**Design of temporal operators for library Requirements**

by Martin Otter, Feb. 18, 2015   
(based on Thuys FORM\_L and discussions with Thuy)

Operations in yellow, are not yet implemented (operators in white are already implemented)

# Temporal operators for fixed window (Boolean or Queued time locators)

**TimeLocators** (output Boolean y: y=true: within time locator; y=false; not within time locator)

~~Always() : Returns always true~~ (same as Sources.BooleanConstant()

Every(interval,duration): Returns true during every interval for a duration length

Until(u) : Returns true, until first rising edge of input

After(u) : Returns true, after first rising edge of input

AfterFor(u,duration): Returns true, after a rising edge for given duration

AfterUntil(u1,u2): Returns true after a rising edge of input 1 until rising edge of input 2

**CheckProperties** (input Boolean condition: whenever condition is true, check is applied)

During(condition, check) // check must be true

MinDuration (condition,check,duration) // check must be true for at least the given duration  
 MaxDuration (condition,check,duration) // check must be true for at most the given duration

BandDuration(condition,check,durationMin,durationMax) // check must have a true duration  
 // between durationMin … durationMax

FixedRising(condition,check,nRising=1) // check must have exactly nRising edges

MinRising(condition,check,nRisingMin=1) // check must have at least nRisingMin edges

MaxRising(condition,check,nRisingMax=1) // check must have at most nRisingMax edges

BandRising(condition,check,nRisingMin=1,nRisingMax=2) // check must have   
 // nRisingMin .. nRisingMax edges

NoRising(condition,check) // check must not have a rising edge

WhenRising(condition, check) // check must be true, if condition has rising edge  
 WhenFalling(condition, check) // check must be true, if condition has falling edge  
 WhenChanging(condition, check) // check must be true, if condition has changing edge

**TimeLocatorSets** (output Queue y: returns queue that stores time instants of start of time locators)

After(u) Operator returns true, after a rising edge of input

AfterFor(u,duration): Operator returns true, after a rising edge for given duration

AfterUntil(u1,u2): Operator returns true after a rising edge of input 1 until rising edge of input 2

**CheckPropertySets** (input Queue condition: apply check on all elements of the time locator set)

// same as Checks, but for all elements of the queues of TimeLocatorSets

MinDuration (condition,check,duration) // check must be true for at least the given duration  
 MaxDuration (condition,check,duration) // check must be true for at most the given duration

BandDuration(condition,check,durationMin,durationMax) // check must have a true duration

MinRising(condition,check,nRisingMin=1) // check must have at least nRising edges

MaxRising(condition,check,nRisingMax=1) // check must have at most nRising edges

FixedRising(condition,check,nRising=1) // check must have exactly nRising edges

BandRising(condition,check,nRisingMin=1,nRisingMax=2) // check must have   
 // nRisingMin .. nRisingMax edges

NoRising(condition,check) // check must not have a rising edge

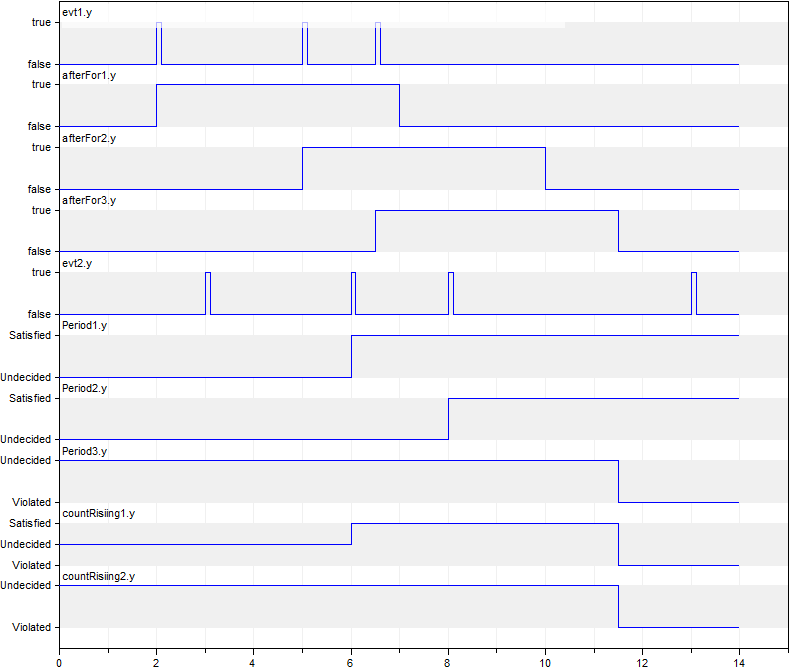
**Precise semantics of TimeLocatorSets/CheckSets needs clarification**. Example:

// after evt1 within 5s check count(evt2)==2  
// (for every 5s period after a rising edge of evt1,

// there must be 2 rising edges of evt2)  
CountRising( condition = AfterFor(condition=evt1, within=5.y,

check = evt2, op = “==”, nRising=2);

The understanding is, that this combination of AfterFor and CountRising results in:



Explanation:   
Whenever evt1 has a rising edge, a new period of 5s is started. The time locators of these periods are overlapping and are displayed with variables afterFor1.y, afterFor2.y, afterFor3.y.  
In every period, the input evt2.y must have 2 rising edges.  
This is true for the first two (overlapping) periods, but not for the third one.

There is **no unique (obvious) semantics** for the value of the output (countRising1.y).  
Proposal:

The resulting value for **one** period is defined as (see Period1.y, Period2.y, Period3.y above):

Before in one period two rising edges occurred, the output is set to Undecided.

After in one period two rising edges occurred, the output is set to Satisfied until the next rising edge of the input evt1.  
After in one period two rising edges did not occur, the output is set to Violated until the next rising edge of the input evt2.

The resulting output is a combination of the values of the overlapping periods.   
Thuy proposed “or” of the values. However, this does not work because “Satisfied” and “Violated” would result in “Satisfied”, which is unintuitive.

Using “and” of the values, is not so intuitive either, because “Satisfied” and “Undecided” would result in “Undecided”.  
In the above diagram, the resulting output (**countRising1.y**) is set according to the following true table (if at least one value is Violated, the result is Violated. If all values are Undecided, the result if Undecided. In all other cases the result is Satisfied):

|  |  |  |  |
| --- | --- | --- | --- |
| *periode2*  *periode1* | *Satisfied* | *Undecided* | *Violated* |
| Satisfied | Satisfied | Satisfied | Violated |
| Undecided | Satisfied | Undecided | Violated |
| Violated | Violated | Violated | Violated |

However, we can also use “and” (but then the result looks more pessimistic as it could). The advantage would be, that this can be more easier communicated to the end user  
(this results in **countRising2.y** above).

**Implementation of FIFO Queue**:

With external Modelica functions. The C-Code uses dynamic memory allocation (the memory is a vector; if memory no longer sufficient, new vector with double length is allocated and old vector is copied to new vector). Every queue element consists of one Real number. In the Time Locator a new queue element is generated and initialized with zero. In the Check block the element value is updated according to the Check function (e.g. counting the number of occurrences of rising edges).

Possible functions and call sequences (this is not yet correct/complete):

In Time locator

when initial() then  
 id = **init**();  
 end when;

when evt1 then // time locator entered  
 // figure out value of yStart (from checkBlock)  
 id2 = **push**(id, value=0.0);

end when;

when evt2 then // time locator is terminated  
 // figure out value of yStart (from checkBlock)  
 id3 = **pop**(id, id2); // push is called before pop if both evt1 and evt2 arise  
 end when;

In Check block (id + id2 + id3 + yStart are communicated in the connector)  
 algorithm  
 when check then // check has a rising edge  
 len := **length**(id,id2,id3) // actual length of queue  
 for i in 1:len loop  
 value := **inquire**(id,id2,id3, i) + 1; // inquire and increment i-th element from the front  
 **set**(id,id2,id3, i, value) // set new value for i-th value from the front  
 end for;  
 // figure out value of y (from all queue elements)  
 y := ….  
 elsewhen yStart then   
 y := yStart;  
 end when;

# Temporal operators for sliding time window

The semantics of the sliding time window is mostly unclear (see below). Furthermore, it seems not possible to separate (sliding) time locator and “check” for the implementation. So, for every suitable “check” block there will be a combined block “sliding time window for this check type”. Here is some analysis:

DuringAny(interval, check) // check must be true in any interval  
 // this can only be true, if check is true for the whole time (independently of the  
 // sliding time interval. Easier and clearer with “Always” + “During”

DurationAny(interval,check,duration) // check must be true for at least the given duration  
 // same issue as with DuringAny. check must be true all the time

InRange(interval,check,start,end) // check must be true within “start” up to “end” duration.  
 // same issue as with DuringAny. check must be true all the time

NoRising(interval,check) // check must not have a rising edge  
 // this can only be true, if check has no rising edge over the whole time span  
 // Easier and clear with a special operator for. “interval” is not needed (holds for every interval)

OneRising(interval,check) // check must have exactly one rising edge  
 // This cannot be fulfilled by a sliding time window (seems to be not useful)

AtLeastOneRising(interval,check) // check must have at least one rising edge  
 // Most likely the same as “AfterFor(u=check, duration=interval)  
 // So operator not needed

CountRising(condition,check,op,nRising) // op = “<”, “<=”, “==”, “>”, “>=”  
 // not clear how to defines this with a sliding time window

Conclusion: According to current understanding, operators for sliding time window are not included, because semantics unclear and/or in combination with check not useful.

# EUROSYSLIB/DA operators (discuss with Eric, Maxim)

Inclusion of the operators from EUROSYSLIB/DA Properties blocks (if “o.k.” block is slightly adapted to the naming and icon convention of Requirements library).  
The blocks could be stored under **SignalAnalysis**

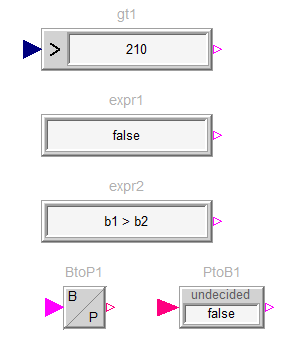
* MovingRamp\_i, MovingRamp\_e not clear. Probably split in several functions to compute the derivative of a variable with different methods:  
   DerExact: y = der(u); // only if input signal can be analytically differentiated  
   DerFiltered: y = der( filter(u, fcrit) ) // low pass filter on u and then derivative (works always)  
   DerFiniteDifference y = (u – delay(u,dt))/dt // derivative over a finite time interval
* TimeMonitoring // o.k.
* OscillationMonitoring -> FrequencyMonitoring // Implementation is unclear
* DomainMonitoring // check implementation
* TimeIntegration -> TriggeredIntegrator // o.k.
* MovingIntegration -> TriggeredMovingAverage // o.k.
* MovingAccumulation -> TriggeredAccumulation // o.k.
* FFT\_real // check implementation and discuss with DLR colleague Martin Kuhn
* UpperLevel -> AnyTrue (“or” of Boolean input vector)  
   AllTrue (“and” of Boolean input vector)  
   CountTrue (number of true values in Boolean input vector)  
   // all based on Modelica.Math.BooleanVectors;   
   // should be stored under “MathBlock” (no temporal operators/no memory)

# Graphical Appearance

I have evaluated several alternatives to improve the graphical layout. My currently best preference is the following:

* The width/length of a block are either width/length of the standard block ({{-100,-100}{100,100}}  
  or a factor 2 smaller or a factor 2 larger. The benefit is that the blocks can be easier located together (see examples below)
* The instance name is displayed above the block, but in light grey, in order that it is not disturbing the layout too much (reducing “Thuys noise”). One could remove the instance name also completely from the icon, but it is then no longer so easy to plot variables.
* All values of the parameter menu are displayed in the icon, in order that it is not necessary to inspect the parameter menu to understand the precise parameterization.  
  All such menu entries are defined as “input fields” (see examples below).
* All components have “default component names”: The class name is without any abbreviation (e.g. BooleanToProperty), whereas the default component name is a meaningful abbreviation in order that the name displays reasonably in the icon (e.g. BToP1).

Examples:



* Components that have a double length and a half width of a standard block:  
  gt1 returns true if u > 210  
  expr1 returns always false  
  expr2 returns b1 > b2
* Components that have a half length and a half width of a standard block:  
  BtoP1 transforms the Boolean input to a Property output
* Components that have the same length and a half width of a standard block  
  PtoB1 transforms a Property input to a Boolean output (via parameter undecided it is defined whether Property.Undecided is mapped to false (as above) or to true.

# Mapping

(Summary of current status; based on Hildings proposal + discussion at the WP2 meeting on Jan. 27/28 2015)

It is assumed that the modeler provides a standardized block in which one or more requirements are present for one system component (e.g. requirements for a pump).