Modelica Change Proposal MCP-0012

Calling Blocks as Functions

Status: In Development

(discussed in ticket #1512)

Summary

The purpose of this MCP is to introduce functions with memory and events into Modelica. This is achieved by enhancing Modelica so that blocks can be called as functions (since memory and events are already supported in blocks). Since functions have a different type system as blocks (e.g. arguments in functions **can** be identified by position, whereas in blocks they **must** be identified by name), the calling mechanism of a block is naturally restricted to named arguments. Since functions have an optional mechanism for named input arguments, but not for named output arguments, functions are generalized for named output arguments first. As a consequence, the optional calling mechanism of functions and the required calling mechanism of blocks become identical.

Revision

Date	*Short description of revision*
April 21, 2016	Adapted to MCP format
Dec. 17, 2014	Major rewrite based on discussions at the 84th Modelica Design Meeting, discussions at the #1512 ticket and a Web/phone meeting on Dec. 17 (rewritten by Martin Otter)
July 25, 2014	Martin Otter: Adapted to the prototype from Dymola (from Hans Olsson)
June 11, 2014	Martin Otter: Initial version.

Contributor License Agreement

Not yet applicable (since no CLA is available).

Table of Contents

1. Generalized expressions

1.1 Named function output arguments

Assume that the following function definition is present:

```
record Rec
  Real y[10];
end Rec;

function fc
  input  Real u;
  output Rec  rec;
  output Real y;
algorithm
  …
end fc;
```

Then fc can be called in an expression to return only one of the output arguments, by explicitly defining the desired output argument in the function call:

```
Real z = fc(u).y;
Rec  r = fc(u).rec;
```

The single return argument of the function is identified by a "." and the output name after the function call.

Prototypes for this feature are planned in January 2015 for Dymola (DS Lund) and for OpenModelica (PELAB).

1.2 Expression selectors

The value of an expression can be further processed by an expression selector. Examples:

```
Real a[10], b[10];
Real c[2] = (a+b)[4:5];    // Selection of array elements

Real d[3] = fc(u).rec.y[5:7];
Real e[3] = (fc(u)).y[5:7];   // (…) selects first output of fc, so rec
```

Prototypes for this feature are planned for a later stage.

2. Calling a block as a function – Level 1

Calling a block as a function is a difficult topic, because blocks may have memory and events. For this reason the task is split into 3 levels of different complexity and usefulness. Prototypes for Level 1 are planned in January 2015 for Dymola (DS Lund) and for OpenModelica (PELAB). Prototypes of the other levels will be made later.

2.1 Example

Take the following Modelica blocks

```
Modelica.Blocks.MathBoolean.OnDelay
Modelica.Blocks.Sources.BooleanPulse
```

that have both one input and one output Boolean signal. It shall be possible to use these block as:

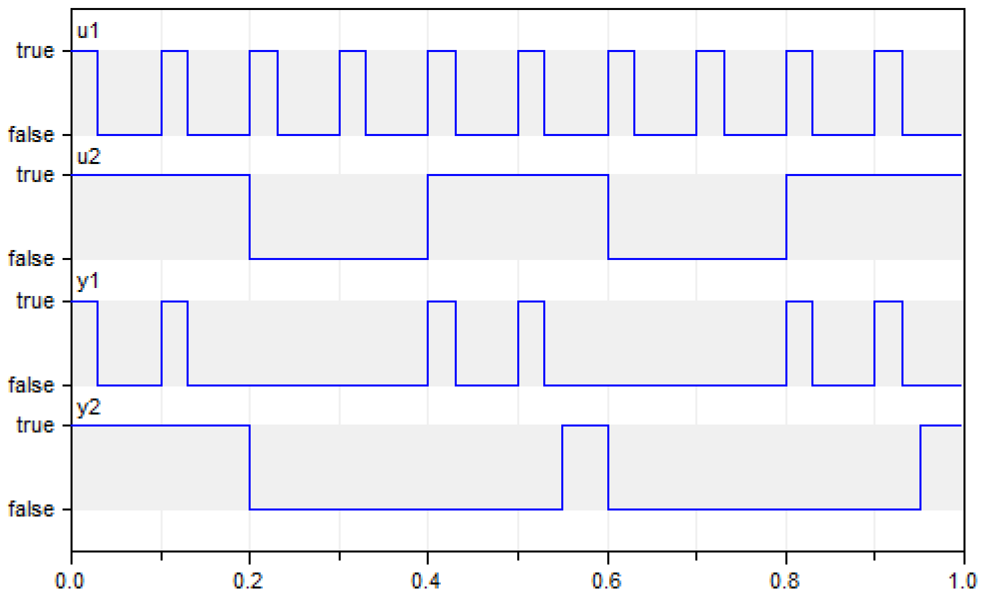
```
import Modelica.Blocks.MathBoolean.*;
import Modelica.Blocks.Sources.*;

Boolean u1 = BooleanPulse(width=30, period=0.1).y;
Boolean u2 = BooleanPulse(width=50, period=0.4).y;
Boolean y1 = (u1 and u2) or OnDelay(u=u1, delayTime=0.3).y;
```

```
  Boolean y2 = OnDelay(u=OnDelay(u=u2, delayTime=0.1).y, delayTime=0.05).y;
```

This is performed by defining a mapping of the above code fragment to the following Modelica 3.2 code:

```
    import Modelica.Blocks.MathBoolean.*;
    import Modelica.Blocks.Sources.*;

    BooleanPulse BooleanPulse_1(width=30, period=0.1);
    BooleanPulse BooleanPulse_2(width=50, period=0.4);
    Boolean u1 = BooleanPulse_1.y;
    Boolean u2 = BooleanPulse_2.y;

    OnDelay OnDelay_1(delayTime=0.3);
    OnDelay OnDelay_2(delayTime=0.1);
    OnDelay OnDelay_3(delayTime=0.05);

    Boolean y1 = (u1 and u2) or OnDelay_1.y;
    Boolean y2 = OnDelay_3.y;
  equation
    OnDelay_1.u = u1;
    OnDelay_2.u = u2;
    OnDelay_3.u = OnDelay_2.y;
```

The mapping is performed by instantiating the blocks and utilizing the input and output variables of the blocks in the expressions. Simulation result of the above example:



2.2 Mapping rule

In the **declaration** section of a Modelica model or block class it is possible to call a Modelica block with the syntax of a function call (with named input and output arguments), provided the following requirements are fulfilled:

1. The class name of the block is used as function name.
2. All input variables of the block that have no default values must be provided as named function input arguments. The inputs can be variables or variable connectors (so a variable that is also a connector, such as Modelica.Blocks.Interfaces.BooleanInput).
3. For all other public (non-output) variables, standard modifiers can be applied.
4. The desired output variable of the block must be used as named function output argument (so appended to the call together with a “dot”).

5. The expression in which the block is called is not conditional (conditional expressions are handled in Level 2).
6. The declaration in which the block is called is not allowed to have an element "inner" or "outer" and is not allowed to have one of the type-prefixes (flow, stream, discrete, parameter, constant, input, output)[1]

Under these pre-requisites, a block "`Block`" with

- inputs "`u1, u2, …` " that have no defaults,
- modifiers "`m1, m2, …`"
- outputs "y1, y2, … "

is called as

```
Block(u1=.., u2=.., …, m1=.., m2=.., …).y2
```

in an expression when y2 of the Block shall be used in the expression. This part of the expression is replaced by `<Block>.y2` and the following statements are introduced:

```
     Block <Block>(m1=.., m2=.., …);
  equation
     <Block>.u1 = …;
     <Block>.u2 = …;
     …
     <Block>.un = …;
```

where `<Block>` is a unique instance name introduced by the tool.

Note, the above mapping rule also holds if the input/output variables are connectors, as in the BooleanPulse and OnDelay example:

```
block OnDelay
   Modelica.Blocks.Interfaces.BooleanInput u;
   Modelica.Blocks.Interfaces.BooleanOutput y;
     …
end OnDelay;

Boolean y2 = OnDelay.y(u=u1, delayTime=0.3).y;
```

2.3 Prototype Implementation

Planned for Dymola and OpenModelica.

3. Calling a block as a function – Level 2

In this level, the restriction of Level 1, no conditional expression, is released: The block can still be only called in the declaration section of a model or block, but it **can** be used in a **conditional expression**.

[1] The reason for this restriction is that (a) the simple mapping mechanism in section 2.2 would not give an intuitive result, and (b) the type-prefixes do not make sense for blocks that have potentially initialization, when-clauses, differential equations (at least, it is not obvious what the result shall be). Note, the synchronous language Lucide Synchrone supports both functions with and without memory and Lucide Synchrone has a similar restriction (a function with memory cannot be called in a constant declaration).

3.1 Example

The mapping rule of section 2.2 cannot be applied, because then the block will be evaluated also if the corresponding if-branch is not active. This is both non-intuitive and can lead to unexpected/unwanted exceptions. Example:

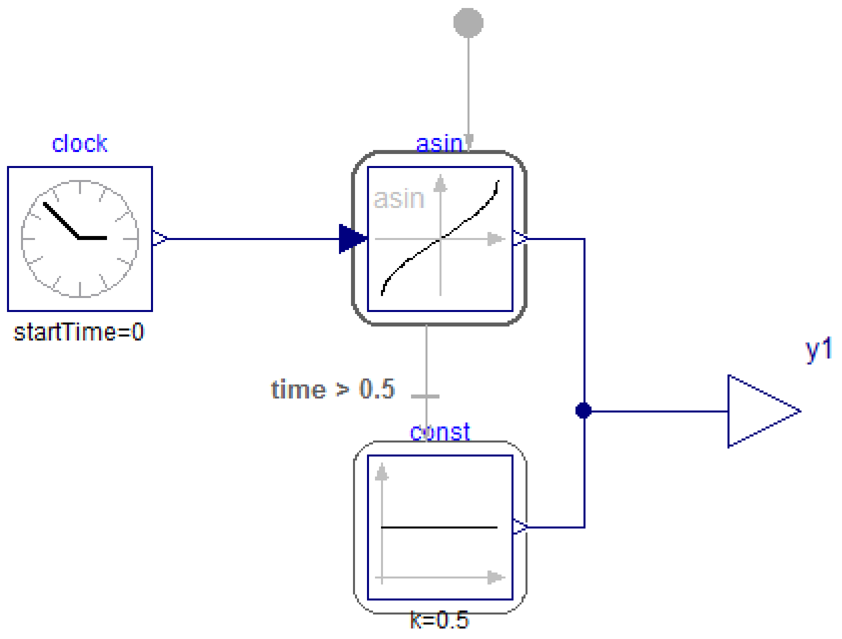
```
  Real y = if time<0.5 then Modelica.Blocks.Math.Asin(u=time).y else 0.5;
```

Using the mapping rule from section 2.2 results in:

```
    Modelica.Blocks.Math.Asin Asin1;
    Real y = if time < 0.5 then Asin1(u=time).y else 0.5;
  equation
    Asin1.u = time;
```

Here the block Asin1 is evaluated all the time, also if time > 0.5. Around time = 1, this will give an exception, because Asin1 is no longer defined for this input argument. The mapping semantics of section 2.2 is therefore both inefficient (because the block is evaluated, although the block ignores the result) and non-intuitive because an exception is generated, that should not occur.

Conceptually, an extended mapping rule is needed that is based on the continuous-time state machine proposed in (Elmqvist et.al. 2014)[2]. Graphically, this can be performed as:



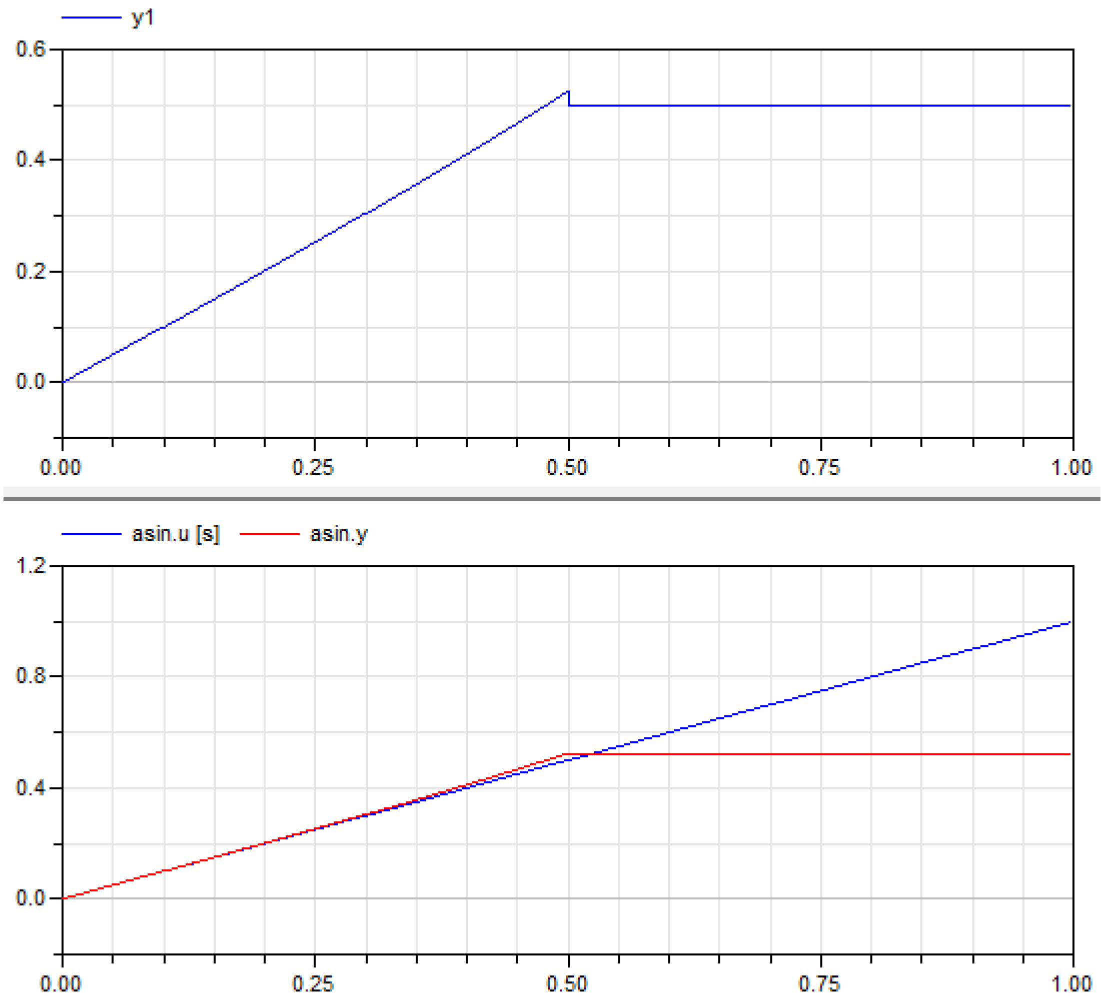
Textually, the declaration can be mapped to:

```
      Modelica.Blocks.Math.Asin Asin1;

      block ElseBranch
        Modelica.Blocks.Interfaces.RealOutput y = 0.5;
      end ElseBranch;
      ElseBranch elseBranch;

      block IfInput
        Modelica.Blocks.Interfaces.RealOutput y = time;
      end IfInput;
      IfInput ifInput;
```

[2] Hilding Elmqvist, Sven Erik Mattsson and Martin Otter (2014): Modelica extensions for Multi-Mode DAE Systems. Modelica Conference 2014.

```
    Modelica.Blocks.Interfaces.RealOutput yTemp;
    Real y = yTemp;
  equation
    initialState(Asin1);
    transition(Asin1, elseBranch, not time < 0.5, reset=false);
    connect(ifInput.y, Asin1.u);
    connect(Asin1.y, yTemp);
    connect(elseBranch.y, yTemp);
```

Simulating the graphically or the textually mapped system results in:



As can be seen, first continuous-time state machines need to be introduced in Modelica and second a much more complicated mapping rule is needed. The mapping rule becomes also quite complicated, because the expressions in the then- and else-branches must be computed with temporary blocks, and all parameters and other variables used in these branches must be propagate to these blocks.

Conceptually, the semantics seems to be clear: The then- and else-branch must be states of a state machine, the transition between these states is formulated with the transition operator (with reset=false), and all variables in a state are kept constant, if the state is deactivated. The actual mapping is more complicated as in the example sketch, because the “initialState” is not know at compile time. Therefore, a third state, the initial state, is needed and during initialization the if-expression is used to decide whether to make a transition to the if- or to the else-branch state.

4. Calling a block as a function – Level 3

In this final level, a block can also be called in an equation or algorithm section inside a model or block class (but not in a Modelica function). One has to analyze in more detail, which usages are still forbidden, and how to adapt the mapping rules. In all cases, one has to assume that a general block is called as function and this block contains initial equation sections, differential equations, for-loops, while-loops, when-clauses. Here is a first sketch:

Most likely it is not allowed to call a block as a function in

- initial equation or initial algorithm sections
  (because a mapping to standard Modelica is not possible: Even with continuous-time state machines it is not possible to formulate a model where a block instance is only active during initialization, but not during continuous-time integration.)
- when clauses
  Conceptually, calling a block as a function in a when clause means that a when-clause (in the block) is used inside another when-clause. This semantics is not defined in Modelica. Furthermore, some operators like the derivative operator are not allowed in a when-clause, and the semantics is undefined if a block is called as a function in a when-clause and this block has differential equations.
- for loops
  The number of iterations of a for-loop need to be known at compile time. If a for-loop has n iterations, then n instances of the block (that is called as function in the for-loop) are needed, because in every iteration of the for-loop a different computation takes place (if the input arguments to the block are changing in the loop).
  If the block has a memory, e.g. using the pre-operator on a Boolean variable, then the semantics is unclear: Intuitively, a user would expect that in every iteration of the for-loop the same block is used and then it might be no longer possible to have n blocks.

5. Acknowledgment

This MCP idea was developed within the MODRIO ITEA2 project. Partial financial support of BMBF is highly appreciated.