

# Pitfalls using discrete event blocks in Simulink and Modelica

Peter Junglas  
PHWT Vechta/Diepholz/Oldenburg  
*peter@peter-junglas.de*

Though Simulink as well as Modelica are basically tools for the modeling of continuous systems, they both contain several elements that allow for discrete and hybrid modeling. Comparing the respective block libraries one finds almost identically looking components, but this similarity is deceptive. Using simple digital circuits with flip-flops as examples, one finds subtle differences in the handling of events, which can lead to an unexpected behaviour of a model. A good understanding of the underlying mechanisms, especially of the Modelica standard description, is therefore essential for the successful modeling of discrete systems. But even then it can be difficult to create models which have the required behaviour.

## 1 Introduction

Modern simulation programs with their graphical user interfaces and large block libraries have simplified the task of creating complex models considerably. Superficially the basic procedure is almost identical even for tools with very different philosophies like the signal-oriented Simulink and Modelica-based physical modeling programs [1] like Dymola. This can lead to a concentration on the apparent block structure of a model, disregarding details behind the scene like solver properties or the defining language. That this negligence can result in modeling errors that are hard to understand will be shown in the following.

While Simulink and Modelica have their roots strongly in continuous modeling, they both have basic discrete features, which can be enhanced with additional packages for state machines [2, 3] or process based models [4, 5]. Already the basic libraries contain blocks like `Memory` and `Unit Delay` in Simulink resp. `Pre` and `UnitDelay` in Modelica, which look very similar – but their underlying mechanisms are quite different.

In the following a few examples of simple digital circuits containing flip-flops will demonstrate that seemingly identical models can lead to different results. After a close look at the Modelica language specification a detailed analysis of the models will explain, why they behave differently from their corresponding

Simulink counterparts. Finally it will be shown how to cope with the subtleties and create robust models of flip-flops in Modelica.

Though all ideas presented here surely are well-known to experts, the consequences of "well-known facts" are not always self-evident. Therefore this study may be useful not only for students (and teachers), but for practitioners as well, especially when they are employing different simulation tools.

## 2 Simulating flip-flops in Simulink and Modelica

Trying to model digital circuits containing flip-flops is a rewarding endeavor, because it illustrates the intricacies of discrete modeling in a nutshell. Though the models presented in this section are quite simple, some of them show a behaviour that probably may come as a surprise.

### 2.1 The static RS flip-flop

The simplest flip-flop is the static RS flip-flop, which is usually shown as two nor gates with crosswise connected outputs. This can not be modeled directly, because it leads to an algebraic loop over discrete variables. To break it, one can simply add a "very short"

delay, in Simulink using a Memory block for that purpose (fig. 1).

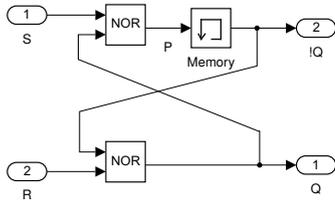


Figure 1: Static RS flip-flop (version A)

The results in fig. 2 (version A) show the behaviour for all possible input combinations, including  $S = R = 1$  at  $t = 10$ , which is called *forbidden*, because the corresponding outputs  $Q$  and  $!Q$  are *inconsistent*, i. e. not inverse to each other. The delays can be seen clearly, at  $t = 7$  they lead to another inconsistency due to the lag of  $!Q$  behind  $Q$ .

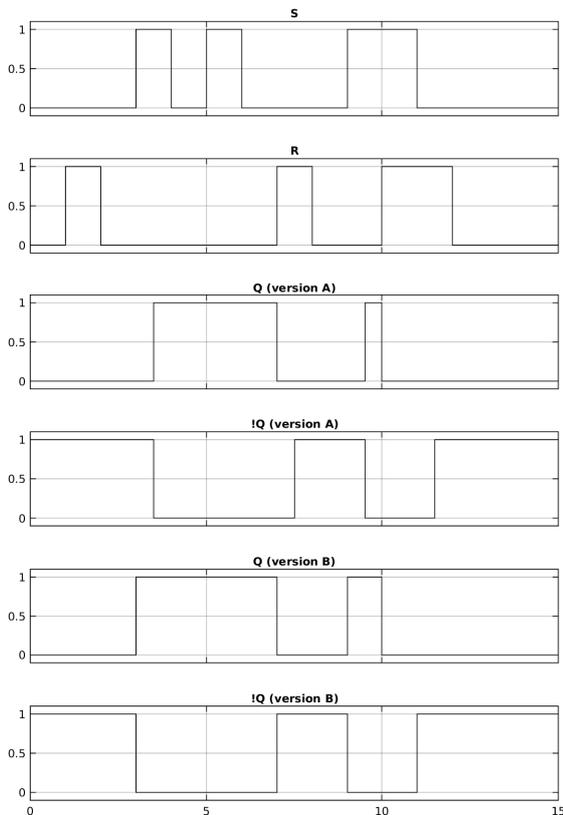


Figure 2: Simulation results of static RS flip-flops

Actually the delays are astonishingly large: The fun-

damental time period of the Simulink solver is set to 0.1, but the delay is 0.5. This value is inherited from the blocks defining the input values  $S$  and  $R$ . With some extra effort one can reduce it, but defining a very small time step leads to long execution times, while the inconsistency still remains – however shortly.

Therefore the Simulink block library contains a different implementation of the RS flip-flop (fig. 3). Here the delay block has been moved into the feedback loop, while the generic Logic block implements the needed boolean logic to compute the output values from the  $S$  and  $R$  inputs and the “last”  $Q$  value. The simulation results (fig. 2, version B) are now as required.

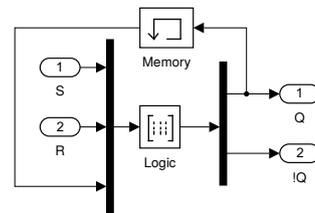


Figure 3: Static RS flip-flop (version B)

In Modelica one can reproduce the simple version A using the Pre block to implement the delay – actually this RSFlipFlop is already included in the basic Blocks library. It directly leads to the desired results as in version B with no output delay at all! Apparently, Simulink and Modelica work differently here. With a little effort one can replicate version B as well, but it simply reproduces the former correct results.

## 2.2 The triggered RS flip-flop

To cope with timing problems the *triggered* flip-flops have an additional input, often denoted CLK, for synchronisation purposes. The state of such a flip-flop changes only, when this input changes in a predefined way, e.g. from true to false (*negative edge*). Fig. 4 shows example signals for the inputs  $S$ ,  $R$  and CLK and the corresponding outputs  $Q$  and  $!Q$ . Basically the flip-flop behaves like before, but the changes take place only at the negative edges of the CLK input.

In Simulink such a flip-flop can be easily modeled by adding a Trigger block to the subsystem of the static RS flip-flop. This creates an additional *trigger* input

for the block, which leads exactly to the required behaviour.

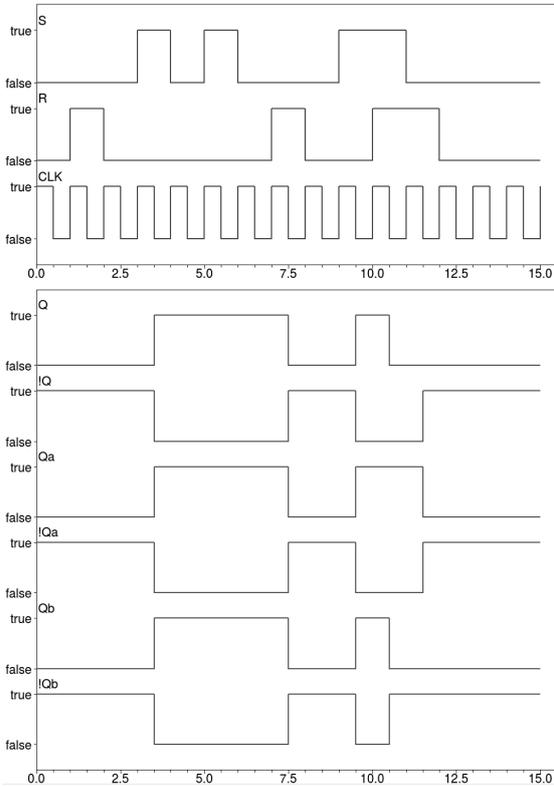


Figure 4: Simulation results of triggered RS flip-flops

In Modelica one can use the FallingEdge block to replicate the Simulink model (fig. 5). At a negative edge of its input signal it creates an “infinitesimally short” output impulse, which unfortunately doesn’t show up in result plots. That it is really working can be seen in fig. 4, where its output Qa and !Qa show the correct results – up to  $t = 10.5$ . At this point the CLK triggers while the inputs have the “forbidden” combination. Both outputs should be false now, but Qa stays at true mysteriously.

Since the reason, why the simple static RS flip-flop works, was puzzling anyhow, one could come up with the idea of replacing it by the elaborated version B, which even leads to a closer correspondence with Simulink. This produces the results shown as Qb and !Qb in fig. 4: It works up to  $t = 10.5$ , but then instead of the correct results (false/false) or the previous ones (true/false) it surprisingly goes to false/true. What is going on here?

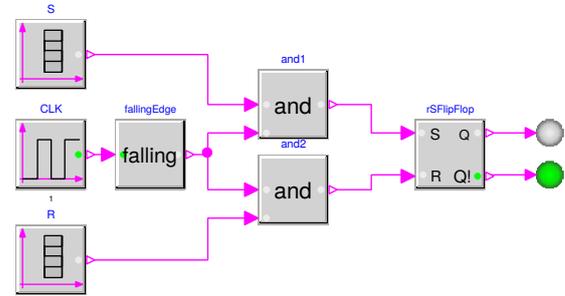


Figure 5: Triggered RS flip-flop (Modelica)

The point is not that it is particularly important to model the RS flip-flop at the forbidden state – in fact in reality one ensures that this state can not appear, e.g. with additional logic as in the JK flip-flop. The real problem is that one would (naively) expect both Modelica models to work, but they don’t. However, one has no chance to understand these results looking at the models only from the block level.

### 2.3 The shift register

The basic memory block is the D flip-flop, which has only one data input D and a CLK input and stores the data value at the negative edge of its CLK input. It can be created easily from a triggered RS flip-flop by identifying S with D and connecting its R input via a NOT gate to D (fig. 6). Since the “forbidden” state is ruled out here, one can hope that the Modelica version will work as expected.

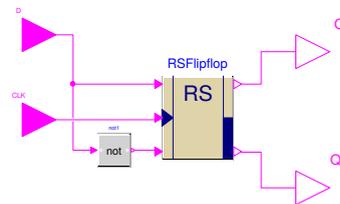


Figure 6: D flip-flop

A simple shift register (fig. 7) will be used to test the D flip-flop. The Simulink model shows the correct behaviour: Each flip-flop delays its input signal by one clock period, so that the incoming signal is “shifted” through the register.

The corresponding Modelica model however doesn’t

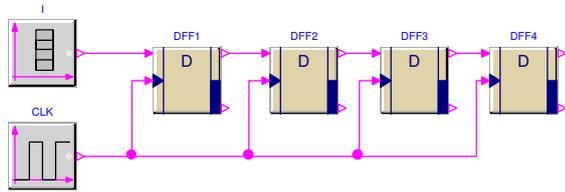


Figure 7: Shift register

work as requested (fig. 8): While the output of the first flip-flop DFF1 is actually the delayed input, the later flip-flops seem to “absorb” a single signal, so that only longer blocks survive.

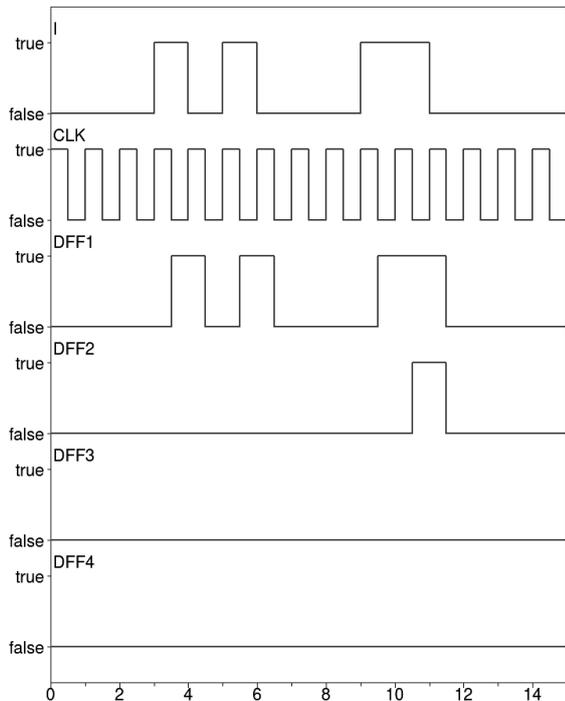


Figure 8: Simulation results of a shift register (Modelica)

Using version B of the static RS flip-flop inside the model, one again gets different, but still wrong results: Though the output of DFF1 is still correctly delayed, all other outputs coincide with DFF1 – the signal runs through the register in one step! The same thing happens incidentally, if one uses the D flip-flop that is hidden in the Modelica library `Electrical.Digital.Examples`.

In contrast to the triggered RS flip-flop, here one can find an easy ad-hoc workaround: If one adds a Pre

block before each D input (optionally including the first one, for symmetry) the shift register works perfectly.

### 3 Detailed analysis of the models

The Modelica results shown above seem to be proper nonsense and the result of grave bugs of the simulation program. But quite the contrary: They are completely in accordance with the Modelica language specification [6], as will be shown in the following.

#### 3.1 A look at the Modelica language specification

The handling of events in Modelica is quite complicated, the relevant descriptions are scattered throughout the standard document [6]. The basic definition of an *event* is given in ch. 8.5 [6, p.92], but the most relevant point for the current discussion is the clarification of the pre operator:

`pre(y)`  
Returns the “left limit”  $y(t^{pre})$  of variable  $y(t)$  at a time instant  $t$ . At an event instant,  $y(t^{pre})$  is the value of  $y$  after the last event iteration at time instant  $t(\dots)$ .

...  
A new event is triggered if at least for one variable  $v$  “`pre(v) <> v`” after the active model equations are evaluated at an event instant. In this case the model is at once reevaluated. This evaluation sequence is called “*event iteration*”. The integration is restarted, if for all  $v$  used in pre-operators the following condition holds: “`pre(v) == v`”. [6, p.29f]

The exact procedure is described with more detail in an appendix [6, p.262f]. It has two important consequences:

1. Discrete variables can run through several values at a fixed time instant, if the event loop consists of more than one iteration.

2.  $\text{pre}(v)$  is the value of  $v$  at the preceding iteration, not the one before the first iteration (i. e. at  $t^-$ ).

This is in contrast to the much simpler Simulink definition, where the *previous* value is related to time, i.e. it is  $y(t^-)$ .

### 3.2 Understanding the static RS flip-flop

To thoroughly understand the behaviour of a discrete model in Modelica, one has to go through the event iterations manually and compute the values of all variables. For version A of the static RS flip-flop (fig. 1) these are the two inputs S, R, the two outputs Q, !Q and the input P of the Pre block (shown as Memory block in the figure).

To understand, why the model has no time delay, it suffices to look at the values around  $t = 3$ . To compute them one starts with the known inputs S, R and the value of !Q, which is given as the P of the “last” step. This immediately gives Q and finally P. Table 1 displays all values, where  $t = 3^-$  is before the event,  $t = 3.x$  during the event loop and  $t = 3^+$  at the end.

t	S	R	P	Q	!Q
$3^-$	0	0	1	0	1
$3.a$	1	0	0	0	1
$3.b$	1	0	0	1	0
$3^+$	1	0	0	1	0

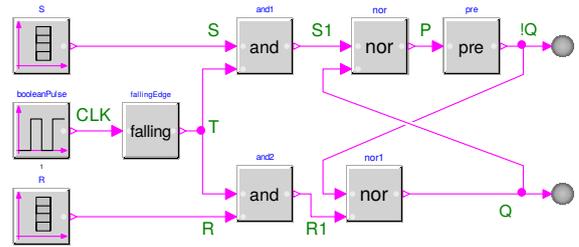
**Table 1:** Event iteration for the static RS flip-flop

The first line is given by the previous state of the flip-flop, the next line reflects the change of the input S. Comparing the two lines shows that P has changed, i.e.  $P \neq \text{pre}(P)$ . This leads to a new event iteration with the result shown in the third line. The change in Q and !Q triggers another iteration giving the result in the last line. This time nothing has changed, the iteration stops and the simulation time continuous with these values.

This analysis clarifies the behaviour of the static flip-flop: It works without output delays because of the immediate re-evaluation after a change. In a similar way one can make sure that version B works as well.

### 3.3 Understanding the triggered RS flip-flop

For a detailed walk-through one starts with a labelling of all signals, including those which are hidden inside subsystems. The result for the triggered RS flip-flop (version A) is shown in fig. 9.



**Figure 9:** Detailed view of the triggered RS flip-flop

The interesting point here is the wrong result for  $R = S = 1$ , which happens at the negative edge of the CLK input at  $t = 10.5$  (cf. fig. 4). The computation starts with the known inputs S, R, CLK and the value of !Q, which is the last value of P. Using them one immediately gets T, S1, R1 and Q and finally P. The result is shown in table 2.

t	S	R	CLK	T	S1	R1	P	Q	!Q
$10.5^-$	1	1	1	0	0	0	0	1	0
$10.5.a$	1	1	0	1	1	1	0	0	0
$10.5.b$	1	1	0	0	0	0	0	1	0
$10.5^+$	1	1	0	0	0	0	0	1	0

**Table 2:** Event iteration for the triggered RS flip-flop

Before  $t = 10.5$ , the flip-flop is set, i.e.  $Q = 1$  and  $P = 0$ , which fixes the first line. The next line shows the change of the CLK input to 0, which sets  $T = 1$  and all other signals accordingly. The following iteration has  $T = 0$ , cutting of S and R again and leading to another iteration. Finally nothing changes, so the event loop stops and the simulation time progresses again.

At  $10.5a$  the “correct” output  $!Q = Q = 0$  appears. A real triggered system like in Simulink would now hold its state until the next input trigger. But in Modelica the event loop iterates again and changes the value of Q back to 1. To understand why the flip-flop works properly for other times, table 3 provides the compu-

tations for  $t = 7.5$ . One easily assures oneself of the correct working of the flip-flop at all other times and of the – strange, but correct – behaviour of version B.

t	S	R	CLK	T	S1	R1	P	Q	!Q
7.5 <sup>-</sup>	0	1	1	0	0	0	0	1	0
7.5.a	0	1	0	1	0	1	1	0	0
7.5.b	0	1	0	0	0	0	1	0	1
7.5 <sup>+</sup>	0	1	0	0	0	0	1	0	1

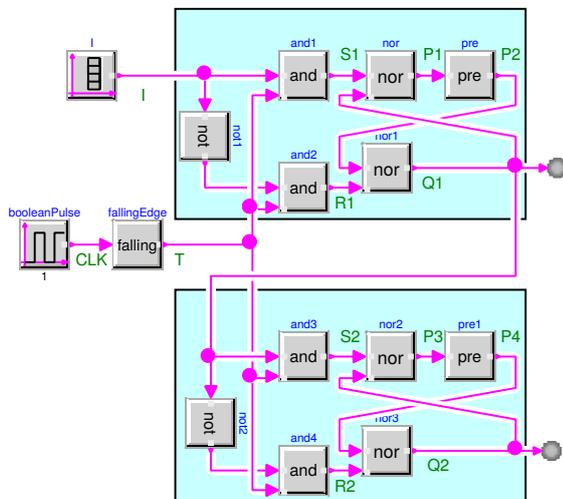
**Table 3:** Event iteration for the triggered RS flip-flop, 2<sup>nd</sup> example

t	I	CLK	T	S1	R1	P2	Q1	P1	S2	R2	P4	Q2	P3
3.5 <sup>-</sup>	1	1	0	0	0	1	0	1	0	0	1	0	1
3.5.a	1	0	1	1	0	1	0	0	0	1	1	0	1
3.5.b	1	0	0	0	0	0	1	0	0	0	1	0	1
3.5 <sup>+</sup>	1	0	0	0	0	0	1	0	0	0	1	0	1
4.5 <sup>-</sup>	0	1	0	0	0	0	1	0	0	0	1	0	1
4.5.a	0	0	1	0	1	0	0	1	0	1	1	0	1
4.5.b	0	0	0	0	0	1	0	1	0	0	1	0	1
4.5 <sup>+</sup>	0	0	0	0	0	1	0	1	0	0	1	0	1

**Table 4:** Event iteration for the shift register

### 3.4 Explaining the shift register

To understand the strange behaviour of the shift register it suffices to examine a short version with two D flip-flops in a row. Fig. 10 shows the detailed model and all signal names.



**Figure 10:** Detailed view of a short shift register

As should be clear by now, one starts with I, CLK, P2 (last value of P1) and P4 (last value of P3), computes T, S1, R1, S2, R2, Q1, Q2 and finally P1, P3. Using the input values of fig. 8 leads to the results shown in table 4.

It is now obvious, why the shift register doesn't work as intended: At event iteration 4.5.a the first flip-flop is already reset, i. e. Q1 has the value 0. But also at this

point the flip-flops are unlocked, i. e. T is 1. Therefore the second flip-flop gets the new value of Q1, not the old one. This makes plausible that an additional Pre block might cure the problem. To make sure though, one has to go through another manual computation.

## 4 Solution of the problems

The last section has shown, how to explain the strange behaviour of the flip-flop example models in the context of Modelica's event system. But this doesn't help with the task of creating reliable, easy to use flip-flop components, because there is apparently no way to implement a triggered system with the semantics that is needed here – at least using Modelica's standard events.

Similar problems have been found in a completely different application area, namely the modeling of reliable state machines [3]. They have been solved there by extending the Modelica language itself: Version 3.3 contains features for synchronous signals that are tied to discrete clocks [7]. A corresponding block library has been created that allows to include these features easily in a graphical environment [8].

Fortunately the new possibilities are exactly what is needed to create a properly working triggered flip-flop. Fig. 11 shows an implementation based on the new Synchronous library. The Sample and Hold blocks transfer standard signals to synchronous ones and vice versa. The EventClock creates a clock signal that is triggered by a positive edge of its input.

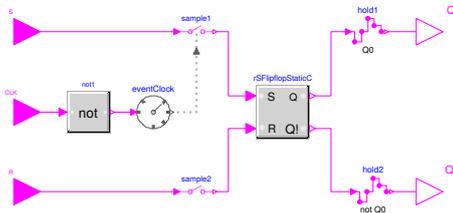


Figure 11: Triggered RS flip-flop with clock

But the main point is hidden inside the static RS flip-flop (fig. 12): Though it looks almost exactly like the version B of the static flip-flop (fig. 3), instead of a Pre block it now contains the new UnitDelay block. It returns the value of its clocked input signal at the previous time step, not at some mysterious event loop iteration.

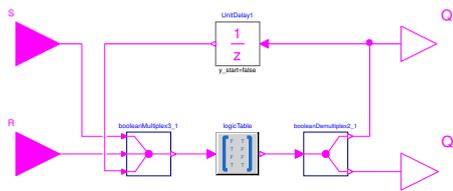


Figure 12: Static RS flip-flop with clock

Since the semantics of the blocks and signals are now identical to their Simulink versions, so are the results: The flip-flop works as expected, even at the “forbidden” state, and the shift register built from it properly shifts its input.

## 5 Conclusions

Modelica’s event system is complicated – to say the least. This can lead to models that behave very differently to what one might expect, especially if one has worked with Simulink before. The main culprit is the pre operator – or corresponding block – that is not defined in a strictly temporal sense, but relates to a previous iteration of the event loop.

The good news is that the Synchronous library and its underlying Modelica constructs add just the kind of temporal semantics that one needs e. g. for the modeling of flip-flops or state machines. This could be used to complement the Digital library with still missing

standard flip-flop components.

But the crucial point to bear in mind is: Even with today’s sophisticated graphical simulation tools modeling is more than connecting blocks from a library! Without a thorough understanding of the underlying mechanisms one is on murky ground, and simulation results may be correct just by pure luck.

## Acknowledgements

The author is thankful to Fabian Köslin from Dassault Systemes for helpful discussions about the intricacies of the Modelica event system.

The author is grateful for the hospitality extended to him by Tom Schramm and his colleagues at the Department of Geomatics, HCU Hamburg.

## References

- [1] P. A. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3*. Wiley & Sons, New York, 2015.
- [2] The MathWorks. *Stateflow: Model and simulate decision logic using state machines and flow charts*. Online: <http://www.mathworks.de/products/stateflow/> (called 2015-11-25).
- [3] H. Elmqvist, F. Gaucher, S. E. Mattsson, F. Dupont. *State Machines in Modelica*. Proc. 9th Int. Modelica Conference, Munich, p. 37-46, 2012.
- [4] The MathWorks. *SimEvents: Model and simulate discrete-event systems*. Online: <http://www.mathworks.de/products/simevents/> (called 2015-11-25).
- [5] V. Sanz, A. Urquia, S. Dormido. *Parallel DEVS and Process-Oriented Modeling in Modelica*. Proc. 7th Int. Modelica Conference, Como, p. 96–107, 2009.
- [6] Modelica Association. *The Modelica Language Specification Version 3.3 Revision 1, July 11, 2014*. Online: <https://modelica.org/documents/>

ModelicaSpec33Revision1.pdf (called 2015-11-27).

- [7] H. Elmqvist, M. Otter, S. E. Mattsson. *Fundamentals of Synchronous Control in Modelica*. Proc. 9th Int. Modelica Conference, Munich, p. 15-26, 2012.
- [8] M. Otter, B. Thiele, H. Elmqvist. *A Library for Synchronous Control Systems in Modelica*. Proc. 9th Int. Modelica Conference, Munich, p. 27-36, 2012.