D2.1.1 – Modelica extensions for properties modeling

Part IV: Modelica for properties modeling

WP2.1 – Properties modelling language
WP2 – Properties modelling and Safety

MODRIO (11004)

Version M40 (3.0)

Date April 21, 2016

Authors

Martin Otter	DLR SR, Germany
Lena Buffoni, Peter Fritzson, Martin Sjölund	Linköping University, Sweden
Wladimir Schamai	Airbus Group Innovations, Germany
Alfredo Garro, Andrea Tundis	University of Calabria, Italy
Hilding Elmqvist	Dassault Systèmes AB, Sweden

Executive summary

The goal in MODRIO WP2.1 is to define **requirements formally** and **check** them **automatically** whenever a system model is **simulated**. It is not intended to perform formal model verification. Details of the overall design are described in deliverable D2.1.1–Part II. This design is informally called "Properties Modeling" in the sequel. One essential part of this design is the newly developed language FORM-L described in deliverable D2.1.1–Part III to formally define requirements in a form that are close to the nomenclature used by system architects.

In the document at hand, D2.1.1–Part IV, a proposal is made how to concretely implement "Properties modeling" with the Modelica Language utilizing the FORM-L language design as inspiration. The following results have been achieved:

- Extensions to the Modelica language have been developed to achieve a more user-friendly, FORM-L like syntax for properties modeling in Modelica. Prototypes in Dymola, OpenModelica and SimulationX have been implemented for these proposals.
- An open source Modelica Library Modelica_Requirements library has been developed (provided in directory D2.1.1\AdditionalMaterial) containing about 150 models/blocks and 60 functions to define requirements formally in Modelica, check them automatically when a model is simulated using blocks from this library and summarizing the result of the check at the end of a simulation. This library has been evaluated with Dymola, OpenModelica and SimulationX.
- Part of the EDF use case for properties modeling, a Backup Power Supply system (D8.1.3-Part I), has been implemented and provided as example in the Modelica_Requirements library.
- Several aircraft specific examples from Dassault Aviation have been added to the Modelica_Requirements library.
- In order that formally defined requirements can be practically used, they must be associated in a convenient way to the corresponding behavioral model (the so called "mapping"). Within the MODRIO project several proposals have been developed, Modelica Change Proposals provided, and prototypes implemented. However, there is not yet a common agreement about the "best" approach. There seems to be agreement for one part of the "mapping" within a Modelica model: Extracting automatically all needed variable values from a model by casting a model instance to a record (containing these variables) and then utilizing this record in a requirements model, see (*MCP-0023, 2016*).

Note, all web-links in this document have been checked for correctness on April 21, 2016.

Contents

1. Overview

The goal in MODRIO WP2.1 is to define **requirements formally** and **check** them **automatically** whenever a system model is **simulated**. Details of the overall design are described in deliverable D2.1.1 – Part II (Bouskela et. al., 2014). One essential part of this design is the newly developed language FORM-L described in D2.1.1 – Part III (Thuy, 2014) to formally define requirements in a form that are close to the nomenclature used by system architects. This design is informally called "Properties Modeling" in the sequel. The FORM-L language is based on Linear Temporal Logic (LTL[1]) and adds many practical details. The main purpose is to check formally defined requirements **by simulation**. It is not intended to perform formal model verification (as done by tools such as SMV, NuSMV, SPIN). For this reason, FORM-L defines quite many high-level operators that can be used for assessment by simulation, but would be impossible to utilize with current state-of-the-art "model checkers" (that perform formal verifications of models).[2]

In the document at hand (D2.1.1 – Part IV) a proposal is made how to concretely implement "Properties modeling" with the Modelica Language (Modelica 2012): The design of the FORM-L language is in an early phase and no FORM-L translator, interpreter or simulator was yet developed, which is also not possible within the budget and time frame of the MODRIO project. Instead, the proposal at hand utilizes the FORM-L language design as inspiration:

- to develop extensions of the Modelica language arriving at a user-friendly syntax for properties modeling in Modelica,
- to implement FORM-L operators with Modelica language elements and provide these operators in a user-friendly way in a Modelica library.
- to evaluate this approach with two Modelica tools: Dymola and OpenModelica.

This document covers only part of the design of (*Bouskela et. al., 2014*). In particular, the following tasks are **not** (yet) **taken care off**:

- Attaching formally defined requirements to a particular system model in a **user convenient way** (the issue is that requirements shall be defined formally without reference to a system model, and there is then the question how to associate such definitions to a specific simulation model). The current status of the discussion is shortly summarized in section 6.3
- Requirement definition in an organization is usually based on information available in databases of the organization. There is the question, how to concretely reference and use information from such databases in the definition of requirements with Modelica.
- Systematically testing the fulfillment of formally defined requirements by performing many (suitable) simulations and presenting the result of the simulations in a compact form to the user.

[1] See for example http://en.wikipedia.org/wiki/Linear_temporal_logic or (Baier and Katoen 2008).

[2] 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: $p \geq p_{cavitate}$. Such a requirement can be assessed by simulation, but not verified by current model checkers.

2. State-of-the-Art of Properties Modeling

In this section, the state-of-the-art to formally define requirements and check them by simulations is shortly summarized. Furthermore, material is collected that might be helpful for implementing properties modeling in Modelica.

2.1 Overview

The standard in industrial applications is to define **requirements in natural language in textual form**, either in reports by using for example Microsoft Word, or with dedicated tool support, the latter especially to get support for collaboration, traceability, coverage analysis. There are a huge amount of tools in this area, for example: Rational DOORS from IBM, Reqtify from Dassault Systèmes, OSRMT (GPL2), formalmind Studio (free). The most important (xml-based) exchange format between requirement tools seems to be ReqIF (OMG 2013).

Defining and processing requirements formally is an area of active research. The mathematics is usually based on temporal logic; see for example (*Baier and Katoen 2008*). There are many publications, but the pure mathematical notation is quite far away from a language that an engineer could use as input in a tool. Furthermore, temporal logic is often not sufficient and a richer framework is needed.

For electronic circuit design, there is a proposal for an Analog Specification Languages (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. The main purpose of the requirement violation monitor is to detect requirement violations without intervening into the analyzed system. The detection is indicated by a verdict that can take 3 values. In a simulation-based testing approach that uses a finite number of executions based on deterministic scenarios, this means that the requirement violation monitor verdict should be enumerated with at least the following values[3]:

- *not evaluated* (default value, provided there is a precondition for the evaluation), which indicates that the requirement was not yet evaluated,
- *violated*, which indicates a violation of the requirement (implying that the requirement was evaluated),
- *not violated*, which indicates that no violation was found (still implying that the requirement was evaluated).

The requirement violation monitor verdict shall be computed at any simulated time instant and can change between *not evaluated*, not *violated*, and *violated* in any possible way. This way violation monitors can be used for online monitoring (Leucker and Schallhart, 2009). The status *not evaluated* means that the precondition is not met and that the requirement cannot be evaluated under the given conditions.

The task of formalizing requirements into violation monitors — i.e., the expression of the violation condition(s) — can be accomplished in many different ways using different formalisms. For example, in runtime verification, monitors are expressed in some higher level specifications (e.g. LTL – Linear

[3] This enumeration is referred to as “three-valued semantics” in (Leucker & Schallhart, 2009) with the literals “inconclusive”, “false” and “true” respectively.

Temporal Logic, RV-LTL – Runtime Verification of LTL, TLTL – Timed Linear Temporal Logic, etc.) which are then used to generate efficient code for the actual monitors (Leucker and Schallhart, 2009). In testing to verify software written in a particular programming language, monitors may be expressed in the very same programming language. Using UML[4] or Modelica models, monitors may be expressed using state machines, equations, or functions. In addition, templates or specialized library functions may be helpful in reducing manual effort and modeling errors.

The choice for using a particular formalism will depend on the modeler's preferences, the skills of the audience (reader of the model or code), project policies, or the list of formalisms supported by the analytical tools that will be used. However, independently of the choice of formalism, the task of converting the natural-language statement into formal language will require correct interpretation and understanding of the requirement statement and the ability to translate that meaning into a model that expresses it identically.

[4] UML: Unified Modeling Language (http://www.uml.org/)

2.2 MIL STD 704f

For the design of language elements, concrete examples and use cases of technical requirements are helpful, such as the EDF Backup Power Supply benchmark (*Thuy, 2013*).

For electrical systems, the publicly available requirements for electrical systems in military aircraft, MIL-STD-704F, should be taken into consideration[5]. Here is an example from this standard how the requirements are formulated:

5.2.4 Abnormal operation. The overvoltage and undervoltage values shall be within the limits of figure 4 for 400 Hz and variable frequency systems.

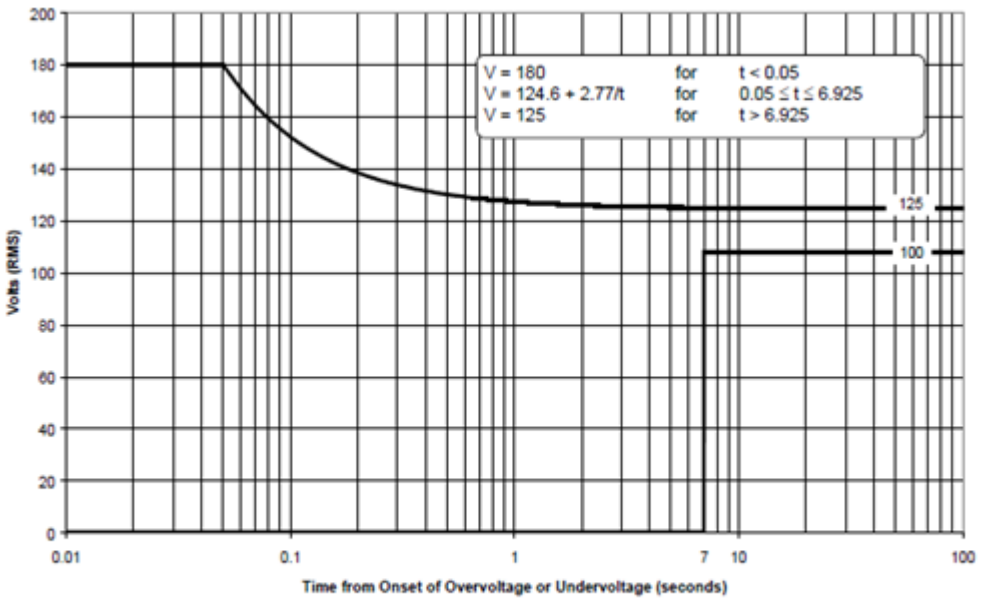


FIGURE 4. Limits for 400 Hz and variable frequency AC overvoltage or undervoltage.

Text and figure from MIL-STD-704F (Dec. 30, 2008).

This indicates that for electrical systems predefined property models should be available to directly represent such characteristics (for example: "*when a Boolean input has a rising edge, then the signal provided as second argument must remain between the limits defined by an analytically given curve*").

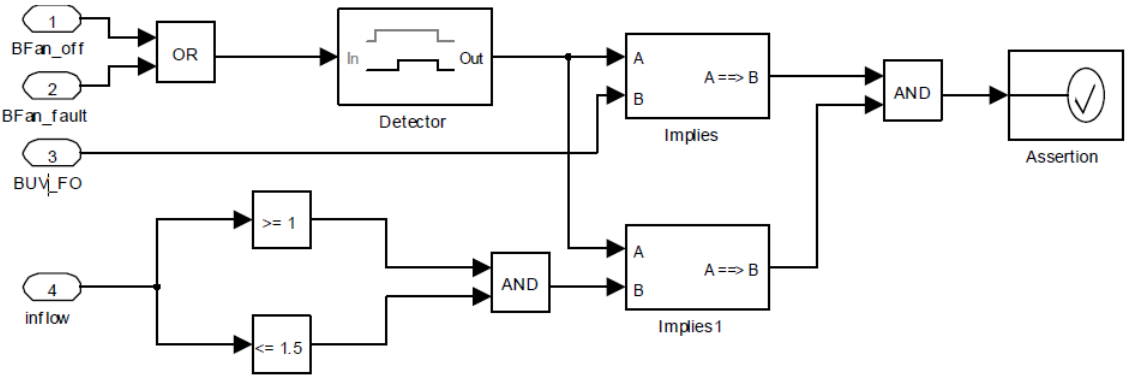
[5] This suggestion is from Martin Kuhn, DLR-SR.

2.3 SIMULINK toolboxes

The Stateflow toolboxes “Validation & Verification”, as well as the “Design Verifier” from Mathworks are used to define formal requirements in SIMULINK and are used to automatically test and verify requirements by simulation, and also for certain restricted models by a model checker. An overview of the “Design Verifier” can be found here.

In the master thesis (*Tunnat 2011*) the two toolboxes have been applied to an aircraft system. Example from the thesis (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):

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.



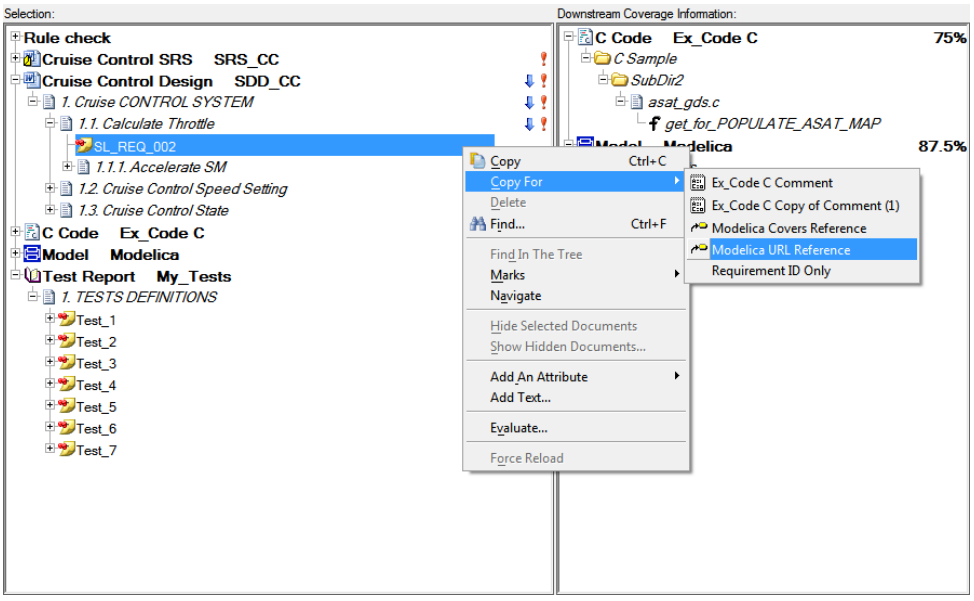
Text and figure from (Tunnat 2011).

As always in SIMULINK, the definition (of the requirements) is implemented in purely graphical form. Note, this is in contrast to the current approach in MODRIO to define requirements directly in a language (FORM-L or Modelica). The example above shows the essential elements: The “Detector” delays and/or synchronizes Boolean signals, the “Implies” block is basically the logical “implies(..)” operator of Boolean algebra, and “Assertion” expects that its input is always true. All about 15 requirement definitions in this thesis use basically these three blocks.

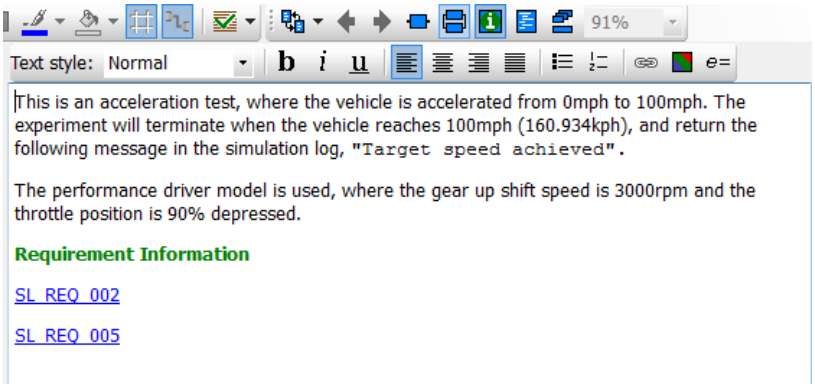
2.4 Reqtify and Dymola

Reqtify and Dymola from Dassault Systèmes support two-way references (Reqtify has a reference to the Modelica Requirements block, where the requirement is formally defined in Modelica, and the Requirements block has a reference to the requirements definition in Reqtify). Reqtify can list which requirements have been tested and are satisfied, etc.

The linking process is illustrated below. To link a Dymola Requirement model to a Reqitify requirement: In Reqtify, right-click on the requirement, select **Copy For > Modelica URL Reference**:



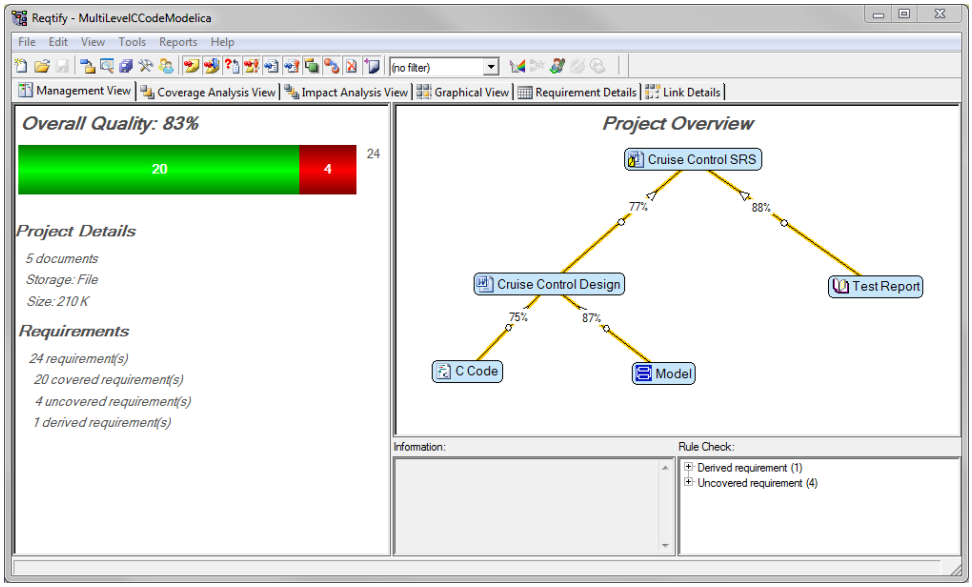
In Dymola, go to the Documentation layer of the model and change to the Info Editor view. Paste the copied link from Reqtify into the model documentation. This creates a hyperlink in the model documentation which has the requirement name. Selecting this link will navigate to the requirement in Reqtify.



To navigate from Reqtify to Dymola, select the Navigate menu item or double-click a requirement in Reqtify to select the corresponding model in Dymola. This opens Dymola (if not yet opened) on the chosen model.

To navigate from Dymola to Reqtify, click on the link in the Info Editor. This opens Reqtify (if not yet opened) on the right document and the chosen requirement is selected.

Reqtify can present coverage of Requirements in Dymola models as shown below:



The summary includes the following categories of requirements:

- Covered category contains requirements covered by Modelica model
- Uncovered category contains requirements not covered by Modelica model
- Derived category contains non top-level requirements covering nothing.

There is currently no coupling between Dymola and Reqtify regarding the outcome of the evaluation of the properties.

2.5 Temporal operators

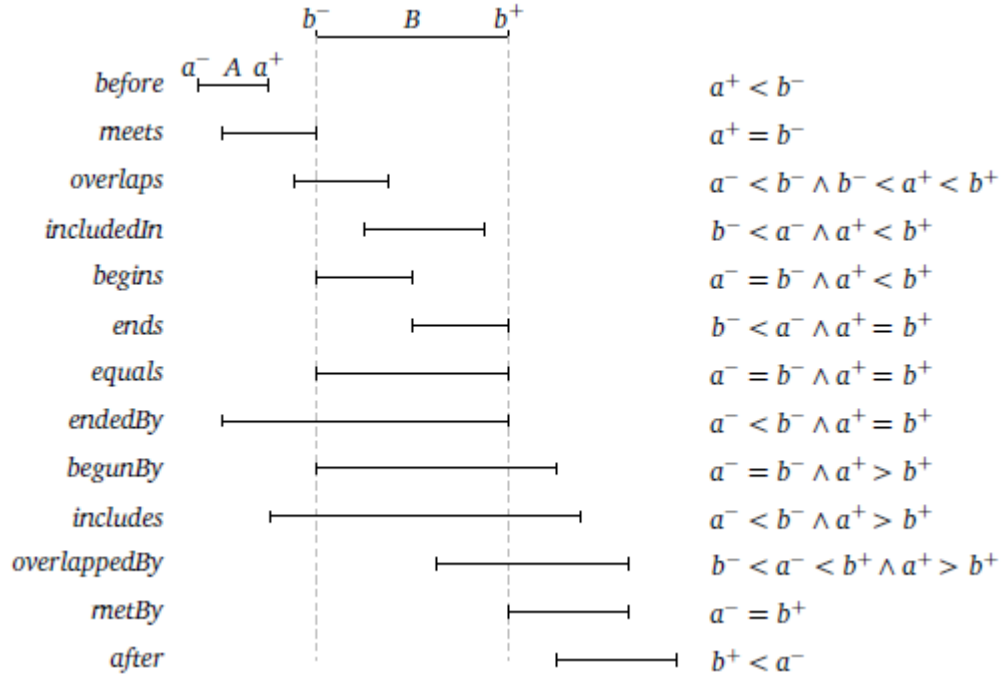
Formal requirements definition usually makes use of temporal operators. In FORM-L also many temporal operators are defined. In this section temporal operators from different references are summarized:

2.5.1 Allen's operators and ISO 19108

The following text and the figure is from (*Grote 2010*):

James F. Allen introduced in 1983 a temporal interval algebra (Allen 1983), which has become a fundamental building block in temporal reasoning, and has been found to be generally applicable to arbitrary types of intervals. It is based on the observation, that only a limited number of topological relationships between two intervals are possible. By enumerating all possible orders of the begin and end points of the intervals, 13 mutually exclusive interval relationships can be identified:

Figure 4.1.: Allen's operators: The 13 possible relationships between two intervals A, B. The 6 relationships below equals are the inverse of the ones above**.**



Text and figure from (*Grote 2010*)

Allens operators are 1:1 used in "ISO 19108 Temporal Operators". In the original publication by Allen, in the dissertation above and in ISO19108 slightly different names are used (in light red the essential differences to ISO19108 are marked):

(Grote 2010)	(Allen 1983)	ISO 19108
before	before	Before
meets	meets	Meets
overlaps	overlaps	Overlaps
includedIn	during	During
begins	starts	Begins
ends	finishes	Ends
equals	equal	Equals
endedBy	finished-by	EndedBy
begunBy	started by	BegunBy
includes	contains	Contains
overlappedBy	overlapped-by	OverlappedBy
metBy	met-by	MetBy
after	after	After

2.5.2 Temporal operators in Stateflow

In Stateflow from Mathworks, the following temporal operators are used:

Syntax	***explanation***
after(n,event)	Returns true if event has occurred at least n times since activation of the corresponding state
before(n,event)	Returns true if event has occurred fewer than n times since activation of the corresponding state
at(n,event)	Returns true at the n-th occurrence of event since activation of the corresponding state
every(n,event)	Returns true at every n-th occurrence of event since activation of the corresponding state
temporalCount(event)	Increments by 1 (start=0) and returns the incremented Integer whenever event has occurred since activation of the corresponding state

3. 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*). FORM-L uses three-valued logic with two different data types:

- `property`, can have values `Violated`, `Undecided`, `Satisfied`.
- `condition`, can have values `false`, `undefined`, `true`.

Important reasons for using three-value logic are:

- In certain situations, it is not possible to state whether a property is `Violated/false` or `Satisfied/true`. For example FORM-L operator `during(condition, check)` is informally defined as: "*As long as the `condition` is true, `check` must be true*". However, what return value to use, when condition is not true? 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 operator is not defined and therefore the best meaningful value to return is `undefined`. With two-valued logic, the user has to somehow arbitrarily 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 a too optimistic or too pessimistic view.
- Simulations with property models should determine whether a required property is always `Satisfied`. A simulation may, however, not evaluate property models in critical regions. With three-valued logic this situation can be signaled to the user by having the value `Undecided` for a property. With two-valued logic it cannot be stated that a simulation did not test all required properties (or an additional flag has to be introduced for this purpose).

On the other hand, three-valued logic has the following severe drawbacks:

- Mixing of two-valued and three-valued logic becomes unintuitive for the user. For example, the two FORM-L operators `during` and `duringAny` are naturally defined with **Boolean input**

arguments and **three-valued logic output argument**. However, then a nested-call is not directly possible:

```
Requirement R8 = during(duringAny(duration=10, check=On),
                        check=Available)
"MPS MUST be declared Available again when On for more than 10 s."
```

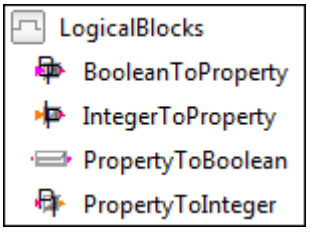
The reason is that `duringAny` returns a three-valued logic type, `during` expects a two-valued logic type and an automatic cast is logically not possible (because it is not clear how to map `undefined` to `false` or `true`).

- There are several three-valued logic definitions, such as Keene's, Lukuasiewicz's, Bochvar's and other logics, see for example (the difference is in the `implies(a,b)` operator):
  http://en.wikipedia.org/wiki/Three-valued_logic
  http://scientopia.org/blogs/goodmath/2010/08/24/introducing-three-valued-logic/
  http://scientopia.org/blogs/goodmath/2011/01/31/more-3-valued-logic-lukasiewicz-and-bochvar/

  For an end-user it is not obvious, which three-valued logic is used in a system.

- It might be not obvious for a library developer to implement an operator or function in three-valued logic. For example, the standard logic operator `implies` (= `during(..)` in FORM-L) has different definitions in different three-valued logical systems (see links above) and it is not obvious which is the "right" one (and the one "natural" for the end-user). Furthermore, if a function has two logical input arguments and one logical output argument, then there are nine different cases if three-valued logic is used, and four different cases for two-valued logic. The probability to make mistakes or define something “unintuitive” is then much higher in three-valued logic.
- Modelica has already many operators and functions for two-valued logic and also the user will have many models utilizing two-valued logic. If only three-valued logic would be used for properties modeling, then a large amount of existing code cannot be used.

It is clear that two-valued logic must be supported in order to utilize existing code and to support the well-known view of the end-user on logical expressions, as well as language elements such as `if/else` or `while`. On the other hand, two-valued logic alone has severe disadvantages for property modeling as sketched above. For this reason, in the Modelica_Requirements document two-valued logic, as well as a restricted form of three-valued logic is used. Basically, an enumeration `Property` is introduced with the three values `Violated`, `Undecided`, `Satisfied`. Additionally, a few operators are introduced that have three-valued logic input and/or output arguments. Furthermore, conversion blocks between `Property`, `Boolean` , `Integer` are provided:



4. Extension Proposals to the Modelica Language

In this section the current proposals are summarized that shall allow to express properties in a user-convenient way in Modelica. All proposals have been tested by prototypes (either in Dymola or in OpenModelica). In this section only a short sketch of the proposals is given. The formal "Modelica Change Proposals (MCP-0010, MCP-0011, MCP-0012, MCP-0021, MCP-0023)" contain all the details and are available from https://svn.modelica.org/projects/MCP/public.

4.1 Calling blocks as functions (MCPI-0012)

Temporal operators are a very central element of the FORM-L language. Example in FORM-L notation:

```
condition C1a = duringAny s4 check Off;
```

In this example `C1a` is true, if `Off` has been true for at least 4 seconds. A corresponding notation in Modelica could be:

```
Boolean C1a = duringAny(4, Off);
```

that is the Modelica function **`duringAny`** has the duration (= 4) and the Boolean variable to be checked (**`Off`**) as input and returns true or false. Unfortunately, an implementation requires a memory and a Modelica function cannot have a memory, because all calls of a (native) Modelica function will return the same result, if the same input arguments are used. Instead, such functionality has to be implemented with a Modelica block or model. For models and blocks, a function/operator-like call syntax is not supported and a block DuringAny would have to be utilized as:

```
DuringAny DuringAny1(duration=4, check=Off);
Boolean C1a = DuringAny1.result;
```

When many such definitions are present (as it is in the EDF properties use case), the definition of temporal properties is no longer practical in Modelica (becomes too clumsy and confusing).

In the Modelica-Change-Proposal MCP-0012, a proposal is made to improve the situation (*MCP-0012, 2014*):

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)[6]

[6] The reason for this restriction is that (a) the simple mapping mechanism would not give an intuitive result, and

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.

With this proposal, the FORM-L example at the beginning of this section can be rewritten as:

```
Boolean C1a = DuringAny(duration = 4, check = Off).y;
```

Signals in the interface of Modelica blocks are accessed by name and not by position as in function calls. Without changing the type system of Modelica, the name of the output signal must be given and this is done in the proposal by appending it after the call.

Dymola Prototype

Dassault Systèmes has made a prototype implementation of the above proposal in Dymola. The current restriction is that blocks can only be called as function in the declaration section of models and blocks and nested calls are not yet supported.

OpenModelica Prototype

The OpenModelica compiler has been extended with a function rewriteBlockCalls (see below) which traverses the abstract syntax internal representation of a model while expanding each function block call by adding an instance variable and equations referring to an instance of the block. A separate instance of the block is created for each block call.

```
function rewriteBlockCalls
  input  TypeName className "The name of the model containg functionBlock-calls";
  input  TypeName inDefs "The package containing the functionBlock definitions";
  output Boolean  success;
end rewriteBlockCalls;
```

Once this function is called on a model containing function block calls, the block calls are resolved to block instantiations.

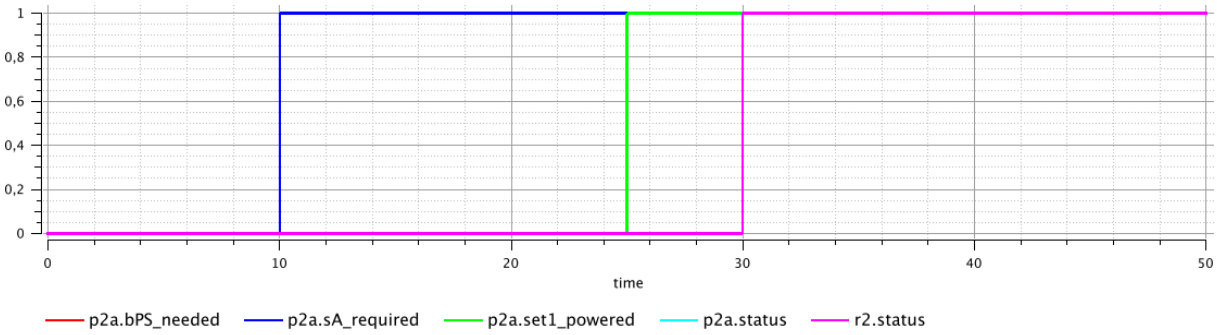
Example:

(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).

```
loadFile("exampleRequirements.mo");
loadFile("exampleVerification.mo");
loadFile("FORMLlib.mo");
rewriteBlockCalls(BPSRequirements.P2a, FORMLlib);
simulate(test.Test, stopTime = 50);
```

`Test` is a model that contains an instance of the requirement/property `P2a` together with a scenario for the verification of the requirement.

The status of the requirement during simulation is undefined until the event that triggers the verification occurs, then it is not_violated as set1 is powered at time 25, within the allocated time for the event:



Once the function block calls are resolved, the model can be saved thus generating a standard Modelica model, compatible with the current Modelica standard and all Modelica tools.

The transformed model with calls to function blocks expanded can be saved to a standard Modelica Model text file using saveModel:

```
saveModel("P2a_StandardModelica.mo", BPSRequirements.P2a);
```

The transformed Modelica model with block calls expanded appears as follows:

```
model P2a "condition P2a is after ((BPS.Needed and SA.Required) becomes true)
           within s20 assert Set1.Powered;"
  WithinAfter _autogen_WithinAfter0(withinT = 20);
equation
  _autogen_WithinAfter0.event = bPS_needed and sA_required;
public
  extends Requirement;
  input Boolean bPS_needed;
  input Boolean sA_required;
  input Boolean set1_powered;
equation
  status = if _autogen_WithinAfter0.out then if set1_powered then not_violated else
violated else undefined;
```

In the current version of the prototype, block calls are unconditionally transformed to block instances. This is performed both in the declaration and in the equation section. In the next version of the prototype restrictions will be introduced that are currently under discussion.

4.2 Reduction operators

4.2.1 Overview

In FORM-L a compact set-oriented formulation is used to extract information from sets. For example:

```
real TotalDemand = sum {S in Steps | S.DemandToBPS};
```

This statement defines variable `TotalDemand` as the sum of the `DemandToBPS` variables of each individual member of set `Steps`, a `Step` being a set of electrical components connected simultaneously to the BPS.

Other examples from FORM-L:

```
aOperator {X in Steps | F(X)}       // integer or a real value
```

`aOperator` is one of ***min*** | ***max*** | ***sum*** | ***prod*** and F is a function applicable to each member of the set to be reordered yielding an ***integer*** or ***real*** result that can be evaluated on all members.

The universal quantifier ∀ can be expressed as follows:

```
card {X in Steps | not(condition(X))} = 0
```

Modelica does not have sets (so an unordered collection of entities, where no entity is present twice). However, Modelica has vectors and arrays and if it is guaranteed that a vector does not have two elements in common, it can be practically treated as a set. Furthermore, all the examples above also make sense for vectors. There is also the issue that it is not obvious when two objects are treated the same and one of them should be removed, when both are stored in a set.

Modelica has “Reduction Functions and Operators”, see section 10.3.4 in (Modelica 2012), that are similar to the proposal above from FORM-L. Especially, the first statement can be written in Modelica as:

```
Real TotalDemand = sum(S.DemandToBPS for S in Steps);
```

The major difference is that the definition of the iteration variable (`S in Steps`) is made at the end and not at the beginning of the expression. Modelica supports the following special operators, where reduction expressions with for-loops can be directly used (for loops can also be nested):

- min(..)
- max(..)
- sum(..)
- product(..)
- array(..) and the short hand notation {…}

The last feature, “{..}”, allows to implement any desired function, by using the array reduction operator to generate an array which is passed as input argument to a (user-defined) function. Examples:

```
card( {not condition(X) for X in Steps} ) == 0
        // is true, if condition(X) holds for all X in Steps
```

Note, the {..} operator generates a vector, and the user-defined card(…) function returns the dimension of this vector.

```
Boolean atleastOnePumpActive = exists( {p.isActive for p in pumps} )
```

checks whether at least one pump in vector pumps is active.

```
Boolean noCavitation = forall( {p.p_a > p_cavitate for p in pumps} )
```

checks that all pumps in vector pumps do not cavitate.

The implementation of these functions is very simple:

```
function card
  input Boolean b[:] "Boolean vector";
  output Integer result "Number of true entries";
algorithm
  result :=sum(if e then 1 else 0 for e in b);
end card;

function exists
  input Boolean b[:] "Boolean vector";
  output Boolean result "= true, if at least one element of b is true";
algorithm
  result :=card(b) > 0;
end exists

function forall
  input Boolean b[:] "Boolean vector";
  output Boolean result "= true, if all elements of b are true";
algorithm
  result :=card(b) == size(b, 1);
end forall;
```

What is not so nice with user-defined functions is that a double number of parentheses is needed:

```
functionName( {... } )
```

and that first the vector is constructed, and afterwards function “functionName” operates on this vector. This may become inefficient for large vectors. In section 4.4.3 an MCP proposal is sketched, how these disadvantages can be fixed in Modelica. A prototype of this proposal is available in OpenModelica.

In FORM-L subsets can be constructed in a compact form:

```
mySubset = {X in MySet | condition(X)};
```

`mySubset` contains all elements `X` of `MySet` for which `condition(X)` is true. The resulting set can then be further processed by other set operations. With current Modelica, expressing subsets is clumsy (not practical), because for every subset condition a separate function would have to be implemented:

```
mySubset = subSetCondition( MySet );
```

Reduction expressions could be slightly extended to also provide subsets in a convenient way (a proposal from Hilding Elmqvist):

```
mySubset = {X for X in MySet if condition(X)};
```

The difference to the current possibilities can be more clearly seen with the following example:

```
Integer v[:]= {1,2,0,3,0};

{x<>0 for x in v}       // returns {true,true,false,true,false}
{x for x in v if x<>0}  // returns {1,2,3}
```

In section 4.4.4 an MCP proposal is sketched, how subset determination can be supported in Modelica. A prototype of this proposal is available in OpenModelica.

4.2.2 Clarification of reduction operators in Modelica 3.3 revision 1

Armin Troy from ITI considered expressing the following property:

> "In all Subsystems, there is at least one pump that is active."

by introducing a syntactic extension to Modelica:

```
b = forall (Subsystem s in S)
        (exists (Pump p in s.P)
            (p.IsActive)
```

Hilding Elmqvist from DS pointed out the existence of array comprehensions in Modelica and constructed the following example:

```
record Pump
  Boolean IsActive;
end Pump;
record Subsystem
  Pump P[:];
end Subsystem;

function Test
  output Boolean b1, b2;
 protected
  Subsystem S[:] = {Subsystem(P={Pump(IsActive=false)}),
                    Subsystem(P={Pump(IsActive=false), Pump(IsActive=true)})};
algorithm
  // Desired notation (unclear in Modelica 3.3; allowed in Modelica 3.3 revision 1)
  b1 := forall({exists({p.IsActive for p in s.P}) for s in S});

  // Most likely, Modelica 3.3 requires a clumsy rewriting
  b2 := forall({exists({S[i].P[j].IsActive for j in 1:size(S[i].P, 1)}) for i in 1:size(S, 1)});
  end Test;
```

It was not clear whether the first, compact notation is allowed in the Modelica specification version 3.3. In Dymola 2015, this gave a syntax error and only the second, clumsy rewriting could be used.

In July 2014, revision 1 of Modelica 3.3 was released, to clarify and fix issues of the specification text. In particular, the issue above was also clarified, so that the first, compact notation is now explicitly allowed in Modelica. This feature was evaluated with a prototype in Dymola, and is officially available since Dymola 2015 FD01.

4.2.3 Allowing user-defined functions in reductions (MCPI-0011)

MCPI-0011 proposes user-defined functions as reductions, in order to write code like:

```
function forall
  input  Boolean b1;
  input  Boolean b2 := true; // start condition
  output Boolean b := b1 and b2;
end forall;

function exists
  input  Boolean b1;
  input  Boolean b2 := false; // start condition
  output Boolean b := b1 or b2;
end exists;

forall(exists(p.isActive for p in s.P) for s in S))
```

There are the following advantages over the currently possible `forall( { exists( {..} ) } )`:

- Less clumsy notation because unnecessary {..} are no longer needed.
- An efficient implementation will not generate first a vector and will then operate on this vector, but will always only work with two elements of the vector, so no need to allocate storage for a vector.

4.2.4 Adding guards to reductions (MCPI-0010)

MCPI-0010 proposes a new syntax for filtering reduction expressions either by filtering either each iterator each combination of iterators. Run-time error-checking must be performed by the tool to ensure that array reductions construct arrays where each element has the same dimension size.

```
model FilterUsingGuards
  Real r0[:,:] = {i+j for i guard i>0 in -10:10, j guard j<0 in -10:10};
  Real r1 = sum(i+j for i in -10:10, j in -10:10 if i>0 and j<0);
end FilterUsingGuard;
```

5. Mapping FORM-L to Modelica

The goal is to utilize FORM-L in a Modelica model. In the MODRIO project two approaches have been analyzed: (a) Expressing properties solely in Modelica by enhancing the Modelica language and providing an appropriate Modelica library in which the operators are implemented, and (b) providing a preprocessor that translates FORM-L like expressions to Modelica. Approach (a) is sketched in the sequel. It shows how to express properties in Modelica in a way that is close to FORM-L. Hereby it is assumed that the mixed 2/3-valued logics is used in Modelica, as sketched in section 3, and that the proposed extensions to Modelica are available that are sketched in section 4.

In the separate working document (*Garro et al., 2016*), the details are given by using a 3-column table: The first column shows the FORM-L element, the second column the corresponding Modelica element, and in the third column additional explanations are given. This document is shortly summarized here:

FORM-L uses a special syntax to express properties. In Modelica either operator syntax is used (see

next table), or reduction expressions (see section 4.4). Examples for operator syntax:

FORM-L	*Modelica*
card set	**card**(set)
first set	**first**(set)
during condition **check** signal	**during**(condition, check=signal)
duringAny duration **check** signal	**DuringAny**(duration, check=signal)
after event	**After**(event)
after event **for** duration	**AfterFor**(event, duration)
when condition **becomes** true **check** signal	**WhenRising**(condition, check=signal)

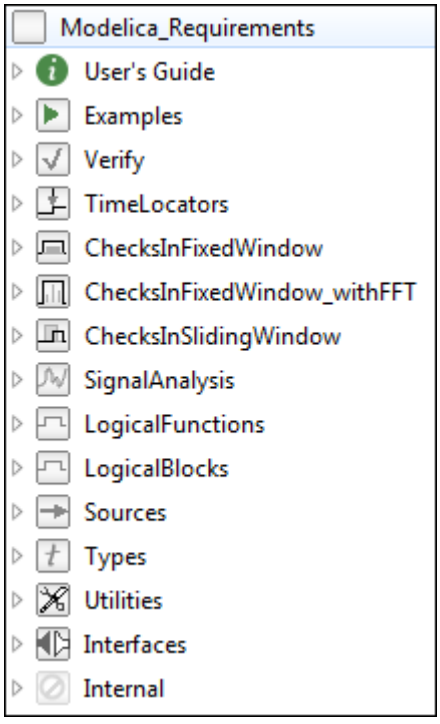
If operators do not have an internal memory, they are represented with Modelica functions and start with a lower case letter (such as **card**, **first**, **during**).
If operators have an internal memory, they are represented with Modelica blocks and start with an upper case letter (such as **DuringAny**, **AfterFor**). The above table shows the desired calling syntax of block operators in Modelica. With the Modelica extension sketched in section 4.1, the actually used syntax is a bit clumsier, for example,

```
DuringAny(duration=duration, check=signal).y
```

6. Modelica_Requirements Library

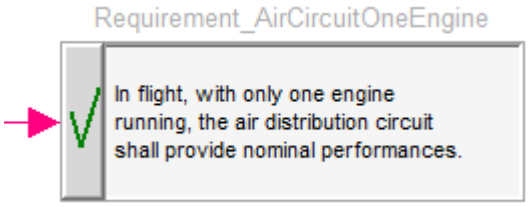
6.1 Overview

In the MODRIO project the Modelica library Modelica_Requirements has been implemented. There had been several versions and variants. For version 0.6 provided with this deliverable all examples have been removed that need the feature “calling blocks as functions”, in order that the library can be utilized by all Modelica tools. In this form, the library consists of about 150 model/blocks and 60 functions. The library is discussed in the paper (*Otter et al., 2015*). The FFT-based requirement blocks introduced into the library have been developed in the CleanSky project (*Kuhn et al., 2015*). In the following, a short overview of the Modelica_Requirements library is given. Hereby text passages from (*Otter et al., 2015*) are used.



The top level view of the library is shown in the figure at the right side.

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. An example for the usage of block Requirement is shown in the next figure:



Example on how to define a required property.

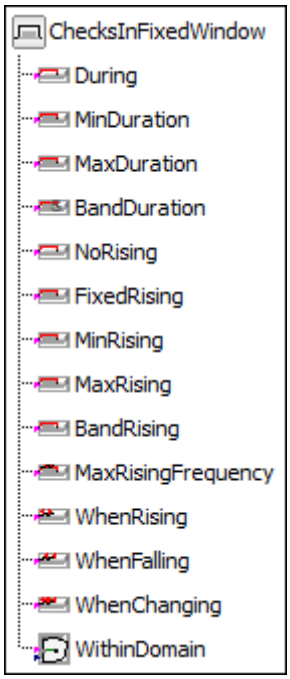
The left hand arrow is an input signal of type Property. In the icon, the content of parameter text is displayed that should contain a textual description of the required property. The Requirement block monitors its property input over a simulation and computes its status at the end of the simulation run:

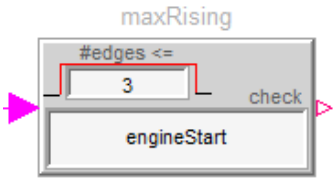
- *Requirement is violated*:
  Input is Violated at least once.
- *Requirement is untested*:
  Input is Undecided for the complete simulation run
- *Requirement is satisfied*:
  Input is Satisfied at least once, and is never Violated.

In sub-library ChecksInFixedWindow (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 below, 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).



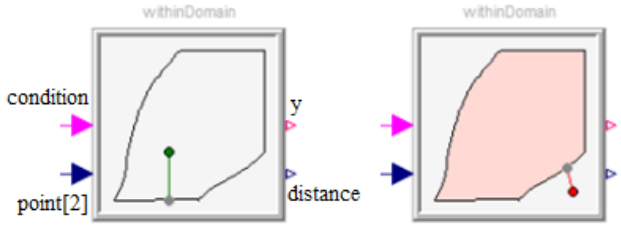


Example for MaxRising block.

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.

WithinDomain is a more complicated block, see left part of figure below. 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). The actual polygon is displayed in the icon, together with the point (= green circle) and the nearest distance of the point to the polygon.



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

After a simulation run, a diagram animation shows the actual status. In the right part of the figure 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.

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 in a future version of the Modelica_Requirements library.

In the next figure an example from (*Thuy, 2013*) is shown using block AfterUntil. This example concerns the generator of a Backup Power Supply system:

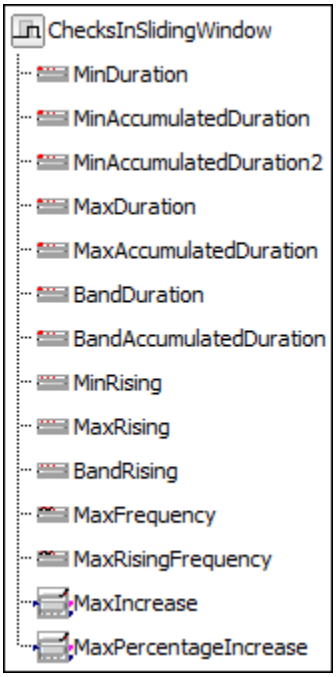
Example for block AfterUntil.

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

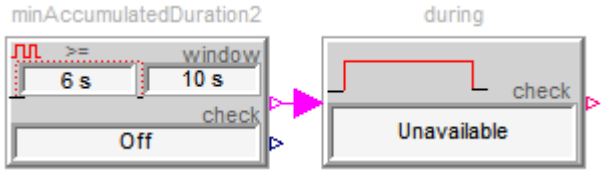
In sub-library ChecksInSlidingWindow (see figure below) blocks are present that determine whether a particular property is fulfilled or not in a *sliding time window*. For example, if a sliding time window has size *T* and *t* is the actual time instant, then in every time range $[t - T, t]$ the property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



Some of the functions operating on this buffer are sketched at hand of block MinAccumulated-Duration2, see next figure:



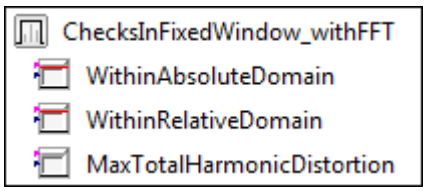
Example for MinAccumulatedDuration2

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.*

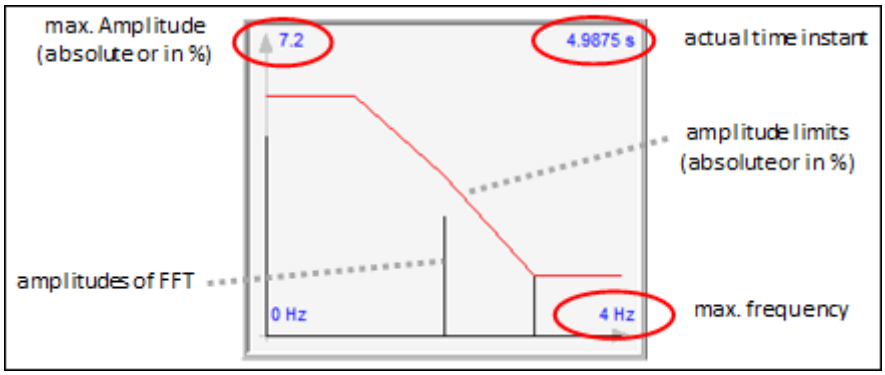
This is achieved in the following way: Component minAccumulatedDuration2 outputs true, if in any time window of length 10 s variable Off was accumulated true for at least 6 s. This signal is the input to component during which requires that whenever the input is true, variable Unavailable must be true 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.



In sub-library ChecksInFixedWindow_withFFT (see figure to the right blocks are present that check properties based on FFT (Fast Fourier Transform) computations in fixed time windows. An FFT determines the frequency content and amplitudes of a sampled, periodic signal, and the blocks in this package check whether these frequencies and amplitudes fulfill certain conditions. The approach is to trigger the sampling of the periodic signal by a rising edge of Boolean input variable condition, storing the values in an internal buffer and computing the FFT once "sufficient" values are available.

In the icon of the blocks the last computed FFT is shown, see example in the next figure:



6.2 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 how to perform this with Modelica, especially (*Schamai 2013*; *Schamai et al. 2014*, *Schamai et al. 2015*) 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. Additionally, two Modelica Change Proposals have been proposed for this purposes and tested with a Dymola prototype (*MCP-0021 2015*, *MCP-0023 2016*). However, an agreement about the “best” approach in Modelica was not yet reached.

6.3 Issues and Future Work

The current development stage allows checking 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.

The Modelica_Requirements library does not yet cover all parts of FORM-L. Currently, a redesign of the library is performed to provide a better implementation of some blocks and add blocks for “overlapping time windows” and other functionality from FORM-L.

7. Literature

Alan J.F. (1983): **Maintaining Knowledge about Temporal Intervals**. Communications of the ACM, Volume 26, Issue 11, Nov. 1983. Download: http://dl.acm.org/citation.cfm?id=358434

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

Bouskela D., Jardin A., Thuy N. (2014): **D2.1.1 – Modelica Extensions for Properties Modeling, Part II: Modeling Architecture for the Design Verification against System Requirements**. MODRIO deliverable, Sept. 2014.

Elmqvist H., Olsson H., Otter M. (2015): Constructs for Meta Properties Modeling in Modelica. Proceedings of the 11th International Modelica Conference, Versailles (France), September 21-23. http://www.ep.liu.se/ecp/118/026/ecp15118245.pdf

Garro A., Tundis A., Otter M. (2016): **D2.1.1 - Modelica Extensions for Properties Modeling, Part IVb: FORM-L and Modelica: syntax and relationships**. MODRIO deliverable, 2016.

Grote C. (2010): **An Aeronautical Publish/Subscribe System Employing Imperfect Spatiotemporal Filters**. Dissertation. University of Darmstadt. Download: http://d-nb.info/1007634510/34

Elmqvist H., Olsson H., Otter M. (2015): **Constructs for Meta Properties Modeling in Modelica**. Proceddings of the 11th International Modelica Conference, Versailles (France), September 21-23. http://www.ep.liu.se/ecp/118/067/ecp15118625.pdf

Kuhn M., Otter M., Giese T. (2015): **Model Based Specifications in Aircraft Systems Design**. Proceedings of the 11th International Modelica Conference, Versailles (France), September 21-23. http://www.ep.liu.se/ecp/118/053/ecp15118491.pdf

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

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

MCP-0010 (2014): **Adding guards to reductions**. Modelica Change Proposal 0010 https://svn.modelica.org/projects/MCP/public/MCP-0010_ReductionFilter

MCP-0011 (2014): **Allowing user-defined functions in reductions**. Modelica Change Proposal 0011, https://svn.modelica.org/projects/MCP/public/MCP-0011_CustomReductions

MCP-0012 (2014): **Calling Blocks as Functions**. Modelica Change Proposal 0012. https://svn.modelica.org/projects/MCP/public/MCP-0012_CallingBlocksAsFunctions

MCP-0020 (2015): **Models as Arguments to Functions**. Modelica Change Proposal MCP-0020 (superseded by MCP-0023). https://svn.modelica.org/projects/MCP/MAinternal/MCP-0020_ModelsAsArgumentsToFunctions

MCP-0021 (2015): **Component Iterators**. Modelica Change Proposal 0021. https://svn.modelica.org/projects/MCP/public/MCP-0021_ComponentIterators

MCP-0023 (2016): **Model to Record**. Modelica Change Proposal 0023. https://svn.modelica.org/projects/MCP/public/MCP-0023_ModelToRecord

Modelica (2012): **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 (2013): **Requirements Interchange Format (ReqIF)**. Download: http://www.omg.org/spec/ReqIF/1.1/PDF/ http://www.omg.org/spec/ReqIF/20110401/reqif.xsd

Otter M., Thuy N., Bouskela D., Buffoni L., Elmqvist H., Fritzson P., Garro A., Jardin A., Olsson H., Payelleville M., Schamai W., Thomas E., Tundis A. (2015): **Formal Requirements Modeling for Simulation-Based Verification**. Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23. http://www.ep.liu.se/ecp/118/067/ecp15118625.pdf

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

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

Schamai W., Buffoni L., Albarello N., Miranda P.F., Fritzson P. (2015): **An Aeronautic Case Study for Requirement Formalization and Automated Model Composition in Modelica**. Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23. http://www.ep.liu.se/ecp/118/099/ecp15118911.pdf

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

Thuy N. (2013): **D8.1.3 – Part 1 The Backup Power Supply**. MODRIO deliverable, Nov. 2013.

Thuy N. (2014): **D2.1.1 – Modelica Extensions for Properties Modeling, Part III: FOrmal Requirements Modelling Language (FORM-L)**. MODRIO deliverable, Sept. 2014.