Synchronous and asynchronous events in Modelica: proposal for an improved hybrid model

Ramine Nikoukhah[1] Sébastien Furic[2]
[1] INRIA Rocquencourt BP 105, 78153 Le Chesnay Cedex, France
[2] LMS-Imagine, 7 place des Minimes, 42300 Roanne, France
ramine.nikoukhah@inria.fr furic@amesim.com

Abstract

The event synchronism in Modelica has been a subject of contradictory interpretations. An interpretation inspired by Scicos formalism [2] has been shown to provide desirable properties. In this interpretation, all independent events are assumed asynchronous; that includes events generated by the **sample** keywords. But in analogy with the way multi-rate systems are modeled in Simulink, it is desirable also to consider **sample** generated events as synchronous. In this paper, we propose a special treatment for the keyword **sample** to overcome this dilemma.

Keywords: Modelica, Scicos, synchronism, real-time code generation

1. Introduction

In [1], it is argued that all Modelica events should be considered asynchronous unless they are derived explicitly from a single event. This is in contrast to Dymola's implementation where all events are considered potentially synchronous, by default. In Dymola, simultaneity is interpreted as synchronism. In [1] it is shown that the asynchronous point of view not only leads to the generation of more efficient code but it can also allow for separate compilation of isolated modules.

It may be argued that with the asynchronous point of view, non-deterministic behavior becomes an issue. But the problem of non-determinism here is not worse than in the fully synchronous context because the reason for non-determinism in hybrid systems is the finite precision of the numerical solver. Indeed, it is not more nondeterministic to assume that two events: **time** > 3 and x < 2, where x is a continuous time variable, are asynchronous than assuming that they are potentially synchronous.

Asynchronous means that even if the two events occur (in theory) at exactly the same time, one is considered to occur just before or after. In the synchronous context, the formalism considers also the case of the two events happening simultaneously and treats it differently. However in practice, since events such as x < 2 are detected by the zero-crossing mechanism of the numerical solver, there is very little chance that two events be detected simultaneously even if theoretically they are simultaneous. So in the synchronous context, the non-determinism is even worse: not only there are three possible outcomes in the presence of two zero-crossing events but the user is lead to believe that it can count on simultaneous detection when in most cases the result is completely unpredictable.

The **sample** generated events, even though not produced by the zero-crossing mechanism during the simulation, should naturally be considered as independent and thus asynchronous as well. But this goes against the usual practices in Modelica where synchronism is often implicitly assumed. In this paper we propose a very special interpretation of the **sample** keyword which not only leads to models in accord with our asynchronous framework but assures synchronism among **sample** generated events.

2. Asynchronous framework

In the asynchronous interpretation of the Modelica specification, two events are considered synchronous only if they can be traced back to a single event source. For example in the following model:

```
when sample(0, 1) then
  d = pre(d) + 1;
end when;
when d > 3 then
  a = pre(a) + 1;
```

```
end when;
```

the event d > 3 is synchronous with the event **sample**(0, 1). The former is the source of the latter. But in

```
der(x) = x ;
when sample(0, 1) then
  d = pre(d) + 1;
end when;
when x > 3 then
  a = pre(a) + 1;
end when;
```

the two events are not synchronous. There is no unique source of activation at the origin of these events. So these events are considered asynchronous even if the two events are activated simultaneously; even if we can prove mathematically that they always occur simultaneously.

The basic assumption is that events detected by the zero-crossing mechanism of the numerical solver (or an equivalent mechanism used to improve performance) are always asynchronous. So even if they are detected simultaneously by the solver, by default they are treated sequentially in an arbitrary order.

3. Special case of sample construct

Under the asynchronous assumption, and by treating the **sample** keyword as a macro, the following program:

```
model M
  Boolean b;
  ...
equation
  b = sample(0, 1);
  when f(b) then
    ... g(b)...
  end when;
  ...
end M;
```

can be expanded as follows:

```
model M
  discrete time Integer k(start=0);
  Boolean b;
  ...
equation
  when time >= k then
    k = pre(k) + 1;
  end when;
  b = false;
  when change(k) and f(true) then
    ... g(true)...
  end when;
  ...
end M;
```

This means that we are lead to assume that different **sample** statements generate asynchronous events (we also lose periodicity information contained in the arguments of the **sample).** For example, in the model:

```
when sample(0, 1) then
  b = a;
end when;
when sample(0, 1) then
  a = b + 1;
end when;
```

the variables a and b are evaluated in an arbitrary order and no algebraic loop is detected..

Dymola on the other hand assumes that all events are synchronous. In particular it assumes that all the equations in both **when** clauses in this example may have to be satisfied simultaneously. That is why Dymola finds an algebraic loop in this example.

This seems reasonable; however Dymola also finds an algebraic loop in:

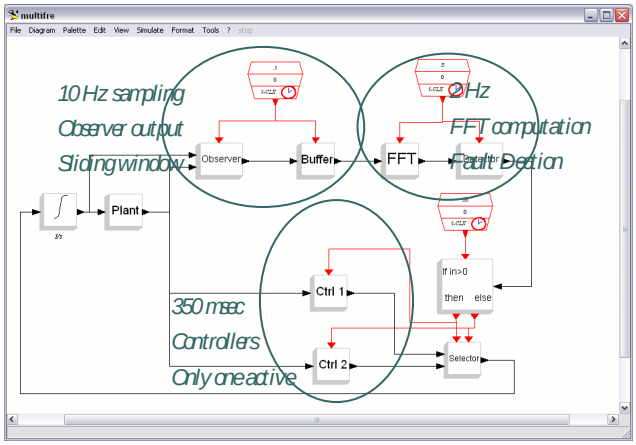
```
when sample(0, 1) then
  b = a;
end when;
when sample(0.5, 1) then
  a = b + 1;
end when;
```

when clearly no algebraic loop exists in this model.

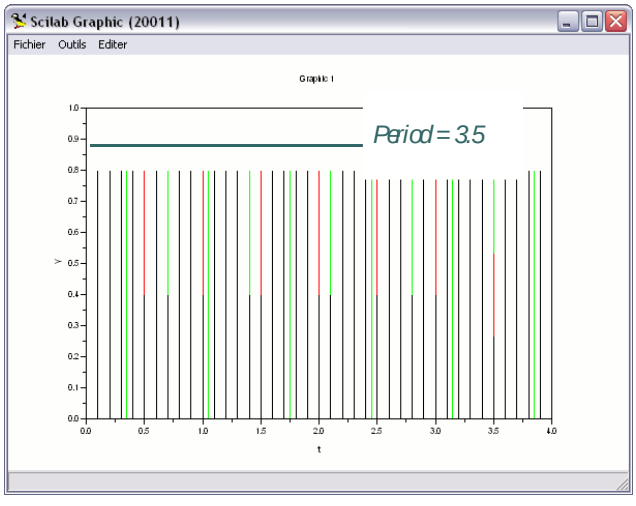
4. Periodicity information

The periodicity information may not be very useful for simulation but it is precious for real-time code generation. It is a lot easier to generate embedded code for a discrete-time system when the system is periodic and all the timing information is available

during the code generation process. Consider for example the following system, which represents a continuous time plant with a discrete-time failure detector and a reconfigurable controller. Different components of the detector/controller mechanism run at different frequencies; we say then that the system is multi-rate.



Consider now the problem of hard real-time code generation for this mechanism, which contains three basic frequencies with periods 0.1, 0.5 and 0.35. The events corresponding to these three clocks are synchronized at different time instants. In general there could be 7 different situations for which static code generation must be performed but in this particular case only 5 situations come up. The important information to note here is that the system will function in a fully periodic way and the timing of all the situations can be computed in advance thanks to information on the periods (and offsets if any) of the clocks. It turns out that in this case, the overall period is 3.5; the timings of different event situations are illustrated below:



To model such a system in Modelica, it is common practice to assume synchronism of independent **sample** sources (this is done in particular, by developers of the Modelica Standard Libraries) and represent each clock by an independent **sample** statement.

But in the asynchronous point of view adopted by us, following the replacement of the **sample** macros with the corresponding Modelica code as presented previously, the clocks become asynchronous. In this framework, it is necessary to use a single clock and derive the other clocks by sub-sampling; otherwise the behavior of the system will not correspond to the desired behavior.

5. Synchronous sample

We have seen that on one hand it is desirable to consider all independent events to be asynchronous and on the other hand, it is convenient to force, depending on their arguments, sample generated events as synchronous.

The type of synchronism considered here has nothing to do with the way Dymola enforces synchronism but it is rather close to Simulink's way of handling multi-rate systems and Scicos' **SampleClk** blocks. The idea is to synthesize a basic clock at a precompilation phase so that all the synchronous clocks defined by **sample** statements can be obtained by sub-sampling the basic clock. The computation of the parameters of this basic clock is straightforward, see [3] for details. Here is a simple example:

```
when sample(0, 2) then
  <expr1>;
end when;
when sample(0, 3) then
  <expr2>;
end when;
```

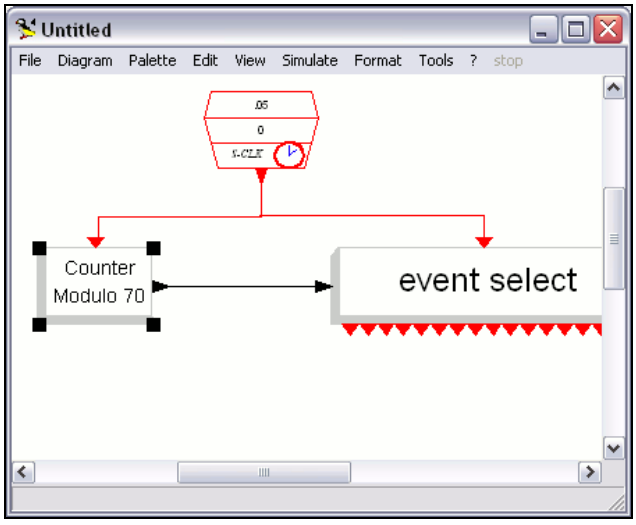
The periods involved in this case are 2 and 3; the period of the basic clock is obtained by computing the greatest common divisor of 2 and 3, which is 1. The overall period in this case is 6, so one way the precompiler could modify the code is as follows:

```
when sample(0, 1) then
  k = mod(pre(k) + 1, 6);
  if k == 0 then
    <expr1>;
    <expr2>;
  elseif k == 2 or k == 4 then
```

<expr1>;
elseif k == 3 **then**
<expr2>;
end if;
end when;

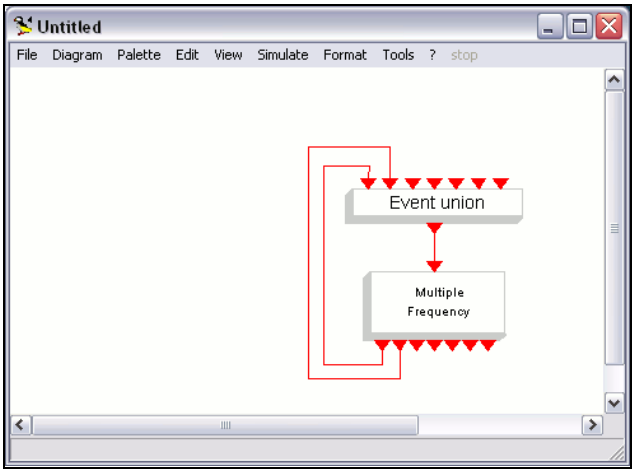
This way of sub-sampling clocks have already been introduced in the Modelica specification (see for example the fast sample, slow sample example on page 81 of [6]).

Going back to our code now, we see that it contains a single **sample** keyword so it is a synchronous code (assuming no **when** constructs are present in the rest of the model). The **sample** construct can now be expanded as previously described. This construction in Scicos is referred to as a periodic construction. For example going back to the detector/controller model from the previous section, the period of the basic clock would be 0.05 (the greatest common divisor of 0.1, 0.5 and 0.35) and the periodic solution would look like the following. Note that the modulo counter counts from 0 to 69 because the period is 3.5 and the basic clock's period is 0.05.



An alternative procedure consists of constructing a vector of time instants where events occur over a single period (in this case [0,2,3,4]) and generate events using independent event sources corresponding to time instances which, modulo 6, are mapped to the elements of this vector.

This construction can be more efficient for simulation but the periodic solution has the advantage of yielding a synchronous code. For the detector/controller example, the non-periodic (asynchronous) construction looks like the following. To keep the diagram simple we have only drawn two of the activation links out of possible 7 (actually 5 in this particular case).



Periodic solutions are also desirable for real-time code generation because the embedded code can be driven by a hardware fixed frequency clock.

6. Implications of the proposal

By admitting that the asynchronous assumption on independent event generators is the correct interpretation, if the special treatment proposed for the **sample** keyword is not used, most discrete-time models in use won't operate properly. The reason is that, despite some recommendations in the language specification, synchronism of independent **sample** sources is assumed by library developers (in particular, by developers of the Modelica Standard Libraries). This practice, mostly driven by analogy with other practices frequently encountered in Simulink-based modeling, conflicts with the asynchronous assumption made in our hybrid model.

To impose synchronism among various discrete-time models, instead of relying on the usage of identical **sample** keywords, synchronization signals should be used. This issue has been discussed in [5] where activation signals have been introduced.

Even though the use of activation signals is a powerful modeling mechanism that should be considered in future Modelica, for the special case of periodic event clocks, the treatment of the **sample** keyword as proposed in this paper avoids the need for there usage. Indeed, by assuming this treatment, backward compatibility for discrete-time models would be guaranteed. The precompilation phase makes the necessary modi-

fications that assure the synchronization of isolated models that are related to each other simply because they include identical **sample** keywords. The backward compatibility is also assured in the case of multi-rate systems (when non identical **sample** keywords are present in the model)..

7. Conclusion

We have proposed to interpret the **sample** keyword in Modelica in a special manner in such a way as to assure synchronism between these keywords yet staying within the asynchronous framework proposed in [1].

The implementation consists of isolating the **sample** keywords in the flat Modelica model. If only one such keyword is present, then it is transformed as explained in the paper. If more than one **sample** is present in the model, the necessary clock computations are performed and all the **sample** constructs are replaced by conditional statements driven by a single **sample**, as illustrated on an example in the paper. This **sample** is then transformed as in the previous case.

Beside backward compatibility (no Modelica model needs to be altered), allowing the usage of independent **sample** keywords to model synchronous multi-rate systems provides valuable information for real-time code generation. However some issues remain to be solved with this approach, especially the way **sample** constructs are translated at the type level (they probably can not be abstracted away during type computation since the public information they carry may interfere with compatibility checks of models).

Using Modelica for real-time embedded code generation has great potentials. Unlike most code generation environments, in Modelica the execution semantics can be broken up into very fine grains and manipulated symbolically. In most cases this means that real-time code can be obtained without having to use preemption. This issue will be examined in a future paper.

References

[1] Nikoukhah, R., "Hybrid dynamics in Modelica: Should all events be considered synchronous", in Proc. EOOLT Workshop at ECOOP'07, Berlin, 2007.

[2] Campbell, S. L., Chancelier, J. Ph., and Nikoukhah, R., "Modeling and Simulation in Scilab/Scicos", Springer, 2005.

[3] Caspi, P., Curic, A., Maignan, A., Sofronis, C. and Tripakis, S., "Translating Discrete-Time Simulink to Lustre", in Embedded Software, Lecture Notes in Computer Science, Springer, 2003.

[4] Otter, M., Elmqvist, H. and Mattsson, S. E., "Hybrid Modeling in Modelica based on the Synchronous Data Flow Principle", CACSD'99, Aug; 1999, Hawaii.

[5] Nikoukhah, R., "Extensions to Modelica for efficient code generation and separate compilation", in Proc. EOOLT Workshop at ECOOP'07, Berlin, 2007.

[6] Modelica Association, "Modelica - A Unified Object-Oriented Language for Physical Systems Modeling. Language Specification, version 3.0", 2007, available from **www.modelica.org**.