Towards a full integration of Modelica models in the Scicos environment

Ramine Nikoukhah[1] Sébastien Furic[2]
[1]INRIA Rocquencourt, France
ramine.nikoukhah@inria.fr
[2]LMS Imagine, France
sebastien.furic@lmsintl.com

Abstract

In this paper, we show that, provided the Modelica language specification is enriched with the notion of *external connector*, interesting applications follow, where Modelica can be used to describe *synchronous event-activated* blocks that communicate efficiently with the "outside world". Such blocks described in Modelica and compiled separately can still be connected together to form new blocks. Scicos (www.scicos.org), a freely available modeling environment, can make efficient use of those blocks, that in addition enable seamlessly integration of Modelica into a causal environment.

Keywords: external connector; events; synchronization

1 Introduction

Scicos is a software tool for modeling and simulation of dynamical systems [1]. It provides a hierarchical graphical editor for the construction of complex dynamical systems, a simulator and a code generator. Scicos is distributed with the free scientific software package ScicosLab (www.scicoslab.org).

Currently Modelica is partially supported by Scicos. In particular it is possible to use component models in Scicos diagrams where the dynamics of the component has been described in Modelica [2]. This means that, in the same diagram, sections of the model are designed using standard Scicos blocks and others using Modelica blocks. So far, the Modelica section has only been used to model piecewise continuous-time, leaving all discrete-time behaviors to the Scicos-block section [3]. This configuration, in which Scicos and Modelica blocks co-exist in the same environment, has been available in Scicos for a number of years and is very similar to the Simulink/ Simscape product recently released by the Mathworks.

To go further in the integration of Modelica within the Scicos environment, the question of discrete dynamics in Modelica and its synchronization with the outside world needs to be addressed. There are two difficulties in extending the current integration:

- Modelica is conceived to be used to model the whole system. Except external functions and I/O routines with very limited functionalities and under restricted conditions, a Modelica model is written entirely in the Modelica language
- the event synchronization properties in Modelica itself are not unambiguously specified [4].

In this paper, we use the extensions proposed in [4] to propose a solution for the full integration of Modelica components in Scicos or in any other causal simulation environment with synchronous block activation semantics.

2 Current Scicos/Modelica interaction

2.1 Mixed Scicos diagrams

In a *mixed Scicos diagram*, the interaction between standard Scicos and Modelica blocks is made through *mixed type blocks*. These blocks have both regular Scicos input and/or output ports (i.e., *causal ports*) and Modelica connectors (i.e., *acausal ports*). In the example given below, we can see that Scicos ports are drawn as arrow heads whereas Modelica connectors are drawn as plain rectangles. Clearly it does not make sense to connect directly a causal port to an acausal one, and the Scicos editor forbids it.

The interaction between the controller part of this diagram, which is designed using standard Scicos blocks, and the hydraulic Modelica model is done through two mixed-type blocks: the controlled valve and the reservoir. In the case of the reservoir, for instance, the regular output port corresponds to a level sensor.

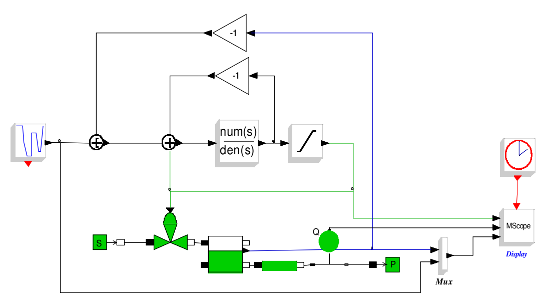

Figure 1: A mixed Scicos diagram including standard Scicos and Modelica blocks

The way Scicos compiler deals with mixed diagrams is to group all the Modelica blocks (including mixed type blocks, which are written in Modelica[1]) and generate the corresponding Modelica program. This program is then compiled with Modelicac (the free Modelica compiler distributed with Scicos), generating a C code that is transparently compiled and linked with Scicos so that, at the end, the whole Modelica section becomes a regular Scicos block where its (now causal) ports correspond to the input/output ports of the mixed block types in the original diagrams.

Even though in Modelica the whole system should normally be designed using the language itself, omitting the description of the outside world (by only specifying "external" input/output ports) would not cause major difficulty, as already proven by pure Scicos models already, which already enable separate compilation. Since the Modelica language specification does not explicitly define interaction with outside world, we propose here to fill that lack of specification by defining *external connectors* and by giving the rules that govern their use.

1 Despite Modelica the lack of a way to express true "external" connectors. Typically, top-level input/output declarations are specifically treated by compiler to yield external connectors as a final result. That has also been used in Dymola for generating Simulink blocks.

3 Extension of the Scicos/Modelica interaction

3.1 Activation signals

In Scicos, *activation signals* are used explicitly to specify control actions. For example on Figure 1, the time instants when the Scope block samples its input values and displays them is specified by the activation signal (sequence of periodic events in this case generated by the event clock) it receives through its activation input port placed on top of the block. Basic operations on activation signals in Scicos are used to provide periodic and non periodic clocks, sub-sampling, conditioning, etc. Such functionalities in Modelica are provided through special language constructs such as **sample** and **when**, not through signals that can be communicated through connectors. Actually, Modelica is not designed in the same spirit synchronous languages such as Scicos-the-language are. Consider for example the simple model below:

```
block Counter
  input Boolean activated;
  output Integer count(start=0);
equation
  when activated then
    count = pre(count) + 1;
  end when;
end Counter;
```

Modelica semantics currently impose instantaneous equations to be activated by the *detection of an edge* in Boolean conditions of "when" equations, with the notable exception of equations guarded by the **sample** operator[2]. As a consequence, it is not possible to separately compile a counter that increments its value when the environment into which it will be put simply decides it should do: testing the Boolean expression is still necessary to eventually activate the equations. The purpose of activation signals proposed by Scicos is precisely to obviate that problem: indeed, activation signals only take one value that gives the time instant *when* the model should be activated. The rest of the time, the signal is simply *absent* (i.e., no value is present at the model's input port). If we make an analogy with booleans, we can say that activation signals can only take the `true` value, *when they have a value*[3].

2 Hence the necessity to define it as a primitive of the language.

Going back to our example, the optimal code typically generated for the "when" equation would be (suppose the target language is C):

```
if (activated) count = count + 1;
```

Each time the value of `activated` is updated, the simulation environment has to run the code above, which performs the test, as explained above, and then update `count` if required.

Now suppose we have an `Event` type (see [4]) in Modelica to represent the type of activation signals. Our counter could then be written:

```
block Counter
  input Event activated;
  output Integer count(start=0);
equation
  when activated then
    count = pre(count) + 1;
  end when;
end Counter;
```

We have only changed the type of the input. But the optimal code typically generated for the "when" equation now becomes:

```
count = count + 1;
```

No test is required since the only possible value for `activated` is equivalent to the Boolean value `true`: incrementing the counter is just a matter of calling the above code, which is the responsibility of the outer environment. Imagine a model that would contain hundreds of (deeply nested) sub-models: the use of activation signals would result in significant increase in performance.

3.2 Activation ports in Modelica

Since the lack of activation signals in Modelica makes the interaction with Scicos (and potentially any other external causal environment) difficult, we simply propose to define external connectors capable of listening/emitting *events*.

An interesting solution would have been to reuse the **`external`** keyword as prefix to the declaration of external connectors as in:

```
external connector EventInput
  Event e;
end EventInput;
```

But since a compliant Modelica compiler is required to compile code for interaction with the outer world as soon as "toplevel" declarations of variables with prefix **`input`** or **`output`** exist, we will simply omit the keyword **`external`**, to remain compatible with the current language specification. So the only addition we propose to the language is the `Event` type (with accompanying operations). Interfacing with the outer world is just a matter of special interpretation of toplevel declarations, as usual.

For example, the following model, once compiled at toplevel, yields a Scicos block with two input activation events and one output activation event:

```
block B
  input Event e1, e2;
  output Event e3;
equation
  ...
end B;
```

Of course, the ability to declare activation ports is far from sufficient to enable seamless integration of Modelica into the Scicos environment. In particular, Scicos has to know where the events are generated and which input/output activation ports depend on which, in order to ensure global synchronization of the whole model

After presenting in the next section the `Event` type in more details and operations that apply to its members, we explain the principle of model abstraction in presence of activation events.

3.3 The `Event` type

The `Event` type is the type of all *events*, that is, abstract objects which sole purpose is to represent *activations times* in hybrid models. Notice that by "activation time" we do not speak about normal Modelica values that would represent measures of time, as provided by the pseudo-variable `time`: we are talking about the **domain** of signals which is implicit in Modelica, where programs only manipulate values of signals, i.e., elements of their codomains.

As already said in previous sections, events only have one possible value (but different domains!). Contrary to the design choice made in the Signal language, we do not want compatibility of `Event` with `Boolean` (by declaring the former to be a subtype of the latter). Instead of that, we prefer explicit conversion in expressions when required, to avoid unnecessary complications and user confusion. Only "when" equations accept both events and booleans, mainly for compatibility with existing practice.

3 In the synchronous language Signal, the value of activation signals is denoted *true* and is compatible with the Boolean type.

Since `Event` is such a degenerate type, the only operations that make sense for events involve in reality subsets of the time line, i.e., their domain[4].

Creation of events corresponding to edge detection requires the following syntax:

```
e = event(x >= 0.0);
```

The primitive `event()` creates an event activated whenever its Boolean expression argument becomes `true`. It should not be confused with the primitive `edge()` that, in Modelica, does not generate impulses (see [4]).

It is also possible to program events in the future by specifying a delay from current time:

```
e1 = event(x >= 0.0);
when e1 then
  e2 = e1 + 10.0;
end when;
```

The equation inside the "when" reads: "`e2` occurs ten units of time after `e1`".

Event synchronization, subsampling, etc., can simply be expressed by means of "when" equations, provided their semantics are revised so that tractable syntactic methods would be used to detect clock dependency[5], as explained in [6].

Modelica is now ready for true separate compilation, as explained in the next section.

3.4 Abstraction of models having activation ports

With the new connectors introduced here, an isolated Modelica compiled model can be represented as follows:

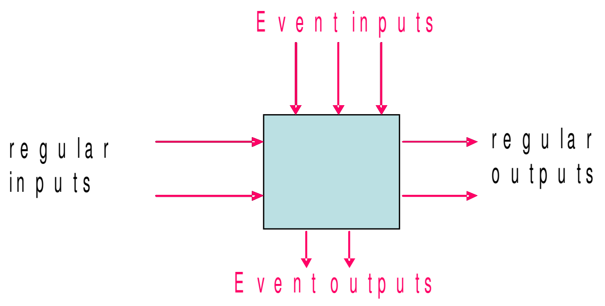

Figure 2: an abstracted separately compiled Modelica model with activation ports

4 Which is fortunate since events have been precisely introduced for that purpose...

5 In other words, clock calculus would become typing.

This compiled model can be inserted and used as an ordinary Scicos block provided an additional constraint is fulfilled: Scicos imposes that, in basic blocks, output events are never synchronous with input events. Fortunately, this property can be checked at model compilation time: indeed, as mentioned above, synchronicity can be determined syntactically.

In Scicos, input events can be synchronized. This synchronization is imposed from the outside of the module (so outside of the Modelica code) depending where the input events come from. If two input events in a model are such that, say, one of them is a subsample of the other, then clearly the synchronicity of the two events must be taken into account within the Modelica program. For example, in the case of the module depicted in Figure 2 above, the activation can come from the first event, the second, event, the third event, the simultaneous first and second events, the simultaneous first and third events, etc. In [4], the "switchwhen" clause was introduced specifically to deal with this problem: that clause simply generalizes the Modelica "when" clause by providing a way to express constraints depending on explicit synchronicity conditions. We propose in this paper a more "Modelica-like" version of that construct, that for clarity however requires some rudimentary pattern matching feature[6]. Here is an example:

```
match {e1, e2, e3} with
  case {-, -, *} then
    <eq1>
    <eq2>
    ...
  end case;
  case {*, -, *} then
    <eq3>
    <eq4>
    ...
  end case;
  case {*, *, * } then
    <eq5>
    <eq6>
  end case;
```

6 Pattern matching has already been proposed in other Modelica-like languages: the MetaModelica language used in the OpenModelica project features pattern matching. See http://www.ida.liu.se/~pelab/modelica/OpenModelica.html

```
end match;
```

The content of one and only one case is activated depending on what combination of events `e1`, `e2` and `e3` is generated. A "case" clause is activated whenever the expression after **`match`** matches one of the *patterns* following each **`case`**. In a pattern, "`-`" and "`*`" denote respectively the absence and the presence of a signal[7]: the first symbol corresponds to the first event (here `c1`) and so on. For example if events `c1` and `c3` are simultaneously generated but not `c2`, then the second case is activated. For this to happen, `c1` and `c2` must be synchronous. By using this clause, the Modelica code inside the module can anticipate every possible combination of input events activating it.

3.5 Composition of separately compiled models

An interesting property of separately compiled models is that they do *compose*. In other words, it is still possible to connect such compiled models to build new models without recompilation. More importantly, models that only define discrete-time signals (by means of "when" clauses) compose without significant loss of efficiency. This is not so surprising: since they are undistinguishable from ordinary Scicos blocks, they share the same properties (and Scicos blocks do compose efficiently!).

4 Conclusion

The Modelica extensions we have proposed allow us to integrate a larger class of Modelica models into Scicos blocks. This functionality should be made available in future versions of Scicos. Also, any causal simulation environment capable of handling activation signals (that, at runtime, only consists in calling blocks in the right order by ensuring synchronization of inputs if required) would benefit of such extensions.

Another important consequence of our proposal is that, implemented by tool vendors, it would lead to a compilation scheme for pure discrete-time **open models** that, for example, could be used by component providers to produce efficient versions of library models to their clients while still preserving IP.

7 Remember that the only useful information carried out by an event is its presence.

References

[1] S.L. Campbell, J. Chancelier, R. Nikoukhah, ***Modeling and Simulation in Scilab/Scicos***, Springer 2005 - Chinese edition, BuptPress, Beijing, 2007

[2] M. Najafi, S. Furic, R. Nikoukhah, ***Scicos: a general purpose modeling and simulation environment***, Proc. of the 4th International Modelica Conference, Hamburg, 2005

[3] M. Najafi and R. Nikoukhah, ***On the Integration of Modelica in the Modeling and Simulation Software Scicos***, Proc. of Modelling, Identification and Control, 2009

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

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

[6] R. Nikoukhah, S. Furic, ***Synchronous and Asynchronous Events in Modelica: Proposal for an Improved Hybrid Model***, in Proc. Modelica Conference, Bielefeld, 2008