# Implementing the Argesim C21 benchmark with Modelica components

Jan-Philipp Disselkamp, Peter Junglas, Alexander Niehüser, Phillip Schönfelder
PHWT Vechta/Diepholz
*peter@peter-junglas.de*

The Argesim C21 benchmark requires to investigate three different systems showing state events: a bouncing ball, an RLC circuit with a diode and a rotating pendulum with a free flight phase. These models can be easily implemented in Modelica by coding the complete state and event equations. But the simulation practitioner is usually working in a graphical environment using components from a library, writing explicit Modelica code at most for special components. We will show how to implement such systems in a component based style by providing switchable and event handling components, thereby coping with the limitations of the complicated Modelica event system as well as with some problems of the simulation environments.

## 1 Introduction

The modelling and simulation of hybrid systems combining a continuous dynamic evolution with state events is still a nontrivial task for most simulation environments. To compare the different approaches the Argesim C21 benchmark [1] defines three systems that show state events with event types of varying complexity: A bouncing ball model using either a simple state change event or a slightly more complex structure change event, an RLC circuit with a diode leading to a parameter or structural change, and a rotating pendulum with a complex structural change between its rotating and free flight phases.

Using Modelica [2] to implement the three models seems to be a simple task at first sight, since Modelica provides a rich set of event related features. In addition the recent synchronous elements [3] define a complementary set of event functions with different semantics. On the other hand Modelica's events are quite involved, sometimes leading to a counterintuitive behaviour [4]. And [5] even claims that Modelica "provides only very limited means" for the description of systems with variable structure, due to restrictions of the Modelica language, which are often related to its equation based foundation.

But Modelica is more than a modelling language: It provides a frame for the definition of components that can be combined with graphical tools to build up a complex model. Since this is the preferred way how Modelica is used in the industrial practice, it would be highly desirable to have components that make the construction of the example models possible. Some appropriate components are already given by the Modelica Standard Library (MSL), in other cases one has to build them using explicit Modelica code.

In the following we will show several different approaches to implement the example models in a component based way. Since the implementations of Modelica often differ in subtle ways, especially with regard to their event behaviour, we will compare our models using the proprietary programs Dymola from Dassault Systemes and MapleSim from Maplesoft. Finally, the conclusions will address the question, to which extent Modelica is able to model structure changing systems.

## 2 Bouncing ball

The first example is a falling ball that bounces off the ground. The C21 benchmark contains two different contact models: The event contact describes a timeless bounce that changes the velocity immediately according to

$$v_{\text{after}} = -\mu\, v_{\text{before}},$$

where the coefficient $\mu$ describes the energy loss due to the bounce. The dynamic contact includes the deformation of the ball using a linear spring and damper model.

A simple approach that replicates the standard Simulink method, uses integrator blocks to reproduce the differential equation of the system directly. For the event contact one can utilize limit and reset integrators as shown in Fig. 1. All blocks come from the MSL

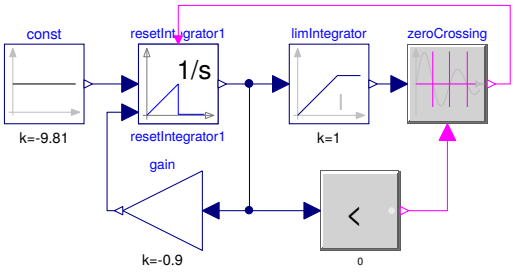


**Figure** 1: Bouncing ball with event contact ("Simulink")

except the `ResetIntegrator`, which can easily be implemented with a `reinit` inside a `when`-construct. Since the MSL `zeroCrossing` block triggers on rising and falling input, it must be disabled for positive velocity. This is superfluous if one uses a selfmade `NegativeZeroCrossing` component instead.

To be more in line with the physical modelling spirit of Modelica one can use concepts and components of the `Mechanics.Translational` part 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. Fig. 2). Its equations

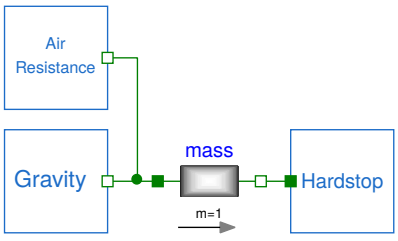


**Figure** 2: Bouncing ball with event contact (1d mechanics)

are based on the example in [2, pp. 96f]:

```
s = flange.s;
v = der(s);
when s <= 0 then
  reinit(v, -mu*pre(v));
end when;
flying = not (s <= 0 and v <= 0);
flange.f = if flying then 0 else -m*g;
```

The interesting part is the use of the additional `flying` variable instead of a simple `s <= 0`. This is necessary to cope with the numerical inaccuracies, which otherwise would lead to very small but negative values of `s` after an event and to the infamous fallthrough of the ball near the scattering singularity.

Problematic is the computation of the counterforce `-m*g` after the flying phase, which needs the mass of the falling ball. Therefore the `Hardstop` component is tightly connected to the `Mass` component and cannot be used as a universally applicable component by its own. A similar problem has already surfaced when trying to include a collision detection mechanism in the Modelica MultiBody Library [6] and has been listed in [5] as one of the obstacles for modelling of variable structure systems in Modelica.

The apparently more complicated dynamic contact can be easily handled in Modelica, since the 1d-mechanics library already contains a corresponding component `ElastoGap`. Actually its equations are a bit more complicated than the ones in the C21 benchmark in order to cope with unphysical effects of the simple equations. But it is a trivial matter to create a copy and strip it down to comply with C21.

# 3 RLC circuit with diode

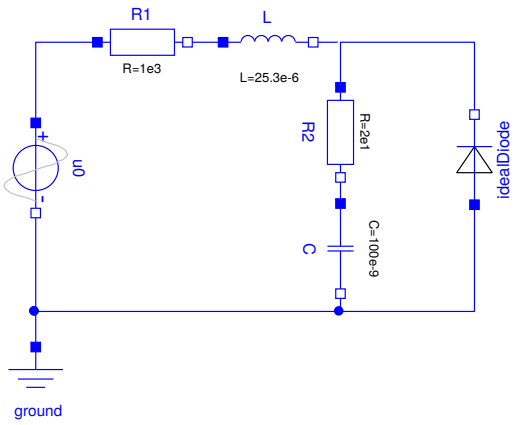


**Figure** 3: RLC circuit with diode

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 compo-

nents from the `Electrical.Analog` part of the MSL (cf. Fig. 3), using the `IdealDiode` for the shortcut diode of C21.

The MSL contains a diode component implementing the common Shockley characteristic, but it differs from the "Shockley diode" of C21, because it has a non-vanishing current in locking phase. Therefore one has to create a special component here. Using a partial model provided by the MSL this is a matter of a few lines:

```
model DIshi1 "diode model after Shockley"
  extends Analog.Interfaces.OnePort;
  parameter Real IS = 1e-8;
  parameter Real UT = 26e-3;
equation
  if v < 0 then
    i = 0;
  else
    v = UT*log(i/IS + 1);
  end if;
end DIshi1;
```

Alternatively the model `DIshu1` uses the inverse relation to define `i` as function of `v` in conducting phase.

C21 defines a "real-time" variant, where the algebraic equation for v in conducting phase is replaced by its derivative in order to reduce the DAE equations of the complete model to an ODE. In the component based model used here things are more complicated: If one uses the derivative equation for v

```
der(v) = (UT/Is)/(i/Is + 1)*der(i);
```

inside the diode model, the variable `i` changes from algebraic to state variable, when the diode switches from locking to conducting phase. To cope with this problem, we added another variant, where the equation in locking phase is derivated as well:

```
der(i) = 0;
```

Another diode variant defined in the benchmark replaces the exponential Shockley characteristic with a piecewise linear interpolated approximation. This can be implemented either graphically using a `LookupTable` component from the MSL together with a `VoltageSensor` and a generic Diode model or by implementing the linear interpolation directly with a Modelica function.

# 4 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. Since its structure changes completely between states of different dimension, this is the most interesting and demanding system of the benchmark.

Even an entirely equation-based solution is challenging. In [1] three modelling concepts are presented: a hybrid decomposition with subsystems of different dimensions, a maximal state space approach and a DAE of high index that uses cartesian coordinates throughout, but changes the constraint equation. Of these the first one seems hopeless in Modelica: Although the variables that are used as states can change dynamically, their total number is fixed by an equal number of equations. The second method can be applied in a straighforward manner, leading to a working - but monolithic - solution. The third idea seems to be most promising, since DAEs of high index are commonplace in Modelica. But though a corresponding model has been formulated by the authors, none of the available simulation programs could cope with it.

Focusing on component based models, two different ideas have been followed, similar in spirit to the two different solution methods described in [7]: The first is a "local" method using switchable components for the structurally different phases, the other takes a top-down approach to switch globally between two different systems.

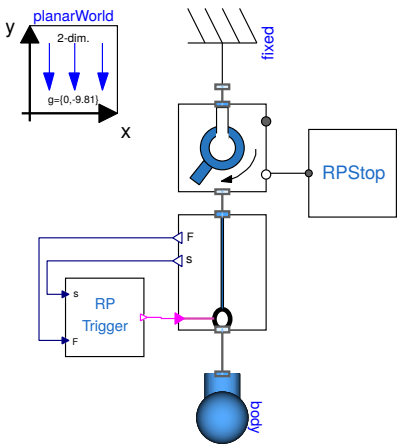


**Figure** 4: Rotating pendulum model (2d mechanics)

The first model uses graphical components as far as possible, coming from the MSL or from the free Planar Mechanical Library [8], which is used here instead of the Modelica MultiBody library, since it is much simpler and works identically in both simulation programs. The complete model consists mainly of a `Revolute` joint defining the swinging motion, a selfmade `Rope` together with a trigger and a `Body` for the point mass (cf. Fig. 4).

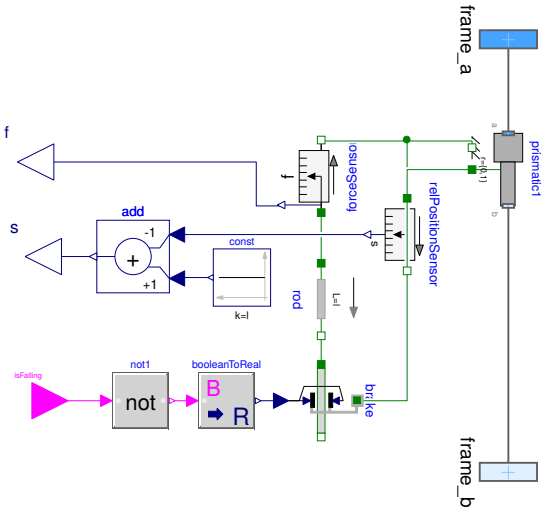


**Figure** 5: Rope component

The central idea is found in the implementation of the `Rope` (cf. Fig. 5): It is basically a `Prismatic` joint which is controlled by a rod of fixed length that is attached to a brake. According to the state of the brake the length of the joint is either fixed or freely variable, thus defining the swinging and falling states of the pendulum. The `RPTrigger` component uses a standard flipflop [4] to represent the state and changes it according to outward force and slack of the rope.

From a practical point of view the model works fine, but technically it is not a complete solution of the benchmark: The transition from falling to swinging state takes a short time due to the large, but finite force delivered by the brake. Presumably this could be fixed with an event-based brake, but this would destroy the charme of the model that consists almost entirely of standard components.

The second implementation takes a global approach: It 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. Fig. 6). The switch contains the event functions $h^F$, $h^S$ as defined in [1] and computes the initial state at a system change.

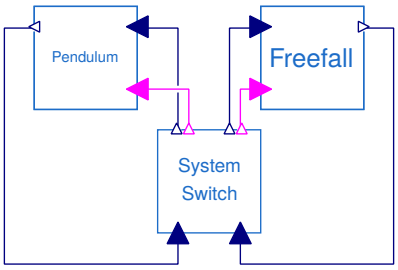


**Figure** 6: Rotating pendulum model (hybrid version)

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;
```

Then the implementation of a concrete system only needs the definition of the state equations, usually in form of an ODE. But even a graphical approach is possible: One starts with a new component that inherits from `SwitchableSystem`, thereby getting the necessary input and output connectors automatically. Inside the model one simply ignores these connectors and constructs the system graphically with components from the MSL or other libraries (cf. Fig. 7).

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

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

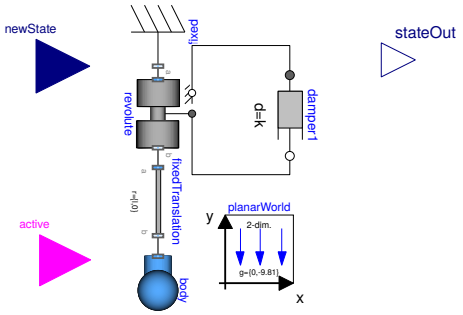


**Figure** 7: SwitchablePendulum component

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. Fig. 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.

# 5 Differences between Dymola and MapleSim models

All models have been tested with Dymola 2018 and MapleSim 2017.3 under Kubuntu 16.04. MapleSim uses an internal (non-Modelica) format to save the models, but can import Modelica libraries. Therefore all components have been collected in a Modelica library, which is identical for both programs. To make this possible we couldn't use the Modelica MultiBody library, because MapleSim uses its own proprietary version, which is incompatible with the Modelica one. Fortunately the free Planar Mechanical Library [8] works in both programs and was sufficient here. Furthermore it allows to access internal variables, which was useful for some tests, but not possible with MapleSim's MultiBody library.

Except for the differences mentioned in the following and for minor numerical deviations all models worked identically in both programs. Execution times were always small except for one notable example: The Shockley diode model `DIshi1`, which defines the characteristic as v(i), was about 100 times slower than the i(v) variant, when using MapleSim. This is probably due to a poor choice of the state variable.

Another problem showed up only in MapleSim: When running the hybrid version of the pendulum model using graphically constructed subsystems, the simulation stops immediately with the error message

```
'reinit' applied to an algebraic variable
'Main.switchablePendulum\_1.state[2]'(t),
                will have no effect
```

Obviously the solver did not use the `state[2]` variable as a proper state variable. Since this is in conflict with the `stateSelect` attribute, it seems to be a bug.

Much more interesting in the present context is the behaviour of the different "real-time" diodes: The variants using derivated equations for both phases ran in both programs, whereas the versions with the equation `i = 0` in locking phase worked only in MapleSim. Dymola stopped the simulation, when the first locking phase appeared, claiming to hit upon a singular linear system of equations. At this point the variable `i` changes from a state to an algebraic variable, which might be the root of the problem.

A very strange bug came up, when we tried to implement a pendulum that gets an additional kick (i. e. a sudden change in angular velocity) as requested in [1]: The first version contained several independent when-clauses with reinits. Dymola claimed this to be nondeterministic and combining two of the whens with elsewhen, in fact, did the trick and worked properly. But not with MapleSim: It always stopped with an internal error. To make it work one had to delete the else-branch in

```
when {state[2] < 0, state[2] > 0} then
  if abs(psi) < psiEnd then
    awaitKick = true;
  else
    awaitKick = false;
  end if;
end when;
```

This is definitively a bug, but more interesting is what happened after this strange workaround: The defective model with independent whens runs properly,

but after the elsewhen correction has been applied, it misses the kick. This strongly suggests some errors deep in the event system.

# 6 Conclusions

The basic question is: Can one implement structure-variable systems of different dimension in Modelica? This is a hotly debated topic, and the answers given in the literature are quite different, e. g.

- *"There is until now no possibility implemented to make graphical model switching for subsystems with different state space dimension."* [7]

- *"Even though ... the set of variables and the set of equations is fixed over time, it is the case that conditional equations in hybrid DAEs can be activated and deactivated. ...*
  *... Thus the* active part *of the hybrid DAE can be* structurally dynamic*, i.e., at run-time change the number of* active *variables and equations in the DAE."* [9].

Even though the authors of the first quote are correct formally, the results presented here emphasize the point of the second one. At least Modelica's event system is rich enough to provide working and interesting solutions to the tasks of the C21 benchmark, even if one concentrates on graphical modelling.

Nevertheless, the most straighforward way to implement the pendulum example - a DAE system using only cartesian coordinates - did *not* succeed in any program, though it seems to be correct formally. Whether this is a problem of Modelica itself or of the implementations, is an open question.

Which concepts work and which do not can depend on the concrete implementing software. There will always be differences because of bugs, but some of the problems seem to indicate that they are rather due to different interpretations of the exact semantics of the event system. For further progress on structure-variable systems in Modelica, this needs to be clarified in a precise manner.

# Acknowledgements

The authors are thankful to the Maplesoft Support for providing the basic idea of the rope model.

# References

[1] A. Körner, F. Breitenecker. *State Events and Structural-dynamic Systems: Definition of ARGESIM Benchmark C21.* Simulation Notes Europe, 26(2), pp. 117-122, 2016.

[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 2018-01-08).

[3] H. Elmqvist, M. Otter, S. E. Mattsson. *Fundamentals of Synchronous Control in Modelica.* Proc. 9th Int. Modelica Conf., Munich, pp. 15-26, 2012.

[4] P. Junglas. *Pitfalls using discrete event blocks in Simulink and Modelica.* Proc. ASIM Symp. STS/GMMS, Lippstadt, pp. 90-97, 2016.

[5] D. Zimmer. *Enhancing Modelica towards Variable Structure Systems.* Simulation News Europe, 17(2), pp. 23–28, 2007.

[6] M. Otter, H. Elmqvist, J. Díaz López. *Collision Handling for the Modelica MultiBody Library.* Proc. 4th Int. Modelica Conf., Hamburg, pp. 45-53, 2005.

[7] G. Zauner, D. Leitner, F. Breitenecker. *Modeling Structural - Dynamics Systems in MODELICA/Dymola, MODELICA/Mosilab and AnyLogic.* Simulation News Europe, 17(2), pp. 49–54, 2007.

[8] D. Zimmer. *A Planar Mechanical Library for Teaching Modelica.* Proc. 9th Int. Modelica Conf., Munich, pp. 681-690, 2012.

[9] H. Lundvall, P. Fritzson, B. Bachmann. *Event Handling in the OpenModelica Compiler and Runtime System.* Technical report, PELAB, Dept. Computer Science, Linköping University, 2008.