Formal Requirements Modeling for Simulation-Based Verification

Martin Otter[1], Nguyen Thuy[2], Daniel Bouskela[2], Lena Buffoni[3], Hilding Elmqvist[4], Peter Fritzson[3], Alfredo Garro[5], Audrey Jardin[2], Hans Olsson[4], Maxime Payelleville[6], Wladimir Schamai[7], Eric Thomas[6], Andrea Tundis[5]

[1]Institute of System Dynamics and Control, DLR, Germany, Martin.Otter@dlr.de
[2]EDF, France, {Daniel.Bouskela,Audrey.Jardin,N.Thuy}@edf.fr
[3]PELAB, Linköping University, Sweden, {Lena.Buffoni, Peter.Fritzson}@liu.se
[4]Dassault Systèmes AB, Sweden, {Hilding.Elmqvist, Hans.Olsson}@3ds.com
[5]DIMES, University of Calabria, Italy, {Alfredo.Garro, Andrea.Tundis}@unical.it
[6]Dassault Aviation, France, {Eric.Thomas, MP}@dassault-aviation.com
[7]Airbus Group Innovations, Germany, Wladimir.Schamai@airbus.com

Abstract

This paper describes a proposal on how to model formal requirements in Modelica for simulation-based verification. The approach is implemented in the open source Modelica_Requirements library. It requires extensions to the Modelica language, that have been prototypically implemented in the Dymola and Open-Modelica software. The design of the library is based on the FOrmal Requirement Modeling Language (FORM-L) defined by EDF, and on industrial use cases from EDF and Dassault Aviation. It uses 2- and 3-valued temporal logic to describe requirements.

Keywords: requirements, verification, physical systems, 3-valued logic, temporal logic.

1 Introduction[1]

1.1 Overview

To ensure the proper operation of complex physical systems such as power plants, aircraft or vehicles, requirements are issued all along the system's lifecycle: from the preliminary design phase to the operation phase. Typically, the requirements capture the spatiotemporal and quality of service conditions that a system should fulfill. They may be quite complex and numerous. Testing the compliance of the system with the requirements may be quite challenging, due to the many items that should be examined and verified for a given test scenario, and the number of test scenarios to be considered to have a satisfying verification coverage.

This paper tries to improve the current situation, by (a) providing the open source library Modelica_-Requirements to define and model requirements in a formal way using 2- and 3-valued linear temporal logic (LTL); (b) associating requirement models with behavioral models; (c) testing whether the defined requirements are violated by the system design currently studied when the underlying behavioral models are *simulated*. This approach requires extensions to Modelica, that have been prototypically implemented in Dymola (Dassault Systèmes, 2015) and in OpenModelica (Open Source Modelica Consortium, 2015). The library has been tested and can be used by both of these Modelica simulation environments.

The main purpose of this approach is to check formally defined requirements by *simulation*. It is *not* intended to perform formal model verification by model checkers as done by tools such as NuSMV[2], SPIN[3], Prover Plug-in[4] for discrete systems or SpaceEx[5], KeYmaera[6] for hybrid systems. For example, a differential-algebraic equation system may be solved numerically to compute a pressure p in a pipe, and the requirement is formulated as $p \geq p_{cavitate}$. Model checkers for discrete systems cannot be used in this case, and verification tools for hybrid systems can only handle simple sets of differential and discrete equations, but not large models of industrial applications like power plants or aircraft.

1.2 State-of-the-art to Define Requirements

The standard in industrial applications is still to define requirements in natural language in textual form. As a typical example see the requirements for electrical systems in US military aircraft MIL-STD-704F (Department of Defense, 1984). Such specifications are defined in reports by using for example Microsoft Word, or with dedicated tool support. The latter especially to get support for collaboration, traceability, coverage analysis of textually defined requirements. Moreover, visual modeling languages for system

[1] This section uses material from the internal reports (*Bouskela et al. 2015*) and (*Otter et al., 2014*).

[2] NuSMV: http://nusmv.fbk.eu/
[3] SPIN: http://spinroot.com/spin/whatispin.html
[4] Prover Plug-in: http://www.prover.com/products/prover_plugin/
[5] SpaceEx: http://spaceex.imag.fr/
[6] KeYmaera: http://symbolaris.com/info/KeYmaera.html

engineering are very common, such as SysML[7], a general-purpose modeling language for systems engineering applications, that defines requirement and parametric diagrams for supporting the modeling of system properties. In particular, requirement diagrams provide constructs and mechanisms to express and compose system requirements, as well as to allocate them to system components; parametric diagrams can be used for supporting performance analysis and quantitative assessment. There are a number of tools in this area, for example: Rational DOORS from IBM[8], Reqtify from Dassault Systèmes[9], OSRMT (GPL2)[10], formalmind Studio (free)[11]. The most important xml-based exchange format seems to be ReqIF (OMG, 2013).

Defining and processing requirements *formally* is an area of active research. The exploited mathematics uses propositional logic, temporal logic, set theory and others; see for example (Baier and Katoen, 2008; Lamport, 2015). There are many publications, but the pure mathematical notation is quite far away from a language an engineering practitioner would be able to use.

For electronic circuit design, there is a proposal for an Analog Specification Language (ASL) by (Steinhorst and Hedrich, 2009), with a detailed proposal of language elements and some examples. In (Schamai, 2013) the idea for formalizing a natural-language requirement into a requirement violation monitor is presented. In runtime verification, monitors are expressed in some variant of linear temporal logic expressions and to generate efficient code for the actual monitors (Leucker and Schallhart, 2009).

The SIMULINK toolbox "Verification and Validation"[12] from MathWorks is used to define formal requirements in SIMULINK and to automatically test and verify requirements by *simulation*. In the master thesis (Tunnat, 2011) the toolbox has been applied to an aircraft system. Figure 1 is an example from this thesis that shows the essential elements (in the thesis a script was implemented for the report generator of SIMULINK, that combines the textual description in a Word file with the screen shot of the formal definition in Stateflow): The *Detector* delays and/or synchronizes Boolean signals, the *Implies* block is the logical implies operator of Boolean algebra, and *Assertion* expects that its input is always true and triggers a requirements failure if this is not the case. Note, that requirements are defined with 2-valued logic.

Back-up Performance
10 seconds after the BFan is faulty or off, the BUV shall be in FO position and the inflow into the avionic compartment shall be less or equal than 1.5 KG/s and greater or equal than 1 kg/s.

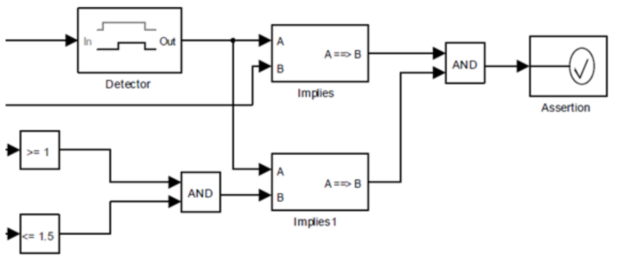


Figure 1. An example of a requirement definition with the SIMULINK toolbox "Verification and Validation". Text and figure from (Tunnat, 2011).

1.3 Modelica_Requirements Prerequisites

In two recent ITEA projects, EUROSYSLIB[13] and OPENPROD[14], part of the research was devoted to how to model requirements in Modelica. The EUROSYSLIB results are reported in (Jardin et al., 2011) and resulted in conceptual work and a prototype Modelica library. The OPENPROD results are partially reported in (Schamai, 2013).

In the ITEA MODRIO[15] project, EDF developed a complete concept for a central industrial scenario: First defining the requirements for a system, then performing an architectural design that shall comply with the requirements and finally evaluating and fine-tuning the architectural design with behavioral models (Bouskela et al., 2015). Furthermore, EDF developed the special language FORM-L (Thuy, 2014) to describe requirements in a formal way but close to the (textual) notation used by system designers. EDF evaluated and refined the language on a larger benchmark example (Thuy, 2013). In (Garro et al., 2014) it was systematically evaluated how to map FORM-L language elements and ideas to Modelica. The above work, including new investigations of Dassault Aviation, finally resulted in the Modelica_Requirements library described in the following sections.

2 Modelica_Requirements Library

The top-level view of this library is shown in Figure 2. The library has about 200 model and block components and about 50 functions. It is provided under the Modelica License 2, and can therefore be used in commercial applications without essential restrictions. The most important sub-libraries are discussed in the following sub-sections.

[7] SysML: http://www.omgsysml.org
[8] DOORS: http://www-03.ibm.com/software/products/en/ratidoor
[9] Reqtify: http://www.3ds.com/products-services/catia/capabilities/requirements-engineering/reqtify/
[10] OSRMT: http://sourceforge.net/projects/osrmt/
[11] formalmind studio: http://formalmind.com/studio
[12] SIMULINK toolbox "Verification and Validation": http://www.mathworks.com/products/simverification

[13] EUROSYSLIB: https://itea3.org/project/eurosyslib.html
[14] OPENPROD: https://itea3.org/project/openprod.html
[15] MODRIO: https://itea3.org/project/modrio.html

2.1 Two- and Three-valued Logic

Defining elements with formal logic requires defining an appropriate data type. All programming languages support two-valued logic. In Modelica, the data type `Boolean` is used for this purpose. FORM-L uses three-valued logic. Also, several publications in this area suggest using three-valued logic, see for example (Schamai, 2013).

Important reasons for using three-valued logic are:

(1) In certain situations it is not possible to state whether a property is violated/false or satisfied/true. For example the FORM-L operator

```
        during(condition, check)
```

is defined as: "As long as the `condition` is `true`, `check` must be `true`". However, what return value should be used, when `condition` is not `true`? (e.g. when the component to be checked is not "in operation"). This case is not defined and therefore the operator should neither return `false` nor `true`, but `undefined`. There are also operators where during a first time range, the return value of the operator is not defined and therefore the best meaningful value to return is `undefined`. With two-valued logic the user has to either return two Booleans to describe this situation, or somehow select a value `false` or `true` in such cases. The problem is that logical expressions that depend on such an arbitrarily selected value may make a required property violated or satisfied, although in reality it is undecided and this may either give an overly optimistic or an overly pessimistic view.

(2) Simulations with requirement models should determine whether a required property is violated. A simulation may, however, not evaluate a defined requirement model (e.g. if only simulations are performed where the model to be checked is not "in operation"). With three-valued logic this situation can be indicated by, e.g. the value `undecided`. With two-valued logic it cannot be stated that a simulation did not test all required properties, and when the simulation run returns with "all required properties satisfied", this might be too optimistic or simply wrong.

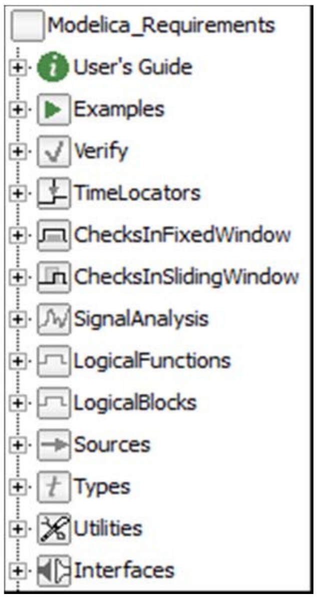


Figure 2. Modelica_-Requirements library.

Three-valued logic has the following drawbacks:

(1) There are several types of three-valued logic definitions, such as Kleene's, Lukasiewicz's, Bochvar's and other logics (Lukasiewicz, 1920; Bochvar, 1937; Breuer, 1972; Rescher, 1969). Some operators, like "`or`" and "`and`" are identical in the different schemes, but the `implies(a,b)` operator is not. For an user it is not obvious which three-valued logic is used in a system and what the consequences are.

(2) Modelica has already many operators and functions for two-valued logic and also users will have many models utilizing two-valued logic. If three-valued logic alone were to be used for requirements modeling, then a large amount of existing code could not be reused.

It is clear that two-valued logic must be supported in order to use existing code and to support the well-known view of the user on logical expressions, as well as language elements such as **`if`**/**`else`** or **`while`**. On the other hand, two-valued logic alone has disadvantages for requirements modeling as sketched above. For these reasons, in the Modelica_-Requirements library two-valued logic, as well as a *restricted form of three-valued logic* is used. The three-valued logic is defined by enumeration `Property` (in sub-library `Types`):

```
 type Property = enumeration(Violated,
                             Undecided,
                             Satisfied);
```

Only functions and blocks with three-valued logic input and/or output arguments are used where the semantics can be defined mathematically in a uniquely accepted way that is also natural and obvious for the user. For example, the function

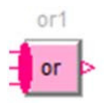
```
        during(condition, check)
```

is provided with `Boolean` input arguments `condition` and `check`, and a `Property` return value. On the other hand, a function `implies(..)` with three-valued logic input/output arguments is not provided because different types of three-valued logics are in use and the result value is not obvious for a user. Also cast functions from `Boolean` and `Integer` to `Property` and from `Property` to `Boolean` and `Integer` are provided. The mapping from `Property` to `Boolean` is not unique, because it is not obvious how to map the value "`Undecided`" to a `Boolean`. This issue is resolved by requiring users to specify the mapping with a second input argument:

```
  Property p = …;
  Boolean  b;
equation
  b = PropertyToBoolean(p,undecided=true);
```

To simplify the view for the user, most functions and blocks have at most one input argument and/or one output argument of type `Property`. The only exceptions are the 3-valued blocks to model the **`or`**, **`and`**, **`not`** operators in 3-valued logic, for which a commonly accepted unique definition exists. For example, the `LogicalBlocks.PropertyOr` block is defined as (in the next figure, three connection lines have been drawn to instance "`or1`"):

```
input  Property u[:];
output Property y;
```



where y = u[1] or u[2] or u[3] or …, and using the truth-table (here for two inputs):

u[1] **or** u[2]	*Violated*	*Undecided*	*Satisfied*
Violated	Violated	Undecided	Satisfied
Undecided	Undecided	Undecided	Satisfied
Satisfied	Satisfied	Satisfied	Satisfied

2.2 Graphical Layout

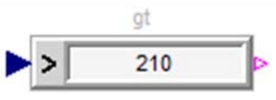
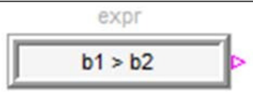
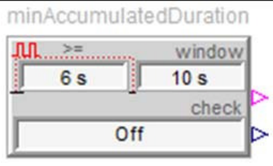
It is expected that the Modelia_Requirements library is utilized by users, such as system architects, without requiring that they be simulation specialists. For this reason an effort was made to improve the usual graphical appearance of models/blocks (within the limitations of Modelica). The following principles are used:

(1) All entries of a parameter menu are displayed in the icon, in order that it not be necessary to inspect the menu to understand the parameterization (as a consequence, a menu, and therefore a block, must be simple and can have at most 3 or 4 input fields).

(2) All such menu entries are defined as "input fields" to make visually clear that the user can provide values (see examples below).

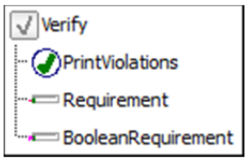
(3) The instance name is displayed above the icon, but in light grey, in order that it not disturbs the layout too much. One could remove the instance name completely from the icon, but it is then no longer so easy to select plot variables by name.

Here are some examples:

$y = u > 210$	
$y = b1 > b2$	
y = true when off has been true for more than 6 accumulated seconds during any 10 second time window.	 minAccumulatedDuration >= window 6 s 10 s check Off

2.3 Definition of Required Properties

In sub-library `Verify` blocks are present to (a) define that a `Property` or `Boolean` signal is a required property and (b) to print a log summary after a simulation (see figure). An example for the usage of block `Requirement` is shown in the next figure:


Verify
PrintViolations
Requirement
BooleanRequirement

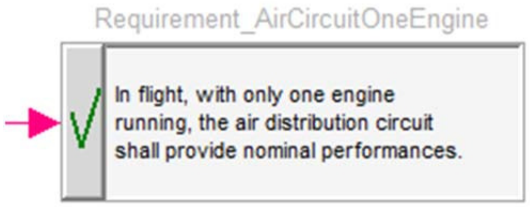

Requirement_AirCircuitOneEngine
In flight, with only one engine running, the air distribution circuit shall provide nominal performances.

Figure 3. Example on how to define a required property.

The left hand arrow is an input signal of type `Property`. In the icon, the content of parameter `text` is displayed that should contain a textual description of the required property. For this, a new annotation "`AutoLineBreak`" is proposed that displays a String parameter in the icon with automatically selected line breaks (so that the text with a given font, here 8pt, is displayed within the surrounding box):

```
parameter String text annotation(AutoLineBreak=true);
```

The Requirement block monitors its property input over a simulation and computes its status at the end of the simulation run:

- *Requirement is violated*:
  Input is `Violated` at least once.
- *Requirement is untested*:
  Input is `Undecided` for the complete simulation run
- *Requirement is satisfied*:
  Input is `Satisfied` at least once, and is never `Violated`.

Determining this status is more difficult than one would expect, because during event iteration a requirement may become temporarily violated, but at event restart the requirement may no longer be violated. To avoid false messages of this type, one has to determine whether a requirement is violated at event restart. This is achieved with the following Modelica code:

```
when not terminal() and change(property) then
   if not pre(atLeastOneFailure) and
     property == Property.Violated then
     atLeastOneFailure = true;
     firstFailureTime  =  time;
  elseif pre(atLeastOneFailure) and
         time <= firstFailureTime and
         (property==Property.Satisfied or
         property==Property.Undecided) then
     atLeastOneFailure =  false;
     firstFailureTime  =  startTime - 1;
  end if;
end when;
```

The when-clause becomes active, whenever `property` changes its value. If `property` became `Violated` the first time, this is marked with `atLeastOneFailure` = `true`. If `property` is changing at the current event iteration, determined by time <= `firstFailureTime`

(time is not changing at an event, and therefore this expression will be true at the same event instant), again a check is made whether `property` is no longer `Violated`. In this case, `atLeastOneFailure` is set back to `false`.

The information about the instance name of the requirement, the requirement text and its status are stored on a log file in textual format. This log file could be processed after the simulation run for example by a script. Additionally, the user can drag the block `PrintViolations` to the top level of his/her model, see Figure 4.

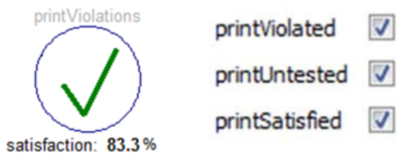
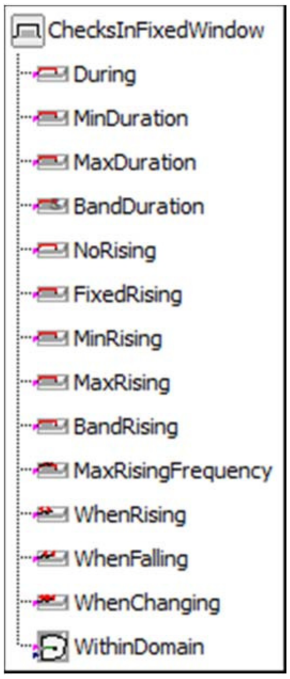


Figure 4. Defining requirement status log
(left figure: icon; right figure: parameters of the block)

This block prints a detailed summary of the status of all requirements to the output window. The output can be configured, see right side of Figure 4. Furthermore, the "satisfaction" factor, that is the percentage of requirements with status = `Satisfied`, are dynamically displayed in the icon (see left side of Figure 4) and stored in the result file, to give a quick overview about the requirement status.

2.4 Checks in Fixed Windows



In sub-library `ChecksIn-FixedWindow` (see figure to the right) blocks are present that determine whether a particular property is fulfilled or not in a *given time window*: Whenever the Boolean input `condition` is true, the property is checked, otherwise the property is not checked (and the output is set to `Undecided`). Properties that can be checked are for example, that input `check`

- must be true for a minimum and/or a maximum duration,
- must have a minimum and/or a maximum number of rising edges.

For example, with block `MaxRising`, see Figure 5, it is stated that the number of rising edges of `check` is limited during every true `condition` phase. The left input arrow is `condition` and the lower input field is `check = engineStart`, so that at most three tries of `engineStart` (becoming `true`) are allowed in the start phase (`condition = true`).

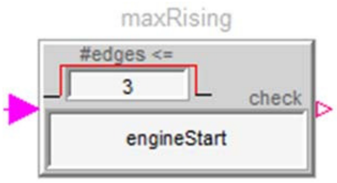


Figure 5. Example for MaxRising block.

In a first design, `check` was not provided by an input field, but by an additional input connector to the left. In larger use cases, like the EDF Backup Power Supply (Thuy 2013), it turned out that the diagram layer of the requirement models became hard to understand due to the many connection lines. This issue could be reduced by using an input field with a name for the `check` signal instead of a connector.

The implementation of most of the blocks in this sub-library is straightforward. For example, the `MaxRising` block is implemented as[16]:

```
initial equation
  countRising = 0;  // number of rising edges
  y = if condition then Property.Satisfied
                   else Property.Undecided;
equation
  when condition then
    countRising = 0;  y = Property.Satisfied;
  elsewhen condition and check then
    countRising = pre(countRising) + 1;
    y = if countRising <= nRisingMax then
         Property.Satisfied  else Property.Violated;
  elsewhen not condition then
    countRising = 0;   y = Property.Undecided;
  end when;
```

A typical simulation result is shown in the next figure:

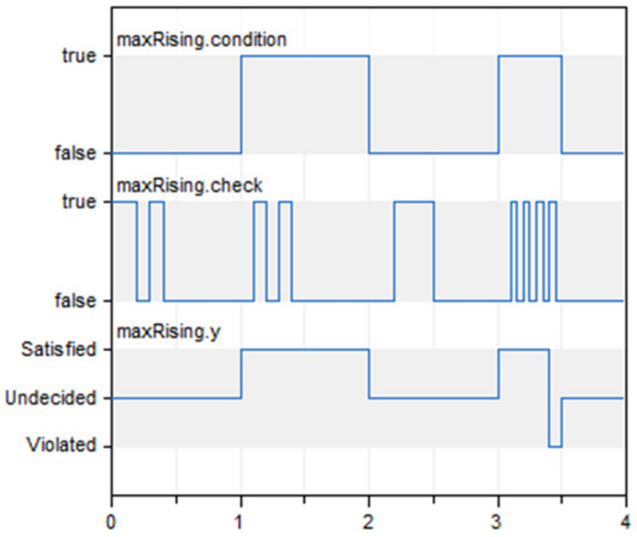


Figure 6. Simulation result for example of Figure 5.

Note, that between 3.4s .. 3.5s the output is `Violated`, because there have been 4 rising edges of check.

[16] Rising edges are not counted at the time instant when condition becomes true or when it becomes false.

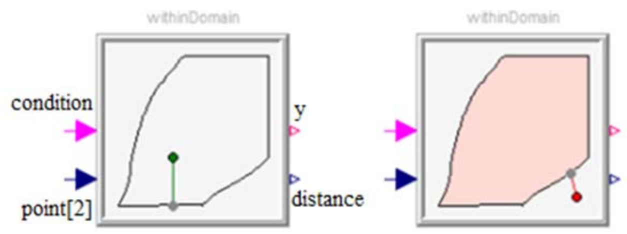


Figure 7. Example for `WithinDomain` block (left figure: point is within the domain, right figure: point is outside the domain)

`WithinDomain` is a more complicated block, see left part of Figure 7. This block defines a domain with a polygon and the requirement is that the input point (a vector of size 2 defining the x- and y-coordinate of the point) must be within this domain. For example, in a passenger aircraft the "*time to complete a cabin pressure change*" (x-coordinate) and the "*cabin altitude rate of change*" (y-coordinate) must be within a given 2-dimensional domain that can be described by the `WithinDomain` block.

The actual polygon is displayed in the icon, together with the point (= green circle) and the nearest distance of the point to the polygon. After a simulation run, a diagram animation shows the actual status. In the right part of Figure 7 the point is outside of the polygon and then the domain and the point is displayed in red. Output `y` is

- `Undecided` if `condition` = `false`,
- `Satisfied` if `condition = true` and the point is within the polygon and
- otherwise it is `Violated`.

Displaying the polygon, the point and the distance in the icon is performed with the standard Modelica annotation `DynamicSelect(..)` that allows an element in an icon to be displayed dynamically. Determining the distance of a point to a polygon is a standard task in computer graphics. In the block a pure Modelica implementation is used. The relationships of one line of the polygon are displayed in Figure 8:

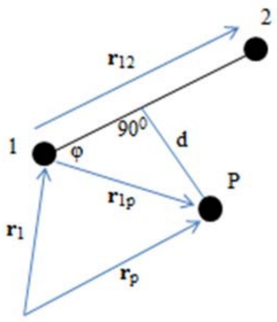


Figure 8. Relationships between one polygon line 1→2, point P and the closest distance d of P to this line.

The corresponding equations are:

$$\begin{aligned} \boldsymbol{r}_{12} &= \boldsymbol{r}_2 - \boldsymbol{r}_1 \\ \boldsymbol{r}_{1p} &= \boldsymbol{r}_p - \boldsymbol{r}_1 \\ \boldsymbol{r}_d &= \boldsymbol{r}_1 + \lambda \cdot \boldsymbol{r}_{12} \end{aligned} \tag{1}$$

The cosine φ of the angle between vectors $\boldsymbol{r}_{12}$ and $\boldsymbol{r}_{1p}$ can be either computed with the relationships in a triangle, or with the dot-product, where λ with $0 \le \lambda \le 1$ characterizes the point $\boldsymbol{r}_d$ on the line with the shortest distance to P:

$$\cos\varphi = \frac{\lambda \cdot |\boldsymbol{r}_{12}|}{|\boldsymbol{r}_{1p}|} = \frac{\boldsymbol{r}_{12} \cdot \boldsymbol{r}_{1p}}{|\boldsymbol{r}_{12}| \cdot |\boldsymbol{r}_{1p}|} \tag{2}$$

and therefore

$$\begin{aligned} \lambda &= \max\left(\min\left(\frac{\boldsymbol{r}_{12} \cdot \boldsymbol{r}_{1p}}{\boldsymbol{r}_{12} \cdot \boldsymbol{r}_{12}}, 1\right), 0\right) \\ d &= |\boldsymbol{r}_{1p} - \lambda \cdot \boldsymbol{r}_{12}| \end{aligned} \tag{3}$$

Equations (3) are applied on every segment of the polygon, and the smallest distance d to all of the segments is selected. Another algorithm computes whether point P is within or outside of the polygon and d is set to a negative value if P is outside of the polygon.

2.5 Time Locators

The condition inputs of the blocks from sub-library `ChecksInFixedWindow` are Booleans that may originate from quite different sources. Due to the importance of these conditions, sub-library `TimeLocators` provides often occurring continuous-time locators, that are temporal operators to define the condition interval of interest (see figure to the right). The outputs of these blocks are Booleans that can be used directly as condition inputs to the blocks of `ChecksInFixedWindow`. FORM-L (Thuy, 2014) has also more complex type of time locators. It is planned to support them as well.

Modelica does not have an "Event" data type. Instead, rising or falling edges of Boolean variables are used to define a time instant of interest that might be described in other modeling systems by an "Event". The following blocks of sub-library TimeLocators are currently available:

- `Every`: Output is true during every interval for a defined duration.
- `Until`: Output is true until first rising edge of input.
- `After`: Output is true after first rising edge of input.
- `AfterFor`: Output is true after rising edge of input for a defined duration.
- `AfterUntil`: Output is true, after rising edge of input 1 until the rising edge of input 2

The implementation of these blocks is straightforward. In Figure 9 an example from (Thuy, 2013) is shown using block `AfterUntil`. This example concerns the generator of a Backup Power Supply system:

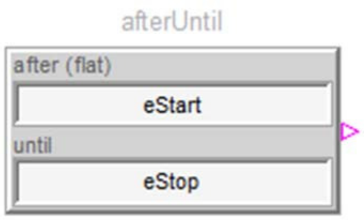


Figure 9. Example for block `AfterUntil`.

The generator can signal several events (= rising edges of Boolean signals), including `eStart` (it has started) and `eStop` (it has stopped). Therefore, Figure 9 defines the time periods where the generator is running. For these time periods required properties might be defined with blocks from sub-library `ChecksInFixed-Window`.

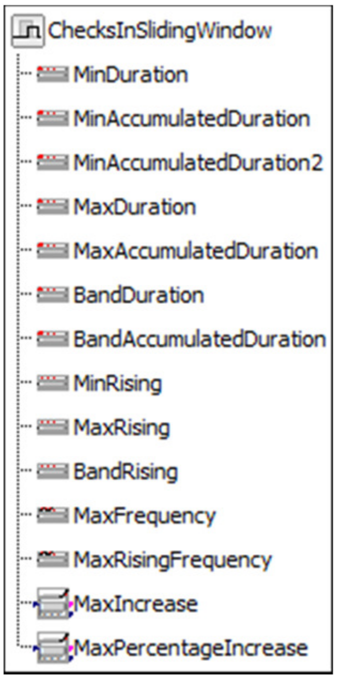
The `AfterUntil` block is implemented as:

```
    input Boolean u1 "Boolean input 1 (after)";
    input Boolean u2 "Boolean input 2 (until)";
    output Boolean y  "= true, after rising edge of u1
                              until rising edge of u2";
  initial equation
    y = u1;
  equation
    when u1 then
       y = true;
    elsewhen u2 then
       y = false;
    end when;
```

A simulation result is shown in Figure 10: The generator is running (`afterUntil.y = true`) between two rising edges of `eStart` and `eStop`.

2.6 Checks in Sliding Windows

In sub-library `Checks-InSlidingWindow` (see figure to the right) blocks are present that determine whether a particular property is fulfilled or not in a *sliding time window*. For example, if a sliding time window has size T and t is the actual time instant, then in every time range $[t - T, t]$ the property must be fulfilled.



Evaluating a property in a sliding time window requires storing the values of the relevant signals in a buffer that covers "essential" signal values in the past at least up to time t - T, and operating on this buffer. For Boolean signals a buffer has been

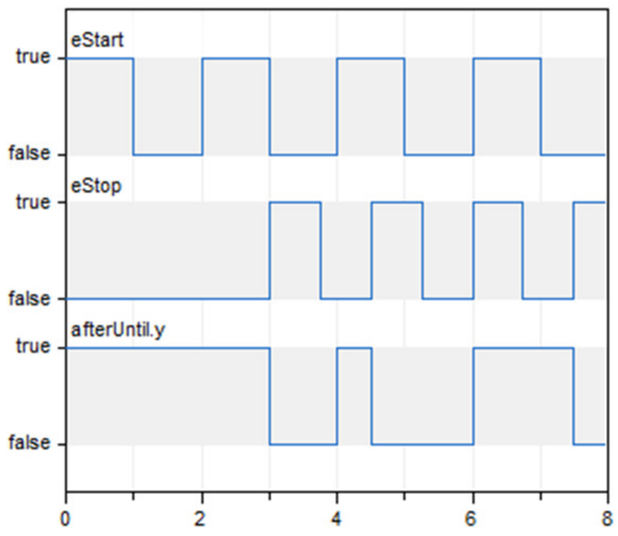
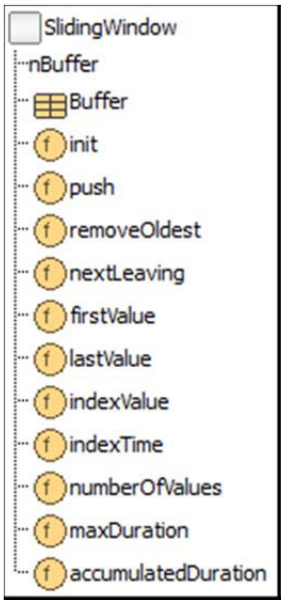


Figure 10. Simulation result for example of Figure 9.

designed which is available as `Internal.Sliding-Window` (see the figure on the right). This is a package consisting of a record `Buffer` in which past values are stored and a set of functions operating on this record. The current implementation is a pure Modelica implementation to gain experience and figure out the right function interfaces. Since a "memory" is needed that is passed between Modelica functions, the size of this memory must be fixed at compilation time and the complete buffer must always be copied, once an element in this buffer is changed. It is planned to replace this implementation by a C-implementation with a Modelica `ExternalObject` to get rid of these restrictions.



The `SlidingWindow` buffer package is basically a queue where elements with a time stamp t are inserted at the top and elements with a time stamp older then t-T are removed at the bottom. The memory of the queue is defined as (where `nBuffer`=20 is a defined constant):

```
record Buffer "Memory of sliding window"
  Modelica.SIunits.Time T "Length of sliding time win.";
  Modelica.SIunits.Time t0 "Time instant where sliding
                              time window starts";
  Modelica.SIunits.Time t[nBuffer] "Time instants";
  Boolean b[nBuffer] "Values at corresp. time instants";
  Integer first          "Index of first element in buffer";
  Integer last           "Index of last element in buffer";
  Integer nElem          "Number of elements in the buffer";
end Buffer;
```

Some of the functions operating on this buffer are sketched at hand of block `MinAccumulated-Duration2`, see Figure 11.

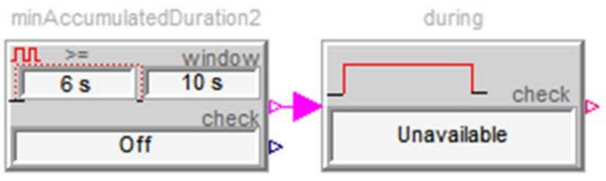


Figure 11. Example for `MinAccumulatedDuration2`

This example models the following requirement from (Thuy, 2013):

> *When the MPS (Main Power Supply system) is switched off, signaled by Boolean Off, then the MPS must be declared Unavailable when it has been off for more than 6 accumulated seconds during any 10 seconds time window.*

This is achieved in the following way: Component `minAccumulatedDuration2` outputs `true`, if in any time window of length 10 s variable `Off` was accumulated `true` for at least 6 s. This signal is the input to component `during` which requires that whenever the input is `true`, variable `Unavailable` must be `true` as well. In that case the block outputs `Satisfied`. If the input of `during` is `true` and `Unavailable = false`, the requirement is clearly violated and the `during` block outputs `Violated` (if the input is `false`, the block outputs `Undecided`).

Block `MinAccumulatedDuration` outputs a `Property` whereas `MinAccumulatedDuration2` outputs a Boolean. The difference is only during the initial phase $t < t_0 + T$ where the first block returns `Undecided` if the property is violated, and the second returns `false`. The `MinAccumulatedDuration2` block is implemented in the following way

```
  import Modelica_Requirements.Internal.SlidingWindow.*;
   parameter Modelica.SIunits.Time window;
   parameter Modelica.SIunits.Time lowerLimit;
   input Boolean check(start = false);
   output Boolean y "= true if property satisfied";
   output Real accDuration;
protected
   Buffer buffer "Buffer for sliding window";

initial equation
   buffer = push(init(T,time), time, check);
   pre(check) = check;

equation
   when change(check) then
      buffer = push(pre(buffer), time, check);
   end when;
   accDuration = accumulatedDuration(buffer, time, check);
   y = accDuration >= lowerLimit;
```

The `Buffer` functions have the following tasks:

- `init(T,time)` returns an instance of `Buffer` and initializes it with the length of the sliding time window T and the initial time instant time t.
- `push(init(T,time), time, check)` generates and initializes a `Buffer` and stores one element (= the initial time instant and the value of check) in the buffer. At the same time, this call removes values from the buffer that have a time stamp older then time - T. The function returns a copy of the buffer.
- The code
  ```
   when change(check) then
      buffer = push(pre(buffer), time, check);
  end when;
  ```
  is executed whenever check changes its value (and at that time instant an event occurs). The function call stores the actual time instant and the value of check in the buffer from the last event instant and removes older values from the buffer. The updated buffer is then returned at the actual event instant.
- The code
  ```
  accDuration = accumulatedDuration(buffer, time, check);
  ```
  is executed during *continuous-time integration*, that is whenever the integrator requires a model evaluation. The function call accumulatedDuration(..) computes the accumulated time duration where the values of check in the buffer have been true during the time window time – T and returns this value. The third argument check of this function call is usually ignored, but is used if the buffer is empty.
- The code
  ```
   y = accDuration >= lowerLimit;
  ```
  triggers a state event when the accumulated time duration crosses its limit and y changes its value from `false` to `true` or from `true` to `false` depending on the crossing direction.

2.7 Utility Functions and Blocks

Besides of the already discussed core blocks, the Modelica_Requirements libray has also quite a lot of utility functions and blocks that might be useful to formally define a requirement:

Sub-library `SignalAnalysis` consists of blocks to compute exact or approximate derivatives, an integrator that can be controlled by a trigger signal, a moving average filter, and other blocks.

Sub-library `LogicalBlocks` provides blocks to convert between Boolean, Integer and Property signals, comparing Real signals as well as logical operators (`not, or, and`) on `Property` signals. Some of these blocks are also available in the Modelica Standard Library. However, since they seemed to be often needed for requirements modeling, they have been provided additionally with the graphical layout used for the Modelica_Requirement blocks.

When textually modeling or when implementing blocks, a set of useful functions for 2- and 3-valued logic have been collected in sub-library `LogicalFunctions`. Some of these functions are motivated by the FORM-L language and provide set-like functionality on Modelica vectors. For example function `exist(..)` has a Boolean input vector and returns true if at least one element of this vector is true. In combination with Modelica's reduction expressions, quite powerful compact formulations are possible, as shown in the next example:

```
// Define a set of pumps
Pump pumps[3] = {Pump(isActive=time < 1 or
                          time > 2 and time < 3),
                 Pump(isActive=time < 0.5 or   time > 2.5),
                 Pump(isActive=time > 1.5 and time < 1.9)};

// At least one pump must be active all the time
Boolean atleastOnePumpActive =
                    exists({p.isActive for p in pumps})
```

Sub-library `Examples` contains a large set of examples to demonstrate and assess the components of the library. Every component of the library is present in at least in one example (so class coverage is 100 %).

There is also a growing set of application specific examples that can be used as templates in actual projects. For example, sublibrary `Modelica_Require-ments.Examples.AircraftRequirements` contains typical requirement definitions used in aircraft systems:

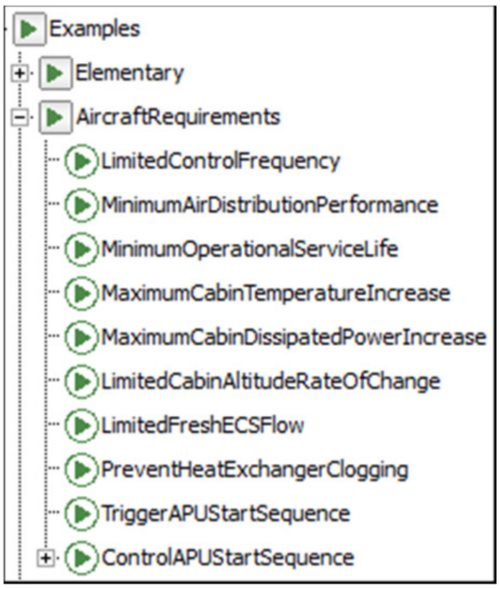

Examples
Elementary
AircraftRequirements
LimitedControlFrequency
MinimumAirDistributionPerformance
MinimumOperationalServiceLife
MaximumCabinTemperatureIncrease
MaximumCabinDissipatedPowerIncrease
LimitedCabinAltitudeRateOfChange
LimitedFreshECSFlow
PreventHeatExchangerClogging
TriggerAPUStartSequence
ControlAPUStartSequence

Figure 12. Sub-library of aircraft specific requirements from Dassault Aviation.

Every example contains a short definition of the requirement (as it is typically present in design documents), the corresponding Modelica model to verify the requirement together with some simple test signals. For example, the requirement "*In the cabin area, the temperature increase should not exceed 3°C per hour.*" is verified with the following model (the input is cabin temperature as function of time defined in a table):

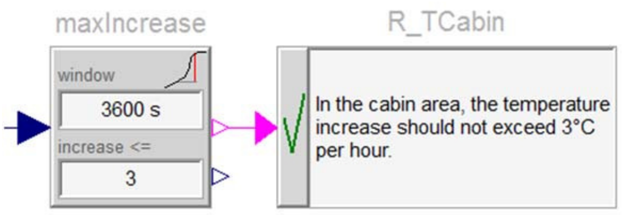

maxIncrease
R_TCabin
window
3600 s
increase <=
3
In the cabin area, the temperature increase should not exceed 3°C per hour.

Figure 13. An aircraft requirement to assess the limited allowed temperature increase in the cabin area.

3 Textual Definition of Requirements

In the previous examples requirements have been defined graphically. Some users prefer, however, a pure textual definition because requirements can be formulated and inspected in a more compact form. It turned out that with current Modelica it is not possible to define requirements in a convenient way, if the requirement model contains a memory. For this reason, section 3.1 sketches a proposal for a Modelica extension to improve this situation.

3.1 Calling Blocks as Functions

The goal is to introduce functions with memory and events into Modelica. Since blocks already support memories and events, the simplest extension seems to be to introduce the feature that blocks can be called as functions. However, functions have a different type system than blocks: Arguments in functions can be identified by position, whereas in blocks they must be identified by name. For this reason, the "function 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. Afterwards, the optional calling mechanism of functions and the required calling mechanism of blocks are identical.

The basic idea is simple: (a) A block is called using its class name, (b) the inputs to the block call are defined by the usual modifiers of a block declaration, (c) one output of a block must be defined as return value of the call, by appending its name with ".name" to the "function call". Take for example the block `MaxRising` of Figure 5. It could be expressed as a declaration in a pure textual form:

```
import Modelica_Requirements.ChecksInFixedWindow.*;
import Modelica_Requirements.Types.Property;

Property property = MaxRising(condition = start,
                              check = engineStart,
                              nRisingMax = 3).y;
```

Note, (…).y defines that output variably y of block Modelica_Requirements.ChecksInFixedWindow.MaxRising is computed and assigned to variable property. The above declaration is transformed (conceptually) to standard Modelica with a formal mapping rule resulting in:

```
MaxRising MaxRising_1(condition = start,
                      check = engineStart,
                      nRisingMax = 3);
Property property = MaxRising_1.y;
```

This shows that a tool has to introduce a declaration for an auxiliary component (here: MaxRising_1) and use the output of this block (here: MaxRising_1.y) in the expression where the call of `MaxRising` occurred.

The block calling can be nested in expressions. However, in order that the simple mapping rule above can be applied by a tool, several restrictions are necessary. Most importantly: A block can be called as a function *only* in the declaration section (with the additional restriction that it cannot be called in an if-expression). The proposed extension above was implemented in prototypes of Dymola and OpenModelica (Buffoni and Fritzson, 2014)

3.2 Examples

In the Modelica_Requirements library several textual examples are present in sub-library Examples.Textual, especially part of the EDF Backup Power Supply benchmark (Thuy, 2013). Example code:

```
Requirement R1(property=WhenRising(condition=Off,
                                   check=MPSVoltage < 170).y,
               text="MPS CAN be declared Off when
                     the voltage gets below 170 V");
Requirement R2(property=during(MPSVoltage < 160,
                               check=Off),
               text="MPS MUST be declared Off when
                     the voltage gets below 160 V");
```

It is a matter of taste whether a user prefers a graphical or a textual definition – the Modelica_Requirements library supports both choices.

4 Utilizing Requirement Models

Once requirements are defined they are typically associated with behavioral models and various techniques are used to verify these requirements based on simulations. Integrating the modelled requirements manually in test scenarios of behavioral models may be a tedious task and there is a clear need to automate this process. Several proposals have been discussed within the MODRIO project for this purpose, especially (Bouskela et al., 2015; Schamai, 2013; Schamai et al., 2014) and also on using Modelica scripts for associating requirements with behavioral models. In (Elmqvist et.al, 2015) two new Modelica language constructs are proposed to simplify this "automatic binding" task. These language elements are also useful for other applications, for example to compute the total mass of a multibody system or for contact handling.

The current development stage allows to check in every simulation run whether the defined requirements are satisfied or violated (or are not tested). Industrial applications would typically involve additional software on top of this base functionality, such as:

- *Monte Carlo Simulation*
  Various initial conditions, operating points, and/or external disturbances are randomly generated within meaningful bounds and for every scenario simulations are performed. This brute force method for evaluation of dynamic systems is standard in many software tools.
- *TestWeaver*
  TestWeaver (Junghanns et al., 2008) is a software tool from QTronic to construct automatically test scenarios, especially also for Modelica models. The goal of the tests is to drive the system in a state where it violates its specifications. A major application area are systems where the inputs have a countable number of values or areas (and these values vary over time).
- *Anti-Optimization*
  A technique used at DLR-SR to evaluate controller designs, see e.g. (Joos, 2011): A special parameter optimization problem is formulated, in order to find an operating point of the system (e.g. height or speed of an aircraft), where the controller works as badly as possible. The major application area are systems where the operating region and the requirements are described by continuous signals.

5 Conclusions and Outlook

In this article the design of a new, open source Modelica library was presented to formally model requirements for industrial applications. The design was driven by applications of EDF (power plants, electrical systems) and Dassault Aviation (aircrafts). The basic design is based on the FOrmal Requirements Modeling Language FORM-L from (Thuy, 2014). The library in the current form (July 2015) is in an Alpha version. It is planned to additionally implement FORM-L components with overlapping sliding time windows, to include dynamic response and FFT requirement blocks from (Kuhn et al., 2015), to introduce continuous indicators for the properties where this is possible (in order that property blocks can be directly used as constraints or criteria for optimization-based methods), to add use cases of EDF and Dassault Aviation, and to connect the library to existing verification frameworks, such as TestWeaver.

Acknowledgements

This paper is based on research performed within the ITEA2 project MODRIO. Partial financial support of the German BMBF, the French DGE, and the Swedish VINNOVA are highly appreciated.

Helpful discussions with Martin Kuhn (DLR) are appreciated. Furthermore, helpful discussions with members of the Modelica Association to define the fine details of the "calling-blocks-as-functions" approach, are also appreciated. Finally, improvement suggestions of the reviewers of this paper are appreciated.

References

C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press. ISBN 978-0-262-02649-9, 2008.

D. Bochvar *Ob odnom trekhznachnom ischislenii i ego primenenii k analizu paradoksov klassicheskogo rasshirennogo funkciona'nogo ischislenija*. In Matematicheskij Sbornik 4, no 46, pp. 287–308. 1937.

D. Bouskela, N. Thuy, and A. Jardin . *D2.1.1 – Modelica extensions for properties modelling, Part II: Modeling Architecture for the Design Verification against System Requirements.* Internal report, ITEA2 MODRIO project, March 2015.

M.A. Breuer. *A Note on Three-Valued Logic Simulation*. IEEE Transaction on Computer C-21, no. 4, pp. 399-402, 1972.

L. Buffoni and P. Fritzson. *Expressing Requirements in Modelica*. Proceedings of the 55th International Conference on Simulation and Modeling (SIMS 2014), October 21-22, Aalborg, Denmark, 2014.

Dassault Systèmes. *Dymola 2016*., 2015. http://www.Dymola.com

Department of Defense *Aircraf.t Electric Power Characteristics (MIL-STD-704F)*. 1984. Download: http://everyspec.com/MIL-STD/MIL-STD-0700-0799/MIL-STD-704F_1083/

A. Garro, A. Tundis, and M. Otter. *D2.1.1 – Modelica extensions for properties modelling, Part IVb: FORM-L and Modelica: syntax and relationships*. Internal report, ITEA2 MODRIO project, Sept. 2014.

H. Elmqvist, H. Olsson, and M. Otter. *Constructs for Meta Properties Modeling in Modelica*. Accepted for Modelica'2015 conference, 2015.

A. Jardin and D. Bouskela. *D2.1.1 – Modelica extensions for properties modelling, Part I: Users motivation*. Internal report, ITEA2 MODRIO project, Sept. 2014.

A. Jardin, D. Bouskel, N. Thuy, N. Ruel, E. Thomas, L. Chastanet, R. Schoenig, and S. Loembé. *Modelling of System Properties in a Modelica Framework*. Proceedings 8th Modelica Conference, Dresden, Germany, March 20-22., pp. 579-592, 2011. Download: http://www.ep.liu.se/ecp/063/065/ecp11063065.pdf

H. D. Joos. *Worst-case parameter search based clearance using parallel nonlinear programming methods*. In: Optimization based Clearance of Flight Control Laws. Lecture notes in control and information sciences, 416. Springer, pp. 149-159, 2011. ISBN 978-3-642-22626-7. ISSN 0170-8643.

A. Junghanns, J. Mauss, and M. Tatar. *TestWeaver – A Tool for Simulation-based Test of Mechatronic Designs*. Proceedings of the Modelica'2008 Confererence, pp. 341-348, March 3-4, 2008. Download: https://www.modelica.org/events/modelica2008/Proceedings/sessions/session3c4.pdf

M. Kuhn, M. Otter and T. Giese. *Model Based Specifications in Aircraft Systems Design*. Accepted for Modelica'2015 conference, Sept. 2015.

L. Lamport. *Principles and Specifications of Concurrent Systems*, 2015. Hyberbook: http://research.microsoft.com/en-us/um/people/lamport/tla/hyperbook.html

M. Leucker, and C. Schallhart, C. *A Brief Account of Runtime Verification*. Journal of Logic and Algebraic Programming 78, no. 5, pp. 293-303, 2009.

J. Levy, S. Hassen, and T.E. Uribe. *Combining Monitors for Runtime System*. Electronic Notes in Theoretical Computer Science 70, no. 4, pp. 112-127, 2002.

J. Łukasiewicz. *On three-valued logic*. In L. Borkowski (ed.), Selected works by Jan Łukasiewicz, North–Holland, Amsterdam, pp. 87–88, 1920. ISBN 0-7204-2252-3.

Modelica Association. *Modelica, A Unified Object-Oriented Language for Systems Modeling. Language Specification, Version 3.3,* May 9, 2012. https://www.modelica.org/documents/ModelicaSpec33.pdf

OMG. *Requirements Interchange Format (ReqIF)*, 2013. Download: http://www.omg.org/spec/ReqIF/1.1/PDF/ http://www.omg.org/spec/ReqIF/20110401/reqif.xsd

Open Source Modelica Consortium. *OpenModelica*, 2015. https://openmodelica.org/

M. Otter M, L. Buffoni, P. Fritzson, M. Sjölund, W. Schamai, A. Garro, A. Tundis, and H. Elmqvist. *D2.1.1 – Modelica extensions for properties modelling, Part IV: Modelica for properties modeling*. Internal report, ITEA2 MODRIO project, Sept. 2014.

N. Rescher. *Many-valued Logic*, McGraw-Hill, 1969.

W. Schamai. *Model-Based Verification of Dynamic System Behavior against Requirements: Method, Language, and Tool*. Ph.D. Thesis, No. 1547, University of Linköping, 2013.. Download: http://liu.diva-portal.org/smash/record.jsf?pid=diva2:654890

W. Schamai, L. Buffoni, and P. Fritzson. *An Approach to Automated Model Composition Illustrated in the Context of Design Verification*. Journal of Modeling, Identification and Control, volume 35- 2, pages 79—91, 2014.

S. Steinhorst and L. Hedrich . *Targeting the Analog Verification Gap: State Space-based Formal Verification Approaches for Analog Circuits*. CAV 2009, Grenoble, France, 2009. Download http://www.em.cs.uni-frankfurt.de/FAC09/papers/FAC_09_Steinhorst.pdf

N. Thuy. *D8.1.3 – Part 1 The Backup Power Supply*. Internal report, ITEA2 MODRIO project, Nov. 2013.

N. Thuy. *D2.1.1 – Modelica extensions for properties modelling, Part III: FOrmal Requirements Modelling LAnguage (FORM-L)*. Internal report, ITEA2 MODRIO project, Sept. 2014.

M. Tunnat. *Integration modellbasierter Methoden in den Entwicklungsprozess hybrider Flugzeugregelungssysteme am Beispiel des Ventilation-Control-System*. Master thesis, Technical University Hamburg-Harburg, Institut für Flugzeug-Kabinensysteme, 2011.