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

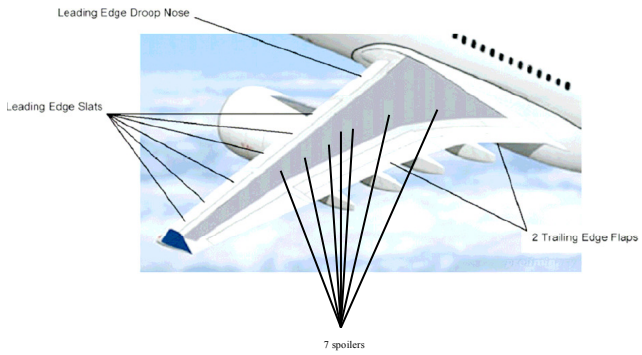

Figure 1. Flight Control System

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.

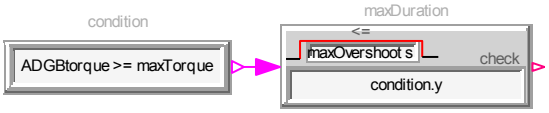

Figure 3. R1 Modelica model

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.

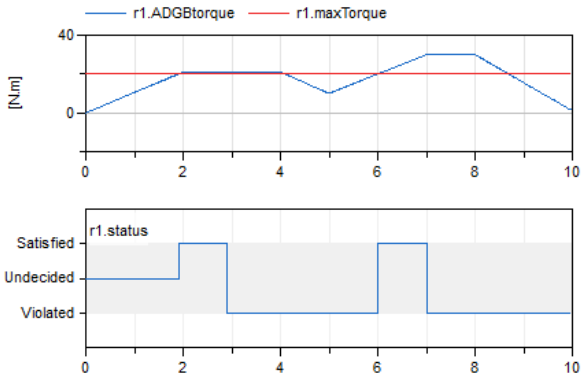

Figure 4. R1 test results

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;
```

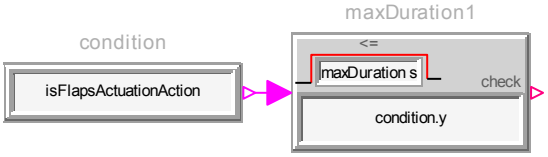

Figure 5. Req 002 Modelica model

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.

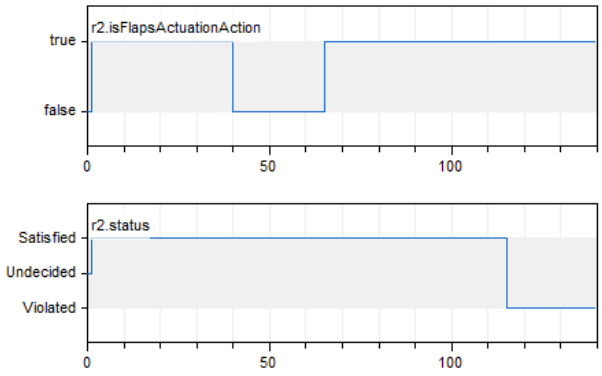

Figure 6. Req 002 test results

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.

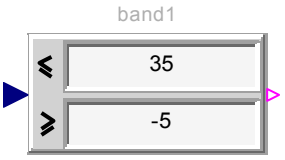

Figure 7. Req 003 Modelica model

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`.

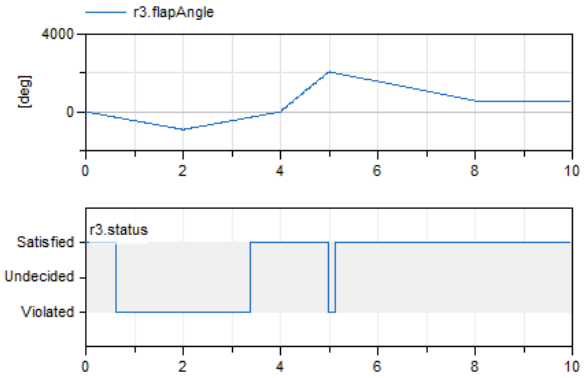

Figure 8. Req 003 test results

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

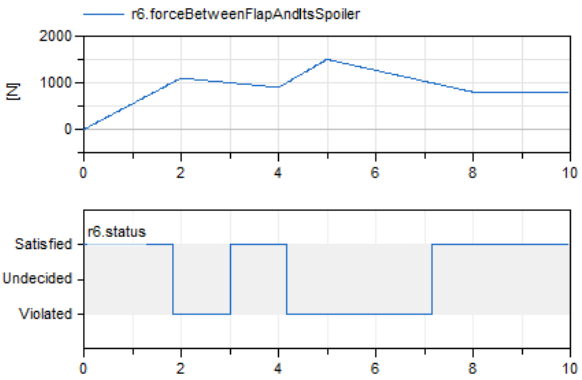

Figure 10. Req 006 test results

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.

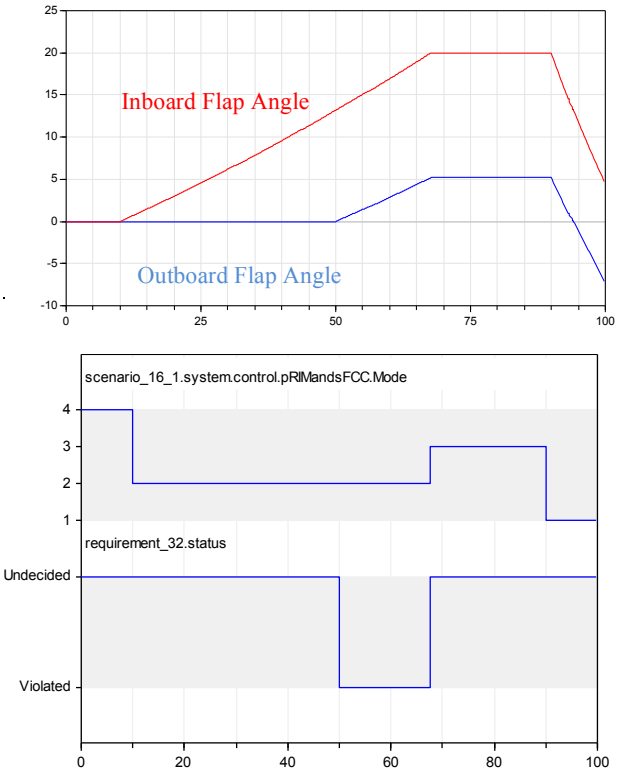

Figure 11. Req 032 test results

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…).

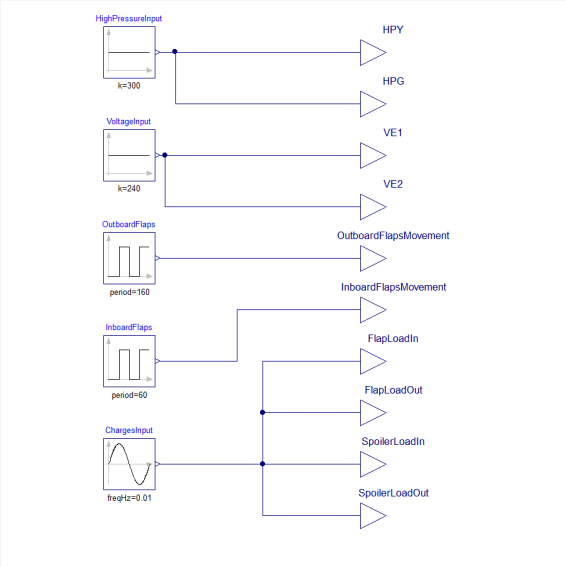

Figure 12. Modeling of a scenario

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.

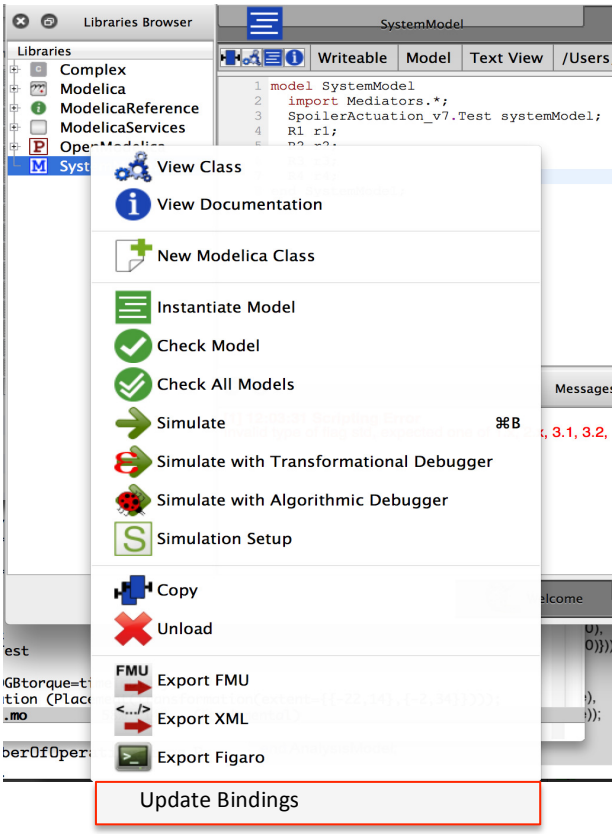

Figure 13 Binding generation in OpenModelica

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.

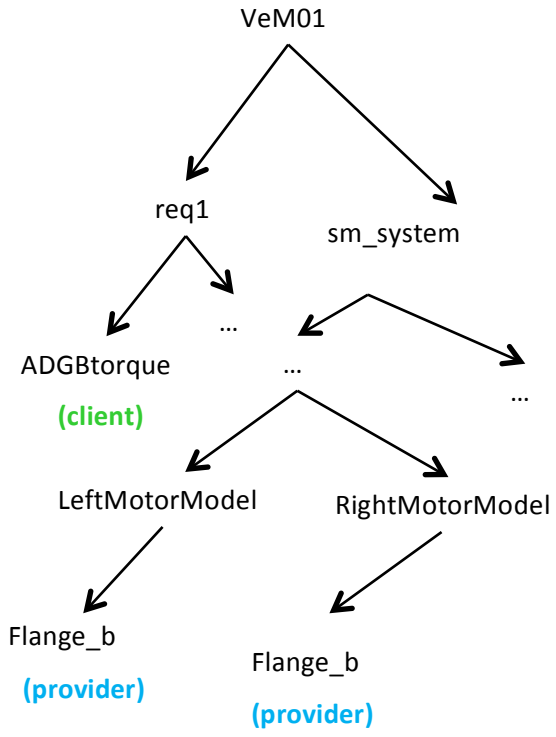

Figure 14 Instantiation tree

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.