Hybrid Dynamics in Modelica: Should all Events be Considered Synchronous

Ramine Nikoukhah

INRIA-Rocquencourt, BP 105, 78153 Le Chesnay Cedex.
Email: ramine.nikoukhah@inria.fr

Abstract. The Modelica specification is ambiguous as to whether all the events are synchronous are not. Different interpretations are possible leading to considerable differences in the ways models should be constructed and compilers developed. In this paper we examine this issue and show that there exists an interpretation that is more appropriate than others leading to more efficient compilers. It turns out that this interpretation is not the one currently adopted by Dymola but it is closely related to the Scicos formalism.

Keywords: Modelica, Synchronous language, Scicos, modeling and simulation.

1 Introduction

Modelica (www.modelica.org) is a language for modeling physical systems. It has been originally developed for modeling systems obtained from the interconnection of components from different disciplines such as electrical circuits, hydraulic and thermodynamics systems, etc. These components are represented symbolically in the language providing the compiler the ability to perform symbolic manipulations on the resulting system of differential equations. This allows the usage of acausal components (equation based) without loss of performance.

But Modelica is not limited to continuous-time models [1]; it can be used to construct hybrid systems, i.e., systems in which continuous-time and discrete-time components interact. Modelica specification [2] tries to define the way these interactions should be interpreted and does so by inspiring from the formalism of synchronous languages. Synchronous languages however deal with events, i.e., discrete-time dynamics. So in the context of Modelica, the concept of synchronism had to be extended to encompass continuous-time dynamics as well. It is exactly this extension which is the subject of this paper.

Scicos (www.scicos.org) is a modeling and simulation environment for hybrid systems. It is free software, included in the scientific software package Scilab (www.scilab.org). Scicos formalism is based on the extension of synchronous languages, in particular Signal [3], to the hybrid environment. The class of models that Scicos is designed for is almost the same as that of Modelica. So it is not a surprise that Modelica and Scicos

have many similar features and confront similar problems. Modelica has many advantages for modeling continuous-time dynamics, especially thanks to its ability to represent models in symbolic form, whereas the Scicos formalism has been specifically designed to allow high performance code generation of discrete-time dynamics.

In this paper, we examine the specification of hybrid dynamics in Modelica and propose an interpretation that is fully compatible with the Scicos formalism. This interpretation, which is not contradictory with the official specification, allows us to obtain an efficient compiler/code generator for Modelica inspired by the Scicos compiler.

Here we start with a flat model (obtained from the application of a front-end compiler), and consider only the problems concerning the design of the phase one of a back-end compiler. This phase breaks down the code into independent asynchronous parts each of which can be compiled separately in phase two. Phase two will be presented in a subsequent paper.

2 Conditioning and Sub-sampling in Modelica

If a model contains no conditioning and all of its parts function at the same rate, then back-end compilation would be a simple task. But in most real life applications, models contain different dynamics resulting from the inter-connection of heterogeneous systems. A model of such a system would often include conditioning and sub-sampling. We use the term conditioning for a change in the model conditioned on the value of a variable (for example *if a>0 then*) and the term sub-sampling for the construction of a new, not necessarily regular, clock from a faster clock.

The *when-elsewhen* and *if-then-else* clauses are the basic language constructs in Modelica for performing conditioning and sub-sampling. The description of the ways these constructs function is ambiguous in the Modelica specification. Comparing with the Scicos formalism, we can consider that Modelica's *if-then-else* clause does conditioning and *when* does sub-sampling. But the situation is somewhat more complex because *when* plays two different roles. And, we need to distinguish these two different types of *when* clauses. But before, we need to examine the notion of synchronism in Modelica.

2.1. Synchronous versus Simultaneous

In our 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.

Our 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. In particular, 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* can be evaluated in any order.

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

To see the way Dymola proceeds, consider the following example:

```
equation
der(x)=1;
der(y)=1;
when (x>2) then
  z=pre(z)+3;
  v=u+1;
end when;
when (y>2) then
  u=z+1;
end when;
```

The simulation shows that the equations (assignments) are ordered as follows:

```
z=pre(z)+3;u=z+1;v=u+1;
```

this means that the content of a *when* clause is split into separate conditional clauses. In stark contrast, in our interpretation of the Modelica specification, the code within an asynchronous *when* clause is treated synchronously and never broken up. Both interpretations are valid and consistent; however our interpretation has many advantages as we will try to show here.

At first glance, the non determinism that may be encountered in our approach when two zero-crossing events occur simultaneously may seem unacceptable. However, treating two simultaneous zero-crossings as synchronous is not a solution because it is not robust. Indeed, when dealing with nonlinear and complex models, there is no guarantee that the numerical solver would detect two zero-crossings simultaneously even if

theoretically they are simultaneous. In general one is detected slightly before or after the other. And in any case, in most cases treating such an accidental synchronism is not of any use for the construction of the model. Even if the model depends for some reason on the simultaneous detection of two events by the solver, a mechanism should be provided by the language to specify explicitly what should be done in that case. One way would be to introduce a *switchwhen* clause [4], which can be used to explicitly specify what equations are activated in every possible case. The possible cases when we have, for example, two zero-crossings are: the first surface has crossed but not the second, the second has crossed but not the first and finally both surfaces have crossed zero together.

Dymola's interpretation imposes constraints, which in most cases are useless. Moreover, when all zero-crossing events are considered synchronous, the complexity of static scheduling increases with the number of zero-crossings. The solution based on using the *switchwhen* clause allows the user to specify explicitly what possible synchronisms must be considered. It turns out that in most cases, no synchronism is to be considered.

2.2. Primary and Secondary *when* Clauses

So far we have seen two types of *when* clauses, or more specifically *when* clauses based on two types of events: events depending on variables evolving continuously in time such as *time>3* or *x<2* where *x* is a continuous variable; and events depending on discrete variables. *when* clauses conditioned on events of the former type are called primary, the latter ones are called secondary.

An event associated with a secondary *when* clause is necessarily synchronous with events associated to one or more primary *when* clauses. These primary clauses are those in which the discrete variables involved in the definition of the event are defined.

But not all *when* clauses can easily be classified as primary or secondary. Let us consider a simple example:

```
when sample(0,1) then
  d=pre(d)+j;
  c=b;
end when;
when time>d then
  b=a;
end when;
```

The question is whether or not the above two *when* clauses are primary or not. Clearly the first one is, but the second hides in reality two distinct *when* clauses that is because the event *time>d* can be activated in two different ways:

- time increases and crosses *d* continuously (zero-crossing event so asynchronous),
- at a sample time *d* jumps, activating the *time>d* condition; this event is clearly synchronized with *sample(0,1)*.

We call such *when* clauses mixed. We handle this situation by implementing the simulation in such a way that *time>d* is activated only when time crosses continuously *d* and placing a duplicate of the content of this *when* where *d* is defined within a condition that guarantees that the content is activated only if *time>d* is activated due to a jump:

```
when sample(0,1) then
```

```
    d=pre(d)+j;
    c=b;
    if ((time>d) and not(time>pre(d))) then
      b=a ;
    end if ;
  end when;
  when time>d then
    b=a;
  end when;
```

The second solution amounts to considering that a clause such as *when c>0* where *c* is a continuous variable is activated only if *c* crosses zero continuously (the way that is detected by zero-crossing mechanisms built into numerical solvers such as LSODAR or DASKR). This seems to be an appropriate way to handle mixed *when* clauses, however to stay compatible with the Modelica specification, at a pre-compilation phase, the content of these clauses must be duplicated as explained above.

Note that the code we obtain after the pre-compilation phase is not correct according to the Modelictcification (because *b* is defined twice). This however is not a problem because this code is only used within the compiler. But in any case, we consider this restriction too restrictive and we think it should be relaxed. This will be discussed later.

There still remains a situation that needs clarification. Consider the following example:

```
discrete Real a(start=0);
Real x(start=0);
equation
der(x)=0;
when x>3 then
  a=pre(a)+1;
end when;
when time>2 then
  reinit(x,x+4);
end when;
```

Here *x* is a continuous variable, but it is also discrete because at time 2 it jumps from 0 to 4 (activation of *reinit*). This jump activates the content of the first *when*. The *reinit* primitive in this case must be considered as a definition of "discrete" *x*, so following the rule discussed previously, the content of the clause *when x>3* must be copied inside the other *when*:

```
discrete Real a(start=0);
Real x(start=0);
equation
der(x)=0;
when x>3 then
  a=pre(a)+1;
end when;
when time>2 then
  reinit(x,x+4);
  if edge(x>3) then
    a=pre(a)+1;
  end if ;
end when;;
```

This transformation is just a special case of the situation we have considered previously. To see this more clearly, note that

```
when time>2 then
  reinit(x,x+4);
end when;
```

should really be expressed as follows:

```
when time>2 then
  x=pre(x)+4;
end when;
```

2.3. Restrictions on the Use of *when* and *if*

Modelica imposes hard constraints on the usage of *when* and *if-then-else* clauses.

In the case of *when*, a variable is not allowed to be defined in two *when* clauses. For example the following code is not allowed in an *equation* section:

```
when sample(0,1) then
    b=pre(b)+1 ;
end when ;
when time>3.5 then
    b=0 ;
end when ;
```

According to the specification, this can lead to a contradiction if the two *when* clauses are activated at the same time. This statement would make sense if the two *when* clauses were synchronous but not in this case. Lifting this restriction, in the case of primary *when* clauses, is without danger and facilitates the task of modeling in many situations. However, it creates an important difference as far some interpretation of the primitive *pre* is concerned. With the current restriction, we are sure that in the following code:

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

pre(b) is the previous value of *b* defined by *b=pre(b)+1* the last time this *when* clause was activated, i.e. one unit of time before. So without even having to examine the rest of the code, we can be sure that *b* indicates the time. This will no longer be true if the constraint is lifted; consider:

```
when sample(0,1) then
  b=pre(b)+1 ;
end when;
when sample(.5,1) then
  b=pre(b)+1 ;
end when;
```

In this case the value of *b* used to update it in each clause is computed by the instruction in the other clause. But this is not a problem as long as the rules are clear.

We thus propose the following modifications: this restriction be lifted for primary *when* clauses and this restriction be lifted in all *when* clauses as long as the equations defining common variables are identical (such identical equations can arise in transformations applied by the compiler which includes duplicating parts of the code). For example for all conditions *c1*, *c2* (synchronous or not), accept:

```
equation
```

```
when c1 then
   b=a;
end when;
when c2 then
  b=a;
end when;
```

The second modification may seem strange. Indeed why would a model contain identical statements in synchronous *when* clauses. The reason is that our Modelica compiler performs a series of transformations each one generating a new Modelica code from a Modelica code in which such a situation may come up (this happens in particular when processing the union of events construct, see Section 2.6). By lifting this restriction, we make sure that we obtain a valid Modelica code at every stage. But a specific test must be applied to the original model to issue at least a warning to the user for such cases.

Another important restriction concerns the use of *elsewhen*. The Modelica specification states that all the branches of a *when-elsewhen* clause must define the same set of variables. We don't believe this constraint is justified. This constraint is probably a consequence of a similar condition on the use of *if-then-else* clauses. Indeed Modelica imposes that the number of equations in different branches of such a clause be identical. This may be acceptable as far as continuous-time variables are concerned[1], but it is not for discrete variables. So we propose to lift this restriction and accept models including for example the following code:

```
equation
when sample(0,1) then
  if u>0 then
    v=1;
  end if;
end when;
```

Normally in Modelica we should have an *else* branch defining *v*. Note that our proposal is not just an editing facility (i.e., a way to avoid writing code which can be added in automatically later); this code is not equivalent to

```
equation
when sample(0,1) then
  if u>0 then
    v=1;
  else
    v=pre(v) ;
  end if;
end when;
```

In the absence of the *else* branch, the variable *v* is sub-sampled. This would not be the case if *v=pre(v)* were used. Even if the simulation result would be the same, the construction by sub-sampling leads to the generation of more efficient code. Lifting this restriction is again important for transformed models. A specific test can be used on the original model to impose the constraint if desired.

[1] Removing the restriction in the continuous case makes it possible to model Simulink's enabled Super Blocks in Modelica.

2.4. Continuous-Time Dynamics

Our objective is to reduce the Modelica code into a number of asynchronous *when* clauses each of which can be treated separately. The continuous dynamics is no exception. What we call continuous dynamics includes everything within the *equation* section but outside *when* clauses. These equations are always active (Scicos terminology) even when a *when* clause is activated. So these equations are synchronous with all the *when* clauses.

The way this situation is handled in Scicos is to introduce a fictitious clock generating a continuous activation signal. To do the same in Modelica amounts to defining a special *when* clause:

```
when continuous then
```

the content of which would be active all the time except at event instances associated to other *when* clauses. Doing so allows us to consider the "continuous" event as asynchronous with the rest and treat this *when* clause as primary. To preserve the dynamics of the original model, the continuous dynamic equations must also be copied inside all the *when* clauses. For example:

```
equation
y=sin(time) ;
der(x)=y ;
when x<.2 then
  a=y ;
end when ;
```

becomes

```
equation
when continuous then
  y=sin(time) ;
  der(x)=y ;
end when ;
when x<.2 then
  y=sin(time) ;
  der(x)=y ;
  a=y ;
end when ;
```

During the simulation, the content of the *when continuous* clause is used to respond to the queries of the numerical solver, and in particular to generate the value of *der(x)* in this case. In other *when* clauses, the equations defining derivative values can be dropped, especially in the explicit case. In the implicit case (DAE case), the computation of the derivatives can be used to help the re-initialization of the solver.

The point to retain from this section is that the clause *when continuous* is primary and that its content can be treated like any other.

2.5. Initial Conditions

In Modelica, variables can be initialized in different ways but in a flat model (after the application of the front end), they should all be grouped within a single when clause:

```
when initial then
```

```
    a=0 ;
    d=3 ;
    …
end when ;
```

This would be a primary *when* clause and would contain the initialization of all discrete and continuous variables.

A *when terminal* clause can similarly be used to specify whatever needs to be done at the end of the simulation.

2.6. Union of Events

The *when* and *elsewhen* clauses can be activated at the union of events. In Modelica the syntax is as follows:

```
when {c1,c2,c3} then
  < eq1 >
  < eq2 >
end when;
```

In this case, *c1, c2, c3* may be synchronous or not. Note that the content of synchronous *when* clauses should not be executed more than once. For example in:

```
when sample(0,1) then
  d=pre(d)+1;
end when;
when {d>2,2*d>4} then
  a=pre(a)+1 ;
end when;
```

a must be incremented only once, passing from zero to one. But in:

```
when sample(0,1) then
  d=pre(d)+1;
end when;
when sample(0,1) then
  e=pre(e)+1;
end when;
when {d>2,e>2} then
  a=pre(a)+1 ;
end when;
```

a is incremented twice (its value must jump from zero to two). But Dymola considers the *d>2* and *e>2* synchronous and increments *a* just once in this case. Similarly in:

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

in Dymola *d>1, e>0* are synchronous (*a* is incremented only once at time 3). As we have said previously, we think that this interpretation must be avoided.

The counterpart of the union of events is the sum of activation signals in Scicos. The two formalisms coincide perfectly in this case.

In one of the early phases of the compilation of Modelica code, we propose the following transformation which removes all event unions. For example the first when clause presented in this section would be transformed as follows:

```
when c1 then
  < eq1 >
  < eq2 >
end when;
when c2 then
  < eq1 >
  < eq2 >
end when;
when c3 then
  < eq1 >
  < eq2 >
end when;
```

This code is correct if we take into account all the modifications suggested previously whether the *ci, i=1,2,3,* are synchronous or not.

3 Back-end Compiler

The back-end compiler can be divided into two phases. The objective of the first phase is to transform the model into one in which all the *when* clauses are primary. This will allow us to generate, in phase two, static code for each one independently of the others.

Consider the following example:

```
when time>3 then
  d=pre(d)+1;
end when;
when d>3 then
  a=pre(a)+1;
end when;
when a>3 then
  b=a;
end when;
```

We want to remove the secondary when clauses. Clearly in this case we have to remove the last two when clauses. We pick one (say when a>3) and copy its content everywhere the variables involved in the definition of the corresponding event are computed. In this case the only variable involved is *a*, which is defined in the second *when* clause:

```
when time>3 then
  d=pre(d)+1;
end when;
when d>3 then
  a=pre(a)+1;
  if edge(a>3) then
    b=a;
  end if ;
```

```
end when;
```

and then

```
when time>3 then
  d=pre(d)+1;
  if edge(d>3) then
    a=pre(a)+1;
    if edge(a>3) then
      b=a;
    end if ;
  end if ;
end when;
```

This example shows how secondary *when* clauses can be removed to obtain a single primary *when* clause at the end. If the model contains more than one primary *when* clause, the procedure would still be the same as illustrated in the following example:

```
when time>2 then
  a=1 ;
end when ;
when time>3 then
  b=pre(b)+1 ;
end when ;
when a>b then
  c=1 ;
end when ;
```

In this case the first two *when* clauses are primary. We now remove the secondary *when*:

```
when time>2 then
  a=1 ;
  if edge (a>b) then
    c=1 ;
  end if ;
end when ;
when time>3 then
  b=pre(b)+1 ;
  if edge (a>b) then
    c=1 ;
  end if ;
end when ;
```

In this example, a variable is defined twice in two different primary (so asynchronous) *when* clauses. Clearly, this is not a problem. But the application of the transformation, can also lead to a variable being defined more than once in the same *when* clause. Let us examine the following example:

```
when time>2 then
  a=pre(a)+1 ;
end when ;
when a>d then
  b=pre(b)+1 ;
end when ;
when {a>2,b>2} then
  n=pre(n)+1 ;
end when ;
```

We start by removing the operator “union of events. Then we remove the secondary clauses as previously described. We obtain (in two steps):

```
when time>2 then
  a=pre(a)+1 ;
  if edge(a>d) then
    n=pre(n)+1 ;
  end if ;
  if edge(a>2) then
    b=pre(b)+1 ;
    if edge(b>2) then
      n=pre(n)+1 ;
    end if ;
  end if ;
end when ;
```

This code, once *edge* replaced with its definition, may seem to be ordered properly and usable as a sequential code. But this is not the case since *n=pre(n)+1*, in some cases, can be executed twice instead of once. As discussed in the previous section, it is allowed to have a variable defined twice synchronously as long as the equations defining it are identical. This is of course the case here (this is the case in general when it happens because of the application of our transformations).The second phase of the compilation will transform the code into a sequential code correctly.

4 Conclusion

We have examined the notion of synchronism in Modelica and have shown that by abandoning the fully synchronous assumption, it is possible to design more efficient compilers without loss of rigor in the language specification. We have done that by proposing a methodology to implement the first phase of a back-end compiler. The second phase, which is closely related to the second phase of the Scicos compiler, will be presented in a future.

References

1 M. Otter, H. Elmqvist, S. E. Mattsson, “Hybrid Modeling in Modelica based on the Synchronous Data Flow Principle”, CACSD’99, Aug; 1999, Hawaii, USA.

2 Modelica Association, “Modelica - A Unified Object-Oriented Language for Physical Systems Modeling. Language Specification, version 2.2”, 2005, available from www.modelica.org/.

3 A. Benveniste, P. Le Guernic, C. Jacquemot.,"Synchronous programming with events and relations : the Signal language and its semantics", Science of Computer Programming, 16, 1991, p. 103-149.

4 R. Nikoukhah, “Extensions to Modelica for efficient code generation and separate compilation”, in Proc. EOOLT Workshop at ECOOP’07, Berlin, 2007.

5 P. Fritzson - “Principles of Object-Oriented Modeling and Simulation with Modelica 2.1”, Wiley-IEEE Press, 2003.

6 S. L. Campbell, Jean-Philippe Chancelier and Ramine Nikoukhah, “Modeling and Simulation in Scilab/Scicos”, Springer, 2005.