D2.1.5 – A Prototype for Requirement Modeling in OpenModelica

MODRIO (11004)

Version 1.0

Date 25/04/2016

Authors

Lena BUFFONI	LiU
Peter FRITZSON	LiU

Summary

This document presents the prototype implementation for binding of requirements (also called properties) to design alternatives in OpenModelica, completing the description in Paper [1].

The three documents [1], [2], [3] are appended at the end of this document.

1. Abstract

This document presents the prototype implementation for binding of requirements (also called properties) to design alternatives in OpenModelica, completing the description in Paper [1].

2. Introduction

Building complex systems from models that have been developed separately without modifying existing code is a challenging task faced on a regular basis in multiple contexts including design verification. To address this issue an approach has been developed for automating dynamic system model composition by defining the minimum set of information that is necessary to the composition process.

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. In this report we discuss how a requirement model represented in Modelica can be connected to the system model in a semi-automated manner, as we might want to test different model configurations in different scenarios against different sets of requirements. To automatically create all the possible combinations is time consuming and error prone.

This approach can be used in combination with the requirement library developed in the context of MODRIO and has been used in a real study case [3], but it can also be used with any other way of modeling requirements in Modelica.

The binding approach does not assume prior knowledge of each other by the respective models and therefore increases decoupling and allows reuse of existing models and libraries. As mediators can be defined in several steps this means that different people can provide the information necessary to connect the models at different stages in the design process. Furthermore, it enables a formal traceability between client and provider models. For example, determining which requirements are implemented in the system design model at hand can be achieved by looking at the bindings for mandatory requirement clients.

3. Binding Model

In this section we give the definitions of the base concepts used in our approach. To this end we introduce the notions of clients and providers. Clients require certain data; providers can provide the required data. However, clients and providers do not know each other a priori.

Moreover, there may be multiple clients that require the same information. On the other hand, data from several providers may be needed in order to compute data required by one client. This results in a many-to-many relation between clients and providers.

In order to associate the clients and the providers to each other we introduce the mediator concept, which is an entity that can relate a number of clients to a number of providers, as illustrated in Figure 1. References to clients and providers are stored in mediators in order to avoid the need for modifying client or provider models.

The purpose of the presented concepts is to capture all the information needed for inferring (i.e., generating) binding expressions for clients. We define a binding as follows:

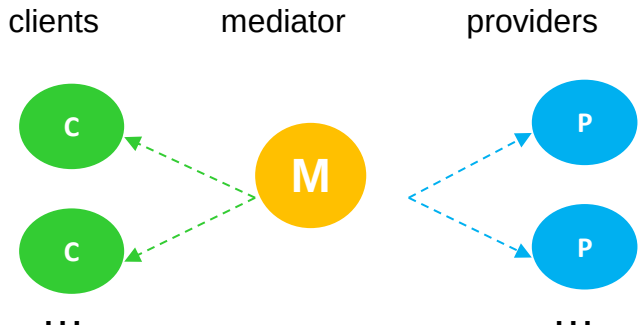
client instance reference = binding expression

3.1. Binding structure

In this section we synthetize the minimum information that is necessary for the use with the binding algorithm proposed by EADS/LIU, more details on this can be found in [2] and [3]:

- The Mediator name reflects what is needed by clients
- The Mediator type must be compatible to its clients
- The Client or provider id is the qualified name of the client or provider model (e.g. Package1.Model1.component1)
- The Client `isMandatory (default=true)` indicates whether the client must be bound.
- (If needed) Mediator template may only contain function calls with a special : operator that will be then expanded to an array of all the instances of the client in the model (e.g. sum(:), toArray(:), card(:), min(:), max(:), etc.)
- (If needed) Client or provider template may contain expressions including references to components within the client or provider models (e.g. in Modelica using the dot-notation)

The binding is n to m and the clients do not know each other a priori:



A structure to represent the client side:

```
record Client
  Boolean isMandatory = false "Defines whether the client needs to be bound
                for the requirement to be checked. False by default";
  String className "A qualified name for the client";
  String instance "The instance within the class";
  String template = "" "A transformation that can be applied to the client.
                      If left empty, no transformation will be applied.";
end Client;
```

A structure to represent the provider side:

```
record Provider
  String className "A qualified name for the provider";
  String instance "The instance within the class";
  String template = "" "A trasformation that can be applied to the
         provider. If left empty, no transformation will be applied.";
end Provider;
```

A mediator to group n clients and m providers:

```
record Mediator
  String name = "" "Reflects what is needed by the clients. Optional.";
  String type = "" "Optional if left empty will not be checked";
  String template = "" "A trasformation that can be applied to the set of
       providers (add(:), max(:),...). If left empty, no transformation
       will be applied.";
  Client[:] clients "List of clients";
  Providers[:] providers;
end Mediator;
```

The idea is to have a data structure that can deal with the most complex cases, but then can be hidden behind function calls for simpler cases. For example for simple one to one variable bindings:

```
function oneToOneBinding "A simple constructor for one to one bindings"
  input String providerName;
  input String clientName;
  output Mediator mediator = Mediator(
            clients = {Client(className = clientName)},
            providers = {Provider(className = providerName)});
end oneToOneBinding;
```

Then in order to generate the bound model, we need an API call that will take a set of requirements, a physical model and a set of mediator definitions and will generate the necessary modifiers.

```
generateBindings(targetName = "VerificationModel1",
                           systemClass = "System",
                           systemInstance = "sys";
                           requirements = {Requirements( class = "Req",

        instance = "req1",

        mediator = Mediator( ....))
                                                          })
```

3.2. An example

A system contains several pumps

Informal requirement:

- *At least 2 pumps shall be in operation at any time.*

We define a requirement violation monitor, that will check the requirement:

```
 model Req
   input Integer numberOfOpPumps = 0 "number of operating pumps";
   constant Integer minNumberOfOpPumps = 2 "min. number of operating pumps"
;
```

```
   output Integer status(start=0,fixed=true) "indication of requirement vio
lation,
      0 = not evaluated, 1 = not violated, 2 = violated";
 algorithm
   if numberOfOpPumps < minNumberOfOpPumps then
      status := 2 "2 means violated";
   else
      status := 1 "1 means NOT violated";
   end if;
end Req;
```

And a possible design for the system:

```
model System
    PA pump1;
    PB pump2;
    PB pump3;
  end System;

  model PA
    input Boolean on = false "to turn the pump on or off";
  end PA;

  model PB
    input Boolean switchedOn = false "to switch the pump on or off";
    Real volFlowRate "volume flow rate of the pump";
  equation
    if switchedOn then
       volFlowRate = 1;
    else
      volFlowRate = 0;
    end if;
  end PB;
```

The system contains two types of pumps A and B for which the way of deducing whether a pump is in operation is different. In order to connect the requirement to the system we define a mediator that inherits from the standard mediator structure:

```
record numberOfOperatingPumps
  extends Mediator(
    name = "NumberOfOperatingPumps",
    mType = "Integer",
    template = "sum(:)",
    clients = {Client(className = "Pack1.MinPumpNumber",
      instance = "numberOfOpPumps")},
      providers = {
        Provider(className = "Pack2.PA",
                 template = "if %name.on then 1 else 0"),
        Provider(className = "Pack2.PB",
                 providerTemplate = "if %name.volFlowRate > 0 then 1 else
0")});
end numberOfOperatingPumps;
```

In the `numberOfOperatingPumps` mediator we define a mediator for a client that requires an Integer value. This client has the id Pack1.Req.numberOfOpPumps.

For pumps of type A this information can be obtained by looking at `Pack2.PA.on`. To transform this into the number of pumps operating, we apply the template "`if providerPath then 1 else 0`".

On the other hand, for pumps of type B this information can be obtained by looking at `Pack2.PB.volFlowRate`. We then need to apply a different template to transform this into the number of pumps operating.

Based on this information we can then generate the following modifiers:

```
model VerificationModel1
  Req req1
    (
      numberOfOpPumps = sum({
      (if sys.pump1.on then 1 else 0),
      (if sys.pump2.volFlowRate > 0 then 1 else 0),
      (if sys.pump3.volFlowRate > 0 then 1 else 0)})
    );
  System sys;
end  VerificationModel1;
```

An additional requirement:

If we want to also define the maximum number of pumps that should be operating simultaneously, then we define another requirement

```
model MaxPumpNumber "The maximum number of operating pumps at any given
   time should be 5"
input Integer numberOfOpPumps;
  constant Integer maxNumberOfOpPumps = 5;
  output Integer status(start = 0, fixed = true) "This property indicates
    the violation monitoring status. The values mean the following: 0=not
    evaluated, 1=evaluated and NOT violated, 2=evaluated and violated. ";
algorithm
  if numberOfOpPumps > maxNumberOfOpPumps then
    status := 2 "2 means violated";
  else
    status := 1 "1 means NOT violated";
  end if;
end MaxPumpNumber;
```

Then we add an additional client to the mediator:

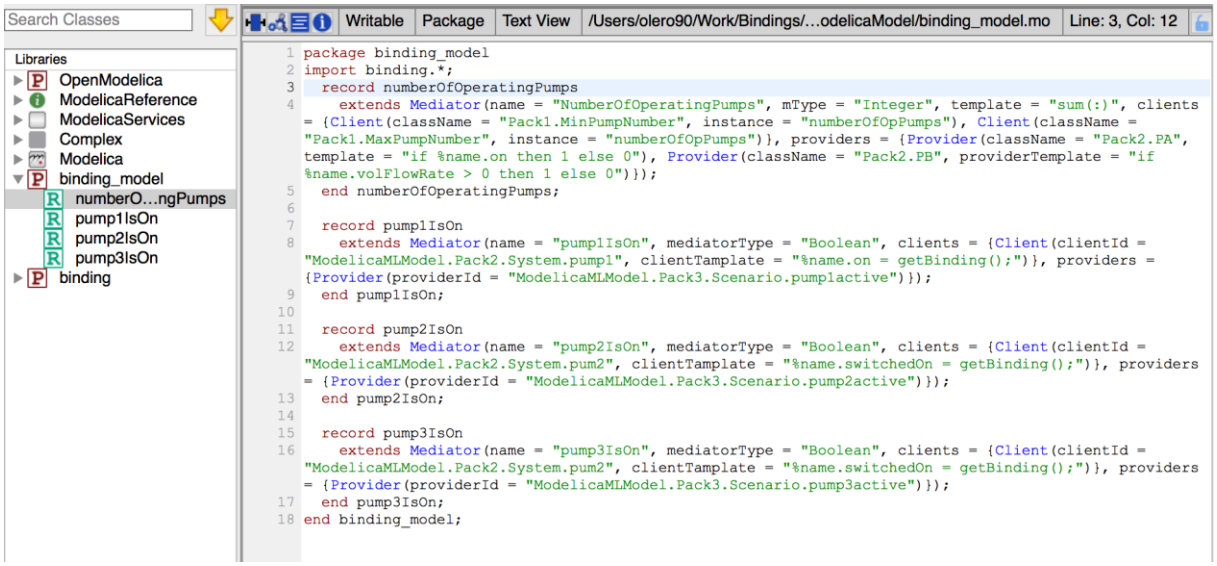
```
record numberOfOperatingPumps
  extends Mediator(name = "NumberOfOperatingPumps",
    mType = "Integer",
    template = "sum(:)",
    clients = {Client(className = "Pack1.MinPumpNumber",
                      instance = "numberOfOpPumps"),
```

```
                Client(className = "Pack1.MaxPumpNumber",
                       instance = "numberOfOpPumps") },
    providers = {Provider(className = "Pack2.PA",
          template = "if %name.on then 1 else 0"),
          Provider(className = "Pack2.PB",
            providerTemplate = "if %name.volFlowRate > 0 then 1 else 0")});
end numberOfOperatingPumps;
```

4. Implementation in OpenModelica

In OpenModelica we have implemented the algorithm that generates the « glue code » that connects the model, the requirements and the scenarios together.

In order for the code to be generated, we need to load the file that contains the binding information :

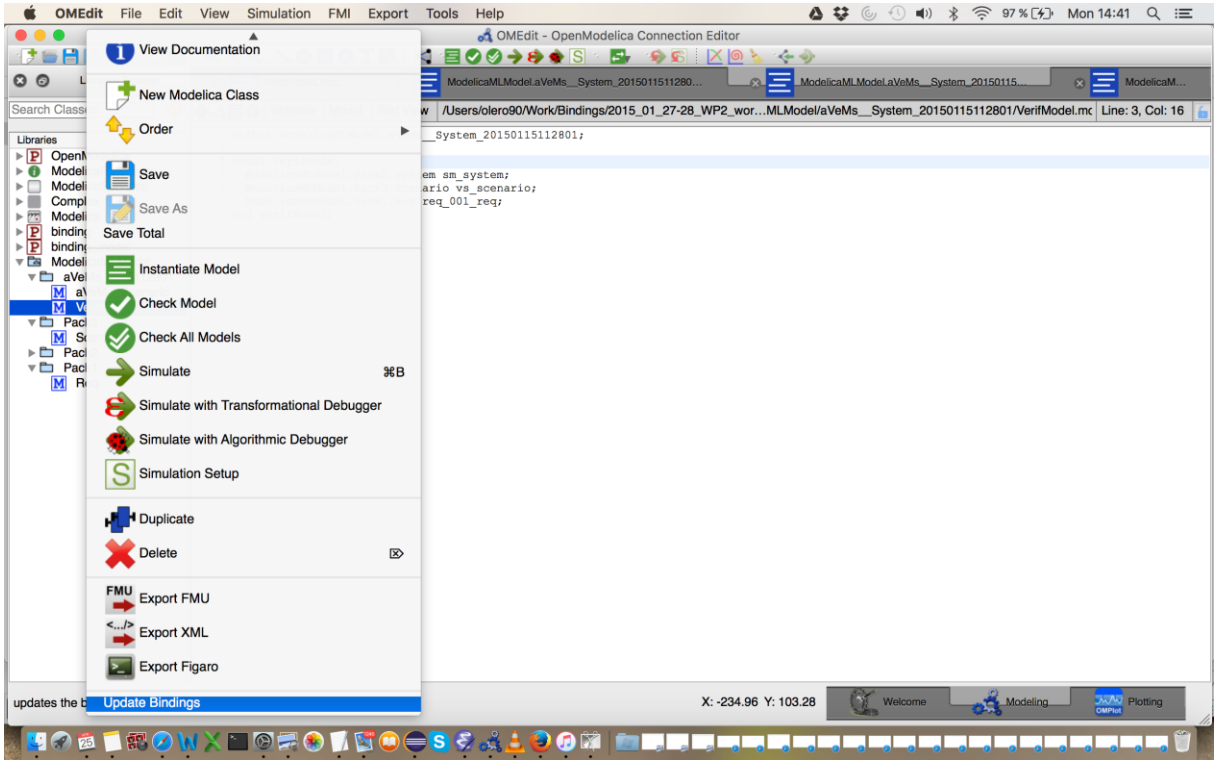


We define the verification Model as a standard Modelica Model, with the requirements we want to test, the design alternative and the scenario :

```
within ModelicaMLModel.aVeMs__System_20150115112801;

model VerifModel
  ModelicaMLModel.Pack2.System sm_system;
  ModelicaMLModel.Pack3.Scenario vs_scenario;
  ModelicaMLModel.Pack1.Req req_001_req;
end VerifModel;
```

In OMEdit, in the right click menu we choose « update bindings ». This will use the algoritm described in detail in [2] to compute the necessary modifier equations to connect the design model, the requirement model and the scenario :



The model will be updated :

```
within ModelicaMLModel.aVeMs__System_20150115112801;

model VerifModel
  ModelicaMLModel.Pack2.System sm_system(pump1.on = vs_scenario.pump1active, pump2.switchedOn =
vs_scenario.pump2active, pump3.switchedOn = vs_scenario.pump3active);
  ModelicaMLModel.Pack3.Scenario vs_scenario;
  ModelicaMLModel.Pack1.Req req_001_req(numberOfOpPumps = sum(i for i in {if sm_system.pump1.on
then 1 else 0, if sm_system.pump2.volFlowRate > 0 then 1 else 0, if sm_system.pump3.volFlowRate >
0 then 1 else 0}));
end VerifModel;
```

This model can then be simulated as normal (see below).

The status of the requirement is 1 throughout the simulation which means that it is fulfilled all the time.

5. References (appended to this document)

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

[2] Schamai, Buffoni, and 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

[3] Wladimir Schamai, Lena Buffoni, Nicolas Albarello, Pablo Fontes De Miranda, and Peter Fritzson. An aeronautic case study for requirement formalization and automated model composition in modelica. In Peter Fritzson and Hilding Elmqvist, editors, Proceedings of the 11th International Modelica Conference. Modelica Association and Linköping University Electronic Press, September 2015.

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

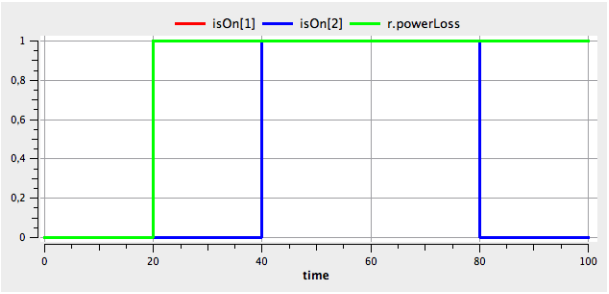


Figure 1: The power loss of the main power system and the switching on/off of backup units 1 and 2

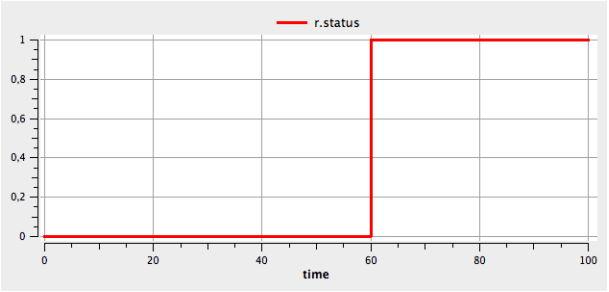


Figure 2: The requirement status, where 0 represents undefined, -1 violated and 1 not_violated

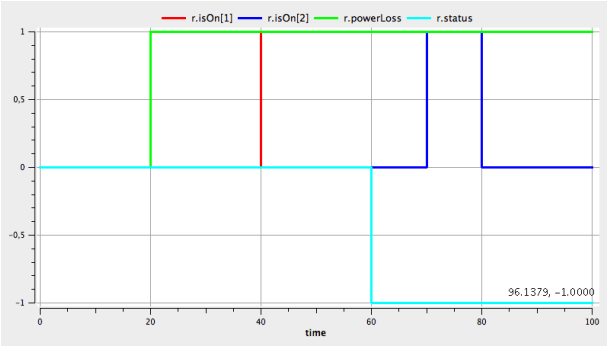


Figure 3: The requirement is violated because the power block 2 is not turned on time.

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.

Modeling, Identification and Control, Vol. 35, No. 2, 2014, pp. 79–91, ISSN 1890–1328

An Approach to Automated Model Composition Illustrated in the Context of Design Verification

Wladimir Schamai [1] Lena Buffoni [2] Peter Fritzson [2]

[1] *Airbus Group Innovations, Hamburg, Germany E-mail:* wladimir.schamai@airbus.com

[2] *Linköping University, SE-581 83 Linköping, Sweden. E-mail:* {lena.buffoni, peter.fritzson}@liu.se

Abstract

Building complex systems form models that were developed separately without modifying existing code is a challenging task faced on a regular basis in multiple contexts, for instance, in design verification. To address this issue, this paper presents a new approach for automating the dynamic system model composition. The presented approach aims to maximise information reuse, by defining the minimum set of information that is necessary to the composition process, to maximise decoupling by removing the need for explicit interfaces and to present a methodology with a modular and structured approach to composition. Moreover the presented approach is illustrated in the context of system design verification against requirements using a Modelica environment, and an approach for expressing the information necessary for automating the composition is formalized.

Keywords: Bindings, model composition, requirement formalization, design verification

1. Introduction

With the increasing complexity of cyber-physical systems, determining whether a particular system design fulfills or violates requirements that are imposed on the system under development can no longer be done manually and requires formalizing a requirement into some computable form. For such a formalized requirement to be verified, it will need to obtain the necessary information from the system model that is being verified, that is, it needs to be combined together with the system. In complex systems with large numbers of requirements, there is a need for an automated approach for composing the requirements with a given system design for the purpose of verification. This task is further complicated by the fact that the requirement models and the physical models are developed separately, and can operate on different quantities and data-types therefore combining them together in order to enable design verification is far from trivial.

This paper builds upon a new approach that enables automated composition of models by expressing the minimum information necessary to compose the models in the form of *bindings* (Schamai, 2013a). It presents a proposal for implementing the *bindings* concept in a Modelica-based environment (Modelica Association, 2013). In contrast to an approach that is based on defining interfaces that models have to implement, our approach enables the integration or composition of models without the need for modifying those models.

Although the concepts proposed in this paper are designed to bind any kinds of components together, we illustrate them in the context of design verification (IEEE1220, 2005; NCOSE, 2006; Kapurch, 2010). More specifically, bindings are used in vVDR (Virtual Verification of Designs against Requirements), a method that enables model-based design verification against requirements. In our examples we wish to verify a particular system design, represented by a Modelica model, against requirements that are formalized as requirement violation monitors models in Modelica.

doi:10.4173/mic.2014.2.2

The need for the bindings concept, presented in this paper, was identified in Schamai (2013a) when trying to find a way to automatically compose simulation models that can be used for design verification. The approach in Schamai (2013a) shows how natural-language requirements (see Hull et al. (2005) for examples) are formalized in order to monitor their violations during simulations. The formalization approach of requirements into monitors is inspired by concepts from run-time verification (Leucker and Schallhart, 2009).

In order to compose simulation models [1] automatically, i.e., to combine the formalized requirements (i.e., violation monitor models), system design models and scenario models, the bindings concept is elaborated in Schamai (2013a) and prototyped in the ModelicaML language (Schamai, 2013b). ModelicaML enables using UML (OMG, 2013) diagram notations for modeling complex physical system and using Modelica for simulations. It supports the model-based design verification method proposed in Schamai (2013a).

The main contribution of this paper is a proposal for leveraging the bindings concept in Modelica directly. The new feature is expected to avoid modeling errors, to reduce the manual modeling effort for integrating models, to enable automated model composition of Modelica models, as well as to establish traceability between models.

In order to illustrate the use of bindings in Modelica, we use the simplified example of a cooling system containing a number of pumps. For the purpouse of this paper, the pumps are modeled in a very simplistic manner, as we are only interested in whether the pumps are turned on or off. However, even such a simple property can be modeled in different ways for different kinds of pumps. We will use this setup to illustrate how the requirement model can be decoupled from the way that a specific property is modeled in the physical representation of the system.

This paper is organized as follows: Section 2 presents the basic concepts. Sections 3 and 4 present two different alternatives for capturing bindings and explain through the use of examples how the model composition can be automated. Section 5 discusses use of bindings in the context of the vVDR methodology and finally Section 6 sums up the work presented and discusses further research directions.

2. Basic Concepts

In this section we give the definitions of the basec concepts used in our approach, and illustrate them by using simple examples of water pumps from a cooling system. To this end we introduce the notions of *clients* and *providers*. Clients require certain data; providers can provide the required data. However, clients and providers *do not know each other* a priori.

Moreover, there may be *multiple clients* that require the same information. On the other hand, data from *several providers* may be needed in order to compute data required by *one client*. This results in a many-to-many relation between clients and providers.

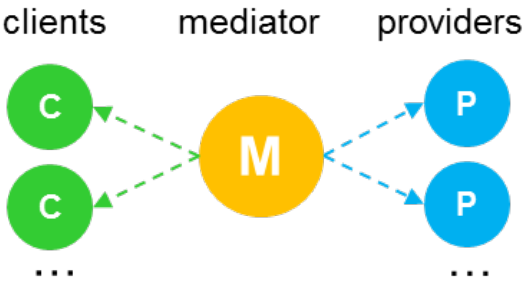


Figure 1: Concept of clients, mediator and providers

In order to associate the clients and the providers to each other we introduce the *mediator* [2] concept, which is an entity that can relate a number of clients to a number of providers, as illustrated in Figure 1. *References* to clients and providers are *stored in mediators* in order to avoid the need for modifying client or provider models.

The purpose of the presented concepts is to capture all the information needed for inferring (i.e., generating) binding expressions for clients. We define a binding as follows:

```
client instance reference = binding expression
```

A binding is a causal relation, which specifies that, at any point in simulated time, the value of the referenced client instance shall be the same as the value computed by the *right-hand expression*. When composing models, the *right-hand side*, i.e., the binding expression for the client at hand, is *inferred* using the corresponding mediator and its referenced providers.

Mediator, client or provider references may have attached template. Templates are code snippets that contain placeholders for context sensitive information. These placeholders or macros are replaced or expanded when the binding expression is generated. For example,

[1] In Schamai (2013a), requirement violation monitor models and other models are ultimately translated into Modelica models for simulations.

[2] The idea of mediators is similar to mediator pattern from the software development domain (Gamma et al., 1995), which promotes the idea of a loose coupling of objects to enable their communication without them explicitly knowing each other's details.

a template attached to a mediator can contain macros for either reducing lists of providers to a single element (e.g. min(:), max(:), sum(:), etc.) or to return arrays (e.g. toArray(:)) or information about the array (e.g. size(:)). Templates attached to client or provider references can contain code snippets for unit or type conversion or references to sub-components.

3. Alternative 1: Binding Specification in Modelica

This section presents the approach for capturing information required for an automated model composition using an extended version of the Modelica language.

Assume that we have a system that contains several pumps, and an informal requirement, such as, *"At least 2 pumps shall be in operation at any time"*. Independent of a specific system design (i.e., the number and type of pumps contained in the system model), this requirement can be formalized into a requirement violation monitor Modelica model as follows:

```
package Requirements
model Req
input Integer numberOfOpPumps = 0 "
   number of operating pumps";
constant Integer minNumberOfOpPumps
   = 2 "min. number of operating
   pumps";
output Integer status(start=0,fixed=
   true) "indication of requirement
    violation, 0 = not evaluated";
  equation
    if numberOfOpPumps <
       minNumberOfOpPumps then
      status = 2 "2 means violated";
    else
      status = 1 "1 means not
         violated";
    end if;
  end Req;
end Requirements;
```

This requirement violation monitor model has as an input `numberOfOpPumps` (that is, the number of operating pumps). When this requirement violation monitor model will be used (i.e., instantiated within another model) in simulations, this input component will need to be bound to some expression that calculates the number of operating pumps within the system for a given system design. Simulating the requirement violation monitor model is pointless without binding it to an expression that calculates this information during simulation.

Further, assume we have the following system design model that contains 3 pumps of 2 different types.

```
package Design
  model System
    PA pump1;
    PB pump2;
    PB pump3;
  end System;

  model PA
input Boolean on = false
"to turn the pump on or off";
  end PA;

  model PB
input Boolean switchedOn = false
"to switch the pump on or off";
Real volFlowRate
"volume flow rate of the pump";
  equation
    if switchedOn then volFlowRate =
        1;
    else volFlowRate = 0;
    end if;
  end PB;
end Design;
```

Additionally, there is a scenario[3] model that can be used for stimulating the system model, e.g., it will turn on and off different pumps during simulation.

```
package Scenarios
  model Scenario
output Boolean pump1active "turn on
   pump1 = true, turn off pump1 =
   false";
output Boolean pump2active "turn on
   pump2 = true, turn off pump2 =
   false";
output Boolean pump3active "turn on
   pump3 = true,  turn off pump3 =
   false";
  algorithm
    if time > 0 and time <= 10 then
          pump3active := false;
    elseif time > 0 and time <= 20
       then pump2active := false;
    elseif time > 0 and time <= 30
       then pump2active := true;
    else
      pump1active := true;
      pump2active := true;
```

[3] In Schamai (2013a) such models are referred to as *verification scenarios*. They are used to stimulate the system model such that a set of requirement violation monitors can be evaluated.

```
      pump3active := true;
    end if;
  end Scenario;
end Scenarios;
```

We create a new model (e.g. `AnalysisModel`[4]) that contains instances of the requirement monitor model (`req1`), the system model (`sys`), and the scenario model (`scen`). Now we wish to automatically find all components (i.e., clients) that require data from other components (i.e., providers) and *bind* them.

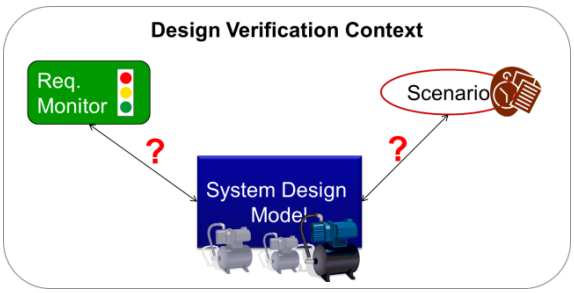


Figure 2: Integrating models for design verification

In Modelica there are two possible ways to integrate models: *acausal* and *causal*. *Acausal* connections (Modelica connect equations) are typically used for modeling physical energy or material flow. In our setting there is no need for acausal connections because requirement monitors are observers (i.e., they must not impact the system model) and scenarios stimulate the system and may observe it. *Causal* connections (Modelica connect equations, component modifiers) are sufficient for connecting requirement monitors, system design and scenarios.

Another question that we need to answer, is how we want to connect models in Modelica: using equations (that require ports of predefined compatible types to be connected) or using component modifiers (i.e., by replacing the declaration equations for input components)? Connections, i.e., connected ports with predefined interfaces, would require an extension or modification of the involved models. This approach may clutter up models with instrumentation code that is not part of the system design, requirement or scenarios. In contrast to pre-defined interfaces, component modifiers can be added when needed and adapted to the context. In the following we will apply the later approach: we will integrate models in a causal way using Modelica component modifiers. The reasons for this decision are twofold. First of all a physical model needs to be designed separately as a standalone model, and therefore having unconnected connectors that are only used in the presence of the requirement model, which would be an issue. Moreover, the binding expressions are generated automatically; therefore their complexity does not impact the user, however if we use connectors, which are in turn used to automatically generate connection equations we add an extra level of complexity to the computation model.

In our example, binding expressions shall be generated for the input component `Requirements.Req.numberOfOpPumps`, and for the pump components within the system. What we want to achieve is shown below:

```
model AnalysisModel
  Requirements.Req req1(
    numberOfOpPumps = sum({
      (if sys.pump1.on then 1 else
         0),
      (if sys.pump2.volFlowRate > 0
         then 1 else 0),
      (if sys.pump3.volFlowRate > 0
         then 1 else 0)}));
  Design.System sys(
    pump1(on = scen.pump1active),
    pump2(switchedOn =
       scen.pump2active),
    pump3(switchedOn =
       scen.pump3active));
  Scenarios.Scenario scen;
end AnalysisModel;
```

In `AnalysisModel` the component `numberOfOpPumps` of the instantiated requirement model is now *bound* to an expression that will calculate the number of operating pumps within the system during simulations by obtaining data from the system model. The system model components `pump1`, `pump2` and `pump3` are bound to the scenario model instance.

In order to do this automatically, the following information will need to be captured by experts because it cannot be deduced automatically. Experts (i.e., people that are familiar with either clients or provider models) will need to specify:

- which models or components are clients,
- which models or components are the corresponding providers,
- and, in case there is no 1:1 compatible mapping of client and provider components, what template should be used to generate the appropriate binding expressions?

In the following section we present a proposal for a Modelica concrete syntax for capturing this *minimum set of information* required to enable such automated generation of binding expressions.

[4] In Schamai (2013a) such models are referred to as verification models

3.1. Modelica Extension for Defining Mediators

As mentioned in Section 2, we suggest storing all information, required to enable inferring binding expressions, in a so called *mediator*. This concept does not exist in Modelica. All Modelica code hereafter is a new proposed extension to the Modelica concrete syntax.

In order to modularize the process, the information captured by mediators can be defined in multiple mediators that use inheritance (i.e., `extends` relation in Modelica).

Assume that in our process we have the role of a *requirement analyst*, a person in charge of elicitation, negotiation and formalization of requirements. When formalizing the requirements, the requirement analyst will define mediators in order to expose the information needed by the clients. For example, the requirement analyst would create the mediator `NumberOfOperatingPumps_C` and associate the component `numberOfOpPumps` (the input component of the requirement violation monitor model) to it as follows:

```
package PartialMediators
  mediator NumberOfOperatingPumps_C
     requiredType Integer;
     clients
      mandatory Requirements.
        Req.numberOfOpPumps;
  end NumberOfOperatingPumps_C;
end PartialMediators;
```

Note, that the only way to reference a client is to use its *model qualified name* (e.g., `Requirements.Req.numberOfOpPumps`). The model `Requirements.Req` may be used in different context, i.e., instantiated in another model and will be given an *instance path* (e.g., req1.numberOfOpPumps in the context of `AnalysisModel`). However, we cannot know the instance path a priori.

So far, the mediator only contains references to *clients*, i.e., models or components that require the information, however, no description of how to get this information yet. In that sense this mediator is *incomplete*.

For the sake of simplicity, there is only one client model that requires the number of operating pumps within the system in our current example. In a larger example there are likely to be more models (e.g. other requirement violation monitors) that will also need the same information. In that case there will still be only one mediator that would then contain references to more clients in the `clients` section. This way, mediators allow a *grouping* of models or components *that require the same information* and enable a concise definition of bindings.

This mediator also indicates the type of data to be provided to the clients, the `requiredType` reference, which must be compatible with the type of all associated clients. This way, the mediator reflects what is needed by clients. There is no need to analyze each client anymore, it is sufficient to only look at mediators. This is especially useful if a different person, with no knowledge of the requirement model is in a charge of defining the providers in the model.

Moreover, the client reference can have the prefix `mandatory`, to indicate that this client must be bound[5]. In our example it is the input of the requirement violation monitor model that has to receive the corresponding value during simulation.

At some point in time, another person, e.g., *system designer* will specify which models can provide the information required by clients and how to compute it from a particular *provider model*. As mentioned above, for doing so, system designer will only need to look at mediators. There is no need to analyze the referenced clients (which may be many in larger models) because the mediator unambiguously reflects the content and the type of required data.

To this end, system designer creates a new mediator `NumberOfOperatingPumps_P`, that extends the mediator `NumberOfOperatingPumps_C`. By using inheritance, the new mediator obtains all client references. Now system designer can add references to provider models and specify how the binding expression should be generated in case there is no 1:1 mapping between clients and providers.

```
mediator NumberOfOperatingPumps_P
  extends PartialMediators.
      NumberOfCaviatingPumps_C;
  template sum(:) end template;
  providers
     Design.PA.on
        template if getPath() then
            1 else 0 end template
           ;
     Design.PB.volFlowRate
        template  if getPath() > 0
            then 1 else 0 end
           template;
end NumberOfOperatingPumps_P;
```

If there is no 1:1 mapping, in addition to the client or provider references, templates can be used to specify how the expression code should look like and where to insert context-sensitive data.

For example, a template can be attached to the mediator in order to either specify how to aggregate in-

[5] An example for a not mandatory client is shown at the end of this section.

formation in case several providers are in place, or to specify that it only should contain constant data.

In our example, the mediator will be used to generate an expression that will calculate the number of operating pumps for any design, i.e., for any number and type of pumps contained in a particular system design model. The strategy for computing this information in this example is the following. Each provider model (i.e., a model of a pump) shall indicate whether the pump is in operation by returning 1 if it does and 0 otherwise. Then the *sum*(:) operator, specified in the mediator `"template` *sum*(:) `end template;"`, will be turned into an expression to calculate the total number of operating pumps in the system during simulations.

The colon in *sum*(:) indicates that this operator expects an array of unordered items. In our example *sum*(:) will be mapped to the Modelica array reduction function *sum*(A) where A is an array of expressions each calculating whether a pump instance is in operation. Note that such expressions may be different for different pump model instances within the system model at hand. In general, the mediator templates are allowed to include any template function that accepts as input an unordered list.

Now, for each referenced provider model of a pump (i.e., in our example there are `Design.PA` and `Design.PB`), we specify how to determine whether the pump is in operation using the provider template. Based on the strategy of the mediator, the following template for the provider model component Design.PA.on `"if getPath() then 1 else 0"` specifies that 1 is returned if the pump is in operation and 0 otherwise. This is different for the second pump type. Here we will be using the component `Design.PB.volFlowRate` and the template `if getPath() > 0 then 1 else 0`.

For referencing provider model sub-components, the `getPath()` operator is used. It is a placeholder that will be replaced[6] by the instance path of the provider model in a particular context (i.e., in our example in the context of `AnalysisModel`, see below).

The mediator `NumberOfOperatingPumps_P` is *complete*. It contains all the information needed to automatically generate the binding expression for all referenced clients[7]. Note that this mediator is independent from the number of pump instances in a given design. As long as the referenced provider models (i.e., `Design.PA` and `Design.PB`) are used in a system design model, this mediator can cope with any number of pump instances (e.g., 20 pumps instead of 3 pumps in our example model `Design.System`).

The inferred binding expression will then be passed[8] to each client that requires the information about the number of pumps that are in operation during a simulation run.

Also, note that the purpose of creating another mediator is to show that this approach can be used for separating concerns and making definitions reusable. In the same way, we could use the first mediator `NumberOfOperatingPumps_C` and add the missing information to it. Separating concerns may be necessary because different people will be the owners of different mediator models, or because the same mediator, that contains client references, may be reused for different designs with provider models specific to the design at hand. Figure 3 summarizes the information that needs to be captured in abstract syntax.

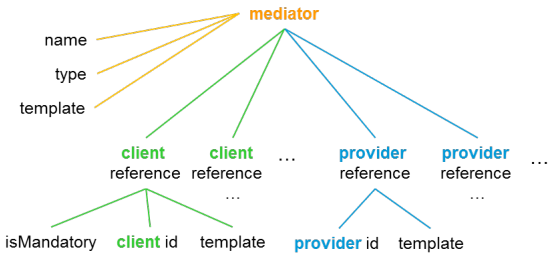


Figure 3: The information that needs to be captured in abstract syntax.

The mediator name[9] reflects what is needed by clients. The mediator `requiredType` must be compatible to each of its clients[10].

Client or provider id is the *qualified name* of the client or provider model or component (e.g. `Requirements.Model1.comp1`). The attribute `mandatory` (`true` by default) indicates whether the client must be bound. If not, the client component must have a default value.

All templates are optional. For example, if there is only one provider that returns exactly what is needed by clients, there will be no need for neither the mediator template nor for the provider template.

In contrast, if the binding expression will refer to several provider models for which the code for accessing the right data is different (like in our running example), then probably a mediator template and a provider template will be necessary. A client template will be

[6] The `getPath()` operator will be replaced with the instance paths of providers when the binding expression will be generated for a particular client.

[7] Recall, in our example we associated clients in the `NumberOfOperatingPumps_C` mediator that is extended by `NumberOfOperatingPumps_P`

[8] In our proposal the generated binding expression will be passed via the Modelica component modifier to the client component.

[9] Comments may be included like for any Modelica model.

[10] In addition, the mediator should indicate the lowest variability of its clients (this is not discussed in this paper).

needed as soon as the actual client is not the referenced client model but a sub-component of it.

Client or provider templates are expressions that can contain instance paths (e.g. in Modelica using the dot-notation) for referencing sub-components within the client or provider models. A mediator template can only contain predefined macros or built-in functions (e.g. $sum(:)$, $toArray(:)$, $card(:)$, $min(:)$, $max(:)$, etc.) or constant data.

A client template is needed in order to enable pointing to sub-components. For example, in our system model there are several pumps that can be turned on or off. System designer can expose the potential stimuli of the system by creating a new mediator and associating the clients as shown below. The provider references will be added by the tester who will be creating the scenario models. Now, the client template in the mediator `Pump1IsOn` is necessary because we need to point) a particular pump instance within the system model, in this case the pump1 component within `Design.System`.

```
package Mediators
...
mediator Pump1IsOn
      requiredType Boolean;
   clients
      Design.System.pump1
         template getPath().on =
            getBinding();
         end template;
    providers
      Scenarios.Scenario.pump1active
         ;
end Pump1IsOn;
...
end Mediators;
```

If there will be more scenario models that turn on and off the pump1, the mediator will still be the same and will only need to include the additional scenario model references in the `providers` section. Similarly, if there is a new design alternative, say one that contains 20 pumps of the same type instead of 3 like in the example above, the bindings will still be generated correctly. If there will be another type of pump (i.e., other than the two types from the example above), then we will merely need to add a new provider reference to the mediator.

To sum up, in order to integrate the binding concept into the Modelica language, we define a new type of class, mediator, which has a particular structure. However the scope of these extensions is limited, as all the extensions are confined to within the mediator, and moreover, once the bindings are generated the models only contains standard Modelica, which means that they will be compatible with any Modelica tools.

3.2. Generating Binding Expressions

In this section we illustrate how we use the information contained in the mediators to automatically generate binding expressions.

3.2.1. Templates

Let us first come back to the question why we need client, mediator, and provider *templates*. A template defines the form of the binding expression to be generated. It specifies where to insert context-sensitive information, such as the instance path of components (i.e., which will replace the placeholder `getPath()`) in order to enable pointing to particular client or provider subcomponents.

The purpose of the mediator template is to specify how to reduce arrays to a single value, or to specify that constant data should be passed to the associated clients. Client and provider templates are primarily used to enable pointing to sub-components. Client templates are also used for overwriting binding definitions. A consideration of possible cases and general validation rules for templates can be found in Schamai (2013a). They are used for generating valid bindings. A *binding* is said to be *valid* if the binding expression can be inferred for client ci_k and the resulting type of the right-hand binding expression is compatible with the left-hand-side expression.

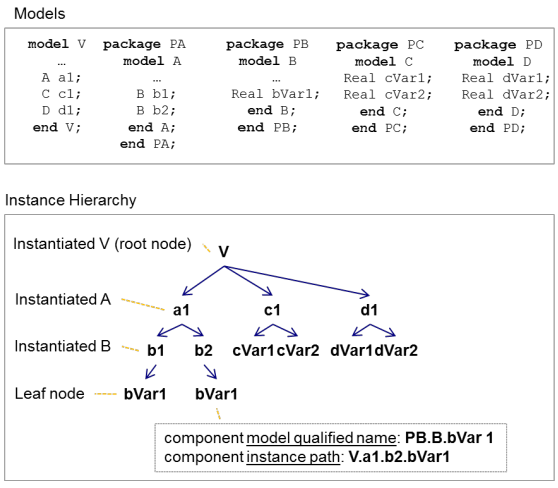


Figure 4: The information that needs to be captured in abstract syntax.

3.2.2. Instantiation Tree

Let us first now introduce a structure, called *instantiation tree*, which is used for inferring binding expressions. It is a tree that starts with the root node representing the model being instantiated (see Figure 4).

Each child node represents a component of the parent model. The recursive tree construction stops at leaf nodes. Leaf nodes represent components of primitive types, which do not have any further internal structure. Figure 4 shows an example of Modelica models[11] and the correspondent instance hierarchy.

Model qualified name is the path (structured name) of a model element - e.g. a class or an attribute of a class - that identifies the element within the structure used to organize the model (e.g., by means of packages or nested classes). Instance path identifies a component within an instantiated model (for example, in our setting, within `AnalysisModel`).

Each tree node contains all the relevant information about the element (e.g. the component model qualified name, component instance path etc., see Figure 4). They are necessary in order to match clients and provider instances based on the model qualified names within mediators.

3.2.3. Algorithm

In `AnalysisModel`, we first import mediators that should be used for generating the binding expressions (i.e., we add an import clause import `Mediators.*` to import the mediators `Mediators.NumberOfOperatingPumps_P`, `Mediators.Pump1IsOn`, etc.).

Now, in order to generate binding expressions for each client in `AnalysisModel` we trigger the new tool feature "Update bindings". Figure 5 below shows how such an invocation could look in a Modelica tool[12].

The algorithm will first create an instantiation tree[13] and collect all client components, mediators to be used for inferring binding expressions and all the referenced provider components contained[14] in `AnalysisModel`.

While creating the instantiation tree (see Figure 6) the algorithm takes the imported mediators into account in order to identify (i.e., match) nodes (i.e., components) that are *clients* or *providers*. Clients and

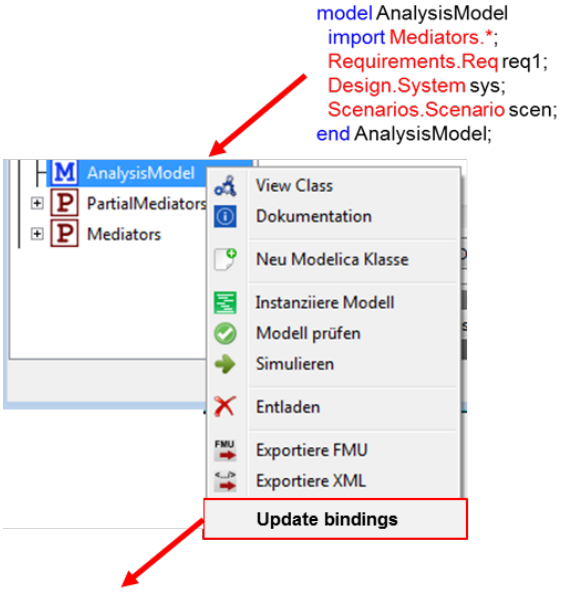


Figure 5: The information that needs to be captured in abstract syntax.

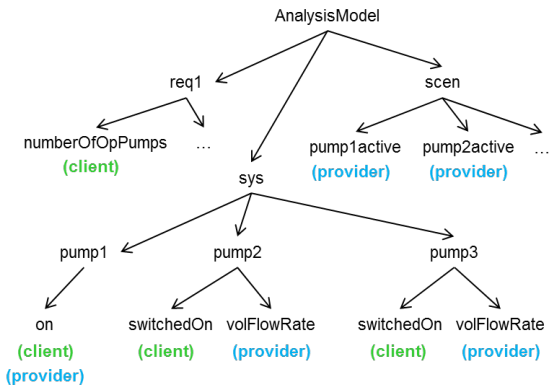


Figure 6: The information that needs to be captured in abstract syntax.

[11] In Modelica the primitive types, such as Real, Integer, String, and Boolean, still have one level of internal structure of predefined properties, see Modelica Association (2013).

[12] Such as the OpenModelica graphical editor OMEdit [4]

[13] The algorithm for constructing the instantiation tree is not shown in this paper. It is a traversal that is straight forward to implement.

[14] Note that a mediator may contain much more client or provider references. However, now, in a specific context, the algorithm will only consider those that are contained in the model at hand.

providers are identified by comparing the *model qualified name* of each component node in the instantiation tree with client and provider references of the imported mediators.

For example, `req1.numberOfOpPumps` is a client because there is a mediator (`Mediators.NumberOfOperatingPumps_P1`) that references this component using its model qualified name (`mandatory Requirements.Req.numberOfOpPumps`). Moreover, it is a mandatory client. This client must be bound to some expression. This means that if no binding expression can be inferred for that client, an error should be reported.

Now the algorithm for inferring the binding expression can be triggered. It is described in pseudo-code in the Appendix [15]. The algorithm requires as input an instantiation tree node which is a client, and the set of mediators to be used for inferring binding expressions.

Mediator that contains client references:

```
mediator NumberOfOperatingPumps_C
  requiredType Integer;
  clients
    mandatory Pack1.Req.numberOfOpPumps;
  ...
```

Mediator that also contains provider references:

```
mediator NumberOfOperatinggPumps_P1
  extends NumberOfOperatingPumps_C;
  template sum(:) end template;
  providers
    Pack2.PA.on
      template if getPath() then 1 else 0 end template;
    ...
```

Model with updated bindings:

```
model AnalysisModel
  import Mediators.*;
  Pack1.Req req1(
    numberOfOpPumps = sum({
      (if sys.pump1.on then 1 else 0),
      (if sys.pump2.volFlowRate > 0 then 1 else 0),
      (if sys.pump3.volFlowRate > 0 then 1 else 0)}));
  ...
end AnalysisModel;
```

Figure 7: The information that needs to be captured in abstract syntax.

The algorithm first identifies the actual client[16] and finds the corresponding mediator. Then, based on the mediator template, the associated providers (e.g. `sys.pump1.on`, `sys.pump2.volFlowRate`, etc.), and the provider templates (e.g. `template if getPath() then 1 else 0 end template` for provider model component `Design.PA.on`) the algorithm tries to generate the binding expression.

For the requirement model component client `Requirements.Req.numberOfOpPumps` the binding expression can be inferred. Figure 6 explains which parts of the mediator specification were used to generate which parts of the binding expression that is passed to the client via Modelica component modifier (see component modification for `numberOfOpPumps` in `req1`).

Note that all placeholder occurrences of `getPath()` are now replaced by the corresponding instance path (e.g. `"sys.pump1."`) of the corresponding components within `AnalysisModel`. The mediator template template `sum(:) end template`; is mapped to the Modelica built-in function $sum(A)$ where A is the array of values, which in our example results from providers and expressions generated based on their templates.

Finally, the inferred binding expression is passed to the client (e.g. to the instance of `Requirements.Req.numberOfOpPumps`) via the Modelica component modifier in its first-level component (i.e., `req1`). The updated `AnalysisModel` is shown in Figure 5.

4. Alternative 2: Binding Specification in XML

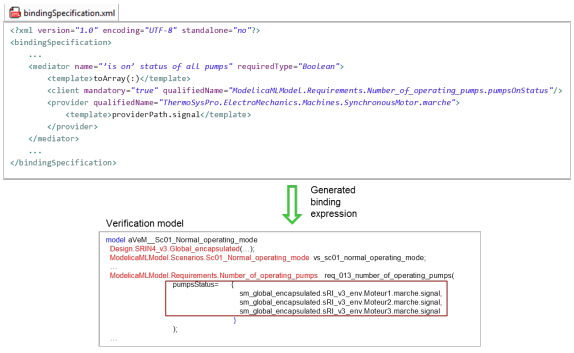


Figure 8: Bindings specification is used to generate binding expressions.

In this Section we present an alternative approach for capturing binding specification. In contrast to the approach presented in Section 3, instead of writing the bindings specification (i.e., the mediator classes) in an extended version of Modelica, we capture this information using XML (W3C, 2013).

[15] In Schamai (2013a)] templates are referred to as operations.

[16] When looking for the client c there may be component at a higher hierarchy level that uses its client template to point to it . This is the actual client because, similar to Modelica *component modification* mechanism, the higher-level components overwrite definitions for lower-level components.

Assume we have the following requirement *"013 - When the system is in operation, there should be no less than two pumps in operation for more than 2 seconds"*. The Modelica requirement violation monitor may have the following inputs:

```
model Number_of_operating_pumps
 input Boolean isNormalOperatingMode
    ;
 input  parameter Integer
    numberOfPumps=0;
 input Boolean pumpsOnStatus[
    numberOfPumps];
...
```

Consider the input `pumpsOnStatus[:]`. It is an array of type Boolean. It's size is equal to the number of pumps within the system. Each array entry indicates whether the pump is in operation (i.e., = true) or not (i.e., = false). This array will need to be constructed based on the system model at hand, i.e., based on the pumps within the system, and passed to the instance of the requirement violation monitor model above.

Figure 8 shows an example of an XML file that captures the bindings specification (only for the part relevant for our example). Inferring of binding expressions would work same as for the Modelica-based version described in Section 3. However, instead of parsing the extended version of Modelica here the tool will need to parse XML. The advantage of this approach is that there is no need for extending the Modelica language. The only thing to be agreed on is the XML Schema, i.e., the structure of such XML files.

5. Further Applications for Bindings

This section mentions one possible application for using the presented bindings concept. Assume that we wish to combine a given system design model with a scenario model and a set of requirements the design shall be verified against. Such a combination is a new model, called *verification model*.

To do so, first, we need to find combinations of one *system design model*, one *scenario model*, and a *set of requirements*. Thereby, we want to ensure that only those scenarios should be considered which can stimulate the system model. In addition, we want only to consider requirements that are already addressed in the design.

Figure 9 illustrates[17] how bindings can be leveraged to create such verification models. It starts with the one design alternative selected by the user. Then it iterates over all found scenarios[18].

For each of the scenarios it checks whether all mandatory clients of the design and this particular scenario model can be satisfied. If yes, the scenario is selected because we can assume that it will stimulate the system design model and it will receive all necessary feedback from the design model. If not all mandatory clients can be satisfied, then this scenario is discarded.

Satisfying a client means that based on the given set of mediators and providers in place, for this client it is possible to infer a valid binding expression.

Further, having selected a scenario, the algorithm iterates over all requirements that are referenced by this particular scenario with a relation indicating that this scenario can be used to verify designs against this particular requirement. For each requirement, again the algorithm checks whether all mandatory requirement violation monitor clients can be satisfied. If yes, the requirement is added to the combination. If no, then we assume that this requirement is not completely or not correctly addressed in the given design and cannot be evaluated yet.

An outer-loop iteration terminates with a selected scenario and one or more requirements. This combination can be translated into a verification model by instantiating the design, selected scenario, and requirements and passing the inferred bindings to clients.

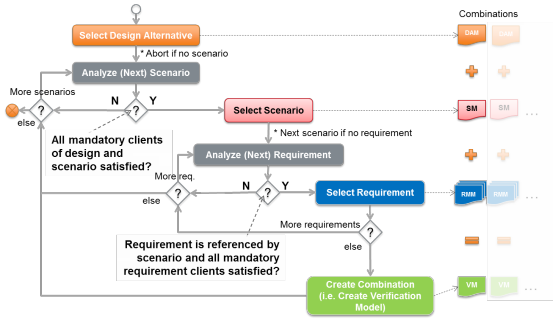


Figure 9: : Illustration of the verification models generation algorithm.

6. Conclusion

In this paper we have presented a new concept that enables the automation of Modelica model composition. We have proposed two different approach to support the process of expressing the information necessary to

[17] Pseudo code for this algorithm can be found in Schamai (2013a)

[18] Scenarios are models that are annotated to be a scenario. Such, for example, Modelica models stimulate the system model during simulation in order to enable evaluation of requirement violation monitors.

enable the automated generation of binding expressions: one based on an extension version of the Modelica language and one based on XML. We illustrated the concept on examples in the context of design verification. In particular, we have shown how bindings of components can be generated for a given Modelica model.

This approach does not rely on an interface mechanism and therefore increases the decoupling of the models as it does not require prior knowledge of the interfaces by the models. Just as with classic interfaces, the binding concept respects the encapsulation principle, and only the information that is displayed publicly by the model can be bound. If a private attribute of a system model is required in order to obtain the information required for the composition of the models, then the model should be rethought, and eventually the required information made public. Furthermore, the possibility to define mediators in several steps means that the information can be provided by different people at different stages of the design process resulting in a more flexible and modular approach.

The advantages of using such an automated generation of binding expressions are the following:

1. Exposing and grouping of information about what data is needed by clients will reduce analysis work because the number of mediators will be smaller than the number of clients. The more clients will require the same information the greater will be the gain in terms of information reuse.

2. Automated generation of binding expressions will reduce modeling errors and the manual modeling effort. This is in particular true for models with highly interrelated components, or for complex binding expressions (e.g., Modelica component modifiers) too complicated to be written manually.

3. The binding concept enables a number of applications. For example, it enables automated composition of verification models from Schamai (2013a). Furthermore, it enables a formal traceability between client and provider models. For example, to determine which requirements are implemented in the system design model at hand, can be achieved by looking at the bindings for mandatory requirement clients.

References

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

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

IEEE1220. Ieee standard for application and management of the systems engineering process. IEEE, 2005.

Kapurch, S. *NASA Systems Engineering Handbook*. DIANE Publishing Company, 2010. URL http://books.google.se/books?id=2CDrawe5AvEC.

Leucker, M. and Schallhart, C. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 2009. 78(5):293 – 303. doi:10.1016/j.jlap.2008.08.004. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS07).

Modelica Association. Modelica 3.2 revision 2 specification. 2013. URL www.modelica.org.

NCOSE. *Systems Engineering Handbook (Version 3 ed.)*. INCOSE., 2006.

OMG. Object Management Group (OMG). 2013. URL www.omg.org.

Schamai, W. *Model-Based Verification of Dynamic System Behavior against Requirements*. Ph.D. thesis, Method, Language, and Tool Linköping: Linköping University Electronic, PressDissertations, 1547, 2013a.

Schamai, W. ModelicaML - UML Profile for Modelica. 2013b. URL www.openmodelica.org/modelicaml.

W3C. Extensible Markup Language (XML). 2013. URL www.w3.org/XML.

Appendices

A. Algorithm for Generating Binding Expressions

Algorithm: $infer Binding(ci, M\_S)$: Infer binding for a client

input : Client ci (node in instantiation I) for which binding is to be inferred
Set of mediators M_S that should be used

output: *bindingExpression* (binding expression, i.e., the right-side expression)
isBindingPossible (is possible to infer binding including manual decisions?)
ci_a (actual client used, i.e., ci or one of its parents in I)
m_a actual mediator used
$p_0 ... p_n$ set of providers referenced by m_c and contained in I

```
1  begin
       // Set all to undefined or false
2      isBindingPossible = false;
3      bindingExpression, ci_a, m = null ;                    // Set all to null (meaning they are undefined)
4      dataCollection = empty ;                                                          // Empty list
       // Get the instantiation (starting from root node) that contains ci
5      I = getInstantiation(ci) ;
       // Find the actual client (ether ci itself or an upper level node in I). Note that upper level client
          templates may over-write bindings from lower levels.
6      {n_0...n_n = ci} = getParentsTopDown(ci) ;                  // Get parents top-down including ci
7      for n_k in {n_0...n_n} do
8          m_0...m_n = getMediators(n_k, M_S) ;                  // Ordered list of mediators that reference n_k
           // For each n_k find those mediators with client references containing specifications for ci.
9          for m_k in {mk_0...mk_n} do
10             nco_k = getClientTemplate(m_k, n_k) ;                          // Get client template for n_k
               // If the left-side expression (placeholder replaced with instancePathOf(n_k)) is equal to
                  instancePathOf(ci) → then select. If there are there multiple matches, select only the
                  topmost entry because this overwrites lower entries. If we reached ci then select ci to be
                  the actual client.
11             if specifiesBindingFor(nco_k, instancePathOf(ci)) or n_k == ci then
                   // Check whether the client template and client reference are valid. A client template
                      must contain the placeholder (e.g. getPath()) to be replaced by a concrete client
                      instance path in I.
                   // If n_k is ci the client template is discarded because it can only specify lower-level
                      clients, which are not of interest at this point.
12                 assert (isValidClientTemplate(nco_k)) → abort otherwise
13                     "INVALID: Client template nco_k, attached to n_k specifying ci is not valid."
                   // Ensure that there is only one reference from mediator to client.
14                 assert (getClientReferences(m_k, n_k) ==1) → abort otherwise
15                     "INVALID: m_k references client n_k several times."
                   // Continue if asserts are true.
16                 ci_a = n_k ;                                                  // Actual client found.
                   // Get providers (and their templates) that are referenced by m_k and contained in I.
17                 {p_0...p_n}, {o_p0...o_pn} = getProviders(m_k, I) O_m_k = getMediatorTemplate(m_k)
                   // Mediator template can be empty or can contain reduction function macros.
18                 assert (isValidMediatorTemplate(O_m_k, sizeOf({p_0...p_n}))) → abort otherwise
19                     "INVALID: Mediator template O_m_k, of m_k is not valid for inferring binding for ci."
                   // Continue if asserts are true.
20                 dataCollection.add(ci, ci_a, o_ci_a, m_k, o_m_k, {{p_0...p_n}, {o_p0...o_pn}}) ;   // Add a raw to
                   dataCollection.
                   // It will be possible to infer binding assuming manual selection.
21                 isBindingPossible = true;
22             end if
23         end for
           // End of iteration over mediators.
24     end for
90     // Top-down iteration looking for c_ia.
```

25

// Check that there is only one entry, i.e., only one valid mediator.

26 **assert** *(sizeOf(dataCollection) == 1)* → **abort otherwise**

27 or **ask for manual decision** "Select a mediator from {...} for inferring binding for ci."

// Check whether the mediator template can handle multiple providers or if providers should be selected manually.

28 **if** *sizeOf*($\{p_0...p_n\}$) > 1 ***and not*** *(isMultiProviderMediatorTemplate(*O_{m_a}*) or preferredBindingExists(ci,* $\{p_0...p_n\}$*))* **then**

29 **abort** or **ask for manual decision** "Select provider from $\{p_0...p_n\}$ for inferring binding for *ci*." ;

30 **optionally** store the selection in **preferred bindings** by using instancePathOf(*ci*), modelQualifiedNameOf(*ci*) and instancePathOf(p_k), modelQualifiedNameOf(p_k)

31 **end if**

// Continue if assertions are true.

// If there is only one mediator, or it was selected manually, the first row contains all relevant data.

32 $ci_a, o_{ci_a}, m_a, om_a, \{\{p_0...p_n\}, \{o_{p0}...o_{pn}\}\} \leftarrow$ the first row in dataCollection

// A provider template may be empty or must contain placeholder (e.g. getPath()) for future provider instance path otherwise.

33 **assert** *(areAllValidProviderTemplates(*$\{\{p_0...p_n\}, \{o_{p0}...o_{pn}\}\}$*)* → **abort otherwise**

34 "Selected providers ... have invalid templates ..."

// If the actual client and the mediator are found and it is possible to infer binding.

35 **assert** *(isDefined(*ci_a*) and isDefined (*m_a*) and isBindingPossible)* → **abort otherwise**

36 "No binding could be inferred for *ci*."

// Continue if assertions are true.

// At this point we have found the actual client, mediator, providers, and all templates. Now we can generate the binding expression.

// In all provider templates, replace the placeholder with the concrete provider instance path in I.

37 $\{ot_{p0}...ot_{pn}\}$ = translateProviderTemplates($\{p_0...p_n\}$)

// Expand reduction function macros in the mediator template.

38 ot_m = translateMediatorTemplate($m_k, \{ot_{p0}...ot_{pn}\}$)

// Replace the placeholder with the client instance path in *I*; replace the placeholder with the inferred binding expres-sion.

39 *bindingExpression* = translateClientTemplate(ci_a, o_{ci_a}, ot_m)

40 **return** *bindingExpression, isBindingPossible,* ci_a*,* m_a*,* $\{p_0...p_n\}$

41 **end**

Any `abort` in the algorithm above means that the function returns the current status of the outputs. Functions such as `isValidClientTemplate(...)` test the validity of the template as described in Schamai (2013a). The `preferredBindingExists(...)` function (not explained in this paper, see Schamai (2013a) for more details) determines whether, in the given set of providers, there is one provider that should be used for the given client.

An Aeronautic Case Study for Requirement Formalization and Automated Model Composition in Modelica

Wladimir Schamai[1] Lena Buffoni[3] Nicolas Albarello[2] Pablo Fontes De Miranda[2] Peter Fritzson[3]

[1]Airbus Group Innovations, Germany wladimir.schamai@airbus.com

[2]Airbus Group Innovations, France {pablo.fontes-de-miranda, nicolas.albarello}@airbus.com

[3]IDA, Linköping University, Sweden, {lena.buffoni,peter.fritzson}@liu.se

Abstract

Building complex systems from models that have been developed separately without modifying existing code is a challenging task faced on a regular basis in multiple contexts including design verification. To address this issue an approach has been developed for automating dynamic system model composition by defining the minimum set of information that is necessary to the composition process. In this paper a design and implementation of this approach for standard Modelica is presented in the context of an application case study – the verification of a new design for spoiler activation against requirements.

Keywords: bindings, requirements, model composition, design verification

1 Introduction

Complex cyber-physical systems within safety critical application domains such as avionics need to take a lot of standard and specifications into account (Kepurch, 2010). For complex system, design verification is often challenging due to large number of requirements to be tested. For such systems an automated approach for connecting together system and requirement models is necessary.

Design verification takes place in system development steps starting from early concept evaluation to detailed system component design. The purpose of the presented approach to support design verification activities[1] by automating the task of simulation model composition.

This paper builds upon an approach that enables automated composition of models by expressing the minimum of information necessary to compose the models automatically (Schamai, 2013). In our case study, we show how binding specification can be defined using standard Modelica language (Modelica Association, 2012; Fritzson 2014), and show how the algorithm for automated binding generation can be implemented in OpenModelica. In contrast to an approach that is based on defining interfaces that models have to implement, this approach enables the integration and/or composition of models without the need for modifying those models. This means that requirement models and system models can be developed separately and existing models can be used without any modifications.

Explicitly exposing and grouping the information that is needed to interconnect the models will reduce analysis work. For example, when several requirements need the same information the same binding specification can be reused.

Additionally, automated generation of binding expressions reduces the risk of introducing errors and reduces modeling effort, in particular in models with highly interrelated components and/or complex binding expressions.

In the case study presented here we wish to verify a particular system design for spoiler activation, represented by a Modelica model, against requirements that are formalized in Modelica using the Modelica Requirements Library (Otter et al, 2014).

This paper is organized as follows: Section 2 presents the case study used in the paper. Section 3 describes the proposed syntax for defining bindings and illustrates it on the case study. Section 4 discusses the implementation of the algorithm for binding generation, and finally Section 5 summarizes the results presented in the paper.

2 Case Study Description

The selected case study is the design verification of the secondary flight system of an aircraft.

The secondary flight control system allows modifying the wing geometry, and consequently the aerodynamic behaviour of the aircraft, during the different flight phases and notably at take-off and landing. It is composed of spoilers, flaps and slats.

[1] Note, since this contribution focuses on implementing of

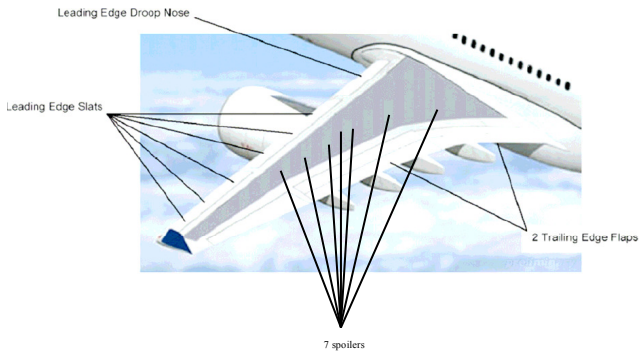


Figure 1. Flight Control System

On the Airbus A350 XWB, some new functions have been attributed to the SFCS system (Strüber, 2014):

- DFS (Differential Flap Setting): possibility of a differential inboard/outboard deflection for loads and drag control
- VC (Variable Camber): Uniform flaps deployment in cruise for drag control
- ADHF (Adaptive Dropped Hinge Flaps): the gaps between the flaps and the spoilers are optimized to reduce turbulences at high and medium speed.

These new functions induced a new architecture of the system and new control logics which both need to be tested.

Figure 2. ADHF configurations

In this new architecture, the actuation of the flaps is done by an actuation chain made of:

- Hydraulic motors
- Electrical motors
- Gears

The actuation of the spoilers is done via actuators being servo controlled actuators (SCA) or Electric Backup Hydraulic Actuators (EBHA). In order to simulate the system behaviour a Modelica model has been developed. The inputs of the system are flap commands, aerodynamic loads on surfaces (flaps and spoilers), and failures of some components. The model essentially uses blocks from the Modelica Standard Library except blocks modeling hydraulic components which were developed specially for this application. For confidentiality reasons, the content of the model cannot be disclosed.

2.1 Requirements Formalization

A system is developed based on requirements which are captured up-front typically using natural language (Hull, 2005). To test requirements they need to be formalized, i.e., they need to be translated into a machine readable form. In our case study we use the new Modelica Requirements Library (Otter et al, 2014) developed in the MODRIO project and the extension for calling blocks as functions implemented in OpenModelica (Buffoni and Fritzson, 2014).

In the following we show some examples of natural language requirements and their corresponding versions in Modelica. Each requirement is modeled such that it explicitly specifies the inputs it requires for evaluation. These inputs will need to be provided by the system or test scenario models. Further, each requirement has an explicit `status` attribute which is the requirement verdict that can take the values *undecided*, *violated* or *satisfied*.

Req.001 "*The torque of any ADGB electrical motor shall not be superior to 20 N.m for more than 1 sec.*"

This is translated into the following Modelica model.

```
model R1
  …
  input Torque ADGBtorque = 0;
  constant Torque maxTorque = 20;
  constant Duration maxDurationForTorqueOvershoot = 1;

  Property status(start = Property.Undecided, fixed = true);

  Modelica_Requirements.ChecksInFixedWindow.MaxDuration ma
xDuration(durationMax=maxDurationForTorqueOvershoot,check=c
ondition.y);
  Modelica_Requirements.Sources.BooleanExpression condition(y=
ADGBtorque >= maxTorque);
equation
  status = maxDuration.y;
  connect(condition.y, maxDuration.condition);
end R1;
```

The R1 model has one input `ADGBtorque`. It is the actual torque of any electrical motor. The value will need to be provided the R1 instance by the system model when testing this requirement using simulations.

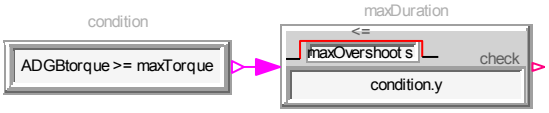


Figure 3. R1 Modelica model

Figure 3 shows the graphical view of the R1 model. It has two components: `Condition` and `maxDuration` from the Modelica Requirements Library. The `condition` component outputs *true* if the actual torque of the motor is greater than the defined threshold and *false* otherwise.

DOI 10.3384/ecp15118911

This output is used as input for the `maxDuration` component that outputs the status of the requirement violation. At the very beginning, as long as the condition is false it outputs `undecided`. This is because at this point the requirement was not yet evaluated. As soon as the condition returns `true` the `maxDuration` component will return `violated` if the condition was `true` for longer than 1 sec. or `satisfied` otherwise.

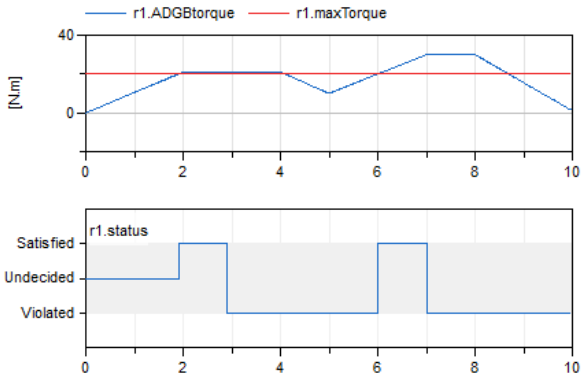


Figure 4. R1 test results

Req.002 "*The time of any action of flaps actuation (extension/retraction) shall be less than 50 sec.*".

```
model R2
  …
  input Boolean isFlapsActuationAction = false;
  constant Duration maxDuration = 50;

  Property status(start = Property.Undecided, fixed = true);

  Modelica_Requirements.ChecksInFixedWindow.MaxDuration ma
xDuration1(durationMax=maxDuration,check=condition.y);
  Modelica_Requirements.Sources.BooleanExpression condition(y=i
sFlapsActuationAction);
equation
  connect(condition.y, maxDuration1.condition);
  status = maxDuration1.y;
end R2;
```

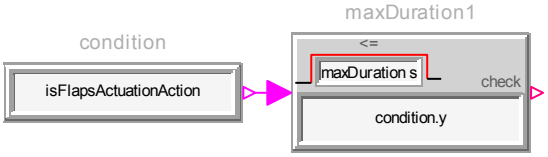


Figure 5. Req 002 Modelica model

The input to this model is `isFlapsActuationAction`. It is a Boolean type value to be provided by the system model when testing this requirement using simulations. Note that at this point it is not clear how to determine whether the flaps actuation action takes place.

In fact there may be several ways of accessing this information of one system design model, and there may be several system design alternative models. It is the task of the person who develops the design models to specify how this data can be accessed. Section 3 discusses how this can be done.

Figure 5 shows the graphical view of the R2 model. It includes two components: `condition` and `maxDuration` which are instances of models from the Modelica Requirements Library. The `condition` component outputs `true` as long as the action flaps actuation action takes place (i.e., extension or retraction) and `false` otherwise. This output is used as input for the `maxDuration` component that outputs the status of the requirement violation.

At the very beginning, as long as the condition is `false` it outputs `undecided`. This is because at this point the requirement was not yet evaluated at all. As soon as the condition the flaps actuation action starts, the `maxDuration` component will measure the time. It returns `satisfied` if the action took less than 50 sec. and `violated` otherwise.

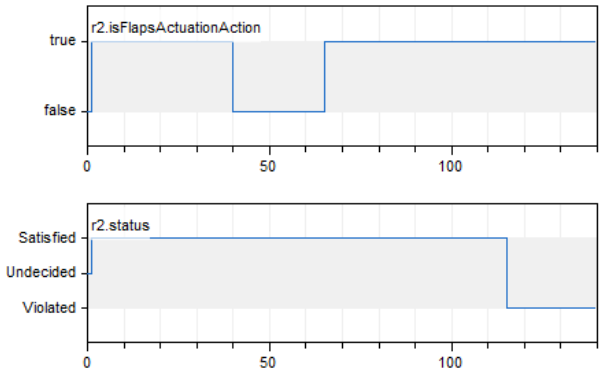


Figure 6. Req 002 test results

Req.003 "*The flap angle shall be comprised in the range [-5°;35°]*".

```
model R3
  …
  input Angle flapAngle;

  Property status(start = Property.Undecided, fixed = true);

  Modelica_Requirements.LogicalBlocks.WithinBand band1(u_max
=35, u_min=-5, u=flapAngle);
equation
  if (not band1.y) then
    status = Property.Violated;
  else
    status = Property.Satisfied;
  end if;
end R3;
```

The input for the model R3 is the `flapAngle`. Since there will be several flaps this requirement will need to be checked (i.e., instantiated) for each flap. The model `WithinBand` from the Modelica Requirements library is used for computing the verdict for this requirement.

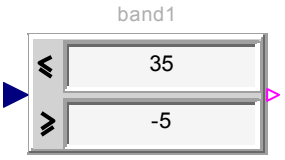


Figure 7. Req 003 Modelica model

The status can be evaluated at any time right from the beginning, i.e., there will be no time instant at which the status is `undecided`.

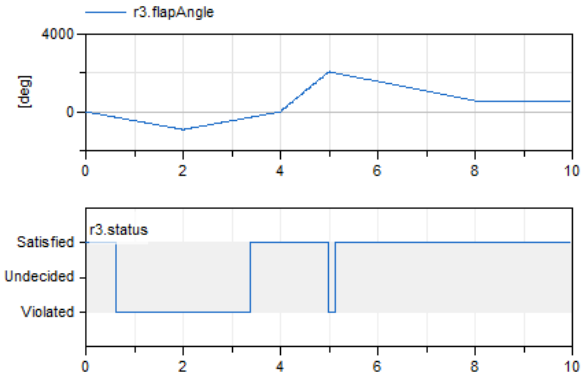


Figure 8. Req 003 test results

Req.004 "*When the flap is moving, the distance (gap) between a flap and its spoiler shall be less than 10 cm*".

Req.005 "*When the flap is not moving, the distance (gap) between a flap and its spoiler shall be less than 3 cm*".

The formalization of the requirements **Req.004** and **005** is similar to **Req.006** (see below)**.**

Req.006 "*The effort between a flap and its spoiler shall be less than 1000N*".

```
model R6
  …
  input Force forceBetweenFlatAndItsSpoiler=0;
  constant Force maxAllowedForce=1000;

  Property status(start = Property.Undecided, fixed = true);

  Modelica_Requirements.LogicalBlocks.LessThreshold l(threshold
=maxAllowedForce, u = forceBetweenFlatAndItsSpoiler);

equation
  if not l.y then
    status = Property.Violated;
  else
    status = Property.Satisfied;
  end if;
end R6;
```

Figure 9. Req 006 Modelica model

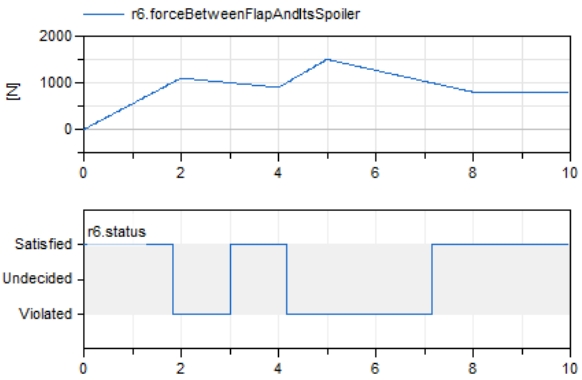


Figure 10. Req 006 test results

Req.015 "*The high lift system shall be able to hold the high lift surfaces in their current position:*

- *Under all load conditions;*
- *Under all relevant environmental conditions;*
- *After total loss of electric and hydraulic power (permanent or transient).*

```
model Requirement_15
…
 input Boolean hydraulicFailure;
 input Boolean electricalFailure;
 input Angle outboardValue;
 input Angle inboardValue;

parameter Real minDerivative = 0.01 "Values in degrees/s";
Property status(start = Property.Undecided, fixed = true);

Modelica_Requirements.ChecksInFixedWindow.During during1(ch
eck=not (flapsMoving.y));
Modelica_Requirements.Sources.BooleanExpression totalFailure(y
= hydraulicFailure and electricalFailure);
 Modelica_Requirements.Sources.BooleanExpression flapsMoving(
y=abs(der( SI.Conversions.to_deg(outboardValue))) > minDerivativ
e or abs(der( SI.Conversions.to_deg(inboardValue))) > minDerivati
ve);

equation
status = during1.y;

end Requirement_15;
```

Req.016 **"***Transients in normal system operations and in case of failure shall not cause excessive loads to components.***"**

This requirement is quite challenging to formalize since the conditions of excessive loads for each component must be defined.

The following formalization of the requirement uses arrays to gather variables coming from the different instances of the different components (ADGBs, flaps, PCU brakes). The binding mechanism will feed these inputs with the corresponding variables depending on the number of instances present in the model. The *PropertyAnd* block synthesizes the values of the different statuses with a 3-valued "and" logic.

DOI 10.3384/ecp15118911

```
model Requirement_16
…
  input Angle flapSpeed[:]; //left and right, inboard and outboard
  input Real ADGB_MotorTorque[:]; //left and right
  input Real ADGB_BrakeTorque[:]; //left and right
  input Real PCU_BrakeTorque[:]; //green and yellow brakes

  parameter Torque maxMotorTorque = 20;
  parameter Real maxDerivative = 2;
  parameter Torque maxADGB_BrakeTorque = 1e6;
  parameter Torque maxPCU_BrakeTorque = 1e6;

  Property status(start = Property.Undecided, fixed = true);

  Property ADGB_MotorStatus[size(ADGB_MotorTorque,1)];
  Property ADGB_BrakeStatus[size(ADGB_BrakeTorque,1)];
  Property PCU_BrakeStatus[size(PCU_BrakeTorque, 1)];
  Property flapOverspeedStatus[size(flapSpeed, 1)];

  Modelica_Requirements.LogicalBlocks.PropertyAnd andStatus(nu
=size(ADGB_MotorTorque,1)+size(ADGB_BrakeStatus, 1)+size(P
CU_BrakeStatus, 1)+size(flapOverspeedStatus, 1));

equation
  andStatus.u = cat(1,ADGB_MotorStatus,ADGB_BrakeStatus,PCU
_BrakeStatus,flapOverspeedStatus);
  andStatus.y = status;

  for i in 1:size(ADGB_MotorTorque,1) loop
    if abs(ADGB_MotorTorque[i])>maxMotorTorque then
      ADGB_MotorStatus[i]=Property.Violated;
    else
      ADGB_MotorStatus[i]=Property.Satisfied;
    end if;
  end for;

... (same for ADGB_BrakeTorque, PCU_BrakeTorque and
flapSpeed)

end Requirement_16;
```

Req.032 "*A single electrical failure shall not prevent an inboard flaps only movement.*"

```
model Requirement_32
    …
  input Boolean electricalFailure;
  input Boolean hydraulicFailure;
  input Angle outboardValue;
  input Angle inboardValue;
  input Integer mode; //mode as computed by SFCC

  Property status(start = Property.Undecided, fixed = true);

  parameter Real minDerivative = 0.01 "Value in rad/s";

  Boolean inboardMovement = abs(der(inboardValue))>= minDeriv
ative;
  Boolean outboardMovement = abs(der(outboardValue))>= minDer
ivative;

equation
  if (mode == 2 and electricalFailure) then //mode 2 = Inboard Differ
ential Flap Setting
    if (inboardMovement and not
                    (outboardMovement)) then
      status = Property.Satisfied;
    else
      status = Property.Violated;
    end if;
   else
    status = Property.Undecided;
  end if;
end Requirement_32;
```

Figure 11. Req 032 Modelica Model

In this formalization, a mode computed by the main control computer is used to check if a "inboard flap only movement" is commanded (mode 2). Other implementation could be possible but this one was chosen for its simplicity.

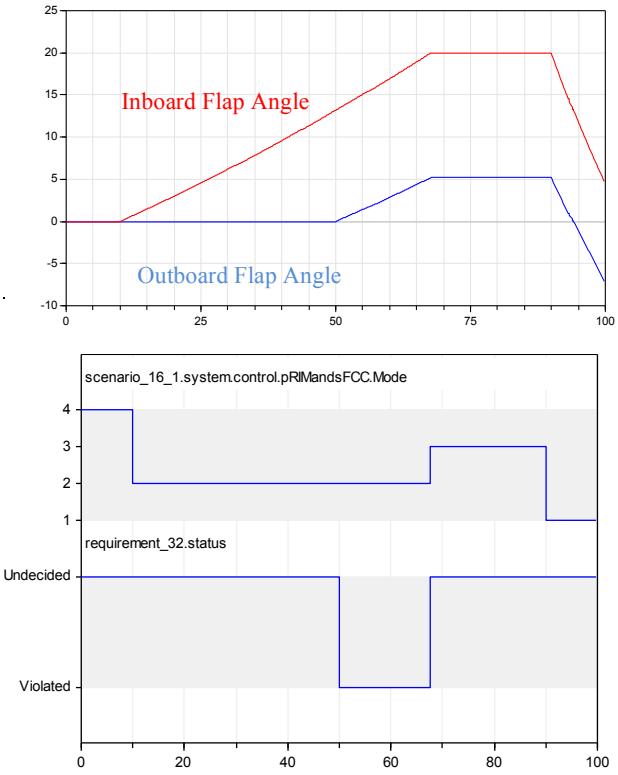


Figure 11. Req 032 test results

The last figure shows the results of a real scenario of system utilization. The model was excited with a pulse entry with 20 degrees of amplitude to move the inboard flaps, with no commands given to the outboard flaps. Also, during the simulation there are cases of an electrical failure distributed in a pulse form.

The requirement is violated during the simulation of the model since the outboard flaps continue to move during the electrical failures. This is due to an error in the model or in the system design and shall be investigated.

2.2 Verification Scenario Formalization

Scenarios are defined to stimulate the system in different conditions. These scenarios are defined as Modelica models providing inputs to the system model (flap commands, loads, failures…).

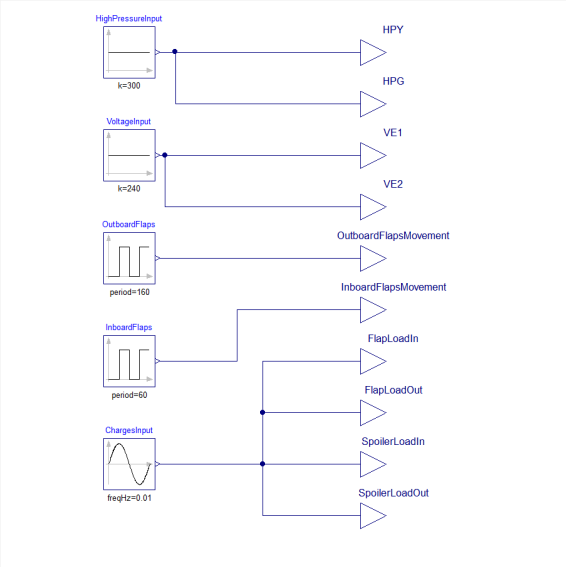


Figure 12. Modeling of a scenario

The scenarios must be defined so that requirements are verified (i.e. some scenarios permit the verification of the requirement). For this, a verification table can be created. This table defines which scenario should permit the verification of which requirements. After simulation of all scenarios, the value of the requirement will permit to check if the table is correct (i.e. to check if the scenarios have triggered the verification of the requirements.

	Sc01	Sc02	…	Sc15	Sc16
R001	X	X			
R002				X	X
…					
R032				X	X

Table 1. Verification table

3 Binding definition

This section describes the syntax for specifying the mediators and generating the bindings. In contrast to our previous proposals for representing bindings which relied either on either the use of an XML representation or on extensions to the Modelica language, the proposal presented in this paper is fully compliant with standard Modelica and relies on records to represent the binding information.

Clients, in this case requirements, require certain data. A record for representing the client, specifies the information necessary from the client side:

```
record Client "Client is a model or component that requires a modifi
er (i.e. a binding)"
  extends Modelica.Icons.Record;

  String id "A qualified name for the client";

  String template = "" "A transformation that can be applied to the
generated binding expression for this client.  If left empty, no transf
ormation will be applied.";

  Boolean isMandatory = false "Defines if the client must to be bo
und or if a binding is optional.";
 end Client;
```

A number of fields that are optional have predefined values, so that they do not need to be specified if not relevant for a specific binding.

Providers make data available to clients. The information specified by a provider is defined in the record below:

```
record Provider "Provider specifies how to access data required for
clients that are linked to the mediator this provider is used for."
  extends Modelica.Icons.Record;

  String id "A qualified name for the provider. ";
  String template = "" "Code snippet with placeholders used for ge
nerating part of binding expression.  If left empty, no transformation
will be applied.";

end Provider;
```

Clients and providers do not know each other a priori. In order to relate a set of clients and a set of providers, we use the mediators, defined by the record below:

```
record Mediator "Mediator captures data required for inferring bin
ding expression for referenced clients using referenced providers."
  extends Modelica.Icons.Record;

  String name = "" "Reflects what is needed by  referenced clients.
Optional.";
  String mType = "" "Reflects the type required by  referenced clei
nts. Optional.";

  String template = "" "A transformation that can include calls to fu
nctions that can handle unsorted arrays(e.g., add(:), max(:), toArray(
:), etc.). If left empty, no transformation will be applied.";

  Client clients[:] "List of clients.";
  Provider providers[:] "List of providers.";

end Mediator;
```

A more detailed description of the mediator concept can be found in *(Schamai, 2013)*.

3.1 Binding Specification

Section 2.1 shows examples of formalized requirements. The corresponding requirement models from require the following data:

- *Current distance between flap and its spoiler* (for R4.distanceBetweenFlapAndItsSpoiler and R5. distanceFlapSpoiler)
- *Current flap angle* (for R3.flapAngle)
- *Current force between flap and its spoiler* (for R6.forceBetweenFlapAndItsSpoiler)

DOI 10.3384/ecp15118911

- *Current torque of electrical motor* (R1.ADGBtorque)
- *Flap is moving* (for R4.isFlapMoving and R5.isFlapMoving)

The system model determines which of the requirements will be tested and how they will be combined with scenario models. Furthermore, there might be requirements which are repeatedly imposed on system parts of the same kind that exist inside the system model (e.g., there are several flaps in our model).

The purpose of the binding specification is to capture the minimum information in order to enable creating any combination of the system model and a set of requirements such that they will be bound correctly in an automated fashion, as well as to enable determining how many times a particular requirement needs to instantiated.

In order to do so, the user will now define *mediators* (Schamai et al, 2014). In our example the mediators reflect (i.e., contain information about) what data will need to be provided by the system model in order to enable testing of particular requirements (the clients).

Consider the mediator M1. It defines that any instance of the requirement models `R4.distanceBetweenFlapAndItsSpoiler` and `R5.distanceFlapSpoiler` (*clients*) have to be bound[2] to some other components in order to retrieve the value during simulating. The value can be accessed inside the instance of the type `Spoilers.Spoiler_SC.elastoGap` (*provider*) by using its sub-component `elastoGap.s_rel` (captured by the `template` attribute) whereby `getPath()` will be replaced by the instance path of the *provider* model. Other mediators are defined in a similar way.

```
 record M1
  import BindingDefinition.*;
  import Req.*;
  import SpoilerActuation_v7.*;

  extends Mediator(
    name = "Current distance between flap and its spoiler",
    mType = "Modelica.SIunits.Distance",
    clients = {
Client(id="R4.distanceBetweenFlapAndItsSpoiler",
isMandatory=true),
Client(id="R5.distanceFlapSpoiler",
isMandatory=true)},
    providers = {
Provider(id="Spoilers.Spoiler_SC.elastoGap",
template="getPath().elastoGap.s_rel")});
 end M1;

record M2
…
  extends Mediator(
    name="Current flap angle",
    mType="Modelica.SIunits.Angle",
    clients={Client(id="R3.flapAngle", isMandatory=true)},
    providers={Provider(id="Flaps.Flap.FlapAngle")});

end M2;

record M3
 …
  extends Mediator(
    name="Current force between flap and its spoiler",
    mType="Modelica.SIunits.Force",
    clients={Client(id="R6.forceBetweenFlatAndItsSpoiler",
isMandatory=true)},
    providers={Provider(id=" Spoilers.Spoiler_SC.elastoGap",
template="getPath().flange_a")});

end M3;

record M4
…
 extends Mediator(
  name="Current torque of electrical motor",
  mType="Modelica.SIunits.Torque",
  clients={Client(id="R1.ADGBtorque", isMandatory=true)},
  providers={
Provider(id="Flaps.ActuationChainComponents.MotorModel.flange
_b", template="getPath().tau")});

end M4;

record M5
…
 extends Mediator(
  name=" Flap is moving",
  mType="Boolean",
  clients={
   Client(id="R4.isFlapMoving", isMandatory=true),
   Client(id="R5.isFlapMoving",isMandatory=true)},
  providers={
   Provider(id="Control.SFCC.Mode", template="getPath() <> 4")
});
end M5;

record M6
…
  extends Mediator(
    name="Flaps actuation action is taking place",
    mType="Boolean",
    clients={Client(id="R2.isFlapsActuationAction",
isMandatory=true)},
    providers={Provider(id="Control.SFCC.Mode",
template="getPath() <> 4")});

end M6;
```

4 Binding generation

Once the bindings are specified a verification model can be created containing the system model and the requirements to be verified.

```
model VeM01
 import Req.*;
 import SpoilerActuation_v7.*;
System sm_system;
 R1 r1; R2 r2; R3 r3; R4 r4; R5 r5; R6 r6;
end VeM01;
```

[2] This is indicated by the attribute `isMandatory=true`

The system model imports the packages where the mediators that can be used in the binding computation are defined.

The OpenModelica API has been extended with a call: `inferBindings(systemModel, program);`

The call accepts as arguments the name of `systemModel` as well as the environment (here `program`) with all the loaded classes where it will look for the mediator definitions and update the `systemModel` with the binding expressions in the form of modifiers.

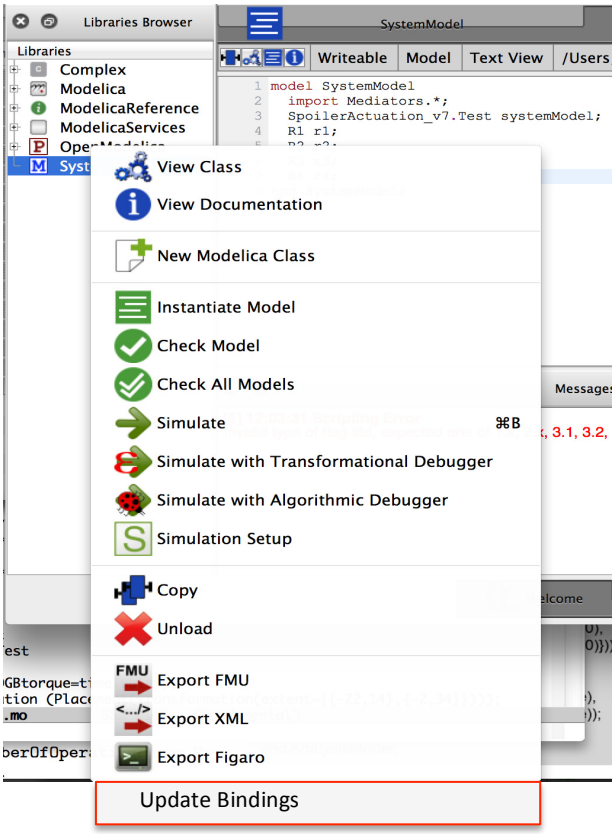


Figure 13 Binding generation in OpenModelica

The algorithm for binding generation is implemented in OpenModelica as it is defined in (Schamai et al, 2014). First an instance tree is built for the model to be bound (see Figure 14). This instance tree is represented in an internal data structure and all the clients and providers are identified by checking whether they match the client or provider paths defined in any mediators. For instance mediator M4 specifies only one client : Client(id="R1.ADGBtorque", isMandatory=true) and therefore ADGBtorque will be marked as a client in the instantiation tree. All the mediator data is also stored in an internal structure with references to all the instances of clients and providers found for each mediator.

Once all the internal structures are created, for each client the bindings are computed by localizing all the providers. If more than one provider is present then a template must be defined in the mediator to describe how the inputs from different clients must be combined. In mediator M4 we only have one provider defined:

Provider(id="Flaps.ActuationChainComponents.MotorModel.flange_b", template="getPath().tau")

As in the model we have two instances of MotorModel, right and left, two provider instances will be found by the algorithm.

The getPath() call is replaced with the path of the component instance in the environment, in this example sm_system.flaps.actuationChain.ADGB_Left.motorModel.flange_b and sm_system.flaps.actuationChain.ADGB_Right.motorModel .flange_b and the template is used to generate a binding expression, in this case to point to tau inside each of the providers.

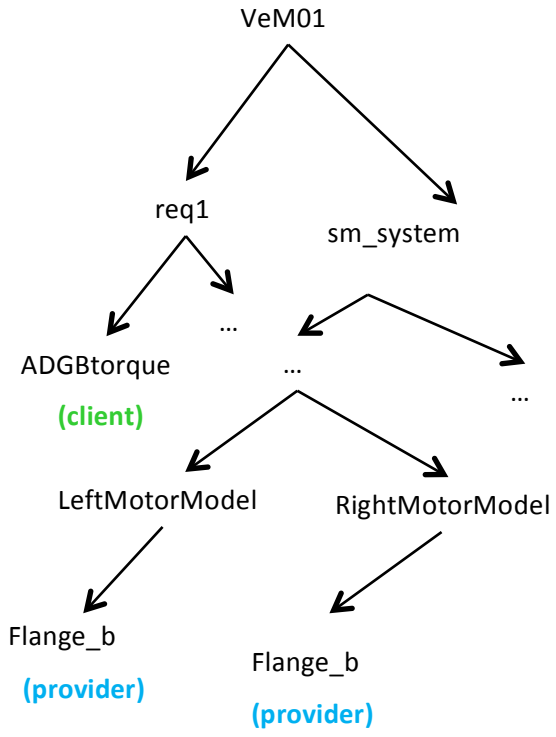


Figure 14 Instantiation tree

The algorithm figures out the required number of requirement instantiations and generates the binding expressions. For instance, in this example we need two instances of R1, one for each motor and as the model used in this use case has four flaps, four instances of R4 will be needed. Binding expressions are implemented as modifiers that will be applied to the components and sub-components of `systemModel`.

If the algorithm cannot find a binding for a mandatory client, or several binding are possible for the same client, the result will be an error message.

DOI 10.3384/ecp15118911

Similarly, when several instances of requirements that require more than one input need to be generated, user involvement may be needed to indicate how to correctly pair up the providers. In future versions of the implementation, support for storing and reusing user decisions will be implemented through the use of annotations.

For example, regarding the system model specified for this case study, the algorithm will generate the following binding expressions:

```
model VeM01
 import Req.*;
 import SpoilerActuation_v7.*;

System sm_system;
R1 req_001_0_r1(ADGBtorque = sm_system.flaps.actuationChain.
          ADGB_Left.motorModel.flange_b.tau);
R1 req_001_1_r1(ADGBtorque = sm_system.flaps.actuationChain.
          ADGB_Right.motorModel.flange_b.tau);
R2 req_002_r2(isFlapsActuationAction = sm_system.control
          .pRIMandsFCC.Mode <> 4);
R3 req_003_0_r3(flapAngle = sm_system.flaps.FlapLI.FlapAngle);
R3 req_003_1_r3(flapAngle
           = sm_system.flaps.FlapLO.FlapAngle);
R3 req_003_2_r3(flapAngle = sm_system.flaps.FlapRI.FlapAngle);
R3 req_003_3_r3(flapAngle
          = sm_system.flaps.FlapRO.FlapAngle);
R4 req_004_0_r4(distanceBetweenFlapAndItsSpoiler =
          sm_system.LISpoiler.elastoGap.elastoGap.s_rel,
          isFlapMoving =
          sm_system.control.pRIMandsFCC.Mode <> 4);
R4 req_004_1_r4(distanceBetweenFlapAndItsSpoiler =
          sm_system.LOSpoiler.elastoGap.elastoGap.s_rel,
          isFlapMoving =
          sm_system.control.pRIMandsFCC.Mode <> 4);
R4 req_004_2_r4(distanceBetweenFlapAndItsSpoiler =
           sm_system.RISpoiler.elastoGap.elastoGap.s_rel,
          isFlapMoving =
          sm_system.control.pRIMandsFCC.Mode <> 4);
R4 req_004_3_r4(distanceBetweenFlapAndItsSpoiler =
           sm_system.ROSpoiler.elastoGap.elastoGap.s_rel,
          isFlapMoving =
          sm_system.control.pRIMandsFCC.Mode <> 4);
R5 req_005_0_r5(distanceFlapSpoiler =
           sm_system.LISpoiler.elastoGap.elastoGap.s_rel,
          isFlapMoving =
          sm_system.control.pRIMandsFCC.Mode <> 4);
R5 req_005_1_r5(distanceFlapSpoiler =
          sm_system.LOSpoiler.elastoGap.elastoGap.s_rel,
          isFlapMoving =
          sm_system.control.pRIMandsFCC.Mode <> 4);
R5 req_005_2_r5(distanceFlapSpoiler =
          sm_system.RISpoiler.elastoGap.elastoGap.s_rel,
          isFlapMoving =
          sm_system.control.pRIMandsFCC.Mode <> 4);
R5 req_005_3_r5(distanceFlapSpoiler =
           sm_system.ROSpoiler.elastoGap.elastoGap.s_rel,
          isFlapMoving =
          sm_system.control.pRIMandsFCC.Mode <> 4);
end VeM01;
```

Once the bindings are defined, if we want to modify the system design, for instance adding backup components to the system or modifying the number of flaps, then the bindings can be regenerated with no additional effort.

Moreover, bindings can be used in batch testing to automatically generate verification models with different scenarios and different requirement subsets. This is something that would be difficult to do using explicit interfaces.

5 Conclusion

In this paper we have presented:

- A new application of design verification on an industrial case study in the field of aeronautics.
- The use of the new requirement modeling library for formalizing the requirements of the case study. We have shown that the binding approach is fully compatible with the new Modelica Requirements library.
- A modified version of the syntax for representing binding specification that is fully compliant with standard Modelica syntax, meaning that binding specifications can be edited and visualized in any Modelica tool. In order to support the binding generation, a tool has to simply implement the binding algorithm in *(Schamai, 2014)*.
- An implementation of the binding algorithm in OpenModelica

The binding approach does not assume prior knowledge of each other by the respective models and therefore increases decoupling and allows reuse of existing models and libraries. As mediators can be defined in several steps this means that different people can provide the information necessary to connect the models at different stages in the design process.

Moreover, the binding algorithm is general and can be used for binding models in other contexts than requirement verification. Furthermore, it enables a formal traceability between client and provider models. For example, determining which requirements are implemented in the system design model at hand can be achieved by looking at the bindings for mandatory requirement clients.

The case study is work in progress, but it has already allowed to detect a number of relevant issues in the model.

Clearly, the effectiveness of the presented approach is jeopardized when bindings are specified such that they result into too many ambiguous matches to be resolved by user manually. Such situation should be detected. Possible resolution could include: Providing hints for modifying the binding specifications; Enabling user to add more information to the binding specification for handling special cases; Or supporting the user by providing the list of all possible combinations to choose from. More complete results, for example, the evaluation of this approach on real projects with large number of requirements is still subject to future work.

Acknowledgements

This work is partially supported by the EU INTO-CPS project and the ITEA 2 MODRIO project via the Swedish Government (Vinnova) and the German and French Government.

References

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

Peter Fritzson. *Principles of Object Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. 1250 pages. ISBN 9781-118-859124, Wiley IEEE Press, 2014.

Hull, E., Jackson, K., and Dick, J. Requirements Engineering. Springer, 2005.

Kapurch, S. NASA Systems Engineering Handbook. DIANE Publishing Company, 2010. URL http://books.google.se/books?id=2CDrawe5AvEC.

Martin Otter, Lena Buffoni, Peter Fritzson, Martin Sjölund, Wladimir Schamai, Alfredo Garro, Andrea Tundis, Hilding Elmqvist. D2.1.1 – Modelica Extensions for Properties Modelling, Part IV: Modelica for Properties Modeling. Internal Report, ITEA2 MODRIO project, Sept. 2014.

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

Wladimir Schamai. *Model-Based Verification of Dynamic System Behavior against Requirements*. Ph.D. thesis, Method, Language, and Tool Linköping: Linköping University Electronic Press, Dissertations, 1547, 2013..

Wladimir Schamai, Lena Buffoni, and Peter 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.

H. Strüber The Aerodynamic Design of the A350 XWB-900 High Lift System. 29th Congress of the International COuncil of the Aeronautical Sciences. St Petersburg, 2014.