EXPRESSING REQUIREMENTS IN MODELICA

Lena Buffoni* **and Peter Fritzson**
Linköping University
SE-581 83 Linköping
Sweden

ABSTRACT

As cyber-physical systems grow increasingly complex, the need for methodologies and tool support for an automated requirement verification process becomes evident. Expressing requirements in a computable form becomes a crucial step in defining such a process. The equation based declarative nature of the Modelica language makes it an ideal candidate for modeling a large subset of system requirements. Moreover, modeling both the requirements and the system itself in the same language presents numerous advantages. However, a certain semantic gap subsists between the notions used in requirement modeling and the concepts of cyber-physical modeling that Modelica relies on. To bridge this gap, in this paper, we illustrate through the use of dedicated types, pseudo function calls and function block libraries, how the Modelica language can be tailored to fit the needs of requirement modeling engineers.
Keywords: Requirements, Equation-based modeling

INTRODUCTION

Functional safety is a key concern in all industry sectors, be it nuclear plants, medical appliance manufactures or the automotive industry. The functional correctness of a component is the guarantee that the component behaves the way it should and fulfils all the functional requirements of the system. As the complexity of cyber-physical systems increases, maintaining coherent requirement specifications and using them to verify models of physical systems requires the formalisation of the requirements in a computable manner [2, 4]. In this paper, we propose an approach to formalising the requirements in the same language as the model of the physical system. For this purpose we choose Modelica, an object-oriented equation-based language for modeling multi-domain physical systems [5, 1].

Expressing requirements in the same language as the physical model has numerous advantages. It improves the maintainability of the overall model, ensures that the requirements stay coherent as the model changes and simplifies the verification process, as the requirements can be simulated together with the system model. However, engineers expressing requirements use domain specific terms and concepts [6]. Although requirement-specific notions can be expressed directly in Modelica, writing them from scrach every time manually can be complicated, and the resulting requirements can be harder to understand at first glance.

To bridge the gap between the requirement designer vision and the Modelica world, we define a set of types and pseudo functions, presented in the following section. A pseudo function is not a real function, since it allows side-effects and the use of time-dependent operators and equations in its body, which are disallowed in normal declarative Modelica functions. We extend Modelica with a mechanism for calling these pseudo functions, to simplify the readability of requirements. We illustrate these concepts on a simple example of a backup power system.

The paper is organized as follows. Section 2 introduces the notions used to map the requirements, Section 3 illustrates how the requirement verification is done, Section 4 discusses related works and finally

*Corresponding author: E-mail: lena.buffoni@liu.se

Section 5 summarizes the article and discusses future works.

MODELING REQUIREMENTS

In order to make the expression of requirements in Modelica as intuitive as possible in this section we introduce an approach of mapping concepts from the requirement modeling domain, such as those defined in [6] to the Modelica language.

Requirement Type

To treat requirements in a systematic manner, we need to define a dedicated requirement type. A requirement model should not influence the execution of the physical model, but only access the information from the physical model necessary for the requirement verification. Requirements are defined as special types of blocks: they have several inputs and a single output that represents the status of the requirement. A status can take the following values[11, 8]:

violated when the conditions of the requirement are not fulfilled by the design model;

not violated when the conditions of the requirement are fulfilled by the design model;

undefined when the requirement does not apply, for instance a requirement that describes the behaviour of a power system when it is switched on, cannot be verified when the system is off.

If we take the example of a simple backup power system, which consists of several blocks connected in parallel and operates when the main power supply is lost, we can model a simple requirement *"When the power is on, the backup power-supply must not be activated"*, as follows in standard Modelica:

```
block R1
   extends Requirement;
   input Boolean powerOn;
   input Boolean bPSOn;
 equation
   status = if powerOn then
               if bPSOn then
                  violated
               else not_violated
            else undefined;
 end R1;
```

In the case of such a simple requirement, no additional construct are necessary.

"Pseudo Function" Library

To bridge the semantic gap between the concepts used in requirement modeling and Modelica, we propose to define a set of Modelica function blocks to represent basic requirement modeling constructs. As mentioned, function blocks are a modified version of standard Modelica blocks, with a single output that can be called using a function syntax.

In particular, the time locator properties as defined in [6], such as `after`, `WithinAfter`, `until`, `everyFor` can be defined as Modelica function blocks. These constructs are used to which define a period in time when a requirement should be verified.

For instance `everyFor(duration1, duration2)`, is a time locator that is used to define a requirement that must hold every duration1 seconds, for duration2 seconds.

Such constructs cannot be modeled as simple functions, as they are not context free and rely on time. Therefore to represent this `everyFor`, we can define the following Modelica function block:

```
function block everyAfter
  parameter Real everyT;
  parameter Real forT;
  output Boolean out;
protected
  Real tmp(start = 0);
equation
  when sample(0, everyT) then
    tmp = time;
  end when;
  if time > tmp + forT then
    out = false;
  else
    out = true;
  end if;
end everyAfter;
```

Requirements can then be expressed in terms of these basic building blocks in a more readable fashion. A set of predifined time locators based on the FORM-L specification is available, but the user can also define his own components.

Anonymous Function Blocks Through Function Calls

If we take another simple requirement for a backup power unit, *"Within 40 seconds of the power being lost, at least two sets must be powered"* and attempt to express it in Modelica, we will need to use the function block `withinAfter`, which is defined as follows:

```
function block withinAfter
  parameter Real withinT;
  input Boolean event(start = false)
     ;
  output Boolean out;
  protected
  Real time_event(start = -1);
equation
  when event then
    time_event = time;
  end when;
  if time_event > (-1) and
     time_event + withinT < time
     then
    out = true;
  else
    out = false;
  end if;
end withinAfter;
```

If we use standard Modelica blocks, then we need to explicitly create an instance of an withinAfter block and connect it to the corresponding inputs and outputs, which reduces the readability of the model. Therefore we propose to define a syntax for pseudo functions, where a function block can be called like a function by it's name and with parameters and input variables as arguments. We have implemented an extension in OpenModelica[7], that will automatically generate an instance of the required function block and the corresponding connection equations.

With this syntax, we can define the above requirement in Modelica as follows:

```
block R2
   extends Requirement;
   input Boolean[5] isOn;
   input Boolean powerLoss;
   output Integer status(start = 0)
      ;
   Boolean wA;
  equation
   wA = withinAfter(40, powerLoss);
   when wA then
     status = if countTrue(isOn) >=
         2 then
      not_violated else violated;
   elsewhen not wA then
     status = undefined;
   end when;
end R2;
```

In this example, the function block withinAfter, is called as a function, and the arguments of the call represent the values that the function block should be instantiated with. The parameter withinT should take the value 40, and the signal powerLoss should be connected with the input event.

To generate this transformation we call the function `rewriteFunctionBlockCalls(modelToRewrite, libraryPackage)` in the OpenModelica API. This function will take two arguments, the model that needs to be rewritten and a package containing the function block definitions. It will then parse all the function calls, and replace all the calls to functions with the same names as the function blocks in the package passed in parameters with instantiations of the corresponding function blocks in the declaration section, and the result of pseudo function call will be the single output of the function block. The updated model is then reloaded into memory and can be simulated.

The argument passing works in the same way as for normal function calls, the positional instantiation will bind the values passed to the function call to the parameters and input variables of the function block in the order in which they are defined. Arguments can also be named explicitly, in which case the corresponding input value or parameter will be instantiated with the expression passed to the function.

Saving a model after `rewriteBlockCalls` was called on it will generate standard Modelica code, for instance for the example above:

```
block R2
    extends Requirement;
    input Boolean[5] isOn;
    input Boolean powerLoss;
    output Integer status(start = 0)
       ;
    Boolean wA;
    withinAfter _agen_withinAfter1(
       withinT = 40);
  equation
    _agen_withinAfter1.event =
       powerLoss;
    wA = withinAfter(40, powerLoss);
```

```
    when wA then
      status = if countTrue(isOn) >=
          2 then 1 else -1;
    elsewhen not wA then
      status = 0;
    end when;
  end R2;
```

The extra step of generating standard Modelica code is important, as it allows to export the resulting models in standard Modelica, compatible with any Modelica tools, therefore function blocks are mapped to standard Modelica blocks.

It is improtant to destinguish between a requirement and a function block. The requirement maps to a system requirement, such as the one defined by R2 (*"Within 40 seconds of the power being lost, at least two sets must be powered"*) and can contain one or more function blocks to represent time locators. As illustrated in the previous section, a requirement can also be time independent of time and should then hold continuously.

REQUIREMENT VERIFICATION

Once the the requirement model and the system model are combined, they can be simulated together in order to verify the requirements. Each requirement has a status value which can subsequently be plotted to see at which times the requirement is violated.

The advantage of having the requirements in the same language as the system model is that no additional work is necessary to simulate the system.

In the verification scenario in our example, the power is lost at time 20, and the back-up units 1 and 2 are turned on at time 40 (Figure 1). Therefore the requirement is not violated. The units 1 and 2 are turned off again at time 80, however since this behaviour does not affect the requirement, it remains not violated (Figure 2).

If we modify the verification scenario so that unit 2 is turned on at time 70, the requirement will be violated as illustrated in Figure 3.

RELATED WORK

In this paper we have shown how textual requirements can be formalised in Modelica, however when dealing with large numbers of requirements and simulation scenarios, there is a need for an automated

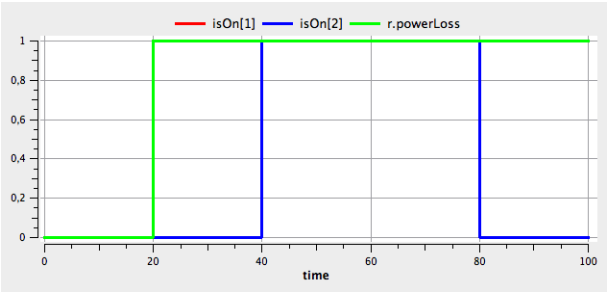

Figure 1: The power loss of the main power system and the switching on/off of backup units 1 and 2

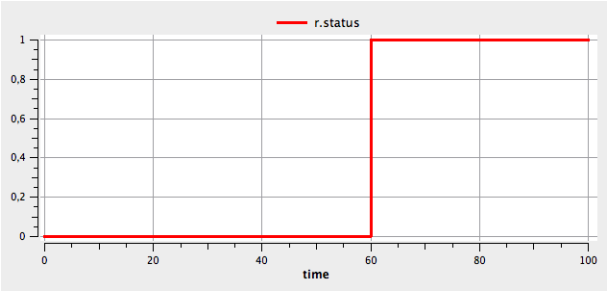

Figure 2: The requirement status, where 0 represents undefined, -1 violated and 1 not_violated

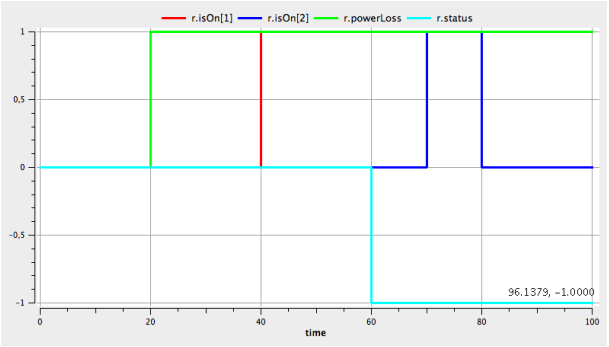

Figure 3: The requirement is violated because the power block 2 is not turned on time.

approach for composing the requirements with a given system design for the purpose of verification. In [10, 9] an approach for automating this process through the use of binding is proposed with an implementation in ModelicaML, a Modelica profile for UML. In [8] the requirement verification methodology is adapted to Modelica syntax. This work complements the work on formalising requirements in Modelica presented in this paper.
FORM-L language (FOrmal Requirements Modelling Language) is a language specification developed by EDF dedicated to expressing requirements and properties in a clear and concise manner [6]. In the work presented in this paper, concepts from FORM-L were mapped to Modelica function blocks in order to use them when modeling requirements in Modelica.

CONCLUSION

In this paper we have illustrated how through a minimal set of extensions, we can use Modelica to formalise requirements and then verify them with respect to a specific system design.
Expressing requirements in the same language as the physical model brings the advantages of a modular, object-oriented language for system design to the process of requirement formalisation, and allows for a runtime verification of requirements. This work is part of a larger ongoing research project aiming to develop tool and methods [3] for model-driven, integrated system verification and fault analysis. Moreover, expressing the requirements in Modelica allows to formalise them and remove the ambiguity present in a verbal description.
The next step in this work is the integration with the work in [8] for an automatic generation of verification scenarios as well as tool support for batch processing of requirements.

ACKNOWLEDGEMENT

This work is partially supported by the ITEA 2 MODRIO project.

REFERENCES

[1] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley IEEE Press, 2004.

[2] E. Hull, K. Jackson, and J. Dick. *Requirements Engineering*. Springer, 2005.

[3] ITEA 2 Projects. MODRIO. http://www.itea2.org/.

[4] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009.

[5] Modelica Association. Modelica 3.2 revision 2 specification, 2013. www.modelica.org.

[6] Thuy Nguyen. FORM-L: A MODELICA Extension for Properties Modelling Illustrated on a Practical Example. In *Proceedings of The 10th International Modelica Conference*, 2014.

[7] Open Source Modelica Consortium. Openmodelica project, 2013. www.openmodelica.org.

[8] Wladimir Schamai. *Model-Based Verification of Dynamic System Behavior against Requirements*. PhD thesis, Method, Language, and Tool Linköping: Linköping University Electronic, PressDissertations, 1547, 2013.

[9] Wladimir Schamai, Lena Buffoni, and Peter Fritzson. An Approach to Automated Model Composition Illustrated in the Context of Design Verification. *Modeling, Identification and Control*, 35(2):79–91, 2014.

[10] Wladimir Schamai, Peter Fritzson, and Chris Jj Paredis. Translation of UML State Machines to Modelica: Handling Semantic Issues. *Simulation*, 89(4):498–512, April 2013.

[11] Andrea Tundis, Lena Rogovchenko-Buffoni, Peter Fritzson, Alfredo Garro, and Mattias Nyberg. Requirement verification and dependency tracing during simulation in modelica. In *Proceedings of EUROSIM Congress on Modelling and Simulation*, September 2013.