Solving ARGESIM Benchmark C21 'State Events and Structural-dynamic Systems' with Simulink

Rebecca Heitmann, Peter Junglas*, Lisa-Kristin Petrucha

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

SNE 31(1), 2021, 33-42, DOI: 10.11128/sne.31.bn21.10556
Received: July 14, 2020; Revised: October 10, 2020;
Accepted: October 20, 2020
SNE - Simulation Notes Europe, ARGESIM Publisher Vienna
ISSN Print 2305-9974, Online 2306-0271, www.sne-journal.org

Abstract. To pinpoint the problems that come with the modeling and simulation of hybrid systems, the ARGESIM C21 benchmark 'State Events and Structural-dynamic Systems' describes three such systems and a lot of corresponding tasks. It is solved here using the well-known simulation environment Simulink, the solutions are based on a direct modeling of the ODEs or DAEs describing the systems. To this end, special schemes have been necessary sometimes, some of which are already provided by Simulink, others had to be modeled explicitely.

Introduction

The Argesim C21 benchmark [1] requires to study three different examples of hybrid systems: a bouncing ball, an RLC circuit with a diode and a rotating pendulum with a free flight phase. A complete solution has been published before that is based on Modelica components [2]. The results presented in the following use the well-known Simulink simulation environment from Mathworks [3] without relying on additional packages for discrete system modeling such as Stateflow [4] or SimEvents [5]. Unlike in [2] modeling doesn't start with the physical components, but from the differential equations that are used to describe the systems.

The complete definition of the example systems and the tasks can be found in the benchmark definition [1]. For conciseness we will not reproduce results that are identical to those in [2], but simply quote the corresponding plots and tables. Instead we will concentrate mainly on the different implementation methods.

All models and scripts necessary to reproduce the results presented here are available from [6]. They have been prepared using Simulink Version 10.1 (R2020a) under Kubuntu 18.04.

1 Case Study Bouncing Ball

The first example describes a mass falling under gravity and air drag, which bounces off the ground. The bouncing process is described either as a simple event or as a continuous process, using a basic material model.

1.1 Event Contact Model

Description of model implementation. The bouncing ball model (without air drag) and the difficulties it poses for zero-crossing detection, have been widely studied by Mathworks [7]. In fact it is used as an example in Simulink that is discussed at length in the documentation. It is implemented there in two different ways. The first one with two separate integrators (and parameters adapted to the benchmark) is shown in Figure 1. It uses a lower saturation limit of the x integrator and external resets of x and v integrators to create the bounce event.

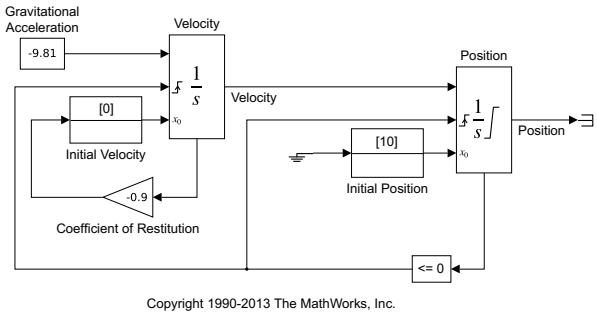

Figure 1: Bouncing ball with event contact and two integrators ([8], parameters adapted).

With standard solver parameters it stops with an error ("too many consecutive zero crossing events"), when approaching the Zenon point. To overcome this problem one can use the optional adaptive algorithm for zero-crossing. It stops bracketing the event after too many events in a short time or if the function variation becomes too small. This works in principle, but the velocity results show a lot of remaining chatter. Changing the relative tolerance of the solver from 1e-3 to 1e-6 and manually adapting the signal threshold of the zero-crossing algorithm, the chatter amplitude is reduced by a factor of 20.

To get rid of the chatter the Simulink library contains a second-order integrator, which is mainly the combination of two connected integrators. The external reset of v, when x reaches 0, is done internally, and a consistent behaviour of x and v at saturation is enforced. With this block the benchmark model including air drag becomes very simple (cf. Figure 2). It uses the standard zero-crossing algorithm, but works fine nevertheless: x and v are exactly zero after the Zenon point.

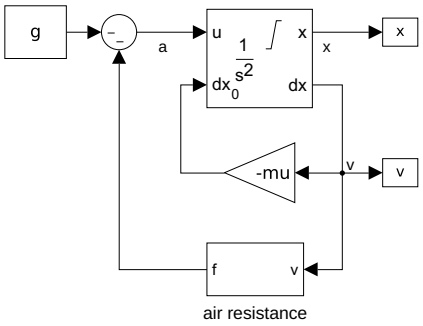

Figure 2: Bouncing ball with event contact and second-order integrator.

Simulation until last bounce – scattering prevention. The Zenon point in the free fall case (without air drag) is easily computed from [1, eq (16)] giving $t_{B,\infty} = 27.129019$ s. A workaround to prevent event scattering is not necessary, the standard model shown in Figure 2 just works fine. It uses the ode23 solver and a relative and absolute tolerance of 10^{-6}. A simple Matlab script that finds the start time of the final zero values of x gives the results $t_{B,\infty} = 27.129019$ s for the free fall case and $t_{B,\infty} = 25.589465$ s when adding air resistance.

Testing accuracy of event handling. To determine the bounce times the standard model exports the simulated values of x and a Matlab script extracts the times where x equals 0.0. This is possible easily in spite of the usual floating point problems, since the saturation event enforces the exact value of 0. Figure 3 shows the difference between the theoretical values and the simulation results for a model without air resistance.

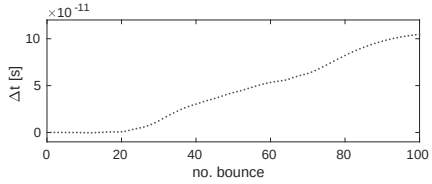

Figure 3: Accuracy of bounce times.

Compensation of linear model deviation. This task asks to introduce an initial velocity v_0 to the linear model so that the same Zenon point is reached as in the nonlinear model. As has been shown in [2] this is not possible, but one has to a add a velocity to the nonlinear model instead. Its value has been computed there as $v_0 = 4.39563\,\mathrm{m/s}$. Using this value in the standard model, the simulation results are the same as in [2, Fig. 3].

1.2 Model with Continuous Contact

Description of model implementation. The bouncing ball model with continuous contact is implemented by directly reproducing the differential equations, where the changing of the right hand sides according to the phase is done with `Switch` components. The complete model is shown in Figure 4, where the equation of the ball deformation is hidden inside a subsystem (cf. Figure 5).

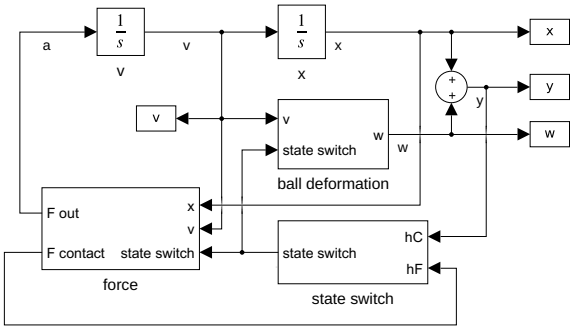

Figure 4: Bouncing ball with continuous contact.

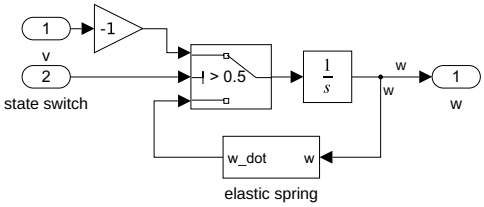

Figure 5: Ball deformation.

The interesting part is the state switch (cf. Figure 6), which toggles between 0 and 1, when the active event function – h_F or h_C according to phase – becomes negative. The actual toggle switch is implemented in a typical way as a triggered subsystem containing a `Unit Delay` with a feedback loop.

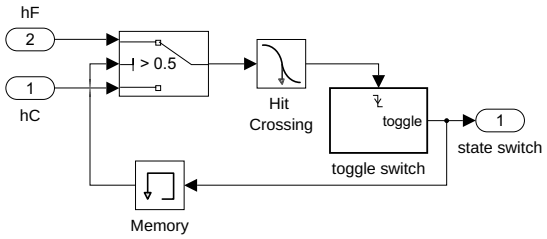

Figure 6: State switch.

The additional `Memory` component is necessary to break an algebraic loop, without it the simulation produces the error "Ambiguous sorted order". Usually one tries do avoid such a block, because it introduces an additional delay. Here it does no harm, since a few solver steps will always occur, until the phase changes again. Quite odd is the appearance of a `Hit Crossing` component directly before the trigger input: One would expect that the toggle switch could be triggered directly by the event function (i.e. the output of the `Switch`). But without the help from the `Hit Crossing` block, the corresponding events are missed. Either the authors are lacking the necessary understanding of the inner workings of Simulink here – or it is simply a bug!

Since all three state variables are used throughout and only the forces change according to the phase, one could call this implementation a "switching model parts" approach.

Dependency of results from algorithms. Simulink offers seven adaptive ODE solvers for initial value problems, among them the Dormand-Prince solver `ode45` and the NDF-based `ode15s`, which is recommended as standard solver for stiff problems. All solvers are well-known and extensively documented [9].

To compare them, the standard model has been simulated with output values at fixed steps of 1e-3 s using ε_{abs} = 1e-12, ε_{rel} = 1e-12 for a reference solution with `ode45` and ε_{abs} = 1e-6, ε_{rel} = 1e-6 for the actual comparison. Using other solvers for the reference solution leads to almost identical results. A typical error plot is shown in Figure 7. The large spikes for the velocity errors are due to small timing differences of the bouncing phases.

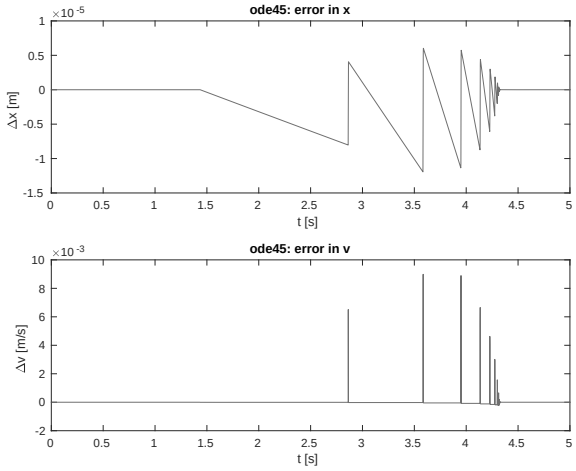

Figure 7: Errors for solver `ode45`.

The maximal errors against the reference solution are given in Table 1. The best one by far is the standard solver `ode45`, the next best `ode23` being far off with a tenfold error. All other solvers, among them all stiff solvers, are much worse with errors 60 to 80 times as large.

	s [mm]	v [m/s]
ode45	0.0119	0.0090
ode23	0.1146	0.0858
ode113	0.7299	0.5678
ode15s	0.6518	0.4722
ode23s	0.8247	0.6440
ode23t	0.8210	0.6410
ode23tb	0.6442	0.4999

Table 1: Absolute errors for different solvers.

The effects of changing two very special solver parameters have been examined, using the `ode45` solver and normal tolerances: Switching on the *Shape preservation* leads to better use of derivative information, which – according to the documentation – should increase accuracy for models with strongly changing derivatives. Looking at the velocities this is the case here, and the errors decrease by 25% accordingly. The other parameter is called `MinimalZcImpactIntegration` and should "reduce the impact of zero-crossing on the integration of continuous states" [10]. Though this sounds promising, it has no effects for our model.

Investigation of contact phase. For this task an additional `HitCrossing` block has been added to the standard model that outputs a signal whenever the velocity becomes zero (in either direction). This allows to extract the maximal and minimal heights easily. To get plots of the state and output variables during the three phases the high accuracy of $\varepsilon_{abs} = \varepsilon_{rel} =$ 1e-10 (as in the corresponding task of [2]) has been used. Figure 8 shows the first contact phase, it and the plots of the other two phases are identical to [2, Fig.7 - Fig.9].

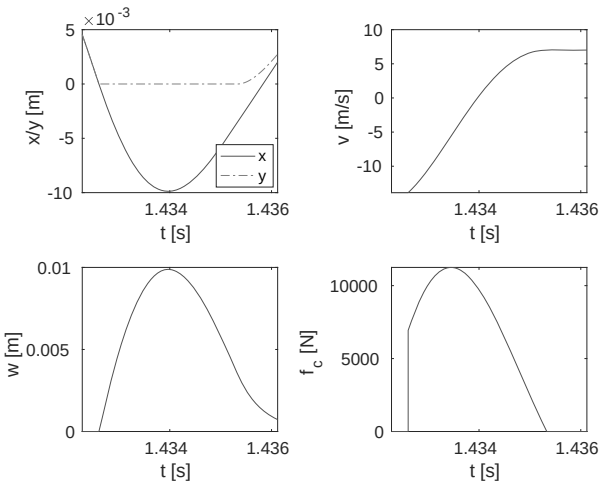

Figure 8: First contact phase.

The first ten values for the maximal height and maximal depression are shown in Table 2. Additional 21 values describe vibrations during the final contact phase.

In spite of the high accuracy the values given here differ from those in [2] by up to 0.05%. To find out, which results are better, additional values have been computed with an even higher accuracy of 1e-12 in Simulink and Maplesim. While the relative error between both Simulink results is only 1e-6, the Maplesim results differ by 1e-4. Furthermore the difference between the higher accuracy results in Simulink and Maplesim stays the same as before. This suggests that the Simulink results have the higher accuracy. This is confirmed by the general behaviour of the solvers that are used here: While `ode45` is known to be generally quite accurate, the Maplesim computations have been done with one of the stiff solvers, which are generally less accurate.

n	h_{max} **[m]**	w_{max} **[mm]**
1	10.00000000000	9.87399975
2	2.49475864884	4.97196356
3	0.63279737764	2.51186760
4	0.16059402954	1.26915027
5	0.04046743537	0.64034298
6	0.01002956662	0.32202875
7	0.00239944058	0.16086111
8	0.00053018975	0.07923061
9	0.00009492532	0.03784056
10	0.00000588511	0.01701582

Table 2: Maximal heights and depressions.

Parameter studies. Changing the values of k or d in the standard model reproduces the results from [2, Fig.10, Fig.11]. The only interesting observation here is the behaviour of the stiff version (k = 1e8, d = 500): With the standard solver `ode45` and accuracy 1e-6 the simulation produces a fall-through behaviour. Decreasing the tolerance to 1e-8, the model works fine. Of course, for this model one would prefer a stiff solver anyhow in order to decrease the computation time.

Bouncing ball on Mars. The standard model reproduces the results from [2, Fig.12].

2 Case Study RLC Circuit with Diode

The second test case of the benchmark is an RLC circuit containing a diode, where several models for the diode

have to be studied: a simple shortcut model, the well-known Shockley model and an approximation thereof.

Description of model implementations. The model of the shortcut diode implements the differential equations directly. It computes the event function and uses the Simulink `Switch` component to change between the locking and the conducting phase. Therefore it is a classical "switching model parts" approach (cf. Figure 9).

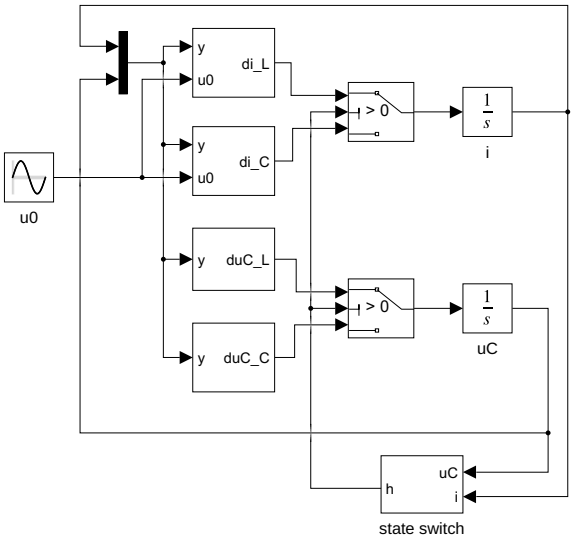

Figure 9: Short cut diode.

To model the Shockley diode one can eliminate the variable i_D from [1, eq. (28)] using the defining relation [1, eq. (34)] to get a semi-explicit DAE system of index 1 for the state variables u_C and i and the algebraic variable u_D. This can be modeled in Simulink in a standard way [11] by using algebraic loops to solve the constraint equation $g(u_C, i, u_D) = 0$ (cf. Figure 10). The discontinuous constraint can be written with the Heaviside step function and easily implemented with Simulink's standard `Signum` component (cf. Figure 11).

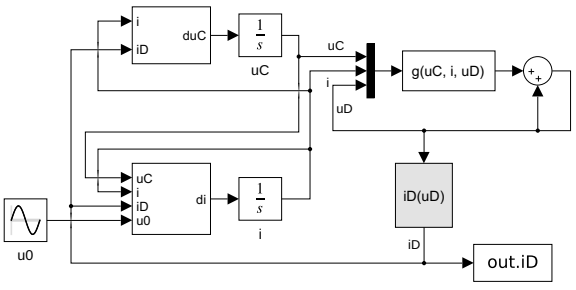

Figure 10: Shockley diode with implicit computation of uD.

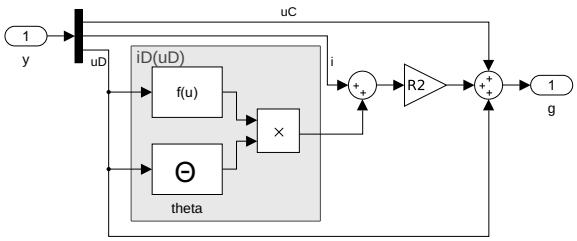

Figure 11: Algebraic constraint of Shockley diode.

So instead of the "switching model parts" approach used before, this is just an ordinary DAE system with a discontinuous constraint equation. It works fine in spite of a warning ("Discontinuities detected within algebraic loop(s), may have trouble solving") and produces the expected results. But: If one deletes the output block `out.iD`, the warning becomes an error ("2 zero crossing signal(s) identified below caused 1000 consecutive zero crossing events..."). Though this can be cured by changing the zero-crossing control algorithm to "adaptive", for the user it is impossible to understand what is going on in detail, why one model works while the other doesn't.

Alternatively one could write the equations of the Shockley diode using an implicit computation of i_D and adopt the same approach as before. The constraint equation is now implemented with a `Switch` block to set $g(u_C, i, i_D) = i_D \stackrel{!}{=} 0$ in the locking phase.

The implementation of the interpolated Shockley diode looks exactly like Figure 10, only the `Fcn` block that computes the $i_D(u_D)$ function in Figure 11 is replaced by a `Lookup Table`. The necessary table values are computed in the `Init Fcn` callback. The model doesn't run with the `ode45` solver, but needs the stiff solvers `ode23t` or `ode23tb`.

One way to cope with DAEs is to simply differentiate the constraint equation, giving the "explicit Shockley diode" model that again can be implemented with the "switching model parts" approach (cf. Figure 12).

The problem here is that the variable u_D only becomes a state variable in conducting phase, while it still is an algebraic variable in locking phase. Therefore the `uD_conducting` block has an additional integrator block, while the `uD_locking` subsystem only contains algebraic computations. In spite of this strange

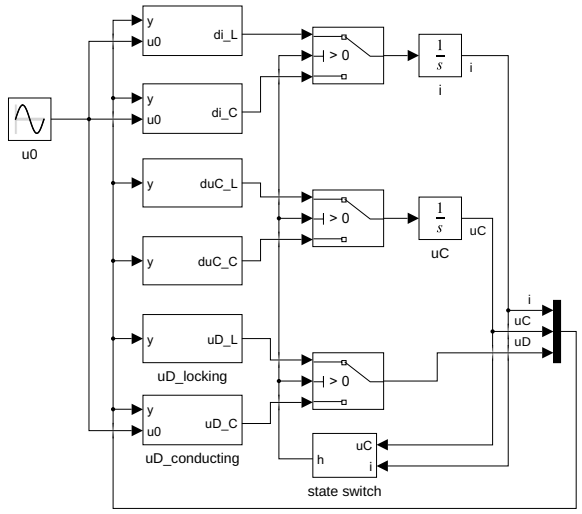

Figure 12: Explicit Shockley diode.

construction the model works without problems.

Dependency of results from algorithms. Following the same procedure as in Section 1.2 the results for the interesting variables i_L, u_C, i_D and u_D have been computed using the seven Simulink solvers and are shown in Table 3 for the shortcut diode and in Table 4 for the Shockley diode.

	ε_{iL}	ε_{uC}	ε_{iD}	ε_{uD}
ode45	0.0435	0.0000	0.0435	0.0011
ode23	30.5122	0.0173	30.5123	0.8522
ode113	50.6304	0.0239	50.6305	0.5859
ode15s	444.0654	181.7101	444.0668	137.8873
ode23s	57.0346	3.2628	57.0348	4.9717
ode23t	283.6237	17.7122	283.6245	13.2222
ode23tb	57.1190	4.0521	57.1192	2.8067

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

According to these results the solvers can be grouped as follows:

- `ode45` is by far the most accurate solver,
- `ode23`, `ode113` have medium errors for i, i_D and u_D, but small errors for u_C,
- `ode23s`, `ode23tb` have medium errors throughout,

	ε_{iL}	ε_{uC}	ε_{iD}	ε_{uD}
ode45	0.0470	0.0006	0.0146	0.0286
ode23	31.5010	0.3780	9.7677	19.1429
ode113	51.7119	0.6202	15.9115	31.4251
ode15s	454.1598	1045.878	765.7054	651.4087
ode23s	57.4423	3.7010	17.8112	34.9072
ode23t	565.4285	59.3439	177.6517	350.7907
ode23tb	57.5270	3.5943	17.8380	34.9595

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

- `ode15s`, `ode23t` have large errors throughout.

Usually one would choose a stiff solver for a DAE system, therefore this behaviour comes unexpected. Apparently, the combination of `ode45` with a Newton solver for the algebraic loops does a good job here.

The time behaviour of the errors is quite different for different solvers and variables. Figure 13 shows a few examples for the shortcut diode and the variable i_d.

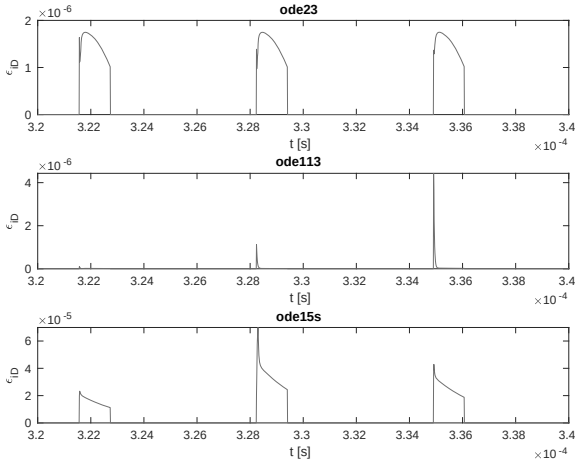

Figure 13: Shortcut diode: Relative errors for i_D

Comparison of shortcut and Shockley diode model. Comparing the results of the shortcut and Shockley diode model leads to the same results as in [2, Fig. 15].

To get accurate simulation times, both models have been run seven times and the mean value of the last five

values has been computed. The corresponding results are shown in Table 5.

	shortcut [s]	Shockley [s]	Sh / sc [%]
ode45	1.0377	2.6004	250.59
ode23	0.7594	1.7771	234.02
ode113	0.6529	1.4567	223.10
ode15s	1.2696	2.9795	234.67
ode23s	7.4267	2.4208	32.60
ode23t	0.8030	1.6715	208.16
ode23tb	0.9082	2.0373	224.33

Table 5: Simulation times for different solvers.

The differences between the solvers are astonishingly small (with the exception of `ode23s`) – one would expect that the stiff solvers like `ode15s` or `ode23s` are much faster for the implicit Shockley model. This is even more surprising in view of the much better accuracy of the `ode45` results.

Approximation of Shockley diode model. The comparison of the Shockley and approximated Shockley diodes (cf. Figure 14) almost reproduces the results shown in [2, Fig. 16]. Only the errors differ: They again have the form of spikes, but they are much more pronounced here.

The real problem is to get the approximated model to run at all: Trying different solvers and precisions and all three values for the number N of breakpoints, one very often gets an error. The proposed solution to use a higher accuracy or switch to a LineSearch-based algorithm usually makes things worse. The only solvers that are working for all N with standard parameters are `ode23t` and `ode23tb`. Since `ode23t` has shown to be very inaccurate, `ode23tb` is used for the task.

Relevance of choice of algebraic state. The Shockley models using u_D or i_D as algebraic variable produce the same results, with differences much lower than the solver tolerance. The runtimes are different, though: The i_D model is 2.4 times slower than the u_D version.

Quite interesting is the warning that appears, when using the Runge-Kutta solvers `ode45` or `ode23` for the i_D version: It states a convergence problem when

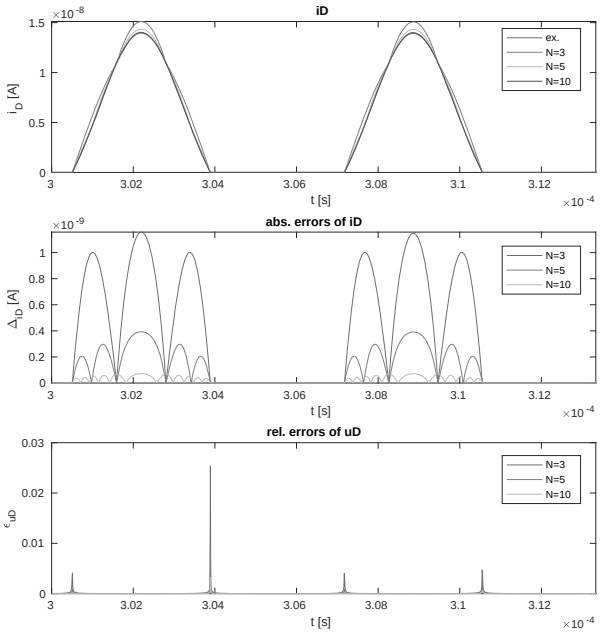

Figure 14: Comparison of Shockley and approx. Shockley diode.

solving the algebraic loop, and falls back to a strategy from an older Simulink version.

Investigation for real-time simulation. Differentiating the constraint equation of u_D for the Shockley diode and replacing the appearing time derivatives, one gets

$$\begin{aligned}\dot{u}_D &= \left[\left(\frac{(R_1+R_2)R_2}{L}-\frac{1}{C}\right)i+\frac{R_2}{L}u_C \right.\\ &\quad \left.+\left(\frac{R_2^2}{L}-\frac{1}{C}\right)I_S\left(e^{u_D/U_T}-1\right)-\frac{R_2}{L}u_0\right]\\ &\quad \cdot\left(\frac{R_2 I_S}{U_T}e^{u_D/U_T}+1\right)^{-1}\end{aligned}$$

This equation is hidden inside the `uD_conducting` subsystem in Figure 12.

The three models of the shortcut, approximated and explicite Shockley diode have been run each with a variable step solver (`ode45`, ε_{rel} = 1e-6) and a fixed step solver (`ode4`, ε_{rel} = 1e-8). As before the approximated model needed special attention: The algebraic loop solver had to use the TrustRegion algorithm with `ode45` and the LineSearch algorithm with `ode4`. The differences between both runs are smaller than the

solver tolerances for all models.

In addition, the explicit model using u_D as state variable and its alternative version based on i_D have been compared to the results of the implicit Shockley model. Again all results agree, especially there is no drift of the diode current as has been seen in [2].

3 Case Study Rotating Pendulum With Free Flight Phase

The third test model is a point mass swinging on a rope of fixed length. According to the forces acting on the mass, the movement can switch between swinging and falling phases.

Description of model implementations. For the implementation of the swinging mass a classic hybrid decomposition method is applied (cf. Figure 15). The two subblocks `Swinging` and `Falling` model the corresponding differential equations, they both output the current state values and have control inputs to enable or disable them. When enabled, new initial values are supplied by another input.

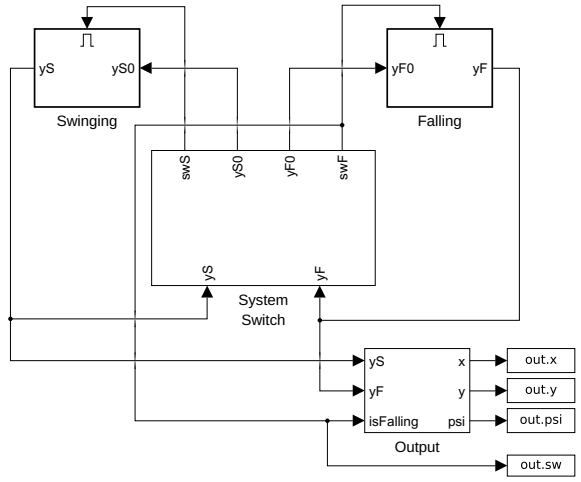

Figure 15: Rotating pendulum model.

The interesting part is the `System Switch` component (cf. Figure 16). It gets the current values of the state variables and outputs the control signals and new initial values for the two "physical" subsystems. To this end it computes the values of the event functions h_F (the rope force) and h_S (the rope slack) and uses them to trigger a state switch, which stores the current state. Such a component has already been used in the bouncing ball model (cf. Figure 6). Again `Memory` blocks are necessary to break the algebraic loops.

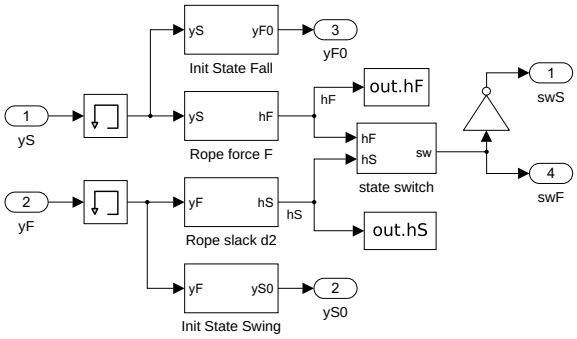

Figure 16: System switch.

Basic simulation of phases. The basic model is extended by a stop mechanism in the `Output` component: A small subsystem stops the simulation, when the angle ψ (measured against the lowest point) is below $\pi/10$. This subsystem is triggered, when the angular velocity goes through zero (at either direction). Running this model produces the results shown in Figure 17, which coincide with those in [2, Fig. 19]. The stop time differs slightly, here it is t = 7.5965376 s, compared to the value t = 7.5962714 s in [2].

Dependency of results from algorithms. The standard procedure for comparing solver precisions has been used again, the results are shown in Table 6. All solvers are very precise here with the notable exception of `ode15s`. Figure 18 displays some plots showing the error over time.

	x [1e-6 m]	y [1e-6 m]	ψ [1e-6]
ode45	0.0000	0.0000	0.0000
ode23	0.0001	0.0000	0.0001
ode113	0.0287	0.0080	0.0298
ode15s	35.3295	9.8355	36.6730
ode23s	0.0681	0.0504	0.0707
ode23t	0.5685	0.4418	0.5901
ode23tb	0.0535	0.0445	0.0555

Table 6: Absolute errors (compared to reference solution).

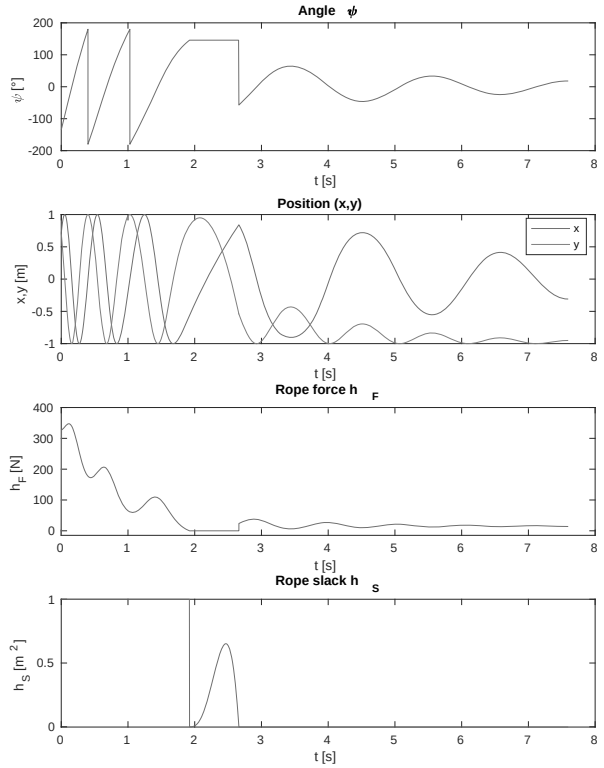

Figure 17: Results of pendulum model.

Additionally, the impact of the tuning parameters "Shape preservation" and "MinimalZcImpactIntegration" has been studied. While the second one has no consequences, the first one has interesting effects: The results of `ode113` get better by more than a factor of 100, while those of `ode15s` get even worse by a factor of 10.

External energy supply. The kick factor γ is determined with a backwards running pendulum model, which gives the same results as in [2, Table 6].

The real challenge here is the implementation of the kick mechanism with its chain of interdepending events. Since there are several very different mechanisms in Simulink to create events, there are often special tricks which make models simpler - but which are hard to come up with. Instead we will try to design the model in an as simple and straightforward manner as possible.

The kick is realized inside the `Swinging` block by using the reset mechanism of the `omega` integrator and providing two different initial values: one that is defined by the `System Switch` when changing back to the "swinging" phase, the other one is used at the kick

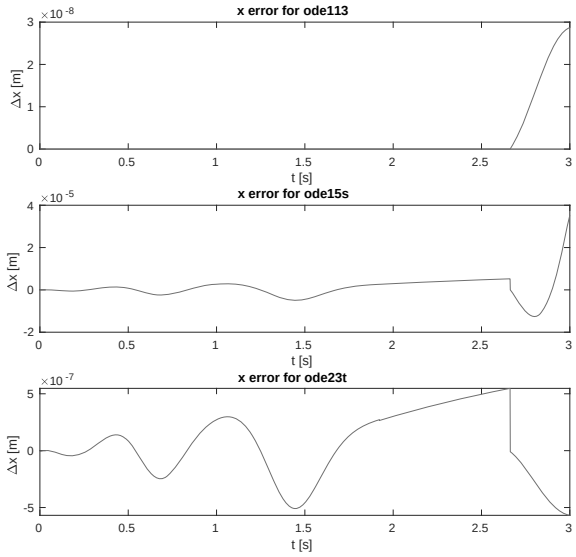

Figure 18: Absolute errors in x for three solvers.

(cf. Figure 19). The signal `isKick` is used here in two ways: Its value (0 or 1) switches between the two different initial values, its change ($0 \to 1$) triggers the kick. To make this work, the signal has to return to 0 quickly to make way for the usual system switching mechanism.

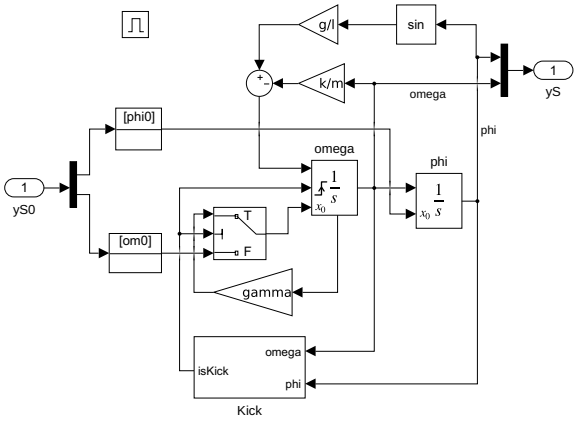

Figure 19: `Swinging` component with kick.

The chain of events is handled inside the `Kick` subsystem (cf. Figure 20): As a first step the `awaitKick` block outputs 1 as soon as the ψ amplitude gets small enough, and enables the `doKick` block, which outputs 1 when ψ reaches 0 afterwards. Both blocks contain only a `Constant` 1 and their output is initialised to 0. To guarantee that `isKick` returns to zero as fast as possible a `Hit Crossing` block is added that creates

an "infinitely short" signal at the time of the $0 \rightarrow 1$ transition.

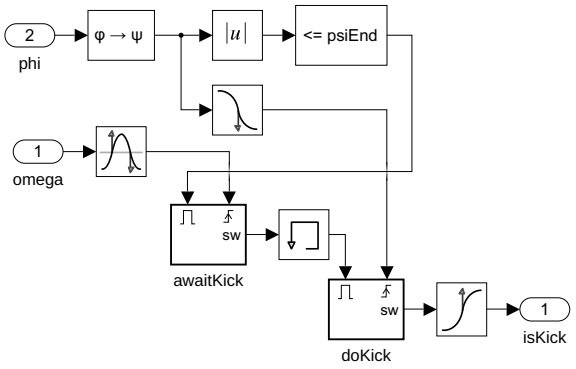

Figure 20: `Kick` subsystem.

4 Conclusions

Simulink provides a lot of different tools to create events and react to them together with special tricks such as the second-order integrator component or tuning parameters for the zero-crossing algorithm. But sometimes they behaved inconsistently, without any reason apparent to the user. Even worse, a few models only ran after playing with different workarounds, like adding `Hit Crossing` components or using very special implementations of simple subtasks.

In every simulation environment one runs into a set of typical problems, when the implementation of simple ideas collides with basic patterns or paradigms within the environment. Usually one builds up a collection of workarounds for common problems. One goal of the ARGESIM benchmarks is to provide a collection of such solutions, maybe in the spirit of the *Design Patterns* of object-oriented programming [12]. But unlike in the OO world, the solutions found in modeling usually are bound to the specific simulation environment – or even to a specific version.

Comparing the mathematical description of the models and their implementation in Simulink, one finds a lot of components that have no direct mathematical counterpart. It would be much better if basic mathematical ideas could be implemented in standard ways, independent of the concrete environment used. But this still remains a task for future simulation programs.

References

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

[2] Disselkamp JP, Junglas P, Niehüser A, Schönfelder P. A Solution to ARGESIM Benchmark C21 'State Events and Structural-dynamic Systems' based on Modelica Components. *SNE Simulation News Europe*. 2018; 28(2):39–48. doi: 10.11128/sne.28.bn21.10411.

[3] The MathWorks. *Simulink: Simulation and Model-Based Design*. `http://www.mathworks.com/products/simulink/`.

[4] The MathWorks. *Stateflow: Model and simulate decision logic using state machines and flow charts*. `http://www.mathworks.com/products/stateflow/`.

[5] Clune MI, Mosterman PJ, Cassandras CG. Discrete Event and Hybrid System Simulation with SimEvents. In: *8th International Workshop on Discrete Event Systems*. Ann Arbor. 2006; pp. 386–387.

[6] Junglas P. *Argesim C21 models and scripts*. URL `http://www.peter-junglas.de/fh/simulation/argesimc21.html`

[7] Zhang F, Yeddanapudi M, Mosterman PJ. Zero-crossing location and detection algorithms for hybrid system simulation. *IFAC Proceedings Volumes*. 2008; 41(2):7967–7972.

[8] The MathWorks. *Simulink: Simulation of a Bouncing Ball*. `https://www.mathworks.com/help/simulink/slref/simulation-of-a-bouncing-ball.html`.

[9] Shampine LF, Gladwell I, Thompson S. *Solving ODEs with matlab*. New York: Cambridge university press. 2003.

[10] The MathWorks. *Simulink: Enable minimal zero-crossing impact integration*. `https://www.mathworks.com/help/simulink/gui/enable-minimal-zero-crossing-impact-integration.html`.

[11] Shampine LF, Reichelt MW, Kierzenka JA. Solving index-1 DAEs in MATLAB and Simulink. *SIAM review*. 1999;41(3):538–552.

[12] Gamma E, Helm R, Johnson R, Vlissides J. *Design patterns: elements of reusable object-oriented software*. Reading, Mass.: Addison-Wesley. 1995.