Enforcing Reliability of Discrete-Time Models in Modelica

Sébastien FURIC
LMS Imagine
7, place des minimes 42300 Roanne
sebastien.furic@lmsintl.com

Abstract

Modelica models involving discrete-time aspects may lead to surprising results due to the way events are currently handled in the language. Indeed, *simultaneity* is interpreted as *synchronism* (see [2] for details) and, as a consequence, two unrelated sources of events may interfere in unexpected ways.

In this paper, we present minimal examples of models that exhibit unexpected or surprising results, then we explain the general causes of such behaviors and propose to introduce the notion of *clock* in the language to solve the issues. In contrast to [1] and [2], we focus here on models resulting from the *composition* of other models: we aim at showing that the current discrete-time theoretical model of Modelica is not robust with respect to model composition. For the final user, it means that it is generally not possible to build reliable models involving discrete-time aspects by simply connecting generic library models: manual adjustments are often required to obtain the expected behavior[1].

Keywords: discrete-time modeling; clock calculus

1 Introduction

Modelica has been designed to primarily solve *continuous-time systems* of differential and algebraic equations. Unfortunately, discrete-time aspects have not been considered with the same level of interest. The result is that essential features of synchronous languages (e.g., Signal, Lustre, Esterel) are not present in Modelica today. Consider for instance the following Modelica model:

```
model M
  Real x, x_dot;
  Integer count;
initial equation
  x = 0;
  x_dot = 1;
  count = 0;
equation
  x_dot = der(x);
  der(x_dot) = -x;
  when
    { x > 0.5, sin(time) > 0.5 }
  then
    count = pre(count) + 1;
  end when;
end M;
```

According to the Modelica specification, that model is correct, so we can try to simulate it. One may wonder which is the value of count at the end of a simulation performed between 0 and 100 seconds for instance. Quite surprisingly, the Modelica specification does not give the answer: any value between 16 and 32 is possible even if — it is the case here — every event can be numerically detected with accuracy so that none is lost due to the limits of time tolerance of the solver[2]. Indeed, the *when clause* that is used to update `count` is activated by two unrelated sources of events (put between curly braces in Modelica syntax) that may accidentally be seen as synchronous during simulation, as explained in [2]. Actually the final value of `count` depends on:

- the "quality" of the translator implementation
- the kind of solver eventually required to solve the final system
- the parameters of the solver, in case a solver is necessary.

In this paper, we aim at explaining the consequences of such a design choice in terms of reliability and reusability of models. The paper is organized as follows: section 2 gives an analysis of the prob-

1 This also begs for a related question, which is: how can we *know* that our models actually require adjustments!

2 The purpose of this paper is not to discuss numerical solver issues, in particular event detection in case of non-trivial continuous-time systems: we will only focus on discrete-time aspects. The introductory example is presented with the hope that it will help readers with physical background to get a feeling of what discrete-time issues are.

lem; section 3 introduces the proposed solution; section 4 shows how to transpose the solution in the context of the Modelica language; section 5 presents an example of application; section 6 gives a conclusion.

2 Analysis of the problem

The problem with the model above comes from the fact that discrete-time aspects are somewhat "approximated" in Modelica's semantics: it is not possible to know for sure, in that model, whether both sources of events corresponding to `x > 0.5` and `sin(time) > 0.5` are synchronous or not. It is not even possible to know for sure, in case they are seen as synchronous by the simulator at the beginning of a simulation, whether they will remain synchronous until the end or not. Indeed, according to the Modelica specification, events instants are "probed" during simulation (only time associated to their occurrence is retained) so deciding whether two events happening at the same measured time are really synchronous (i.e., have the same cause) or whether it is pure coincidence is impossible. Unsurprisingly, this has unfortunate consequences over the design of event-based models in Modelica. Consider for instance the following purely discrete Modelica models:

```
connector Out = output Boolean;

model EventSource "Simple event
source"
  parameter Real t0, T;
  Out out;
equation
  out = sample(t0, T);
end EventSource;

connector In = input Boolean;

model Counter "Simple event
counter"
  parameter Integer n;
  In ins[n];
  Integer count;
initial equation
  count = 0;
equation
  when ins then
    count = pre(count) + 1;
  end when;
end Counter;
```

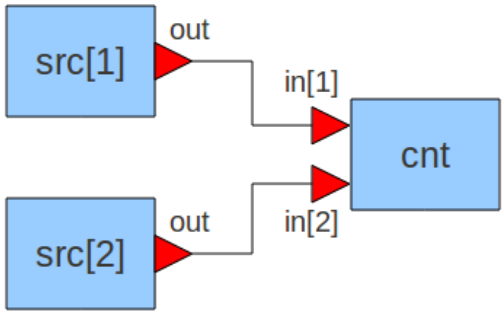

Figure 1: A simple test model

Instances of `EventSource` emit events[3] via their unique output port and instances of `Counter` count the number of events received via their input ports. Consider the following model built upon `EventSource` and `Counter` (Figure 1 gives its graphical representation):

```
model TestCounters
  EventSource src[2](t0 = { 0, 3 },
T = { 1, 2 });
  Counter cnt(n = 2);
equation
  connect(src.out, cnt.ins);
end TestCounters;
```

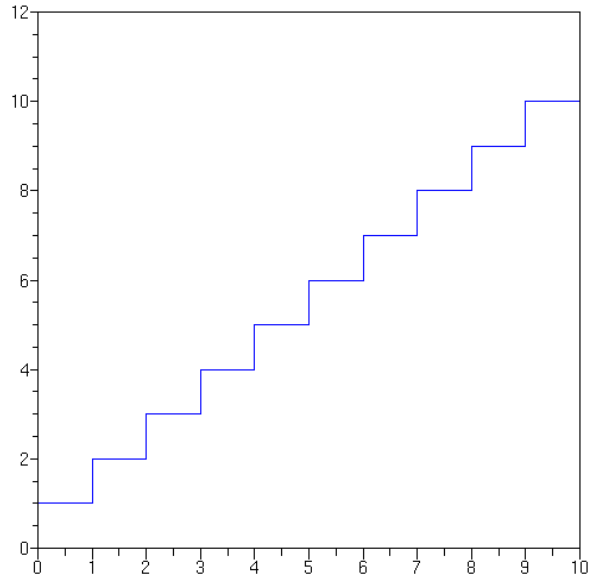

Figure 2: Simulation results of model in Figure 1

Simulation of an instance of `TestCounters` between 0 and 10 seconds gives a rather surprising result (see Figure 2). Indeed, between 0 and 10 seconds the sources emit a total of 11 + 4 events but

3 Boolean values in reality, events are not explicitly emitted.

only 11 of them are “saw” by the instance of `Counter`. That result is explained by Modelica's way of handling discrete events. Indeed, some events are “lost” because, as explained above, only the *measured time* of events matters in Modelica, so two events happening at the same time cannot be distinguished: the simulator does not know whether they have been emitted by the same source connected to both input ports of the instance of `Counter` (as in Figure 3) or by two distinct sources (as in Figure 1).
One may wonder why is it not possible, by default, to consider every local port of a model like `Counter` as an independent local source of events: indeed, in the case of `TestCounters`, it would give the correct answer for `count`. But consider the following model which graphical representation is given in Figure 3:

```
model TestCounters2
  EventSource src(t0 = 0,  T = 1);
  Counter cnt(n = 2);
equation
  connect(src.out, cnt.ins[1]);
  connect(src.out, cnt.ins[2]);
end TestCounters2;
```

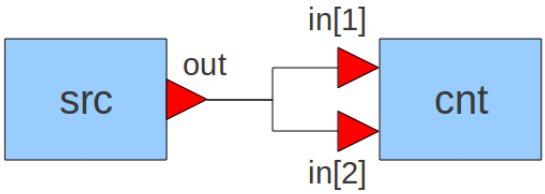

Figure 3: Another simple test model

In any instance of that model, if every port of `cnt` would be considered as a local source of events then twice the correct number of events would be found since there is only one real source of events (with duplicated outputs). One may notice that Modelica's default behavior would lead to the correct result (by accident, however) in that very special situation.

Going back to the original model, one way to avoid the event loss problem in Modelica would be to associate one “subcounter” per input port and to sum the results into the global counter `count`, as in:

```
connector In = input Boolean;

model ImprovedCounter
  parameter Integer n;
  In ins[n];
  Integer count;
protected Integer subcount[n];
initial equation
  subcount = zeros(n);
  count = 0;
equation
  for i in 1 : n loop
    when ins[i] then
      subcount[i] =
pre(subcount[i]) + 1;
    end when;
  end for;
  when ins then
    count = sum(subcount);
  end when;
end ImprovedCounter;
```

However, this solution is far more space- and time-consuming than the original `Counter` model (because a number of additional state variables proportional to the number of listened sources has to be declared and the whole sum of subcounters has to be recomputed each time an event is detected on any input port). Also, that new solution still fails to count the correct number of events in case of a configuration like the one in Figure 3. We may even want to consider configurations like the one in Figure 4, where neither `Counter` nor `ImprovedCounter` would give the correct answer.

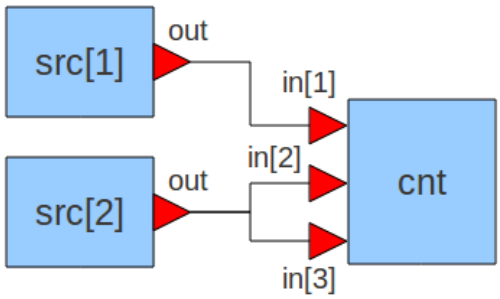

Figure 4: A slightly more complex version of previous test models

A last remark can be made regarding correctness of models. Going back to the first test model, we managed to correct it by providing `ImprovedCounter` (an *adapted version* of `Counter`) to circumvent the issue with simultaneous events. It is important to notice that the correction was possible because we *knew* that our original model had problems with respect to event handling. But in real-world situations, where correctness of models in not known *a priori*, Modelica compilers will not be able to detect such errors since, as shown above, the Modelica language itself does not retain the required information.

As a result, users will have to determine by hand whether their models are correct or not. Of course the task is impossible to complete as far as models get too big or contain encrypted parts for instance.

We conclude from those observations that Modelica needs some improvements to enable the definition of reliable models involving discrete events. The following section explains how that can be achieved. The proposal is based on a preliminary work by INRIA and LMS Imagine in the course of the SimPA 2 project ([1], [2]).

3 Proposal to enable the definition of reliable discrete models in Modelica

3.1 Introduction to clocks and signals

The most important feature of synchronous languages that is currently missing in Modelica is *clocks*. In the Signal language ([3]) clocks give *logical instants* at which signals are said to be *present*, i.e. instants at which values of signals are accessible. Signals sharing the same clock are said to be *synchronous* (their values are present at the same logical instant). Clocks give the *domain* of signals, and types (e.g., Boolean, Integer, Real, etc. in Modelica) give their *codomain*. Consider the following Modelica program:

```
model M
  Integer count;
initial equation
  count = 0;
equation
  when sample(0, 1) then
    count = pre(count) + 1;
  end when;
end M;
```

Interpreted in terms of clocks and signals, this program would define the *discrete-time signal* `count`. One way to see `count` would be as a mapping from events to values (Event is the set of all events):

$$\begin{aligned} \texttt{count}: \quad & \text{Event} \longrightarrow \text{Integer} \\ & e_0 \longmapsto 1 \\ & e_1 \longmapsto 2 \\ & e_2 \longmapsto 3 \\ & \dots \end{aligned}$$

Of course, we would also need to associate a "physical time" with each event, as required by the definition of `sample()`:

$$\begin{aligned} & e_0 \longmapsto 0.0 \\ & e_1 \longmapsto 1.0 \\ & e_2 \longmapsto 2.0 \\ & \dots \end{aligned}$$

It is fundamental to notice that the mapping from events to physical time is not a bijection: two distinct events may be associated with the same physical instant, in which case those events are said to be *simultaneous*. We saw that in Modelica there is no way to tell whether two simultaneous events have the same origin since we only look at physical time. By looking at logical instants, we have a more accurate view of the flow of events: that is the basis of the *synchronous approach* to event handling.

3.2 Why do clocks and signals solve the issues

Let's consider an expression like `sample(t0, T)`. Interpreted in terms of signals and clocks, it would represent a sequence of *fresh events*, each of them mapped to physical instants so that e_k ($k \geq 0$) maps to $t0 + kT$. Consider the following program:

```
when sample(0, 1) then
  count = pre(count) + 1;
end when;
```

We say that the *when clause* above is *activated* at each logical instant yield by the sample construct, which defines a clock, and that `count` *inherits* that clock: `count` causally depends on the sample construct that is used to activate the equation.

We now introduce the notion of *clock union*, as in:

```
when { c1, c2, ... } then
  count = pre(count) + 1;
end when;
```

`{ c1, c2, ... }` represents the union clock of `c1`, `c2`, etc. The set of events emitted by that clock is the union of the set of events emitted by each clock used to compose it. And since in our interpretation events can be distinguished one from each other we can, contrary to Modelica with its current interpretation, compute the union accurately:

- without accidentally forgetting any element (as illustrated in `TestCounters` above)
- without accidentally counting the same element twice (as illustrated in `TestCounters2` above).

Of course, taking clocks into account requires a less naive compilation process than those currently implemented in Modelica compilers. In the next sections we describe the steps required to transform a synchronous language program into efficient compiled code.

3.3 Considerations about the compilation of synchronous programs

Take for instance the model class `TestCounters` defined above. If we instantiate it, we get the following flat Modelica program (where connection equations have been replaced by their contribution to the final system of equations):

```
model FlatTestCounters

  // variables introduced by the
first event source
  parameter Real src[1].t0,
src[1].T;
  Boolean src[1].out;

  // variables introduced by the
second event source
  parameter Real src[2].t0,
src[2].T;
  Boolean src[2].out;

  // variables introduced by the
counter
  Boolean cnt.ins[1], cnt.ins[2];
  Integer cnt.count;

initial equation

  // initial equations introduced
by the counter
  src.count = 0;

equation

  // equations introduced by the
first event source
  src[1].out = sample(src[1].t0,
src[1].T);

  // equations introduced by the
second event source
  src[2].out = sample(src[2].t0,
src[2].T);

  // equations introduced by the
counter
  when { cnt.ins[1], cnt.ins[2] }
then
    cnt.count = pre(cnt.count) + 1;
  end when;

  // expanded connection equations
  cnt.ins[1] = src[1].out;
  cnt.ins[2] = src[2].out;

end FlatTestCounters;
```

For the sake of conciseness, we will consider a simplification of the previous program[4] that still contains the essential constructs:

```
model ShortFlatTestCounters
  parameter Real t0[1], T[1],
t0[2], T[2];
  Boolean c[1], c[2];
  Integer count;
initial equation
  count = 0;
equation
  c[1] = sample(t0[1], T[1]);
  c[2] = sample(t0[2], T[2]);
  when { c[1], c[2] } then
    count = pre(count) + 1;
  end when;
end ShortFlatTestCounters;
```

That program defines two asynchronous event sources since we consider that each sample construct introduces its own sequence of fresh events. It follows that `c[1]` and `c[2]` do not have any event in common and then that the union clock `{ c[1], c[2] }` is irreducible. The *clock calculus* we will propose below will have to reflect those considerations, so that a compiler implementing it will automatically derive canonical representation of clocks, as we currently do by hand in this simple case. The constraints in the above program are finally equivalent to this pseudo-code:

```
c[1] = sample(t0[1], T[1]);
c[2] = sample(t0[2], T[2]);
when c[1] then
  count = pre(count) + 1;
end when;
when c[2] then
```

4 Obtained by removing alias variables and associated equations, and renaming remaining variables

```
  count = pre(count) + 1;
end when;
```

Notice that we now have two concurrent equations defining `count` (which is explicitly forbidden in Modelica) but since both when clauses are guaranteed to be activated asynchronously thanks to our interpretation of sample's properties, there is actually no possible conflict[5].

At this point, an important remark has to be made regarding *determinism*. Indeed, with the semantics proposed in this report, we accept to consider that a connection of two independent event sources is acceptable because we force *interleaving of events* so that two simultaneous events cannot be synchronous if they don't have the same origin. However, the order into which simultaneous events will be treated at runtime is purposely left unspecified. From a control engineer perspective that choice seems rather surprising, especially for a language that could eventually be used to design control systems, where determinism is a fundamental aspect to consider. But from a physical modeling point of view, non-determinism is a natural consequence of the physical nature of the world. Since our aim is to design a physical modeling language it seems reasonable to allow some form of non-determinism to avoid rejecting too many programs, especially those that most users having a physical background will consider correct (and which are, given the limits of physics). Notice however that non-determinism can be detected statically by a compiler implementing our proposal: it just requires a stricter criterion to select correct programs (that may be a compiler option). Compared to Modelica in its current state, we offer a way to control determinism so that it fits control or physical needs. Modelica, on the other hand, currently cannot promise anything regarding determinism for the reasons exposed in previous sections.

3.4 Clock calculus

In this section we introduce the elements of our *clock algebra* and give essential rules that govern clock calculus. We propose the following grammar to describe clock expressions (where e represents a term denoting any signal, and b a term denoting any boolean signal):

clock ::=	
never	(empty clock)
| always	(full clock)
| **clock**(*e*)	(clock of *e*)
| **false**(*b*)	(instants at which *b* is false)
| **true**(*b*)	(instants at which *b* is true)
| **initial**	(initialization clock)
| **edges**(*b*)	(instants at which *b* becomes true)
| **sample**(t_0, T)	(sample starting at t_0, with period T)
| *clock* ∨ *clock*	(union of clocks)
| *clock* ∧ *clock*	(intersection of clocks)
| *clock* \ *clock*	(difference of clocks)
| c_1, c_2, ...	(clock variables)

Several comments have to be made:

- we introduce the notion of "full clock", which, in synchronous languages such as Signal, makes no sense (since clocks are discrete). But since here we have to consider continuous-time signals (we want to describe DAE systems among others), we have an implicit maximal clock for any program: the default continuous-time clock, which includes all the instants of the simulation
- the same kind of remark can be made for clock difference: if the first argument is the full clock, then we get the complementary of the second argument as result, which also makes no sense in synchronous languages such as Signal
- **sample**() and **edges**() share the following properties:
  - they are *generative*: they always yield a fresh, pure discrete-time clock
  - two clocks yielded by those constructs are guaranteed to have an empty intersection (which implies for instance that edges(e_1) ∧ edges(e_2) = empty for any e_1, e_2[6])
- **initial** is a special clock that contains only one instant which corresponds to the first simulation instant (i.e., no other instant may happen before this one).

Systems of equations will be described by the following grammar (where $e_i(s_{i_1}, s_{i_2}, \ldots)$ denotes an expression involving signals $s_{i_1}, s_{i_2}, \ldots$,):

system ::=	
null	(system having no constraint)
| *system* || *system*	(parallel composition)
| *system* **when** *clock*	(sampling)
| **let** *c* = *clock* **in** *system*	(let binding)
| $e_1(s_{1_1}, s_{1_2}, \ldots) = e_2(s_{2_1}, s_{2_2}, \ldots)$	(equation)

To illustrate the use of the system description language defined above, let's write the system corresponding to an instance of the model class `Test_Counters` defined above. It gives:

5 Modelica currently has to impose single assignment restrictions precisely because two sources of cannot be statically proven to be asynchronous, as explained before.

6 This is the property we used in section *Considerations about the compilation of synchronous programs* to calculate the union clock.

let *c[1]* = **sample**(*t0[1]*, *T[1]*) **in**
let *c[2]* = **sample**(*t0[2]*, *T[2]*) **in**
count = 0 **when initial** ||
count = **pre**(count) + 1 **when** *c[1]* ∨ *c[2]*

We saw in previous section that it was also possible to write it as follow:

let *c[1]* = **sample**(*t0[1]*, *T[1]*) **in**
let *c[2]* = **sample**(*t0[2]*, *T[2]*) **in**
count = 0 **when initial** ||
count = **pre**(count) + 1 **when** *c[1]* ||
count = **pre**(count) + 1 **when** *c[2]*

Notice that clock expressions should be reduced to a *canonical form* so that single assignment rule can be checked statically to avoid current Modelica issues with respect to discrete-time modeling. We then require that a compiler will have to perform *full reduction* of clock expressions at compile time by eliminating local clock variables (i.e., rewriting **let** *v* = … **in** … terms) and replacing **clock**(…) terms by their actual value. This reduction will have to be carried out in parallel with the resolution of the assignment problem attached to equations. Indeed, in contrast to synchronous languages such as Signal, we have to deal with the acausal nature of the language: given an equation, it is not possible to tell which signal(s) it defines without performing a *global causality analysis under clock constraints* (we require however *schedulability* of clock constraints). So the algorithm we propose is:

1. The initial contextual clock is always
2. Perform partial resolution of the assignment problem for any equation that is not constrained by a when construct, in order to find defined signals, which are added to current context
3. Pick any when construct which activation clock either depends on past (i.e., "pre signals") or on signals that belong to current context; make that activation clock the current contextual clock and go to step 2

If some subsystems remain unselected at the end of the algorithm, the whole system is:

- not schedulable, if any subsystem contains a clock which depends on the signals it actually defines
- under-constrained, if the assignment problem failed for the remaining equations
- over-constrained, if any subsystem only constraints signals that have already been determined.

An additional check has to be performed in order to validate the whole system: the clock of any "pre signal" should be proven equal to the clock of the signal itself. The clock of a given signal can be determined by "summing" the clocks of when constraints that define that signal (union of clocks). The clock of a "pre signal" is the union of the minimal clocks at which that signal is required to be present (i.e., immediate contextual clocks during clock calculus). By "subtracting" (clock difference) both resulting clocks we have to find empty. Notice that the problem is decidable since we required full reduction of clock expressions to canonical form. That algorithm not only validates the original system, it also returns its *constrained dataflow representation* which can be used to generate efficient code.

4 Application to Modelica

One can notice that Modelica's sample expressions yield boolean values that are quite exclusively used to activate discrete equations. The reason is that in Modelica when clauses require a test to be performed to activate/deactivate associated equations. But most of the time, that test is useless: it only makes sense to "activate equations from the outside". Indeed, a when clause which activation constraint only depends on pure events (as ideally generated by sample) would not need to check anything: the activation/deactivation logic would be "lifted" in the control flow. It follows that when clauses can be made more general and efficient by only depending on clocks instead of boolean signals[7]: pure event-based activations do no longer lead to any test in the generated code. That is particularly interesting in presence of external event sources, which in Modelica currently lead to the generation of "event loops" that are extremely resource-consuming (and that eventually require dynamic synchronization with the source). Our proposal avoids the generation of those expensive loops.

In consequence of the above remarks, we propose to equip Modelica with a new type: Clock, the type of clocks (i.e., sequences of logical time events). We also propose to change the semantics of sample expressions, so that they now denote pure event generators[8]. Here is a modification using clocks of the

7 This is the reason why the name "when" has been coined historically in synchronous languages such as Signal.

8 It would be possible to make sample generate booleans *and* events simultaneously to avoid too many compatibility issues, but, since uses of values yielded by sample as regular booleans seems highly suspicious in our opinion, a more restrictive definition of sample would help to find abusive use of event-generating expres-

original model class `TestCounters` defined above:

```
connector Out = output Clock;

model EventSource "Simple event
source"
  parameter Real t0, T;
  Out out;
equation
  out = sample(t0, T);
end EventSource;

connector In = input Clock;

model Counter "Simple event
counter"
  parameter Integer n;
  In ins[n];
  Integer count;
initial equation
  count = 0;
equation
  when ins then
    count = pre(count) + 1;
  end when;
end Counter;

model TestCounters
  EventSource src[2](t0 = { 0, 3 },
T = { 1, 2 });
  Counter cnt(n = 2);
equation
  connect(src.out, cnt.ins);
end TestCounters;
```

As shown in previous section, an instance of the above model can be represented in our intermediate language as:

let *c[1]* = **sample**(*t0[1]*, *T[1]*) **in**
let *c[2]* = **sample**(*t0[2]*, *T[2]*) **in**
count = 0 **when initial** ||
count = **pre**(count) + 1 **when** *c[1]* ∨ *c[2]*

Given a translation from the original Modelica code to our system description language, we can proceed with *static scheduling* of equations by applying the algorithm proposed above. If the system does not contain any implicit variable, the result is simply composed of several sequences of assignments, activated by *primary clocks* (i.e.,source clocks of the system). It is the case in our simple example:

when initial:
count := 0

when sample(*t0[1]*, *T[1]*) ∨ **sample**(*t0[1]*, *T[1]*):
count := **pre**(count) + 1

The above representation of the system reads as follow:

- at initialization, assign 0 to the *program variable* count
- on each activation scheduled by **sample**(*t0[1]*, *T[1]*) ∨ **sample**(*t0[1]*, *T[1]*), assign the last value held by count plus one to count.

We have derived *sequences of assignments* from the functional specification expressed in our system description language.

5 Example

We consider a well-known example in the Hybrid Systems and Control Theory literature (we will use the version presented in [BK09]). Here we consider the system consisting of a water tank, where water arrives at a variable rate $w_i(t) \geq 0$ through one pipe and leaves through another one at the rate controlled by a valve (cf. Figure 5). The output pipe has a maximal throughput capacity C, and the valve position is given by $0 \leq v(t) \leq 1$. Thus, the actual throughput of the output pipe at the moment t is $C \cdot v(t)$. The valve is controlled by a sensor measuring the level l of water in the tank, which aims at keeping this level in a given interval $[L1, L2]$. For simplicity we assume that there is always enough water in the tank to saturate the output pipe and that the incoming flow does not exceed the output pipe's capacity, i.e. $max\ w_i(t) \leq C$.

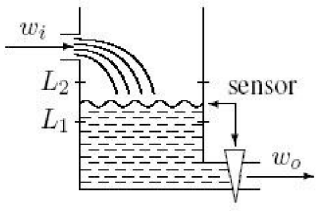

Figure 5: Tank with regulated valve

The transfer function of the complete system has the input space $In = R+$ (incoming flow rate) and the output space $Out = R+ \times +$ (output flow rate and current water level in the tank). This system can be modeled as a composition of three sub-systems (see Figure 6).

sions in existing programs.

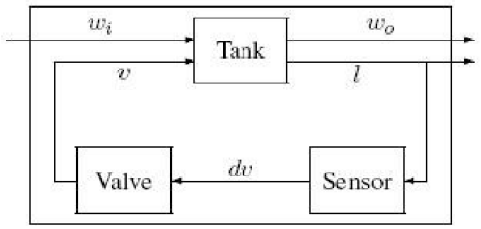

Figure 6: Block-diagram representation of the regulated tank

1. The Tank, taking on input current values of the incoming flow w_i and the position of the valve v and computing the corresponding output flow w_o and water level l from the equations
$$dl = w_i - w_o$$
$$w_o = C \cdot v$$
The corresponding transfer function has the input and output spaces $In_T = R+ \times [0, 1]$ and $Out_T = R+ \times R+$.
2. The Sensor, taking on input the water level l and computing the corresponding valve position adjustment dv from some given equation, e.g.:
$$dv = sign(l - \frac{L1+L2}{2})$$
The corresponding transfer function has the input and output spaces $In_S = R+$ and $Out_S = [-1, 1]$.
3. The Valve, taking on input the adjustment dv and providing on output the corresponding value v.
The corresponding transfer function has the input and output spaces $In_V = [-1, 1]$ and $Out_V = [0, 1]$.

This kind of physical models is straightforward to define as a *continuous-time* Modelica model:

```
// Connector definitions

connector RealInput = input Real;
connector RealOutput = output Real;


// Submodel definitions

model Tank
  parameter Real C;
  RealInput wi;
  RealInput v;
  RealOutput wo;
  RealOutput l;
equation
  wo = C * v;
  der(l) = wi - wo;
end Tank;

model Sensor
  parameter Real L1, L2;
  RealInput l;
  RealOutput dv;
equation
  dv = sign(l - (L1 + L2) / 2);
end Sensor;

model Valve
  RealInput dv;
  RealOutput v;
equation
  /* In order to preserve the range
of v, we
     have to constrain the values
of its derivative */
  der(v) =
    if pre(v) <= 0 then max(dv, 0)
    elseif pre(v) >= 1 then min
(dv, 0)
    else dv;
end Valve;

model Source "Sinusiodal source of
flow"
  constant Real PI = acos(-1);
  parameter Real W0, f;
  RealOutput wo;
equation
  wo = W0 * (0.5 * sin(2 * PI * f *
time) + 0.5);
end Source;

model TankSensorValve "Agregation
of a tank, a sensor and a valve"
  RealInput wi;
  RealInput wo;
  Tank tank(C=10, l(start=1.5));
  Sensor sensor(L1=1, L2=2);
  Valve valve(v(start=0));
equation
  connect(wi, tank.wi);
  connect(tank.l, sensor.l);
  connect(sensor.dv, valve.dv);
  connect(valve.v, tank.v);
  connect(tank.wo, wo);
end TankSensorValve;
```

```
// Example of use

model M "A simple use of above mod-
els"
  TankSensorValve tankSensorValve(
    tank(C=10, l(start=1.5)),
    sensor(L1=1, L2=2),
    valve(v(start=0)));
  Source source(W0=5, f=0.25);
equation
  connect(source.wo, tankSensor-
Valve.wi);
end M;
```

Simulating the above model gives the results in Figure 7.

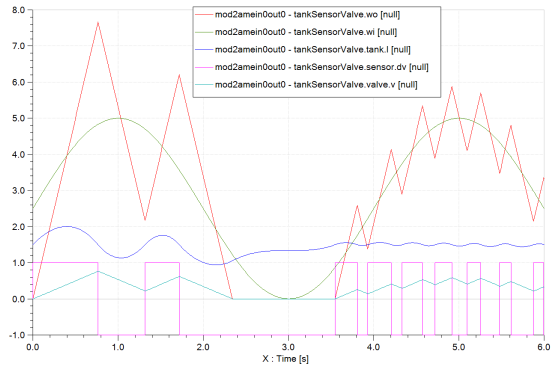

Figure 7: Result of the simulation of a continuous-time model of regulated tank

A *discrete-time* specification of the regulated tank model would offer several advantages over the continuous-time one: it does not require sophisticated solvers to compute simulation results and it may run far faster than the high-fidelity version (the price to pay when using pure discrete-time models is, most of the time, poor accuracy). Here is a new specification in discrete-time Modelica of the submodels needed to build the final model (changes with respect to original version are in red):

```
// Connector definitions

connector RealInput = input Real;
connector RealOutput = output Real;


//useful constants

constant Real MILLI2SEC = 0.001;
constant Real MILLI_PERIOD = 20;
constant Real STRETCH = 400;
constant Real STEP = MILLI_PERIOD *
MILLI2SEC / STRETCH;
```

```
// Submodel definitions

model Tank
  parameter Real C;
  RealInput wi;
  RealInput v;
  RealOutput wo;
  RealOutput l;
equation
  wo = C * v;
  l = pre(l) + (wi – wo) * STEP;
end Tank;

model Sensor
  parameter Real L1, L2;
  RealInput l;
  RealOutput dv;
equation
  dv = sign(l - (L1 + L2) / 2);
end Sensor;

model Valve
  RealInput dv;
  RealOutput v;
equation
  /* In order to preserve the range
of v, we
     have to constrain the values
of its derivative */
  v =
    pre(v) +
    (if pre(v) <= 0 then max(dv, 0)
    elseif pre(v) >= 1 then min
(dv, 0)
    else dv) * STEP;
end Valve;

model Source "Sinusiodal source of
flow"
  constant Real PI = acos(-1);
  parameter Real W0, f;
  RealOutput wo;
/* We explicitly define time as a
discrete signal */
protected Real time;
initial equation
  time = 0;
equation
  when sample(0, STEP) then
    time = pre(time) + STEP;
  end when;
  wo = W0 * (0.5 * sin(2 * PI * f *
time) + 0.5);
end Source;
```

```
model Delay "Ideal delay"
  RealInput x;
  RealOutput zx;
equation
  zx = pre(x);
end Delay;

model TankSensorValve "Agregation
of a tank, a sensor and a valve"
  RealInput wi;
  RealInput wo;
  Tank tank(C=10, l(start=1.5));
  Sensor sensor(L1=1, L2=2);
  Valve valve(v(start=0));
  Delay delay(zx(start=0));
equation
  connect(wi, tank.wi);
  connect(tank.l, sensor.l);
  connect(sensor.dv, valve.dv);
  connect(valve.v, delay.x);
  connect(delay.zx, tank.v);
  connect(tank.wo, wo);
end TankSensorValve;
```

Notice the use of difference equations instead of differential ones and also the use of a when clause that provides the "main clock" exported by the `Source` model class. A new model class is necessary (`Delay`) to break the algebraic loop (controller feedback) by inserting a delay into it, otherwise the model is not schedulable. The `Sensor` model class is kept unchanged: that is not surprising since it only involves algebraic equations. Thanks to activation inheritance, model instances connected to `Source` will be activated by the source's clock. Here is the system of equations resulting from instantiating model class `M` defined above, expressed in our system representation language:

```
m.tankSensorValve.tank.l = 1.5 when initial ||
m.tankSensorValve.tank.wo = 10 *
m.tankSensorValve.tank.v ||
m.tankSensorValve.tank.l =
pre(m.tankSensorValve.tank.l) +
(m.tankSensorValve.tank.wi –
m.tankSensorValve.tank.wo) * 5e-5 ||
m.tankSensorValve.sensor.dv =
sign(m.tankSensorValve.sensor.l – 1.5) ||
m.tankSensorValve.valve.v = 0 when initial ||
m.tankSensorValve.valve.v =
pre(m.tankSensorValve.valve.v) + (if
pre(m.tankSensorValve.valve.v) <= 0 then
max(m.tankSensorValve.valve.dv, 0) elseif
pre(m.tankSensorValve.valve.v) >= 1 then
min(m.tankSensorValve.valve.dv, 0) else
m.tankSensorValve.valve.dv) * 5e-5 ||
m.tankSensorValve.delay.zx = 0 when initial ||
m.tankSensorValve.delay.zx =
pre(m.tankSensorValve.delay.x) ||
m.tankSensorValve.wi = m.tankSensorValve.tank.wi ||
m.tankSensorValve.tank.l = m.tankSensorValve.sensor.l
||
m.tankSensorValve.sensor.dv =
m.tankSensorValve.valve.dv ||
m.tankSensorValve.valve.v =
m.tankSensorValve.delay.x ||
m.tankSensorValve.delay.zx =
m.tankSensorValve.tank.v ||
m.tankSensorValve.tank.wo = m.tankSensorValve.wo ||
m.source.time = 0 when initial ||
m.source.time = pre(m.source.time) + 5e-5 when
sample(0, 5e-5) ||
m.source.wo = 5.0 * (0.5 *. sin(1.5707963268 *
m.source.time) + 0.5) ||
m.source.wo = m.tankSensorValve.wi
```

Sorting that system gives the following assignments:

```
when initial:
  m.tankSensorValve.tank.l := 1.5
  m.tankSensorValve.valve.v := 0
  m.tankSensorValve.delay.zx := 0
  m.source.time := 0

when sample(0, 5e-5):
  m.source.time := pre(m.source.time) + 5e-5
  m.source.wo :=
    5.0 * (0.5 *. sin(1.5707963268 * m.source.time) +
    0.5)
  m.tankSensorValve.wi := m.source.wo
  m.tankSensorValve.tank.wi := m.tankSensorValve.wi
  m.tankSensorValve.delay.zx :=
    pre(m.tankSensorValve.delay.x)
  m.tankSensorValve.tank.v :=
    m.tankSensorValve.delay.zx
  m.tankSensorValve.tank.wo :=
    10 * m.tankSensorValve.tank.v
  m.tankSensorValve.tank.l :=
    pre(m.tankSensorValve.tank.l) +
    (m.tankSensorValve.tank.wi –
    m.tankSensorValve.tank.wo) * 5e-5
  m.tankSensorValve.sensor.l :=
    m.tankSensorValve.tank.l
  m.tankSensorValve.sensor.dv :=
    sign(m.tankSensorValve.sensor.l – 1.5)
  m.tankSensorValve.valve.dv :=
    m.tankSensorValve.sensor.dv
  m.tankSensorValve.valve.v :=
    pre(m.tankSensorValve.valve.v) +
    (if pre(m.tankSensorValve.valve.v) <= 0 then
      max(m.tankSensorValve.valve.dv, 0)
    elseif pre(m.tankSensorValve.valve.v) >= 1 then
      min(m.tankSensorValve.valve.dv, 0)
    else m.tankSensorValve.valve.dv) * 5e-5
  m.tankSensorValve.delay.x :=
    m.tankSensorValve.valve.v
  m.tankSensorValve.wo := m.tankSensorValve.tank.wo
```

The sequences of assignments we have obtained above can be used to feed a real-time embedded code generator. It can be noticed that it can be statically reduced to yiel a final code that is both minimal and reliable. This example has shown how Modelica, if equiped with clocks, would lead to reliable and composition-friendly models: we would get the expected behavior by simply connecting generic models, without further adjustments.

6 Conclusions

In this paper, we have shown how the introduction of clocks solves the issues encountered in event-based models written in Modelica. We have also shown that clocks could be harmoniously integrated into the language without compromising simplicity nor expressiveness, on the contrary: models could be made more generic thanks to the modular-friendly aspects of clock calculus (which would help a lot in the design of industrial-strength libraries) and, since it would be possible to express more subtle relationships between event sources, development, debugging and maintainance of models involving discrete-time aspects would be easier.

The Modelica community, in the course of the Modelica 4 design process, is going to consider the problem of synchrony. We hope that modular aspects and expressiveness resulting from the introduction of a full clock calculus will be retained as key features of the new language.

7 Acknoledgements

Many thanks to Ramine Nikoukhah for having pointed out issues related to synchrony and for having been the first to suggest having a look at the concept of clock in the context of Modelica. I would also like to thank Simon Bliudze and Marc Pouzet for interesting discussions about the theoretical aspects of hybrid systems.

References

[1] R. Nikoukhah, *Activation Inheritance in Modelica*, EOOLT, 2008

[2] R. Nikoukhah, S. Furic, *Synchronous and Asynchronous Events in Modelica: Proposal for an Improved Hybrid Model*, 6th international Modelica conference, 2008

[3] A. Benveniste, P. Le Guernic, and C. Jacquemot, *Programming with events and relations: the Signal language and its semantics*, 1991

[4] S. Bliudze., D. Krob, *Modelling of Complex Systems: Systems as dataflow machines*, 2009