Modelica Extensions for Requirement Modeling and their Implementation

Lena Buffoni (Linköping University),
Wladimir Schamai (EADS)

OpenModelica Workshop 2014
WHY DO WE WANT TO MODEL REQUIREMENTS?
Why are Requirements Important?

• Requirements Engineering is an important part of the early phases in system design V

• Errors in system requirements cause large cost increase in product development, or even completely failed projects

• Requirements engineering should be an integrated part of Modelica-based model-based system development tools

• Some resources:
  – International Requirements engineering conference home page: http://requirements-engineering.org/
Requirement Engineering Activities

- **Requirements inception** or requirements elicitation
- **Requirements identification** - identifying new requirements
- **Requirements analysis and negotiation** - checking requirements and resolving stakeholder conflicts
- **Requirements specification** (System Requirements Specification) - documenting the requirements in a requirements document
- **System modeling** - deriving models of the system
- **Requirements validation** - checking that the documented requirements and models are consistent and meet stakeholder needs
- **Requirements management** - managing changes to the requirements as the system is developed and put into use
Some Requirement Engineering Products

<table>
<thead>
<tr>
<th>Product</th>
<th>Link</th>
<th>Notes</th>
</tr>
</thead>
</table>
Some Requirements Engineering Literature

- Kotonya G. and Sommerville, I. Requirements Engineering: Processes and Techniques. Chichester, UK: John Wiley & Sons
- **Requirements bibliography** Reviewed November 10th 2011
HOW DO WE REPRESENT REQUIREMENTS?
Modeling requirements in Modelica

• We need a way of marking requirements
  – proposal introduce a new type of specialized class: `requirement`

• Alternatives: inherit from a generic class Requirement or use an annotation
Why are requirements different?

• Requirements must have a **status**
• Requirements do **not modify** the physical model
• For specific uses, we need to know which models are requirements, e.g.
  – automatic model composition,
  – requirement verification,
  – ...
• Requirement models can (possibly) contain extensions for expressing requirements in a more requirement-designer friendly way (FORM-L macros for example)
Status of a requirement

Example requirement: when the pump is on the flow must be above a minimum threshold

• The status variable is 3-valued and applies to a specific instant in time:
  – Violated: the requirement is applicable and the conditions of the requirement are not fulfilled – the pump is on and the flow is below the minimum required
  – Not_violated: the requirement is applicable and the conditions of the requirement are fulfilled – the pump is on and the flow is above the minimum required
  – Not_applicable: the requirement cannot be applied or is not relevant - the pumps are off
Status of a requirement (2)

• A set of functions for checking the status over the course of the system simulation
  – `has been evaluated()` – true if the requirement was applicable at some point in time
  – `has been violated()` – true is the requirement has been violated at least once
  – `can become applicable()` – true if the requirement can apply in the future (the time locator can be applicable)
Example 1 - LimitInFlow

requirement LimitInFlow "A2: If pump is on then in-flow is less than maxLevel"

//qOut from the Source component
input Real liquidFlow;
input Boolean pumpOn;
parameter Real maxLevel = 0.05;

equation
  if (pumpOn) then
    if (liquidFlow < maxLevel) then
      status = Status.not_violated;
    else
      status = Status.violated;
    end if;
  else
    status = Status.not_applicable; end if;
end LimitInFlow;
Example (2) : No pump shall cavitate

package Requirements

model Req

/* number of cavitating pumps*/
input Real numberOfCavPumps = 0;

/* min. number of operating pumps */
constant Integer maxNumOfCavPumps = 0;

/* indication of requirement violation, 0 = means not evaluated */
output Status status(start=status_not_applicable,fixed=true);

algorithm

if numberOfCavPumps > maxNumOfCavPumps then
    status := Status.violated;
else
    status := Status.not_violated;
end if;
end Req;
end Requirements;
FORM-L and Modelica

• FORM-L a language for property expression proposed by EDF

• Needed to represent FORM_L constructs in Modelica:

  – A 3-Valued type, e.g. Boolean3 (true, false, undefined)

  – A library of types to represent FORM-L concepts
    Ex:
    ```modelica
    type Condition = Boolean3;
    ```

  – A set of macros to represent FORM-L concepts, that can be expanded to standard Modelica
Example

Modelica:

// Form-L: during (On and (MPSVoltage > V170)) check no (Off becomes true);
requirement Req1_on;
input Real MPSVoltage;
input Boolean on;
input Boolean off;
equation
  if (on and (MPSVoltage > V170)) then
    if (off) then
      status = Status. violated;
    else
      status = Status.not_violated;
    end if;
  else
    status = Status.not_applicable;
  end if;
end Req1;
Example – Using if-expression

Modelica:

// Form-L: during (On and (MPSVoltage > V170)) check no (Off becomes true);
requirement Req1b_is_on;
input Real MPSVoltage;
input Boolean on;
input Boolean off;
equation
    status = if (on and (MPSVoltage > V170)) then
        if(off) Status.violated else Status.not_violated
    else Status.not_applicable;
end Req1bis;
Example(3) – Using Macro AFTER

// requirement R9a =
// during (SingleSensorFailure and after (Op.eVReset + s10)) check NormalPower becomes true;

requirement Req2
input Boolean SingleSensorFailure;
input Boolean eVReset;
input Boolean NormalPower;

equation
  if (AFTER(eVReset, 10) and SingleSensorFailure) then
    if (normalPower) then
      status = Status.not_violated;
    else status = Status.violated;
    end if;
    else status = Status.not_applicable;
  end if;
end Req2;
Example(4) – Standard Modelica

```modelica
// requirement R9a =
// during (SingleSensorFailure and after (Op.eVReset + s10)) check NormalPower becomes true;
model Req2
input Boolean SingleSensorFailure;
input Boolean eVReset;
input Boolean NormalPower;

Boolean wasReset(start=false);
integer tReset;

equation
    when wVReset
        wasReset = true; tReset = time;
    end when;
    if (wasRest and SingleSensorFailure and (tReset + 10 < time)) then
        if (normalPower) ... ?? What should be here? Why is there Op.eVreset in FORML-L and eVReset here? Where is s10? What does it mean??
    end Req2;
```

18
Issues

• Terminology:
  – Requirements? Properties? Assertions? ...

• Extensions
  – New embedded language? Nothing? Something in between?
USING REQUIREMENTS FOR VERIFICATION: A CASE STUDY
Requirement Verification vs System Design

- **Formalized Requirements** that should be verified

- **System Model, i.e., Design Alternative Model**, for which the requirements should be verified

- **Application scenarios** with the system model for which the requirements should be verified

- **Clients (requirement models, scenarios) refer to data from system model components**

- Clients and providers do not know each other a priori
- **Mediators** relate a number of clients to a number of providers
Example: System and Requirement

- A system contains several pumps,
- Requirement: “No pump within the system shall cavitate at any time”

System model:

```plaintext
model SRI
  Machines.PumpA PO1;
  Machines.PumpB PO2;
  Machines.PumpA PO3;
end SRI;

package Machines
  model PumpA
    // Volumetric mass flow-rate inside the pump
    Real Qv = 1;
    // Pressure at the inlet (C1 is the fluid connector at the inlet)
    Real C1_P = 1;
    // Pressure at the outlet (C2 is the fluid connector at the outlet)
    Real C2_P = 1;
  end PumpA;

  model PumpB
    // Volumetric mass flow-rate inside the pump
    Real Qv = 1;
    // Pressure at the inlet (C1 is the fluid connector at the inlet)
    Real C1_P = 1;
    // Pressure at the outlet (C2 is the fluid connector at the outlet)
    Real C2_P = 1;
    // Minimum pressure inside the pump
    Real Pmin = 20;
  end PumpB;
end Machines;
```

Formalized requirement:

```plaintext
package Requirements
  model Req
    /* number of cavitating pumps */
    input Real numberOfCavPumps = 0;
    /* min. number of operating pumps */
    constant Integer maxNumOfCavPumps = 0;
    /* indication of requirement violation, 0 = means not evaluated */
    output Integer status(start=0, fixed=true);

    algorithm
      if numberOfCavPumps > maxNumOfCavPumps then
        status := 2 "2 means violated";
      else
        status := 1 "1 means NOT violated";
      end if;
  end Req;
end Requirements;
```
Example: Aux. Functions from Lib.

package HFunctions

function H_is_cavitating
  /* Volumetric flow inside the pump */
  input Real vol_flow;
  /* Minimum pressure inside the pump */
  input Real Pmin;
  /* Table giving the minimum pressure inside the pump as a function of the volumetric flow inside the pump */
  input Real NPSH;
  /* Boolean stating whether the requirement for non-cavitation is satisfied (=false) or not (=true) */
  output Boolean is_cavitating;

algorithm
  is_cavitating := (Pmin < getNPSH(vol_flow));
end H_is_cavitating;

function getNPSH
  ...
end getNPSH;
end HFunctions;

For determining whether a pump cavitates, call a library function getNPSH(vol_flow)
Example: Analysis model

Formalized requirement:

```plaintext
package Requirements
model Req
  /* number of cavitating pumps*/
  input Real numberOfCavPumps = 0;
  /* max. number of operating pumps */
  constant Integer maxNumOfCavPumps = 0;
  /* indication of requirement violation, 0 = means not evaluated */
  output Integer status(start=0,fixed=true);
algorithm
  if numberOfCavPumps > maxNumOfCavPumps then
    status := 2 "2 means violated";
  else
    status := 1 "1 means NOT violated";
  end if;
end Req;
end Requirements;
```

System model:

```plaintext
model AnalysisModel
...
Requirements Req req1;
SRI sri;
end AnalysisModel;
```

```plaintext
model AnalysisModel
...
Requirements Req req1;
SRI sri;
end AnalysisModel;
```

```plaintext
model SRI
  Machines.PumpA.P01;
  Machines.PumpB.P02;
  Machines.PumpA.P03;
end SRI;

package Machines
model PumpA
  // Volumetric mass flow-rate inside the pump
  Real Qv = 1;
  // Pressure at the inlet (C1 is the fluid connector at the inlet)
  Real C1_P = 1;
  // Pressure at the outlet (C2 is the fluid connector at the outlet)
  Real C2_P = 1;
end PumpA;

model PumpB
  // Volumetric mass flow-rate inside the pump
  Real Qv = 1;
  // Pressure at the inlet (C1 is the fluid connector at the inlet)
  Real C1_P = 1;
  // Pressure at the outlet (C2 is the fluid connector at the outlet)
  Real C2_P = 1;
  // Minimum pressure inside the pump
  Real Pmin = 20;
end PumpB;
end Machines;
```
WHAT DO WE WANT TO ACHIEVE?
What do we want to achieve?

Initial AnalysisModel:

```python
model AnalysisModel
import Mediators.*;
Requirements.Req req1;
SRI sri;
end AnalysisModel;
```

Updated AnalysisModel model with a binding:

```python
model AnalysisModel
import Mediators.*;
Requirements.Req req1(numberOfCavPumps=
sum(
(if (HFunctions.H_is_cavitating(sri.PO1.Qv, sri.PO1.C1_P, 1))
    then 1 else 0),
(if (HFunctions.H_is_cavitating(sri.PO2.Qv, sri.PO2.Pmin, 1))
    then 1 else 0),
(if (HFunctions.H_is_cavitating(sri.PO3.Qv, sri.PO3.C1_P, 1))
    then 1 else 0))));

SRI sri;
end AnalysisModel;
```

Binding:

A **binding** is a causal relation which specifies that, at any simulated time, the value given to the referenced client instance shall be the same as the value computed by the right-hand expression.
WHAT DO WE NEED TO ACHIEVE THIS?
Bindings Concept: Basic Idea

- Some models require data: **Clients**
- Some models can provide require data: **Providers**
- **Clients and providers do not know each other a priori**!

- **Mediators** relate a number of clients to a number of providers
What do we Need to Capture?

- **Mediator** specifies *which models are clients* and exposes what *information* is needed
- **Providers** are used to infer the *binding expression for clients*

In our example:
- A mediator will be used to infer the binding expression that calculates *the number of caviating pumps* in a particular system design model.

Computing strategy:
- Each pump model shall return 1 if it cavitates and 0 otherwise
- Mediator will sum up the values from all the pumps
How do we capture this information?

Example using Modelica Syntax

```modelica
package PartialMediators

partial mediator NumberOfCaviatingPumps_C
    /* Number of cavitating pumps. This mediator is incomplete because no provider are defined yet. */
    type Real;
    clients mandatory Requirements.Req.numberOfCavPumps;
end NumberOfCaviatingPumps_C;
end PartialMediators;

package Mediators

mediator NumberOfCaviatingPumps_P1 extends PartialMediators.NumberOfCaviatingPumps_C;

/* reduces the array of provided values to a single value */
template sum(:) end template;

/* list of providers used to compute the number of all cavitating pumps */
providers
    /* reference to the provider model (i.e., its qualified name) */
    Machines.PumpA
        template
            if HFunctions.H_is_caviating(getPath().Qv, getPath().C1_P, getNPSHTable(A))
                then 1 else 0
            end template;

    /* getPath() is a placeholder that will be replaced with the instance path of the pump model */
    Machines.PumpB
        template
            if HFunctions.H_is_caviating(getPath().Qv, getPath().Pmin, getNPSHTable(B))
                then 1 else 0
            end template;

end NumberOfCaviatingPumps_P1;
end Mediators;
```
What do we need to capture?

Abstract Syntax View

- **Mediator name** (with optional comments) reflects what is needed by clients
- **Mediator type** must be compatible to each of its clients (for Modelica also the lowest variability of clients should be indicated)
- **Client or provider id** is the qualified name of the client or provider model (e.g. `Package1.Model1.component1`)
- **isMandatory** (true by default) indicates whether the client must be bound. If not, the client component must have a default value.
- All value templates are optional (template, preliminary name, an expression that returns a value)
- **Client or provider template** contains expressions that can contain instance paths (e.g. in Modelica using the dot-notation) for referencing components within the client or provider models
- **Mediator template** can only contain predefined macros (e.g. `sum(:), toArray(:), card(:), min(:), max(:), etc.`)
HOW TO ACHIEVE OUR GOAL?
Generated Binding Expression

**Mediator** that contains **client** references:

```plaintext
partial mediator NumberOfCaviatingPumps_C
  type Real
  clients
  isMandatory Requirements.Req.numberOfCavPumps;
```

**Mediator** that also contains **provider** references:

```plaintext
mediator NumberOfCaviatingPumps_P1
  extends NumberOfCaviatingPumps_C;
  template sum(:) end template;
  providers
    Machines.PumpA
    template
      if HFunctions.H_is_cavitating(getPath().Qv, getPath().C1_P, getNPSHTable(A)) then 1 else 0
    end template;
```

Model with generated binding for client req1.numberOfCavPumps:

```plaintext
model AnalysisModel
  ...
  Requirements.Req req1(numberOfCavPumps = sum({
    (if (HFunctions.H_is_cavitating(sri.P01.Qv, sri.P01.C1_P, 1)) then 1 else 0),
    (if (HFunctions.H_is_cavitating(sri.P02.Qv, sri.P02.Pmin, 1)) then 1 else 0),
    (if (HFunctions.H_is_cavitating(sri.P03.Qv, sri.P03.C1_P, 1)) then 1 else 0))});
```

**Note**, `getPath()` is replaced by provider model instance path within the given context.
Generating Bindings Expression

**Instantiation Tree**

Inferred binding expression: \( \text{numberOfCavPumps} = \sum\{(\text{if } \text{HFunctions.H_is_cavitating}(\text{sri.PO1.Qv, sri.PO1.C1_P, 1)) \text{ then } 1 \text{ else } 0), \text{if } \text{HFunctions.H_is_cavitating}(\text{sri.PO2.Qv, sri.PO2.Pmin, 1)) \text{ then } 1 \text{ else } 0), \text{if } \text{HFunctions.H_is_cavitating}(\text{sri.PO3.Qv, sri.PO1.C1_P, 1)) \text{ then } 1 \text{ else } 0))\}

To be stored as modified in AnalysisMode1: req1 (numberOfCavPumps = ...)

Client model qualified name: Requirements.Req.numberOfCavPumps (used to identify it as client based on mediator references)
Client instance path in AnalysisModel1: req1.numberOfCavPumps
Mediator (used to inferring the binding expression): Mediators.NumberOfCaviatingPumps_P1
Mediator operation (used to inferring the binding expression): sum(·)
Providers (used to inferring bindings expression): sri.PO1, sri.PO2, sri.PO3
USING BINDINGS FOR VERIFICATION
MODEL COMPOSITION
Composing Verification Models

main idea

• Collect all **scenarios**, **requirements**, import **mediators**
• Generate/compose **verification models** automatically:
  – Select the **system model** to be verified
  – Find all **scenarios** that can stimulate the selected system model
    (i.e., for each mandatory client check whether the binding expression can be inferred)
  – Find **requirements** that are implemented in the selected system model
    (i.e., **check** whether for **each requirement** for all mandatory clients binding expressions can be inferred)

• Present the list of scenarios and requirements to the user
  – The user can select only a subset or scenarios or requirements he/she wishes to consider
Generating/Composing Verification Models

algorithm

Select Design Alternative

* Abort if no scenario

Analyze (Next) Scenario

More scenarios

Select Scenario

* Next scenario if no requirement

Clients of design alternative satisfied by providers from scenario?

Analyze (Next) Requirement

More req.

Select Requirement

More requirements

Create Combination (i.e. Create Verification Model)

All requirement clients satisfied?

Combinations

DAM

SM

RUM

VM

else

else

...
Issues

• Best representation:
  – Modelica Syntax? XML? Annotations?