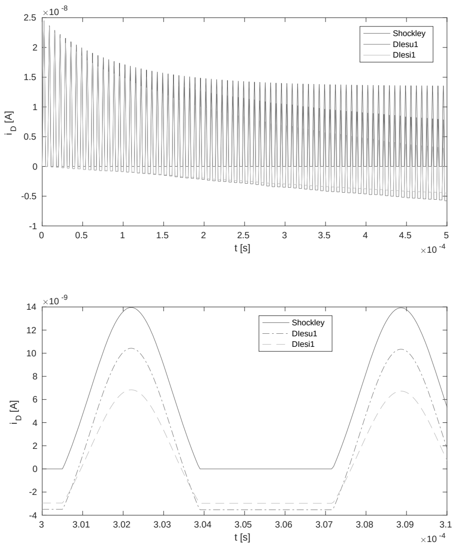

Figure 17: Comparison of Shockley and explicit Shockley diodes.

3 Case Study Rotating Pendulum With Free Flight Phase

The last example is a point mass with air resistance on a rope of fixed length. Its movement switches between swinging and free fall phases according to the direction of the force acting on the mass.

Description of model implementations. The implementation of the rotating pendulum consists of separate blocks for the two different system configurations and a `SystemSwitch` that alternatively activates one or the other system, depending on the state of the active system (cf. Figure 18). The block below computes some additional plot variables.

The switch contains the event functions h^F, h^S as defined in [1] and computes the initial state at a system change:

```
h1 = -g*m*cos(state1[1])+m*l*state1[2]^2;
h2 = l^2 - state2[1]^2 - state2[2]^2;
when h1 < 0 then
  active1 = false;
```

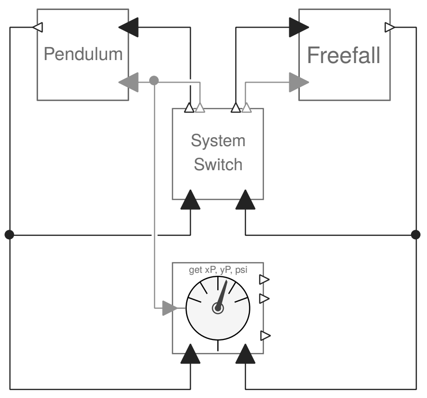

Figure 18: Rotating pendulum model.

```
elsewhen h2 < 0 then
  active1 = true;
end when;
active2 = not active1;

new1[1] = atan2(state2[1],state2[2]);
new1[2] = (state2[2]*state2[3]
           - state2[1]*state2[4])/l^2;
new2[1] = l*sin(state1[1]);
new2[2] = l*cos(state1[1]);
new2[3] = l*state1[2]*cos(state1[1]);
new2[4] = -l*state1[2]*sin(state1[1]);
```

To facilitate the implementation of the two systems a partial model `SwitchableSystem` has been defined that contains the state, the inputs and outputs and the triggering:

```
partial model SwitchableSystem
  parameter Integer N = 2
  parameter Real[N] s0 = {pi/4, 15};
  RealInput[N] newState;
  BooleanInput active;
  RealOutput[N] sOut;
  Real[N] state(start=s0, each fixed=true,
      each stateSelect=StateSelect.always);
equation
  when active then
    reinit(state, pre(newState));
  end when;
  sOut = if active then state else zeros(N);
end SwitchableSystem;
```

Using this the implementation of a concrete system only needs the definition of the state equations, usually in form of an ODE. Even a graphical approach is possible [3], but MapleSim cannot cope with such a model.

Finally one adds a few lines of explicit Modelica code to identify the (inherited) `state` variable with corresponding variables from the concrete model. For the pendulum this is as simple as

```
state[1] = pi/2 - revolute.phi;
state[2] = -revolute.w;
```

The attribute `stateSelect=StateSelect.always` of `state` guarantees that these variables will be used by the solver as the actual states.

This model looks exactly like a hybrid decomposition (cf. Figure 13 of [1]) and for the purpose of constructing the model it really is one: Both submodels can be created independently and almost in the same way as standalone systems. But formally this again is a maximal state space approach: The variables of an inactive system are simply ignored or their derivatives set to zero. In any case they always exist, enlarge the total state space and have to be computed always albeit trivially. On the other hand Modelica compilers routinely handle large systems with lots of trivial equations, so this should not be a large burden.

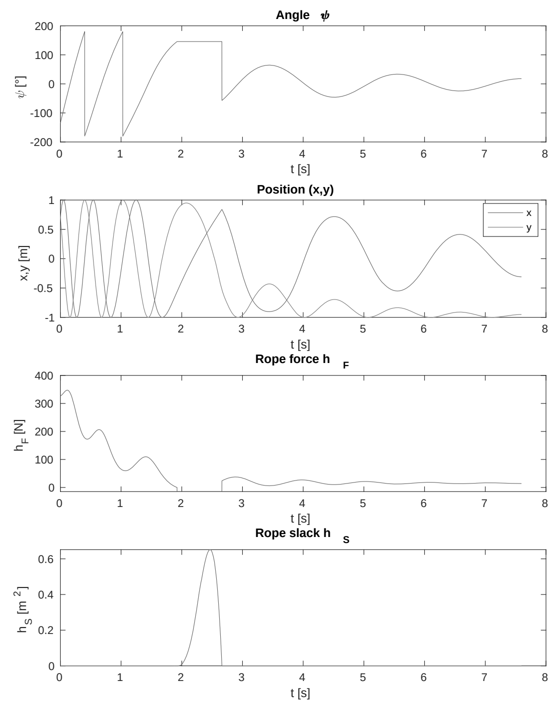

Figure 19: Results of pendulum model.

Basic simulation of phases. Using a simple `when` construction the basic model can be easily extended to stop after the amplitude is below $\pi/10$. This happens at t = 7.5962714 s, the corresponding values of state and event variables are displayed in Figure 19. The angle ψ shown there is measured against the lower equilibrium point and reduced to the interval $[-\pi, \pi]$

Dependency of results from algorithms. The procedure for comparing results that has been used twice before is employed again. The maximal absolute errors for the variables x, y and ψ are given in Table 5. They show that the Rosenbrock solver again has the highest accuracy, while the default solver CK45 performs worst. Figure 20 displays exemplary plots of the error over time.

	RKF45	CK45	ROS
x [10^{-6} m]	0.4804	0.8476	0.1080
y [10^{-6} m]	0.2597	0.5187	0.1941
ψ[10^{-6} rad]	0.4879	0.8801	0.2020

Table 5: Absolute errors (compared to reference solution).

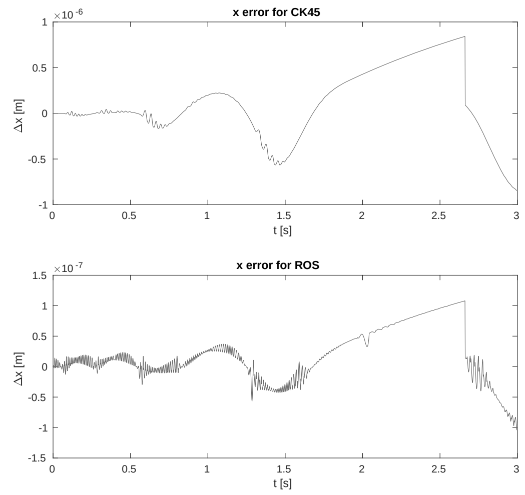

Figure 20: Absolute errors in x for two solvers.

External energy supply. To add the events necessary for the addition of the 'kick' – i. e. a jump of the angular velocity by a factor γ at the lowest point – one needs a `when-elsewhen` construction. This did not work due to a bug in MapleSim, but a simple workaround could be found [3].

The kick value γ is defined by the initial conditions $\psi_0 = 0$ and (unknown) ω_0 and given end conditions ψ_f and ω_f. To find it a simple model `RPkiAux1` has been used that integrates the time inverted pendulum ODE until the (initial) point $\psi = 0$ is reached. γ is then given by the ratio of ω_0 and the known value of ω before the kick. Table 6 shows the final values defining the three cases in [1] and the corresponding values of γ.

The definition of case (iii) "the swinging phase makes two rotations" is ambiguous, it could mean anything between 1.5 to 2.5 rotations before a fall from the top. The given value corresponds to the shortest path.

case	ψ_f	ω_f	γ
(i)	$-(5/4)\,\pi$	15	-21.9911
(ii)	$-\pi$	$\sqrt{g/l}$	-10.1501
(iii)	-3π	$\sqrt{g/l}$	-14.1982

Table 6: Final values and kick factor.

References

[1] Körner A, Breitenecker F. State Events and Structural-dynamic Systems: Definition of ARGESIM Benchmark C21. *Simulation Notes Europe*. 2016; 26(2): 117–122. doi: 10.11128/sne.26.bn21.10339.

[2] Modelica Association. *Modelica® - A Unified Object-Oriented Language for Systems Modeling - Language Specification Version 3.4, April 10, 2017.* Online: `https://modelica.org/documents/ModelicaSpec34.pdf` (called 2018-05-23).

[3] Disselkamp JP, Junglas P, Niehüser A, Schönfelder P. Implementing the Argesim C21 benchmark with Modelica components. In Loose T, editor. *Tagungsband Workshop 2018 ASIM/GI-Fachgruppen*; 2018 Mar; Heilbronn. 197–202.

[4] Junglas P. Argesim C21 models and scripts. Online: `http://www.peter-junglas.de/fh/simulation/argesimc21.html` (called 2018-05-23).

[5] Fritzson PA. Principles of Object-Oriented Modeling and Simulation with Modelica 3.3. Wiley & Sons, New York, 2015.

[6] Nagurka M, Huang S. A mass-spring-damper model of a bouncing ball. In *Proc. American Control Conference*; 2004 Jul; Boston. 499–504.