Modelica extensions: efficient code generation and separate compilation

Ramine Nikoukhah
INRIA

EOOLT 2007

Outline

- Language extensions
  - switch and switchwhen
  - Type Event and primitives event
- Applications
  - Compiler/simulator simplification
  - Separate compilation
  - Scicos interface

Language extensions

switch and *switchwhen*

- ***switch*** generalizes constructor ***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 *n*.

Counterpart in Scicos is realized with *ESelect* block

May accept missing cases with warning (similar to conditions on branches of *if*)

- ***switchwhen*** generalizes constructor ***when-elsewhen***.

- 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 is activated after another (no specified order).

For special cases, the ***switchwhen*** constructor allows to take advantage of this additional information if needed.

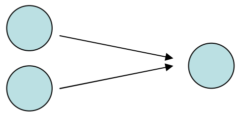
- Very useful in some applications
- Essential for module isolation

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

For example if events *c1* and *c3* are simultaneously detected, then case '101' is activated.

Example of usage of *switchwhen:*

Consider contact between three balls:



Ignoring the possibility of simultaneous contact:

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

Wrong simulation result in case of simultaneous contact

Solution to fix the wrong simulation result

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

Not a flexible solution: does not allow to explicitly specify what happens in case of simultaneous 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 {x1-x3<=1,x2-x3<=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;
```

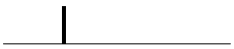
Consider explicitly every case

Type Event and primitive event

Currently events coded by Booleans:

equation

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



Not normal Booleans: impulsive type

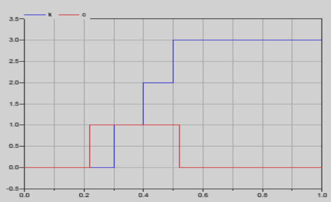
when k>0 then
c=edge(b) ;

edge does not always produce impulsive Boolean

No distinction between Boolean and event

Coding events as Boolean creates confusion

```
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;
```



k is incremented three times during a single ***edge(b)***

Type *Event* codes the time of events as float

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

e2 is an event delayed by one

Delay can be used to emulate *sample(0,1):*

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

Operation on Events

Primitive *event*

Primitive *event* resembles *edge*:

- But it generates *Event* not Boolean.

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

Zero-crossing event detected by numerical solver (asynchronous)

synchronous with *e1*

Argument of *when* must be an *Event*

```
equation
der(x1)=v1;der(x2)=v2;der(x3)=v3;
der(v1)=0;der(v2)=0;der(v3)=0;
E1= event(x1-x3<=1);
E2=event(x2-x3<=1);
switchwhen {E1,E2} then
  case "10":
   reinit(v1,v3);
   reinit(v3,v1);
  case “01":
   reinit(v2,v3);
   reinit(v3,v2);
  case "11":
    <TO DO IN CASE OF
    SIMULATANEOUS CONTACT>
end switchwhen;
```

Generate events.
Not allow:
B1=(x1-x3<=1);

Only *Event* types can be argument of *when* and *switchwhen*

Applications

Compiler/simulator simplification:

Manipulating Events explicitly simplifies model construction:

No need to use artificial tests against time.

Example: Modeling the propagation delay in a digital circuit requires a variable dependent event delay:

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

using Event types →

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

It also simplifies the compiler: compiler no longer needs to “figure out” what “tests” are simple enough to be implemented without solver zero-crossing mechanism.

Canonical representation of flat model and compiled model

Using exclusively *Events* to condition *when* clauses, structures the model.

```
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;
  …..
```

After compiler phase I

All secondary *when*'s removed

e2 is removed: only asynchronous events remain

```
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;
  …..
```

At the end of phase I, only asynchronous *Events* remain.

- 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 user, not guessed by the compiler (may consider allowing compiler to switch type from zero-crossing to predictable when possible)
- Compiler Phase II performs static scheduling independently for the codes associated with each *Event*, and for sections: "continuous", "initial" and "terminal".
- Simulator interacts with the code through Events. It uses an "Event Scheduler" on run-time.

Separate compilation

Module isolation can be realized using *input/output Event*s. Example:

Option: extend definition of function

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

```
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
```

In this case *SlowDownCounter* can be compiled without knowledge of the content of *event_delay* function.

This function can be compiled separately too or written in C (for example a Scicos block routine).

May use ***block*** instead of ***function***, and declare it ***external*** in SlowDownCounter

Isolated modules can be a lot more general than external functions; they can have:

- *Input/output Events*
- Internal states: ***der()*** and ***pre()***
- Conditioning: ***if-then-else*** and ***switch***
- Sub-sampling under isolation condition:

All *Events* within the module must either come from input or be asynchronous

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

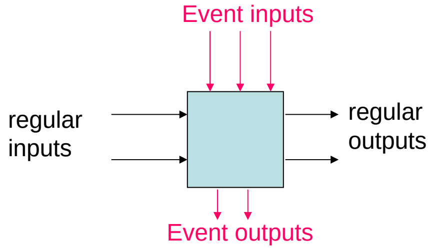
Some information concerning module must be provided:

- What inputs affect outputs directly (direct feed through)
- Is block always active (contains continuous variables)
- If the module contains ***der()***, the continuous state and its derivative must be input and output.

These conditions are exactly the block properties provided to the compiler in Scicos. They are enough for compilation and code generation.

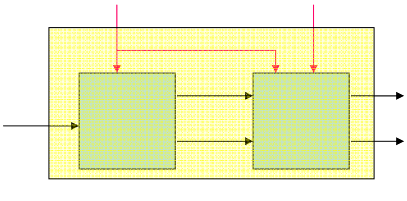
The internal function of the block is not known by the compiler; the code is in general provided as a dll. The associated "black box" routines are called during simulation.

Isolated Modelica module



Can be an external block:
• Scicos block
• Simulink block (under certain conditions)

- Output events are not synchronous with input events
- *switchwhen* sometimes needed inside the block (event inputs can be synchronous, or not)
- Under certain conditions, connected blocks can become a block:



Similar to
Super Block in Scicos

Scicos Interface

- Scicos block can be used in a Modelica model
- Modelica Isolated Module can be used as Scicos block (Simpa Project)

Scicos block interface

```
#include "scicos_block.h"
#include <math.h>
void  my_block(scicos_block *block,int flag)
{
  ...
}
```

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