Extensions to Modelica for efficient code generation and separate compilation

Ramine Nikoukhah

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

Abstract. In the current Modelica specification, the only admitted external entities are memory-less functions. We propose an extension to allow parts of the model containing internal states, conditioning and discrete dynamics, to be definable as external functions. This opens the door to separate compilation of Modelica code. For this purpose, we introduce the language construct *switchwhen* and the type *Event*. These extensions are directly inspired by the Scicos formalism.

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. Modelica can also be used to construct hybrid systems, i.e., systems in which continuous-time and discrete-time components interact. With that respect, Modelica is similar to Scicos (www.scicos.org), a modeling and simulation environment for hybrid systems. Scicos is included in the free open-source scientific software package Scilab (www.scilab.org). Modelica specifications are provided in an official document; the current version is available in [1].

Unlike Scicos, in which model components are defined as grey-box modules (blocks), the Modelica language requires that the complete model be expressed in the Modelica language (except for simple memory-less external functions). It is currently virtually impossible to isolate a sub-model and compile it separately or develop it in a different language. But this is exactly what is needed when simulation environments are used to design, validate and generate code for real-time applications.

This limitation in Modelica may seem surprising, especially when compared to Scicos. Contrary to Modelica, the Scicos compiler requires only some macroscopic information about each block (direct input/output dependency, presence of state, etc.; see [4] for more information on Scicos block structure). The detail of the algorithm used within each block is irrelevant to the compiler; the internal of each block is fully isolated from the outside. A similar mechanism to isolate a sub-model does not exist in Modelica. In this paper, inspired by the Scicos formalism, we propose a solution to this problem. This is particularly important for applications where real-time code generation is sought. In addition, it allows a harmonious integration of the two environments Modelica and Scicos.

2 Language Extensions

The *when-elsewhen* and *if-then-else* clauses are the language constructs in Modelica for performing conditioning and sub-sampling. We present extensions to these clauses, which not only facilitate programming in Modelica, but also in conjunction with the introduction of a new type, *Event*, allows for module isolation and separate compilation.

2.1. switch and swithwhen

The *switch* construct is a very natural extension of *if-then-else*:

```
switch (n)
 case 1 :
  < eq1 >
  < eq2 >
   …..
 case 2 :
  < eq3 >
  < eq4 >
   …..
 case default:
  < eq5 >
  < eq6 >
   …..
end switch;
```

One and only one case is active depending on the value of the integer *n*. The counterpart in Scicos of this construct is realized with the *ESelect* block.

The *switchwhen* construct generalizes *when-elsewhen*.

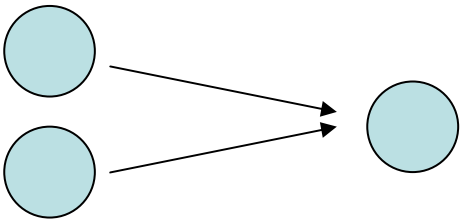
```
switchwhen  {c1,c2,c3}
 case '001' :
  < eq1 >
  < eq2 >
   …..
 case '010' :
  < eq3 >
  < eq4 >
   …..
 case '111':
  < eq5 >
  < eq6 >
   …..
end switchwhen;
```

The content of one and only one case is activated depending on what combination of events *c1*, *c2* and *c3* is generated. For example if events *c1* and *c3* are simultaneously generated but not *c2*, then the case '101' is activated. For this to happen, *c1* and *c2* must be synchronous or be time events which are simultaneously detected (for example by the zero-crossing mechanism of the numerical solver).

In most cases, simultaneous detection of time events (e.g. zero-crossings) needs not be considered as a special case. By default, time events are considered asynchronous, in case of "accidental" simultaneous detection, one event can usually be activated after another (no specified order). For special cases where simultaneous detection does matter, the *switchwhen* construct allows us to take advantage of this additional information. This is useful in some applications as we shall see in the next example:

Example of Usage of *switchwhen*

Consider the problem of contact between three balls:



Ignoring the possibility of simultaneous contact, the dynamics of this system can be modeled as a 1D problem as follows:

```
equation
der(x1)=v1;der(x2)=v2;der(x3)=v3;
der(v1)=0;der(v2)=0;der(v3)=0;
when x3-x1<=1 then
  reinit(v1,pre(v3));
  reinit(v3,pre(v1));
end when;
when x3-x2<=1 then
  reinit(v2,pre(v3));
  reinit(v3,pre(v2));
end when;
```

Here *xi, i=1,2,3,* denote the positions of the balls and we suppose that the balls 1 and 2 situated on left come into contact with the ball number 3 positioned on the right.

The simulation result shows that in the case of simultaneous contact, the model is incorrect. The reason is that after a double contact, after treating one, the jump in the speeds of the balls make it so that the second contact never happens. But this contact is already detected and must be treated. This problem can be avoided by adding a test to make sure that after treating the first contact, the result is taken into account before treating the second:

```
when x3-x1<=1 then
  if v1>v3 then
    reinit(v1,pre(v3));
    reinit(v3,pre(v1));
  end if;
end when;
when x3-x2<=1 then
  if v2>v3 then
    reinit(v2,pre(v3));
    reinit(v3,pre(v2));
  end if;
```

```
end when;
```

This solution is consistent but it is not very flexible. In particular, it does not allow us to specify explicitly what should happen in the case of a double contact. Using *switchwhen*, the dynamics of simultaneous contact can be explicitly expressed:

```
equation
der(x1)=v1;der(x2)=v2;der(x3)=v3;
der(v1)=0;der(v2)=0;der(v3)=0;
switchwhen {x3-x1<=1,x3-x2<=1} then
 case "10":
  reinit(v1,pre(v3));
  reinit(v3,pre(v1));
 case "01":
  reinit(v2,pre(v3));
  reinit(v3,pre(v2));
 case "11":
  <TO DO IN CASE OF SIMULATANEOUS CONTACT>
end switchwhen;
```

We are not suggesting that *switchwhen* is a universal solution to the problem of contacts in mechanics, which is a difficult problem in general. This example was simply chosen to illustrate the usage of *switchwhen* in case of time events.

The most important use of *switchwhen* is for module isolation and applies to the case of synchronous events as we shall see later.

2.2. Type *Event* and Primitive *event*

Currently events in Modelica are coded by Booleans:

e=edge(time>2) ; e=sample(0,1) ;

But these Booleans are not "normal" Booleans: they are of impulsive type. We can consider then that events are coded by this type of Booleans. However, the edge operator, which is usually used to generate events, does not always generate this type of Boolean. For example if *k* is a discrete variable, then

```
when k>0 then
   c=edge(b) ;
```

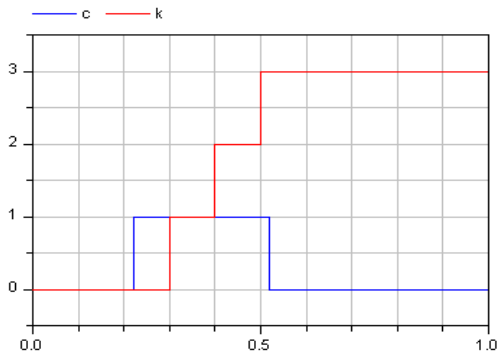
generates a "standard" Boolean. This shows that the *edge* does not always produce an impulsive Boolean (as the name suggests). In fact, *edge(b)* is not just a function of *b*, it depends on the argument of the *when* clause in which it is placed.

We see that there is no real distinction between a "standard" Boolean and one that is used as an event. This can be very confusing as it can be seen in the following example:

```
discrete Real d,k;
Boolean b,c;
```

```
equation
when sample(0,.1) then
  if c then
    k=pre(k)+1;
  else
    k=pre(k);
  end if;
end when;
when sample(.22,.3) then
  b=d>0;
  c=edge(b);
  d=pre(d)+1;
end when;
```

The simulation result over the period [0,1] is given below:



It shows that ***k*** is incremented three times during a single ***edge(b)***! This is not what we would expect and comes from the confusion between Booleans and events.

To remedy this problem, and for more important reasons that we shall see later, we introduce a new type called *Event*. The type *Event* codes the times of events as floats:

```
Event e1(start=0),e2 ;
equation
when e1 then
  e2=e1+1 ;
   ……..
```

In this case, *e2* is an event, delayed by one with respect to *e1*. The Modelica function *sample(0,1)* can be emulated easily using this type:

```
Event e(start=0) ;
equation
when pre(e)  then
  e=pre(e)+1 ;
end when ;
```

We can then use *when e* instead of *when sample(0,1)*. Note that to realize *sample(t0,T)*, we should proceed as follows to avoid accumulation of numerical errors:

```
Event e(start=t0) ;
discrete Integer k(start=0) ;
equation
when pre(e)  then
```

```
  k=pre(k)+1 ;
  e=k*T+t0 ;
end when ;
```

Events can also be generated from zero-crossings. We need a mechanism to generate such *Events*. We have seen that the *edge* operator is not fully appropriate, that is why we propose a new operator: *event*.

```
Event e1,e2;
……
equation
der(x)=sin(x);
e1=event(x>.2) ;
when e1 then
  d=pre(d)+1 ;
  e2=event(d>4);
end when ;
when e2 then
  <xxx>
end when ;
```

Note that the two *event* operators in the above code both generate *Events*, but the mechanism by which they do it is very different. The first one is used outside a *when* clause, so it is realized in the compiler/simulator by the zero-crossing detection mechanism of the numerical solver. The second on the other hand is inside a *when*, it is synchronous and will be removed in the compilation phase. In this case this is done as follows:

```
Event e1;
……
equation
der(x)=sin(x);
e1=event(x>.2) ;
when e1 then
  d=pre(d)+1 ;
  if (d>4) and not (pre(d)>4) then
    <xxx>
  end if ;
end when ;
```

The use of the type *Event* clarifies the situation to a point that we propose that only *Events* be allowed as arguments of *when* clauses[1]. For example the 3 ball problem introduced previously would be expressed as follows:

```
equation
der(x1)=v1;der(x2)=v2;der(x3)=v3;
der(v1)=0;der(v2)=0;der(v3)=0;
E1= event(x3-x1<=1);
E2=event(x3-x2<=1);
switchwhen {E1,E2} then
 case "10":
 ….
```

[1] This can be done gradually to reduce the problems of backward compatibility.

3 Applications

In this section we examine the applications of using the proposed extensions.

3.1. Compiler/Simulator Simplification

The ability to manipulate event times explicitly simplifies model construction. In particular there is no need to use artificial tests against time. For example, consider the problem of modeling the propagation delay in a digital circuit, which requires a variable dependent event delay. This type of delaying operation can be realized as follows:

```
when time>c_time then
  d_time=c_time+u;
end when;
when time>d_time then
  …..
```

But using *Event*, the code can be made simpler:

```
when c_time then
  d_time=c_time+u;
end when;
when d_time then
  …..
```

But representing events explicitly, also simplifies the job of the compiler. In particular, the compiler no longer needs to "figure out" what "tests" are simple enough to be implemented without using the solver's zero-crossing mechanism.

Another simplification comes from the canonical representation of the model when *Events* are explicitly identified (declared) and used in conditioning of the *when* statements. Consider the following example:

```
Event e1,e2,e3,…;
…….
equation
……
e1=event(…
when initial then
   ….
end when;
when e1 then
  k=pre(k)+1;
  e2=event(k>1);
  ….
when e2 then
  e3=time+1;
  ….
end when;
  …..
```

In this model, we can readily identify the *Events* and their types. Clearly *e1* and *e3* are asynchronous *Events*; the first one is of type zero-crossing. But *e2* is synchronous and thus can be eliminated. This simplifies the task of the compiler, which in the first phase

of the compilation, removes *e2* yielding a model containing only primary *when* clauses (see [2] for the definition of a primary *when* clause):

```
Event e1,e3,…;
…….
equation
when continuous then
  e1=event(…
  ….
end when;
when initial then
   ….
end when;
when e1 then
  k=pre(k)+1;
  if (k>1) and not(pre(k)>1) then
    e3=time+1;
    ….
end when;
  …..
```

The important thing to note here is that at the end of the first phase of the compilation, we end up with a model that contains only asynchronous *Events*. Note also that asynchronous events, explicitly declared as *Event*, are of two types:

- Zero-crossing: implemented using zero-crossing mechanism of the numerical solver
- Predictable: e.g., ***e2=e1+1;***

The type of *Event* is coded in the model by the user, not guessed by the compiler (we may consider allowing the compiler to switch type from zero-crossing to predictable when possible).

The phase two of the compilation performs static scheduling independently for the codes associated with each *Event*, and for sections: "continuous", "initial" and "terminal" (see [2] for the definition of these sections). The simulator interacts with the code through these *Events*. It uses an "Event Scheduler" on run-time.

3.2. Separate Compilation

Currently there is no mechanism in Modelica that allows us to isolate a part of a model and compile it separately. But being able to isolate a module has many applications. For example in control applications, often the user models the plant and the controller to validate the performance by simulation and then wants to generate code for the controller part only.

We think that module isolation can be realized using *input/output Event*s. Consider the following example, which is a counter that slows down as time advances:

```
model SlowDownCounter
event_delay BB;
Event E(start=0);
discrete Real U(start=1);
discrete Integer k(start=1);
equation
```

```
when pre(E) then
  k=pre(k)+1;
  (E)=BB(pre(E),U);
end when;
end SlowDownCounter
```

where *event_delay* is considered as an external function that can be expressed in C, Java, Modelica, etc. For that, we consider an extension of the current notion of function, which currently allows only immediate functions with no states, no sub-sampling, etc. The Modelica program for this function could be the following:

```
function event_delay
input Event e1;
output Event e2;
input Real u;
equation
when e1 then
  e2=e1+u;
end when;
end event_delay;
```

In this case *SlowDownCounter* can be compiled without any knowledge of the content of the *event_delay* function. This function can be compiled separately too or written in C (the Scicos block routine of the *Event-delay block* does exactly this). The use of "*function*" may not be the best solution. We may consider using *block* instead of *function*, and declare it as *external* in *SlowDownCounter.*

To see why we need *input Event* in this case, note that without it, flattening the model yields a *when* clause within another *when* clause. Nested *when* clauses are not accepted in Modelica and it is not obvious to see how they can be interpreted if they are allowed.

Isolated modules can be a lot more general than in this example; they can have *input/output Events,* internal states (*der* and *pre), c*onditioning (*if-then-else* and *switch)* and *s*ub-sampling under the isolation condition: **all Events within the module must either come from input or be of asynchronous type (for example of type zero-crossing).**

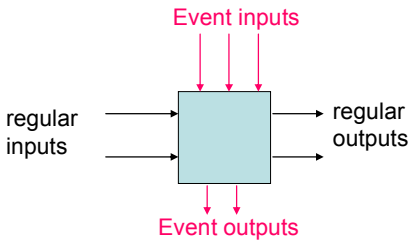
The isolation condition guarantees that the calling environment knows when to call the external module. Specifically it avoids nested *when* clauses which are meaningless.

Note that some information concerning module must be provided to the calling environment so that it can be compiled separately. These information are needed for proper scheduling (finding the right order of execution) but also for proper interfacing with the numerical solver:

- what inputs affect the outputs directly (direct feedthrough),
- is block always active (contains continuous variables),
- if the module contains *der()*, the continuous state and its derivative must be part of the input and the output.

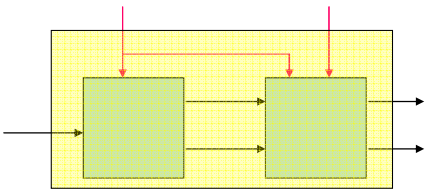
These conditions are exactly the block properties provided to the compiler in Scicos. They are enough for separate compilation and code generation. In Scicos, the internal function of the block is not known to the compiler; its code is in general provided as a *dll*. What the simulator does is to call the "black box" routines associated with the blocks in the right order. This is exactly what the Modelica simulator would do with external modules.

An isolated Modelica module can be represented as follows:



As stated previously, the routines of the block can be written in C (such as a Scicos block, or a Simulink block) or in Modelica. Note that in such a block, output events are never synchronous with input events. Input events however can be synchronized, that is why the *Swtichwhen* clause is sometimes needed inside the block.

Under certain conditions, a sub-model obtained from connected blocks can be converted to a single block:



This construction is very similar to that of a Super Block in Scicos compiled into a block using the code generation mechanism.

3.3. Scicos Interface

As we have seen, the Scicos formalism and the Modelica language become very similar when the type *Event* is introduced in Modelica. It is then natural to consider developing an interface between the two so that:

- Scicos blocks can be used in a Modelica model,
- Modelica isolated modules can be used as Scicos blocks (this has been the subject of the Simpa Project [5]).

The routines associated to blocks in Scicos have a specific prototype:

```
void  my_block(scicos_block *block,int flag)
```

where *scicos_block* is a structure containing block data and *flag* indicates the task that the function must realize. The following table indicates the tasks that the function may have to realize:

Flag	Job
0	Compute state derivative
1	Compute outputs
2	Update states
3	Output event dates
4	Initialization
5	Ending
9	Compute zero crossings and modes

The simulation engine interacts with the block by calling the functions associated with the blocks, with different *flags*, to advance time. This is exactly the way we implement the Modelica simulator. This implementation separates completely the simulation engine from the model. It also allows the use of both Modelica and Scicos components in the same model.

Conclusion

We have introduced extensions to the Modelica language that would allow for model isolation and separate compilation. Besides obvious advantages, this allows Modelica programs to be interfaced with other simulation environments such as Scicos and Simulink.

References

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

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

3 Peter Fritzson - "Principles of Object-Oriented Modeling and Simulation with Modelica 2.1", Wiley-IEEE Press, 2003.

4 Stephen L. Campbell, Jean-Philippe Chancelier and Ramine Nikoukhah, "Modeling and Simulation in Scilab/Scicos", Springer, 2005.

5 http://www.rntl.org/projet/resume2005/simpa2.htm