D2.1.1 – Modelica extensions for properties modelling

Part II: Modeling Architecture for the Verification of Requirements

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

MODRIO (11004)

Version 1.5
Date 01/07/2015

Authors

Daniel Bouskela	EDF
Thuy Nguyen	EDF
Audrey Jardin	EDF

Executive Summary

Verifying that the design and operation of complex physical systems such as power plants satisfy the specified requirements may be quite difficult, due to the large number of requirements and test scenarios that must be considered to have satisfactory test coverage. The systems considered here exhibit a large number of dynamic continuous and discrete states representing respectively physical states and logical modes such as normal, dysfunctional and control system modes. A tool-independent modular modeling architecture is presented to automate as much as possible the testing phase for the verification of such systems.

Four types of models are considered: (1) requirements models to express formally the requirements, (2) architectural models to express the system design, (3) behavioral models to express the system dynamic behavior, and (4) observation operators to observe dynamic variables from the behavioral model. Bindings are used to connect together models (1) to (4) without modifying them in order to produce automatically the full verification model.

The general principle governing the method is that the whole system encompassing the physical process and its control system should be considered, that requirements should be issued independently from design choices, and verified against several possible design architectures. Testing is performed by simulating the behavioral model together with the requirements model used as an automatic observer to detect possible violations in the requirements. Emphasis is put on the original aspects of the method, in particular the use of quantifiers to express generic requirements independently from design choices, the use of observation operators to allow the independent production of the requirements and behavioral models, and the use of 3-valued logic in order to account for the fact that it cannot always be decided using simulation whether requirements are satisfied or not.

The methodological principles governing the different types of models are presented using the intermediate cooling system of a nuclear power plant as an example. The approach developed here is to use the proposal for an extension of the Modelica language to express properties in a textual form named FORM-L.

Then a formal approach is used that is independent of the syntaxes of both FORM-L and Modelica languages to explain how FORM-L complements Modelica.

As a conclusion, two solutions are possible regarding the implementation of FORM-L: FORM-L as an extension of Modelica or as a separate language from Modelica. The first solution has been chosen at the MODRIO WP2 workshop in Munich held on June 13, 2013.

This work is a revised an extended version of an original paper published in the proceedings of the ISA POWID 2014 symposium entitled ‘Novel Open Source Modelling Architecture for the Design Verification against System Requirements using Simulation’.

Summary

AVERTISSEMENT / CAUTION ..... Erreur ! Signet non défini.
Executive Summary ..... Erreur ! Signet non défini.
Summary **3**
1. Introduction **4**
2. Illustrative example: the cooling system of auxiliary equipment of a PWR nuclear power plant **5**
3. Modeling architecture **10**
3.1. Property models 10
3.1.1. Properties, requirements and assumptions *10*
3.1.2. Structure of a requirement *11*
3.1.3. Requirements and architectural models *12*
3.1.4. Example: writing the property model for (Req. 2) *12*
3.2. Behavioral models 14
3.3. Observation operators 16
3.4. Bindings 18
3.4.1. Purpose of bindings *18*
3.4.2. Set binding *18*
3.4.3. Class binding *19*
3.4.4. Instance binding *21*
3.4.5. Algorithm for forming y = H(x) expressions *23*
3.5. Verification models 24
4. How FORM-L complements Modelica **28**
4.1. Introduction 28
4.2. FORM-L expresses constraints 29
4.3. FORM-L uses time locators 30
4.4. FORM-L uses spatial locators 30
4.5. FORM-L is linked to Modelica using bindings 32
4.6. FORM-L uses 3-valued logic 33
5. Conclusion and future work **36**
References **36**

1. Introduction

To ensure the proper operation of complex physical systems such as power plants, requirements are issued all along the system's lifecycle, starting from the system basic design phase until the system's operation phase. The requirements capture the spatiotemporal and quality of service conditions that a system should fulfill. They may be quite complex and numerous. Testing the compliance of the system with the requirements may be quite challenging, due to the many items that should be examined and verified for a given test scenario, and the number of test scenarios to be considered to have a satisfying verification coverage. The purpose of this work is to present a modular modeling architecture to verify compliance to requirements by automating as much as possible the testing phase. Automating the testing phase should help the test engineer to overcome the potential combinatorial explosion of operating modes to be explored. However, in this document, only the generation of the verification model is considered. The generation of complete test scenarios from requirements is not discussed.

The main idea is to build the requirements model on the one hand, and a behavioral model that captures the dynamics of the system for all situations to be tested on the other. However, the two kinds of models are very different in purpose, nature and expertise. Requirements are expressed by modeling the properties that should be fulfilled by the system from a functional perspective. They do not bear on fully identified individual objects, but rather on collections of objects characterized by the desired properties. Generic expression on collections (sets) of objects may be issued using *quantifiers*. On the other hand, the system behavior is expressed by using physical and control laws for each instance of the objects that are considered in the design of the system to fulfill the requirements. Those two types of models, which so to speak come from different horizons, should be linked together without having actually to modify neither of them. In the following, it will be shown how bridging the physical world (represented in behavioral models) to the functional world (represented in property models) may be achieved by using so-called *observation operators*. Also, it is shown why *3-valued logic* is necessary to express requirements in order to account for the fact that it cannot always be decided using simulation whether requirements are satisfied or not. The use of *quantifiers (e.g. universal or existential quantifiers)*, *observation operators, bindings* and *3-valued logic* constitute the main innovative features of the modeling architecture presented in this work.

To be tool-independent, the architecture should be based as much as possible on open source technology. The equations of the behavioral models may be rigorously written using the equation-based Modelica language for physical system modeling [1]. Requirements may be expressed using SysML [2]. SysML, as a UML profile, is used by the systems engineering community to express requirements using graphical notations. These models are not formal and may not be simulated as such. For use in models based on UML, the Object Constraint Language (OCL) [3] was designed to produce formal specifications for distributed software systems. AADL (Architecture Analysis & Design Language) supports abstractions related to software (thread, process…) or hardware (processor, memory, device…) architectures, so this language mainly used in the avionics and aerospace industry is especially effective for model-based analysis and specification of complex real-time embedded systems [4], [5]. Therefore neither OCL nor AADL address the operation of physical systems as such. For this purpose, an attempt was made to bridge the gap between SysML and Modelica in the form of a prototype called ModelicaML that generates Modelica code from ModelicaML models [6]. However, experiments conducted at EDF R&D in the framework of the ITEA 2 OPENPROD project showed that ModelicaML cannot express precise time constraints and more generally cannot express requirements in a way that is independent from design considerations [7]. The formalization of requirements with time constraints is still an active domain of research (see e.g. [8]). The syntax of temporal logic such as LTL (Linear Temporal Logic) or CTL (Computation Tree Logic) is rich for expressing temporal constraints, but their mathematical notations are unclear to most operation engineers.

In order to overcome the difficulties mentioned above, the approach developed in this work is to use the proposal for a new language named FORM-L to express properties [9][10]. The objective of FORM-L is to combine both the semantic rigor needed for automatic processing and the language expressiveness to be understandable by operation engineers. So in the following, all model examples will use the new language FORM-L for property modeling, and the existing language Modelica for

physical and control system modeling.

This work is an extension of an original paper published in the proceedings of the ISA POWID 2014 symposium [11].

2. Illustrative example: the cooling system of auxiliary equipment of a PWR nuclear power plant

The objective of the cooling system is to remove heat from large pieces of equipment of a PWR nuclear power plant. To ensure the proper functioning of the system, requirements are issued that involve the operating conditions of the plant. They may be issued at different stages of the system design, depending on the details of the system architecture involved in the requirements.

Let us take some examples of requirements for the cooling system chronologically sorted according to the advancement of the design process. These requirements may be informally expressed as such:

- Req. 1: 'The maximum heating power to be removed is 25 MW'.
- Req. 2: 'When they are in operation, pumps should not cavitate'.
- Req. 3: 'The water temperature must not vary more than 10°C per hour'.
- Req. 4: 'When the system is in operation, there should be no less than two pumps in operation for more than 2 seconds and the violation of this property should not occur more than 3 times per year with a confidence rate of [a given percentage]".
- Req. 5: 'The mass flow rate in the loop should be quasi-constant'.
- Req. 6: 'A pump must not start more than 3 times per hour'.
- Req. 7: 'When the system is in operation, the water level inside the tank should be comprised between two [given altitudes] above the level of the pumps'.
- Req. 8: 'The temperature at the outlet of the heat exchangers should be close to 17°C within a ±2°C margin with a confidence rate of [a given percentage]'.

The architectural model in Figure 1 shows a possible implementation (or realization) of the requirements. Blocks tagged bc (bc: 'boundary condition') correspond to interfaces with the environment (bc1 to bc4: the cold source) or with other plant systems (bc5: the feedwater supply, bc6 and bc7: the pieces of equipment to be cooled). The number of heat exchangers and pumps result from technological limits: two pumps and two heat exchangers in operation are needed for the amount of energy to be evacuated as stated in (Req. 1), considering the temperature of the cold source. The bypass is needed to satisfy (Req. 5).

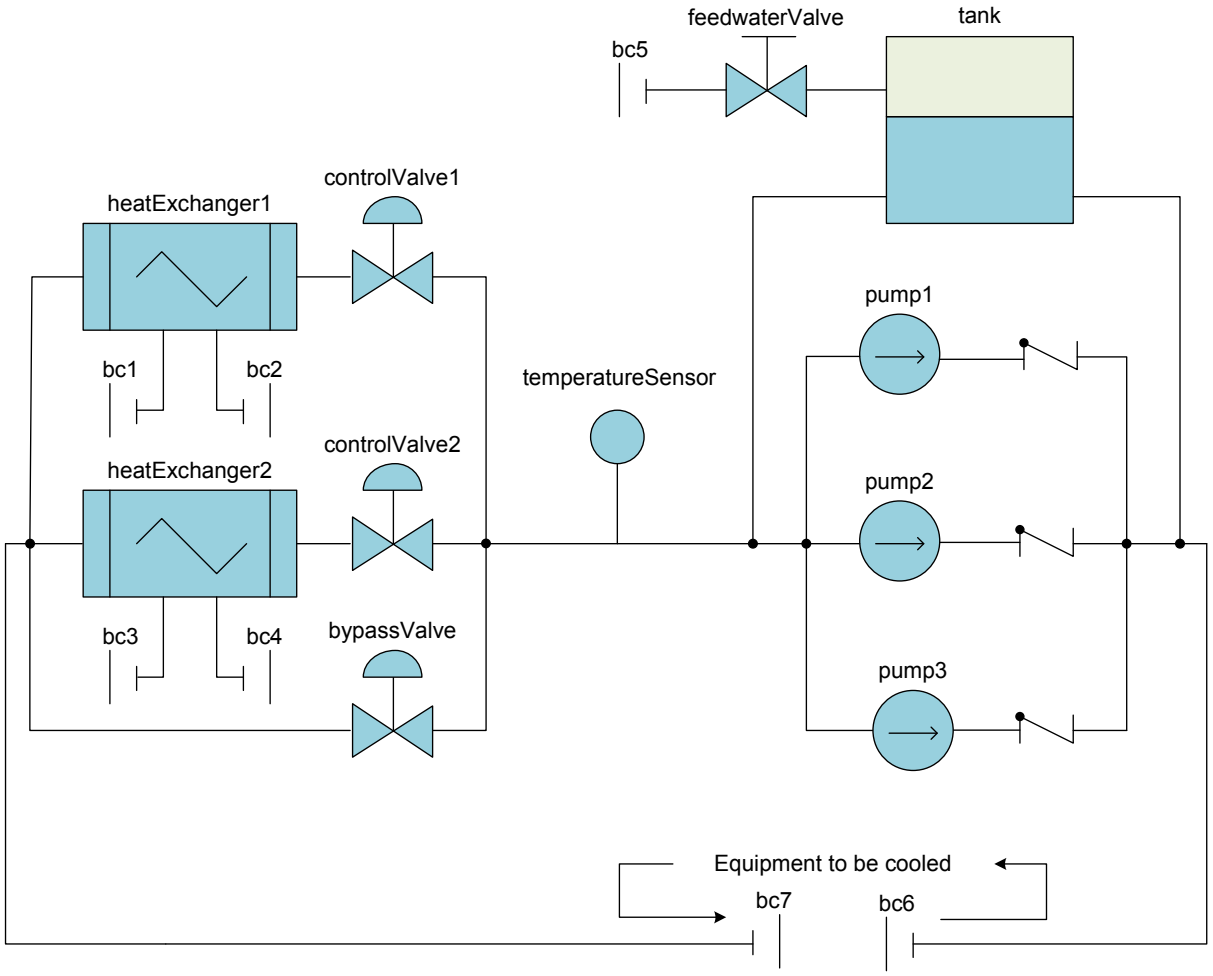


Figure 1: graphical representation of the architectural model of the cooling system

This example shows that requirements involve the functions of the control system at a very early stage of the system design, so that in essence it is not possible to separate out the physical aspects from the control aspects of the system. Therefore the verification of the system against the requirements should involve both the physical and the control parts of the system.

In order to be able to use simulation for the verification of the system, enough information should be provided on the architecture of the system to be able to build a behavioral model of the system. A behavioral model is a mathematical model that predicts the dynamic behavior of the system, given as input the test scenarios through the boundary conditions and the parameters of the model. For instance, in order to verify (Req. 2), the required NPSH (Net Positive Suction Head) curves of the pumps should be known (cf. Figure 2).

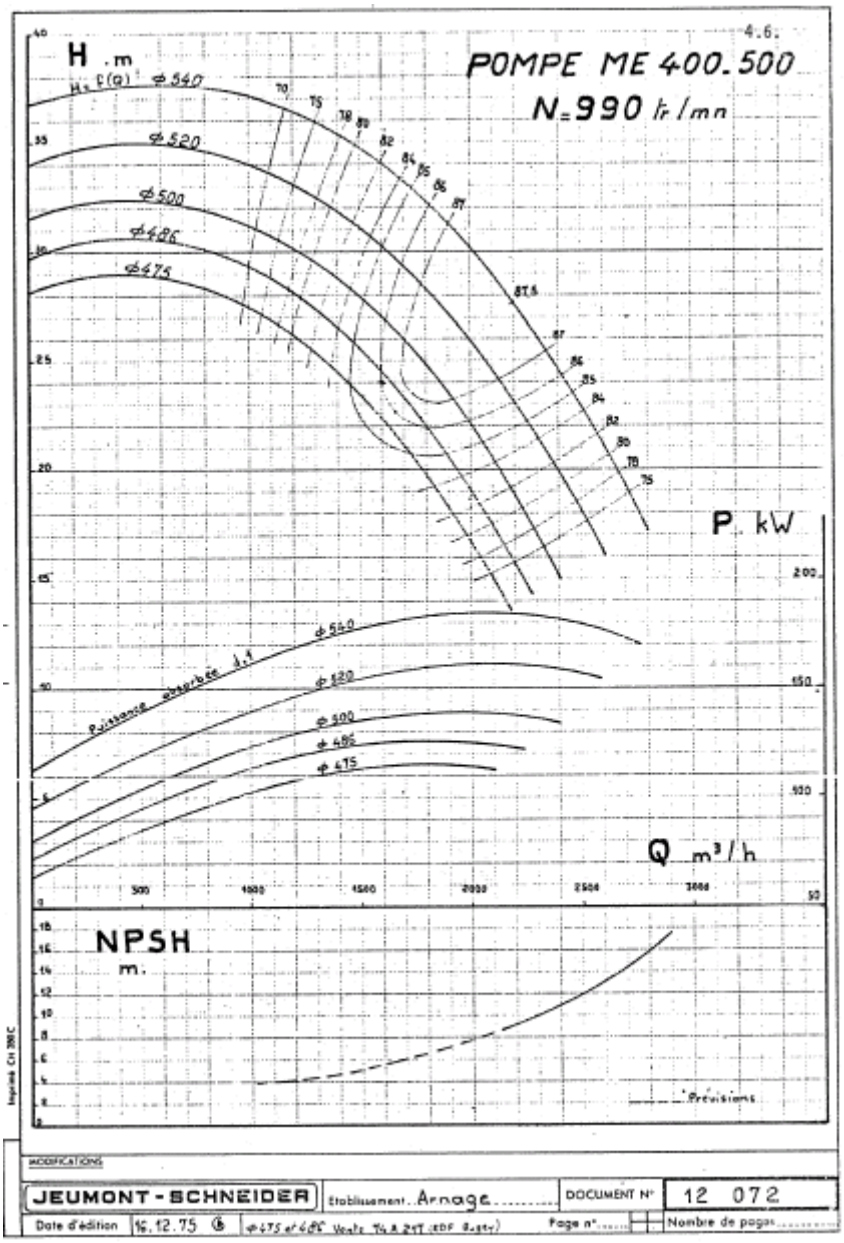
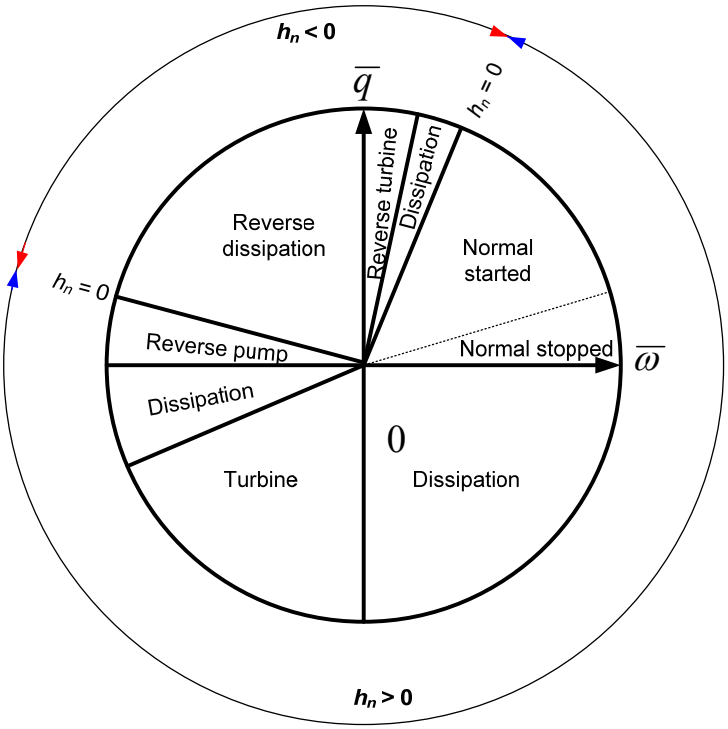


Figure 2: characteristic curves for a centrifugal pump

The possible operation modes for a centrifugal pump are shown in Figure 3 [12].



h_n : head
$\overline{q}$: reduced volumetric flow rate
$\overline{\omega}$: reduced angular velocity

Figure 3: operation modes for a centrifugal pump

Figure 2 shows that the curve giving the required NPSH is only defined when the pump is in the 'normal started' zone with the volumetric flow rate above the threshold given by the dotted line in Figure 3. So the question 'Is the pump cavitating?' can only be answered by *true* or *false* if the pump is in the 'normal started' zone, that is if the pump is started and operating normally. Otherwise, the question cannot be answered.

A possible behavioral model for the cooling system is given in Figure 4.

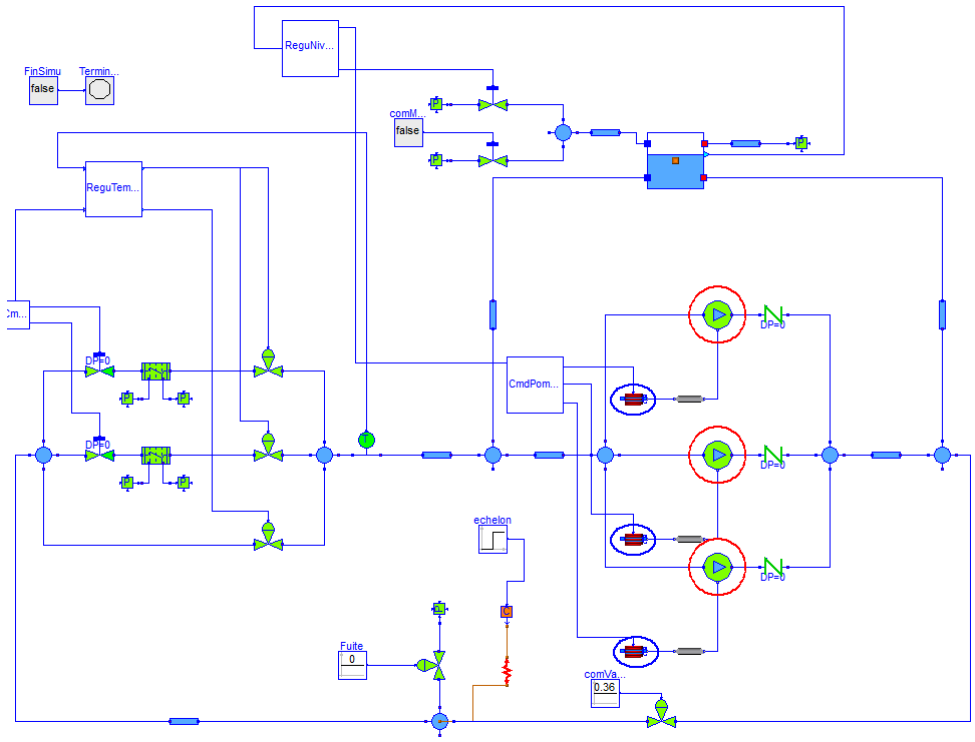


Figure 4: behavioral model of the cooling system built with ThermoSysPro [13]

The behavioral model is built by assembling model components that contain the equations describing the dynamic behavior of the components. The model components must be connected according to specific rules that ensure that the behavioral model is mathematically and physically well formulated. The consequence is that, although it looks like the architectural model, the behavioral model is somewhat more detailed. For instance, pipes and junctions are represented as components in the behavioral model, whereas they are only represented by connecting lines in the architectural model. The pumps are represented in the behavioral model by 3 components: (1) the motor of the pump (inside the blue ellipses), (2) the shaft of the pump, and (3) the hydraulic part of the pump (inside the red circles), whereas in the architectural model, each pump is only represented by one symbol. The control system is represented in the behavioral model, but not in the architectural model. Also the influence on the cooling system of the equipment to be cooled is represented in the behavioral model as a heat source and a valve that respectively model the amount of heat released by the equipment and the pressure losses through the equipment.

The requirements, architectural and behavioral models may have been developed independently from each other, as they require different expertise, serve different purposes and have different lifecycles:

- The requirements model is elaborated by the experts in systems operation. Its purpose is to capture the system's requirements from the operator's point of view. It is modified when operating constraints evolve such as safety or environmental regulations.
- The architectural models are elaborated by the system's designers with mechanical and I&C engineering expertise. Their purpose is to capture the system's possible designs from the manufacturer's point of view. Several architectural models may be proposed to fulfill a given requirements model using different technologies.
- The behavioral models are elaborated by modeling & simulation experts in the field of power plants (with expertise in thermal-hydraulics, neutronics, combustion, control, etc.) in order to predict the physical behavior of the system. Several behavioral models may be used for a given architectural model, with more or less details, depending on the requirements to be verified from the requirements model.

The outline of the modeling architecture is given in Figure 5. The objective is to verify the design against requirements. There are several possible designs for a given set of requirements. There are also several possible behavioral models for a given design. Behavioral models should also be verified against the design considered for verification against requirements. The way to do this is not discussed in this document. The fundamental principle of this architecture is to use the requirement model as an observer of the behavioral model to automatically detect possible violations in the requirements due to design errors or flaws in the requirements.

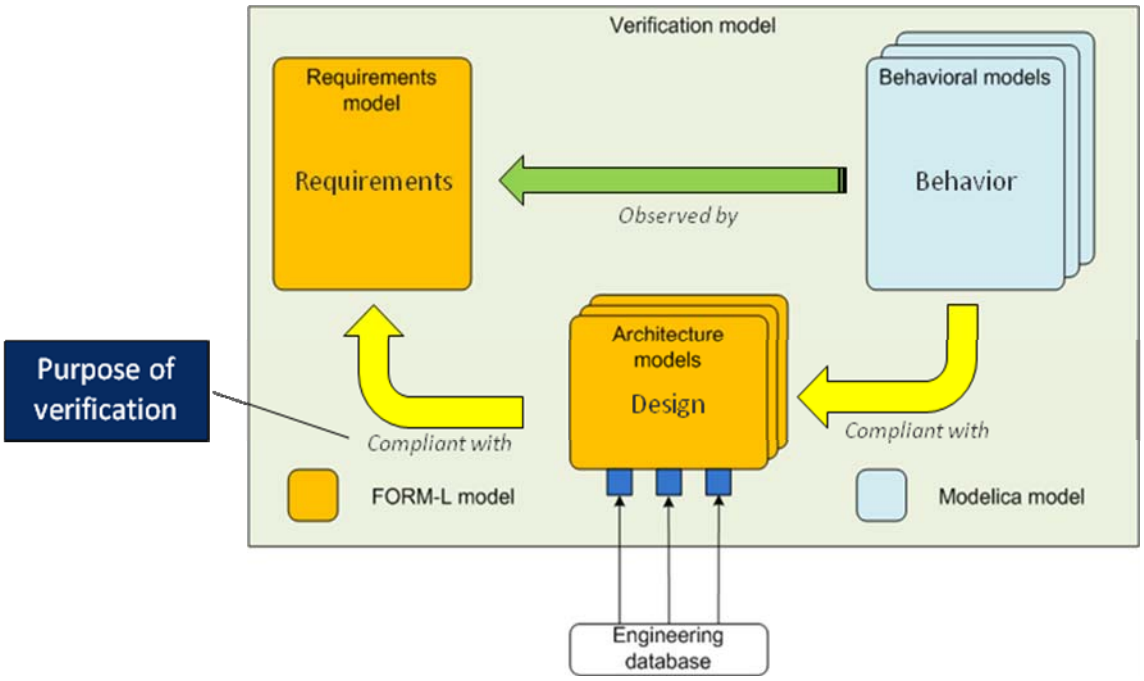


Figure 5: outline of the modeling architecture for the verification of requirements

The problem is now to bind the different models together in order to produce the complete verification model: the behavioral model computes the system's transients from given scenarios as inputs, the architectural model provides the data needed for the computations (such as pumps characteristics) and the requirements model acts as an observer to decide whether requirements are violated or not in the course of the transients. It is important to note that these different kinds of models are in general developed by independent teams, but they must be coupled together in various ways to produce the complete verification models. Hence, the binding should be done by the test engineer in such a way that no modifications are needed on any of the models for the purpose of binding, so that modifications would only be needed if for instance some features were missing in the behavioral model to verify a particular requirement.

Chapter 3 shows how the different types of models are constructed and how the binding is performed in order to produce a so-called *verification model*, which is the complete model for automatic testing.

3. Modeling architecture

3.1. Property models

3.1.1. Properties, requirements and assumptions

A property is a temporal-logic assertion that may be true or false, or undefined when it is not possible to assert whether it is true or false. There are two particular kinds of properties: requirements and assumptions.

- A requirement (or required property) is a desirable property that must be satisfied, hence that should always be true, but that may be violated if the system does not work properly (because of a design error, a component failure…). Requirements express the constraints that the system should fulfill. They do not define any precise system behavior: they just set the limits for authorized system behavior.
- An assumption (or assumed property) is a property that is assumed as satisfied, hence that cannot be violated by definition. Hypotheses constitute the boundary conditions of a requirement model and mostly concern the environment of the system under study. Example of an assumed property: “The temperature of the cold source is between 4°C and 35°C”. All tests will be performed within this temperature range of the cold source.

3.1.2. Structure of a requirement

Requirements are decomposed into:

- Spatial locators (WHERE): they define the sets of objects that are subject to the requirements. The word ‘spatial’ is used here in a broader sense than pure geometrical location as they select objects that meet any kind of criteria. To that end, they are constructed using conditions on attributes of objects in the system.
- Time locators (WHEN): they define the time periods or the instants when the requirement should be fulfilled. To that end, they are constructed using conditions on dynamic attributes of objects in the system. There are two kinds of time locators:
  - Discrete time locators (DTL): they define sets of events. An event defines occurrences of a fact that has no duration.
  - Continuous time locators (CTL[1]): they define sets of continuous time periods. A continuous time period is bounded by two events: the start and the end of the continuous time period.
- Condition to be fulfilled (WHAT): it is the condition to be fulfilled by the objects selected with the spatial locators during the time periods selected with the time locators. It can be one of three types: a Boolean condition that must be true all along the time locator, a constraint on how long a Boolean condition is true during the time locator, or a constraint on the number of occurrences of an event during the time locator. For a given instant, the condition evaluates to ‘true’ or ‘false’ if the instant belongs to the selected time periods. It evaluates to ‘undefined’ otherwise.
- Probabilistic constraint (HOW_WELL): defines a probabilistic constraint on the desirable property (i.e. the three parts WHERE, WHEN and WHAT).

For a given simulation run, a requirement may evaluate to 3 values: satisfied, violated or undecided. The requirement is undecided if the time locator has never been satisfied during the run (i.e. the requirement has not been challenged). It is violated when it has been challenged and the condition evaluated to ‘false’ at least once during the simulation run. It is satisfied when it has been challenged and the condition has never evaluated to ‘false’ during the simulation run.

Example of a requirement: “When the system is in operation, any pump located in room A should not fail more than twice a year with a probability of 99.9%”. This example uses a spatial locator to select a set of pumps (all pumps located in room A) and a time locator in the form of a CTL (a time period of one year sliding all along the time period when the system in operation). The condition states a constraint on the number of occurrences of an event (no more than two failures) and a probabilistic constraint (Pr > 99.9%).

In the example above, the spatial locator is static, so the set of pumps is static.

Example of a requirement: “All pumps in room A that are in operation and whose temperature is above 50°C should be stopped within 1mn”. This example uses a spatial locator to select a set of pumps (all

[1] Not to be confused with Computation Tree Logic.

pumps located in room A that are in operation and whose temperature is above 50°C) and a time locator in the form of a DTL (the event of crossing the 50°C threshold upward) and a CTL (1mn delay). The condition states a constraint on the number of such pumps (the number should be zero).

In the example above, the spatial locator is dynamic so the set of pumps is dynamic, i.e. the number of pumps in the set may vary dynamically depending on the state of the system.

Example of a requirement: “When they are in operation, pumps should not cavitate”. This example uses a spatial locator to select a set of pumps (all pumps in the system that are in operation). The condition states a constraint on the state of such pumps (should not cavitate).

In the example above, the spatial locator is dynamic.

A requirement evaluates to satisfied, violated or undecided whereas an assumption is assumed to be always satisfied.

3.1.3. Requirements and architectural models

Requirements models do not specify precise behavior. Instead, they express the constraints that must be satisfied and the assumptions that are made. Requirements models usually consider the system under study as a "black box", but they may identify key components or subsystems when these have an impact on system requirements. So they may involve implicitly or explicitly both the physical and the control parts of the system. Such models also specify the interactions between the system and its environment, and also the assumptions made on the environment objects.

The architectural models describe the overall design of the system. They classically decompose the system into subsystems or components, express interactions between them, and place design requirements on each of them. They also specify the overall system behavior in terms of when and where specific actions are performed within the system. One can develop several alternative architectural models for the same system requirements model in order to compare their respective strengths and weaknesses.

The example of the cooling system shows that design choices made to satisfy requirements may also in turn become requirements (architecture-dependent requirements) for further refinements of the architecture. This is why requirements models and architectural models are grouped under the same name of *property models*.

It is important to note that at the early stages of system design, requirements usually bear on sets of objects that are defined in intention. The sets will be gradually known in extension later on when the system design is further detailed. A set is defined in intention when its elements are defined by the properties that they fulfill (e.g. “All pumps in room A that are in operation”). A set is known in extension when it is possible to have a sufficiently detailed description of all elements in the set for the purpose of simulation (e.g. “All pumps in room A that are in operation” = { Pump1, Pump2, Pump3 }, with knowledge about the type and characteristics of the pumps and the way they are connected to the rest of the system.

3.1.4. Example: writing the property model for (Req. 2)

First, a definition for the pumps must be given for (Req. 2: ‘When they are in operation, pumps should not cavitate’). This definition is given by the model Pump_R below. The suffix _R recalls that the definition is meant for a requirement model. Two functional dynamic attributes are defined for Pump_R: cavitate, which is a Boolean that tells whether the pump is cavitating or not, and inOperation which is a Boolean that tells whether the pump is started or not. The *external* keyword indicates that the values for cavitate and inOperation are to be found in another model, which in this case is the behavioral model, as indicated by the binding model (cf. §3.4). The keyword *class* indicates that Pump_R may have several instances, and is meant to be a component in a model library.

```
class Pump_R
  external Boolean cavitate;
```

```
   external Boolean inOperation;
end Pump_R;
```

Then as requirement (Req. 2) involves only pumps, the formal expression of (Req. 2) may be written in the requirement model of the cooling system. The set of pumps involved in (Req. 2) is declared as an external set, which means that the pumps are to be found in another model, which in this case is the architectural model, as indicated by the binding model (cf. §3.4). Then (Req. 2) itself is expressed using a logical expression on all elements of pumps using the external variables inOperation and cavitate. The keyword *object* indicates that CoolingSystem_R is a unique instance (i.e. it is not meant to be a library component such as Pump_R).

```
object CoolingSystem_R
        external Pump_R pumps { };
        required property R2 = forAll p in pumps suchThat p.inOperation check not p.cavitate;
        end CoolingSystem_R;
```

Let us now write part of a possible architectural model for CoolingSystem_R. For the example here, it is only sufficient to define the pumps from the system designer's perspective. This definition is given by the model Pump_A below. The suffix _A recalls that the definition is meant for an architectural model. The definition is done in a similar way as Pump_R, but design attributes are introduced instead of functional attributes. The only design attributes introduced for the purpose of the example are the unique identification of the pump and the required NPSH of the pump (NPSH_req). The required NPSH is a curve represented as a vector of real numbers which depends on the instance of the pump (cf. Figure 2). The *external* keyword indicates that the curve is available outside of the architectural model, for instance in an engineering database. The design of the pumps is defined in the architectural class Pump_A which is different from Pump_R in order to be able to evaluate several alternative design choices by defining other architectural classes Pump_A1, Pump_A2, etc. without modifying Pump_R. If only one design is envisaged, then Pump_A could be the same as Pump_R (so the design attributes would be directly inserted in Pump_R, or Pump_A would inherit from Pump_R).

```
class Pump_A
   String id;
   external Real NPSH_req[:, 2];
end Pump_A;
```

Then the architectural model containing the three pumps may be written, as shown below. The three pumps are instantiated from the definition of pumps above and respectively given "PO1", "PO2" and "PO3" as identifiers.

```
object CoolingSystem_A1
        Pump_A pump1 (id="PO1");
        Pump_A pump2 (id="PO2");
        Pump_A pump3 (id="PO3");
      end CoolingSystem_A1;
```

Notice that the intention of making models Pump_A and CoolingSystem_A1 to be possible implementations of Pump_R and CoolingSystem_R is not expressed formally in the models at this stage. It will be expressed later on with the set binding of set CoolingSystem_R.pumps (cf. §3.4.2).

3.2. Behavioral models

The objective of behavioral models is to express possible dynamic behaviors for the architectural models using physical laws for the physical part of the system, and the control laws for the control part of the system. Behavioral models compute the internal physical state of the system as described by its governing equations. By definition, the state of a system is the vector of all independent physical quantities that completely define the state. For instance, in the very simple case of a moving particle, the state is given by $(\vec{r}, \vec{p})$ where $\vec{r}$ is the position of the particle in a coordinate system, and $\vec{p}$ is its momentum, and the governing equations are the field equations of motion that constrain $\vec{r}$ and $\vec{p}$ in space and time.

Going back to the example of the cooling system, the governing equations of a centrifugal pump may be written as such [14]:

$$\frac{\Delta P}{\rho \cdot \omega^2} = f_h\left(\frac{q}{\omega}\right) \tag{3.1}$$

$$C_r = \frac{q \cdot \Delta P}{\eta_h \cdot \omega} \tag{3.2}$$

$$J \cdot \frac{d\omega}{dt} = C_m - C_r \tag{3.3}$$

where q is the volumetric flow rate through the pump, ΔP is the difference in pressure between the outlet and the inlet of the pump, ω is the angular velocity of the rotor, ρ is the fluid density, C_r is the resistive torque on the shaft, C_m is the motor torque on the shaft, J is the inertia of the rotating masses, f_h is the characteristic of the pump and η_h is the efficiency of the pump. These equations may be found in the components of Figure 4 that are inside the red circles.

The behavioral model of the pump is shown below. J and eta are static attributes which depend on the technology of the pumps as chosen in the architectural model. All other variables are dynamic quantities that define the internal state of the pump. Equations (3.1) through (3.3) are written in the equation section below the definition of the variables.

class CentrifugalPump

parameter Real J “Inertia of the rotating masses”;
parameter Real eta “Efficiency of the pump”;

Real Cm “Motor torque”;
Real omega “Angular velocity of the rotor”;
Real q “Volumetric flow rate”;
Real deltaP “Pressure difference between the outlet and the inlet”;
Real Pin “Pressure at the inlet”;
Real Pout “Pressure at the outlet”;
Real rho “Fluid density”;
Real Cr “Resistive torque”;

equation

```
  deltaP = Pout – Pin;
  deltaP/(rho*omega^2) = Functions.fh(q/omega);
  Cr = q*deltaP/(eta*omega);
  J*der(omega) = Cm – Cr ;
   …      // Equations of the specific enthalpy difference between the outlet and the inlet
 end CentrifugalPump;
```

The equations for the electric motor of the pump will not be shown here. In the following, it will be sufficient to know that the motor is started if the supply voltage is above a certain threshold.

```
class ElectricMotor
   Real V "Supply voltage";
equation
   …
end ElectricMotor;
```

The complete behavioral model is obtained by connecting together all the component models of the cooling system, as shown in Figure 4 (pumps, shafts, electric motors, heat exchangers, tanks, valves, pipes, etc.). A partial Modelica model for the cooling system is given below. The keyword *model* is used instead of *object* because the keyword *object* does not exist in Modelica (but the meaning is the same for this particular example).

```
model CoolingSystem_B1
   CentrifugalPump p1 "Model for pump PO1";
   ElectricMotor m1 "Model for electric motor of pump PO1";
   Shaft s1 "Model of shaft for pump PO1";
   CentrifugalPump p2 "Model for pump PO2";
   ElectricMotor m2 "Model for electric motor of pump PO2";
   Shaft s2 "Model of shaft for pump PO2";
   CentrifugalPump p3 "Model for pump PO3";
   ElectricMotor m3 "Model for electric motor of pump PO3";
   Shaft s3 "Model of shaft for pump PO3";
…
equation
   connect (m1.mechanical_outlet, s1.mechanical_inlet);
   connect (s1.mechanical_outlet, p1.mechanical_inlet);
   connect (m2.mechanical_outlet, s2.mechanical_inlet);
   connect (s2.mechanical_outlet, p2.mechanical_inlet);
   connect (m3.mechanical_outlet, s3.mechanical_inlet);
   connect (s3.mechanical_outlet, p3.mechanical_inlet);
   …
end CoolingSystem_B1;
```

3.3. Observation operators

The behavioral model computes the internal state of the system as a function of time, but it does not explicitly say when a pump is started or stopped, or when it cavitates. In other words, the functional state of the system is not given by the behavioral model, only its physical state and the internal states of the control system are. It is the purpose of the so-called *observation operators* to provide a translation between the *physical world*, as expressed in the behavioral models, and the *functional world*, as expressed in the property models. The observation operators translate physical states into functional states: they act as sensors on the behavioral model of the system to read the internal physical state of the system and translate it back to the property model in functional terms. However, there may be situations in which the physical state is not observable. That happens when the physical quantities to be observed are not fully represented in the behavioral model. In such case the value returned by the observation operator is *undefined*.

In the example, the spatial locator selects the pumps that are started. There are several possible criteria to decide whether a pump is started:

1. The supply voltage of the motor of the pump is above a certain threshold.
2. The motor torque supplied to the rotor of the pump is above a certain threshold.
3. The volumetric flow rate inside the pump is above a certain threshold.
4. The pressure elevation through the pump is above a certain threshold.

If the model does not feature the motor of the pump, obviously only criteria (2), (3) and (4) can be utilized. The observation operators for (1), (2) and (3) are given below. They are implemented as functions that take the state variables as input and produce the desired functional criterion as output. The criterion is given as the Boolean evaluation of a conditional expression in an algorithm section rather than an equation section. The reason is quite general and is the following: once the internal state of a system is known (computed by the behavioral model), all other physical quantities in the system may be derived using explicit mathematical relations (without having to solve equations). The algorithm section forbids the use of equations, and therefore guarantees that the output will not influence the input. Therefore the observation operator will not influence the internal state of the system, which is an essential property for the observation operators as it guarantees that the property models will not influence the behavioral models.

For the purpose of the example, record Boolean3 is introduced to handle undefined values for Booleans. Boolean3 is a type that represents Booleans with 3 values: ‘true’, ‘false’ and ‘undefined’.

```
record Boolean3  // Record to implement 3-valued Booleans
  Boolean def; // Value ‘defined’ or ‘undefined’ of the 3-valued Boolean.
  Boolean val;  // Value ‘true’ or ‘false’ of the 3-valued Boolean if def=true. Else unused.
end Boolean3;
```

```
function Obs_PumpStarted_1
  input Real V “Supply voltage”;
  output Boolean3 pumpStarted; // 3-valued Boolean telling whether the pump is started
  parameter Real Vt “Voltage threshold”;
algorithm
  pumpStarted.def := true;
  pumpStarted.val := (V > Vt);
```

```
end Obs_PumpStarted_1;
```

```
function Obs_PumpStarted_2
  input Real Cm "Motor torque";
  output Boolean3 pumpStarted; // 3-valued Boolean telling whether the pump is started
  parameter Real Ct "Torque threshold";
algorithm
  pumpStarted.def := true;
  pumpStarted.val := (Cm > Ct);
end Obs_PumpStarted_2;
```

```
function Obs_PumpStarted_3
  input Real q "Volumetric flow rate";
  output Boolean3 pumpStarted; // 3-valued Boolean telling whether the pump is started
  parameter Real qt "Volumetric flow rate threshold";
algorithm
  pumpStarted.def := true;
  pumpStarted.val := (q > qt);
end Obs_PumpStarted_3;
```

In the example, the condition to be fulfilled is that the pump should not cavitate. The observation operator for cavitation is given below. Notice that P and q denote indeed the physical state of the pump, but NPSH_req is a design attribute of the pump. As the curve giving the required NPSH is only defined when the pump is started as defined by Obs_PumpStarted_3 (cf. explanation below Figure 3), the condition for cavitation cannot be evaluated when Obs_PumpStarted_3 returns 'false'. So Obs_PumpStarted_3 must be added as a condition to be verified before evaluating the condition for cavitation.

```
function Obs_PumpCavitating
  input Real P "Pressure at the inlet of the pump";
  input Real q "Volumetric flow through the pump";
  input Real NPSH_req[:, 2] "Required NPSH";
  output Boolean3 pumpCavitating; // 3-valued Boolean telling whether the pump is cavitating
protected
  Boolean3 pumpStarted;
algorithm
 pumpStarted = Obs_Pump_Started_3(q);
  if (pumpStarted.def and pumpStarted.val) then
    pumpCavitating.def := true;
    pumpCavitating.val := (P < interpolate (NPSH_req, q));
  else
    pumpCavitating.def := false;
    pumpCavitating.val := false;    // Dummy value
  end if;
end Obs_PumpCavitating;
```

It is important to note that an observation operator depends only on the quantity to be computed given the input quantities, regardless of the model that provides the input quantities. For instance, the condition that must be fulfilled by a pump to prevent cavitation is computed in the same way for all pumps, hence for all pump models and only depends on the pressure at the inlet of the pump, the volumetric mass flow rate through the pump and the required NPSH curve provided by the manufacturer. Hence, the same observation operator may be reused with several different pump models. It is therefore useful to conceive libraries of observation operators, and associate them with behavioral model component libraries.

In the following, an observation operator that computes a quantity y will be mathematically denoted H_y to recall that it only depends on y. Therefore $y = H_y(e_1.x, \ldots, e_n.x)$, where $e_1, \ldots, e_n$ are the instances of the model components that contain the quantities x needed to compute y. $e_1, \ldots, e_n$ may be objects in the behavioral model or in the architectural model. x may denote a single value or a vector of values. Of course, x depends on the instance e_i it is associated with (i.e. $e_i.x \neq e_j.x$ if $i \neq j$).

For the external variable y = Pump_R.cavitate, $H_y(e_1.x, \ldots, e_n.x)$ = Obs_PumpCavitating (e_b.P, e_b.q, e_a.NPSH_req), where e_b is an instance of class CentrifugalPump in the behavioral model CoolingSystem_B1, e_a is an instance of class Pump_A in architectural model CoolingSystem_A1.

Hence e_a.cavitate = Obs_PumpCavitating (e_b.P, e_b.q, e_a.NPSH_req).

The way to construct the expression above is fully explained in §3.4.

3.4. Bindings

3.4.1. Purpose of bindings

The purpose of bindings is to solve the general problem of evaluating at simulation time the formal requirement expressions that contain quantifiers on external sets and conditions on external variables.

For (Req. 2), the problem is to evaluate

required property R2 = ***forAll*** p ***in*** pumps ***suchThat*** p.inOperation ***check not*** p.cavitate;

This expression contains a quantifier (*forAll*) that operates on external set pumps and two external variables (inOperation and cavitate).

To be able to evaluate R2 at simulation time, one must indicate which elements populate the external set and how to compute the external variables.

The operator *bind_set* is introduced in order to indicate which elements populate external sets.

As explained in §3.3, external variables y are computed using $y = H_y(e_1.x, \ldots, e_n.x)$ expressions where $e_1, \ldots, e_n$ are the instances of the model components that contain the quantities x needed to compute y and H_y is an observation operator for y.

As there may be a very large number of external variables, the operators *bind_input*, *bind_variable,* and *bind_instance* are introduced to form $y = H_y(e_1.x, \ldots, e_n.x)$ expressions as economically as possible, i.e. to automate the process of forming those expressions to the extent practical.

To that end, the idea is to define as much as possible binding at the class level, and go to the instance level only when class binding is not possible. Binding at the class level is called *class binding* and regroups *bind_input* and *bind_variable*. Class binding enables storing the binding definitions in libraries alongside with the associated classes and reusing them automatically for all instances of those classes. Binding at the instance level is called *instance binding* and regroups *bind_instance* only. Instance binding complements class binding when binding must be done separately for each class instance on a case by case basis.

The following paragraphs show how to construct $p.y = H_y(e_1.x, \ldots, e_n.x)$ expressions for each object p member of external sets, and each external variable p.y declared in the class of object p.

3.4.2. Set binding

The purpose of set binding is to provide the elements p to the external sets.

In the following, set binding will be mathematically denoted s ← { p_1, …, p_n }. This notation means that objects p_1, …, p_n are declared as elements of set s. p_1, …, p_n may be class collections or class instances. A class collection is defined as all instances of a class within a particular model.

The target of s ← { p_1, …, p_n } is the set { p_1, …, p_n } and is denoted target(s ← { p_1, …, p_n }).

Set s may have several declared bindings, but may have at most one active binding. An active binding is the declared binding that is used to construct the verification model (cf. §3.5).

A set binding is declared with the operator *bind_set*.

The binding below shows how to declare that set CoolingSystem_R.pumps contains all instances of class Pump_A in CoolingSystem_A1. The class collection CoolingSystem_A1.Pump_A is used in the binding to indicate that all instances of Pump_A in CoolingSystem_A1 are bound to CoolingSystem_R.pumps. The name of the binding is s1 and may be optionally provided, as indicated by the notation [s1].

```
binding CoolingSystem_R1
  bind_set [s1] (CoolingSystem_R.pumps = { CoolingSystem_A1.Pump_A });
  …
end CoolingSystem_R1;
```

The binding below declares only instances pump1 and pump2 as members of set CoolingSystem_R.pumps. The instances CoolingSystem_A1.pump1 and CoolingSystem_A1.pump2 are used in the binding instead of the class collection CoolingSystem_A1.Pump_A.

```
binding CoolingSystem_R2
  bind_set  (CoolingSystem_R.pumps = { CoolingSystem_A1.pump1, CoolingSystem_A1.pump2 });
  …
end CoolingSystem_R2;
```

Notice that although CoolingSystem_R.pumps is declared as having elements of type (i.e. that are instances of) Pump_R, it contains elements of type Pump_A when the binding is in effect. This implies that elements of type Pump_A should also be considered as being of type Pump_R when bound to set CoolingSystem_R.pumps. Then it makes sense to write p.y with p being of type Pump_A and y declared as a variable of Pump_R. It becomes now clear that instances of Pump_A that are used in architectural model CoolingSystem_A1 implement (i.e. are a possible realization of) Pump_R when bound to CoolingSystem_R.pumps.

3.4.3. Class binding

The purpose of class binding is to form expressions P.y = H_y (E_1.x, …, E_n.x) where P denotes the class of p and E_1, …, E_n denote the classes of e_1, …, e_n. These formal expressions may be computed when classes P and E_1, …, E_n are replaced by their proper instances p and e_1, …, e_n. It is the purpose of set and instance bindings to specify how to perform this operation respectively for p and e_1, …, e_n (cf. §3.4.2 and §3.4.4).

Class binding is performed in two steps: (1) the input binding of H_y and (2) the variable binding of H.y.

Input binding of H_y will be mathematically denoted H_y (u_1, …, u_n) ← (u_1=E_1.x, …, u_n=E_n.x). This expression means that E_1.x, …, E_n.x are declared as the inputs u_1, …, u_n of H_y.

As a consequence of §3.3, observation operator H_y may have several active input bindings.

The signature of H_y is denoted sig(H_y) and defined as sig(H_y) = { E_1, …, E_n }.

Variable binding of P.y will be mathematically denoted P.y ← [H_1, …, H_p]. This means that observation operators H_1, …, H_p are possible candidates for forming the expression p.y = H_y (e_1.x, …, e_n.x), i.e. that H_y will be chosen among H_1, …, H_p according to the algorithm given in §3.4.5. The notation […] denotes an *ordered* set.

Variable P.y may have several declared bindings, but may have at most one active binding.

A *functional role* may be optionally assigned to binding P.y ← [H_1, …, H_p]. The role of binding P.y ← [H_1, …, H_p] is denoted role(P.y ← [H_1, …, H_p]). The purpose of role() is to find the correct inputs for the proper observation operator among the candidates in [H_1, …, H_p] when type matching of the inputs of the proper observation operator do not yield a unique possibility (cf. algorithm in §3.4.5).

An input binding is declared with the operator *bind_input*.

A variable binding is declared with the operator *bind_variable*.

The binding below shows how to declare the inputs for the observation operators used in the example of the cooling system.

```
binding PowerPlant_Lib
  bind_input [i1] (Obs_PumpStarted_1 = (V = ElectricMotor.V));
  bind_input [i2] (Obs_PumpStarted_2 = (Cm = CentrifugalPump.Cm));
  bind_input [i3] (Obs_PumpStarted_3 = (q = CentrifugalPump.q));
  bind_input [i4] (Obs_PumpCavitating = (P = CentrifugalPump.Pin, q = CentrifugalPump.q,
                                                    NPSH_req = Pump_A.NPSH_req);
  …
end PowerPlant_Lib;
```

The binding below shows how to declare the possible observation operators for each external variable in Pump_R. A role is assigned to binding v1.

```
binding Pump_R1
  bind_variable v1 (Pump_R.inOperation = [ Obs_PumpStarted_1, Obs_PumpStarted_2,
                                                    Obs_PumpStarted_3 ]);
  bind_variable [v2] (Pump_R.cavitate = [ Obs_PumpCavitating ]);
  v1.role = “inOperation”;
  …
end Pump_R1;
```

Notice that input bindings are independent of any particular requirements or architectural models. They only depend on the behavioral model libraries that are used to construct the behavioral models. To the contrary, variable bindings depend on the classes of the requirement model where the external variables are declared, but they do not depend on the instances of these classes. This is why in the example above, input bindings are stored in bindings associated with the power plant model component library (PowerPlant_Lib), and variable bindings are stored in bindings associated with Pump_R.

Figure 6 shows that the observation operator is the central object that link together the requirement, architectural and behavioral models at the class level.

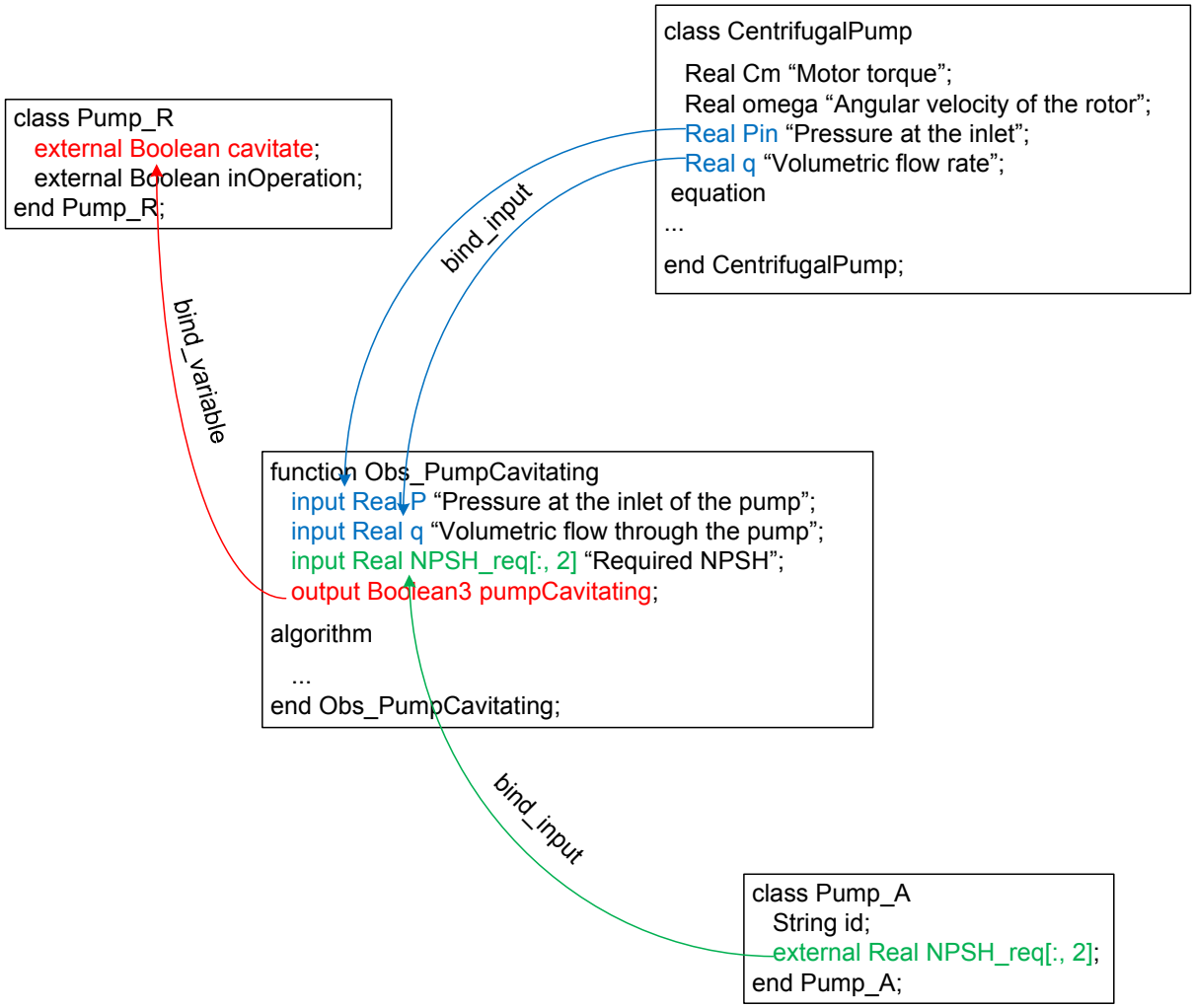


Figure 6: class binding of the observation operator Obs_PumpCavitating

3.4.4. Instance binding

The purpose of instance binding is to form the final expressions $p.y = H_y(e_1.x, \ldots, e_n.x)$ from the expressions $P.y = H_y(E_1.x, \ldots, E_n.x)$ where p is the proper instance of P and $e_1, \ldots, e_n$ are the proper instances of $E_1, \ldots, E_n$.

Instance binding of object a will be denoted $a \leftarrow \{ e_1, \ldots, e_n \}$. This means that instances $e_1, \ldots, e_n$ are bound to instance a. Object a may be an instance in the architectural or requirement model. Objects $e_1, \ldots, e_n$ may be instances in the behavioral, architectural or requirements models.

Object a may have several declared bindings, but may have at most one active binding.

The target of $a \leftarrow \{ e_1, \ldots, e_n \}$ is the set $\{ e_1, \ldots, e_n \}$ and is denoted target($a \leftarrow \{ e_1, \ldots, e_n \}$).

The class of e_i is denoted class(e_i).

The signature of $a \leftarrow \{ e_1, \ldots, e_n \}$ is $\{ class(e_1), \ldots, class(e_n) \}$ and is denoted sig($a \leftarrow \{ e_1, \ldots, e_n \}$).

A *functional role* may be optionally assigned to each e_i in binding $a \leftarrow \{ e_1, \ldots, e_n \}$,. The role of e_i in binding $a \leftarrow \{ e_1, \ldots, e_n \}$ is denoted role(e_i, $a \leftarrow \{ e_1, \ldots, e_n \}$). The purpose of role() is to discriminate between instances of the same class in the target of the binding. Therefore, if e_i and e_j are members of target($a \leftarrow \{ e_1, \ldots, e_n \}$) and if class($e_i$) = class($e_j$), then role($e_i$, $a \leftarrow \{ e_1, \ldots, e_n \}$) should be different from role(e_j, $a \leftarrow \{ e_1, \ldots, e_n \}$) in order to discriminate e_i from e_j.

Figure 7 shows a graphical representation of the binding between the pumps in the architectural model and the objects in the behavioral model that represent their behavior.

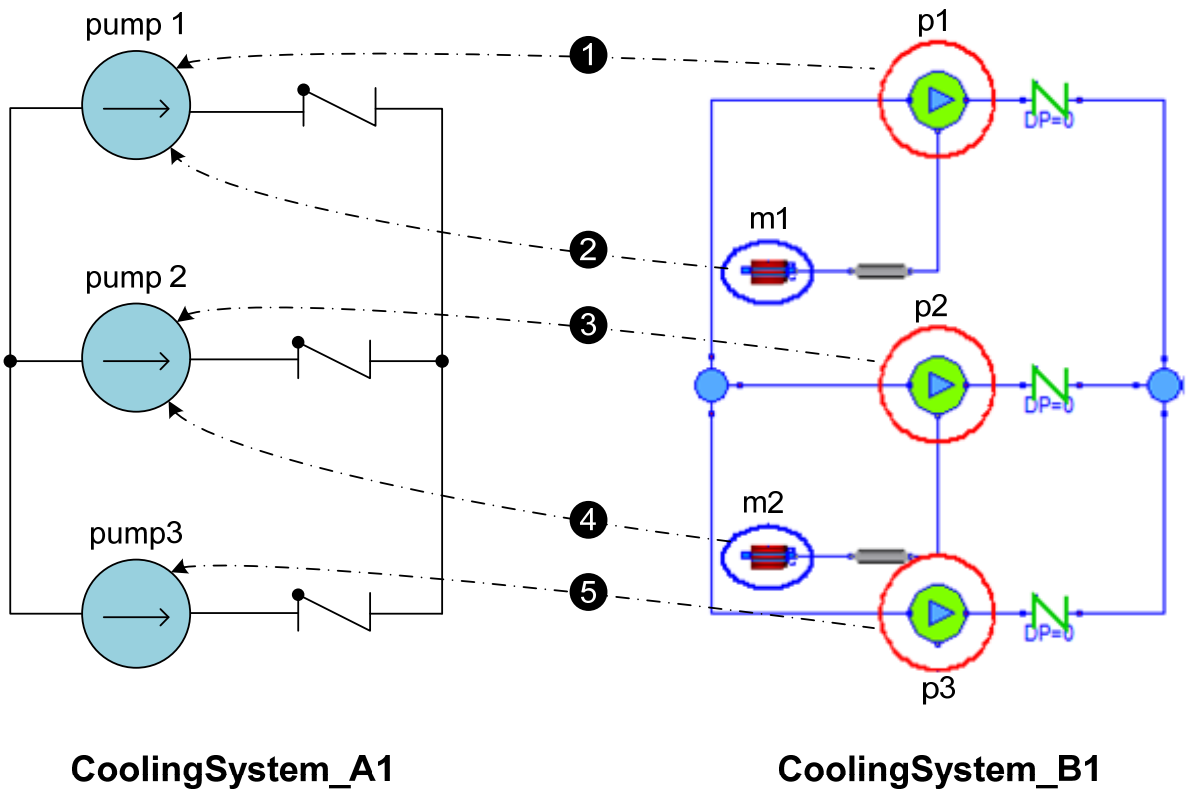


Figure 7: instance binding between the architectural model (left) and the behavioral model (right) for the pumps

In Figure 7, pump3 is only represented by its hydraulic part in the behavioral model, contrary to the original model shown in Figure 4 where pump3 is represented by its hydraulic part, shaft and electric motor. The objective of this modification is to show how the binding can be adapted to particular features of the behavioral model. If the motor of pump3 is not represented in the behavioral model, then requirements related to the control system such as (Req. 4) cannot be verified, but (Req. 2) can still be verified. So this illustrates how the binding may be utilized with incomplete behavioral models that are nonetheless sufficient for verifying a subset of the requirements corresponding for instance to a preliminary design. These models may be completed later on to verify a larger subset of the requirements when the design is more detailed.

The binding below declares the instance binding for all pumps in CoolingSystem_A1 as shown in Figure 7. The same role is assigned to elements m1, m2 and p3 in CoolingSystem_B1.

```
binding  CoolingSystem_A1

  bind_instance i1 (CoolingSystem_A1.pump1={ CoolingSystem_B1.p1, CoolingSystem_B1.m1 });
  bind_instance i2 (CoolingSystem_A1.pump2={ CoolingSystem_B1.p2, CoolingSystem_B1.m2 });
  bind_instance i3 (CoolingSystem_A1.pump3={ CoolingSystem_B1.p3 });

  i1.role (CoolingSystem_B1.m1) = “inOperation”;
  i2.role (CoolingSystem_B1.m2) = “inOperation”;
  i3.role (CoolingSystem_B1.p3) = “inOperation”;

end CoolingSystem_A1;
```

3.4.5. Algorithm for forming y = H(x) expressions

Let us consider:

a. an external variable y declared in a class R: R.y,
b. an external set s of objects of type R declared in an object O: O.s

The problem is to form the expressions $p.y = H_y(e_1.x, \ldots, e_n.x)$ for all objects p in set O.s.

Active bindings used as input of the algorithm

a. Set binding $O.s \leftarrow \{ p_1, \ldots, p_n \}$
b. Variable binding $R.y \leftarrow [H_1, \ldots, H_r]$
c. Input bindings $H_i(u_1, \ldots, u_n) \leftarrow (u_1=E_1.x, \ldots, u_n=E_n.x)$ for i = 1 to i = r
d. Instance bindings $p \leftarrow \{ e_1, \ldots, e_n \}$ for all p in target$(O.s \leftarrow \{ p_1, \ldots, p_n \})$

Algorithm

For each p in target$(O.s \leftarrow \{ p_1, \ldots, p_n \})$

1. Find H of lowest rank in $R.y \leftarrow [H_1, \ldots, H_r]$ such that $sig(H) \subset sig(p \leftarrow \{ e_1, \ldots, e_n \})$. The result is H_y.
2. From the binding $H_y(u_1, \ldots, u_n) \leftarrow (u_1=E_1.x, \ldots, u_n=E_n.x)$, form symbolically the expression $p.y = H_y(u_1=E_1.x, \ldots, u_n=E_n.x)$.
3. For each E_i in $\{ E_1, \ldots, E_n \}$
   a. If there exists a unique instance e_i in the target of $p \leftarrow \{ e_1, \ldots, e_n \}$ such that $class(e_i) = E_i$, then keep e_i. If $role(e_i, p \leftarrow \{ e_1, \ldots, e_n \})$ is different from $role(R.y \leftarrow [H_1, \ldots, H_r])$, a warning may be raised.
   b. Else if there exists multiple instances e_i in the target of $p \leftarrow \{ e_1, \ldots, e_n \}$ such that $class(e_i) = E_i$, then keep e_i such that $role(e_i, p \leftarrow \{ e_1, \ldots, e_n \}) = role(R.y \leftarrow [H_1, \ldots, H_r]$.
   c. If no such unique e_i may be found, then raise an error.
4. Replace in $p.y = H_y(u_1=E_1.x, \ldots, u_n=E_n.x)$ the classes E_i by the found instances e_i. The result is $p.y = H_y(e_1.x, \ldots, e_n.x)$.

As an example of use of this algorithm, let us consider

a. R.y = Pump_R.inOperation
b. O.s = CoolingSystem_R.pumps

Notations

R = Pump_R	y = inOperation	p_3 = A.pump3
O = CoolingSystem_R	H_1 = Obs_PumpStarted_1	e_1 = B.p1
A = CoolingSystem_A1	H_2 = Obs_PumpStarted_2	e_2 = B.m1
B = CoolingSystem_B1	H_3 = Obs_PumpStarted_3	e_3 = B.p2
E_1 = CentrifugalPump	p_1 = A.pump1	e_4 = B.m2
E_2 = ElectricMotor	p_2 = A.pump2	e_5 = B.p3

Active bindings used as input of the algorithm

a. Set binding $O.s \leftarrow \{$ A.Pump_A $\}$
b. Variable binding $R.y \leftarrow [H_1, H_2, H_3]$
c. Input bindings $H_1(V) \leftarrow \{ V=E_2.V \}$, $H_2(Cm) \leftarrow \{ Cm=E_1.Cm \}$, $H_3(q) \leftarrow \{ q=E_1.q \}$
d. Instance bindings $p_1 \leftarrow \{ e_1, e_2 \}$, $p_2 \leftarrow \{ e_3, e_4 \}$, $p_3 \leftarrow \{ e_5 \}$

target $(O.s \leftarrow \{$ A.Pump_A $\}) = \{ p_1, p_2, p_3 \}$.

Applying the algorithm to the first element p in target $O.s \leftarrow \{$ A.Pump_A $\})$:

Algorithm

$p = p_1$

1. Find H of lowest rank in R.y ← [H_1, H_2, H_3] such that sig(H) ⊂ sig(p ← { e_1, e_2 }). The result is H_1.
2. From the binding H_1(V) ← { V=E_2.V }, form symbolically the expression p.y = H_1(V=E_2.V).
3. E_2 has a unique instance in the target of p ← { e_1, e_2 }: e_2. Keep e_2.
4. Replace in p.y = H_1(V=E_2.V) class E_2 by e_2. The result is p.y = H_1(V=e_2.V).

Replacing the symbols by their values yields:

CoolingSystem_A1.pump1.inOperation = Obs_PumpStarted_1 (CoolingSystem_B1.m1.V).

3.5. Verification models

The verification model is the complete Modelica model used for automatic testing of the system design against requirements.

A *configuration* is defined as the set of all models and active bindings needed to produce automatically the verification model, as shown in Figure 8.

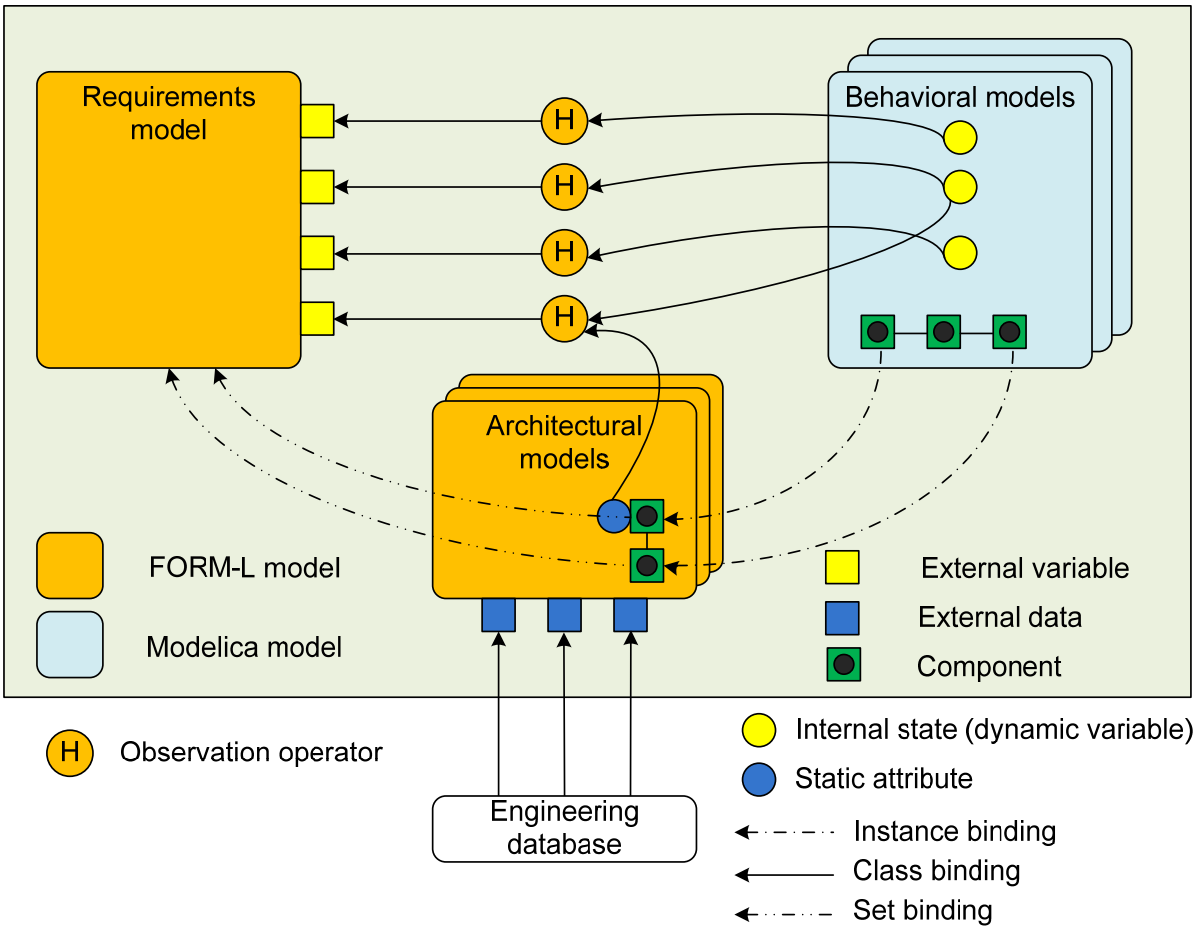


Figure 8: configuration for the verification model

The verification model is in turn used to generate automatically the simulator as shown in Figure 9.

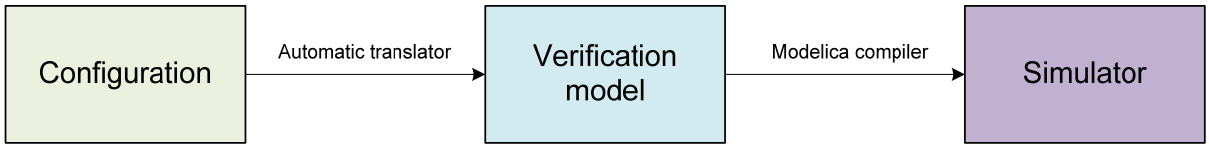


Figure 9: automatic generation of the simulator

The following procedure is given to illustrate how the verification model may be automatically generated from the configuration for (Req. 2):

1. The configuration contains:
   a. The requirement model CoolingSystem_R
   b. The architectural model CoolingSystem_A1
   c. The behavioral model CoolingSystem_B1
   d. The bindings CoolingSystem_R2, PowerPlant_Lib, Pump_R1, CoolingSystem_A1
2. (Req. 2) is given in model CoolingSystem_R by

   R2 = ***forAll*** p ***in*** pumps ***suchThat*** p.inOperation ***check not*** p.cavitate;
3. Apply algorithm given in §3.4.5 to all external variables in R2 for all sets, i.e. to R.y = Pump_R.inOperation, R.y = Pump_R.cavitate for O.s = CoolingSystem_R.pumps.

   The result is:

   pump1.inOperation = Obs_PumpStarted_1 (CoolingSystem_B1.m1.V)

   pump2.inOperation = Obs_PumpStarted_1 (CoolingSystem_B1.m2.V)

   pump3.inOperation = Obs_PumpStarted_2 (CoolingSystem_B1.p3.Cm)

   pump1.cavitate = Obs_PumpCavitating (CoolingSystem_B1.p1.Pin,
   CoolingSystem_B1.p1.q,
   CoolingSystem_A1.pump1.NPSH_req)

   pump2.cavitate = Obs_PumpCavitating (CoolingSystem_B1.p2.Pin,
   CoolingSystem_B1.p2.q,
   CoolingSystem_A1.pump2.NPSH_req)

   pump3.cavitate = Obs_PumpCavitating (CoolingSystem_B1.p3.Pin,
   CoolingSystem_B1.p3.q,
   CoolingSystem_A1.pump3.NPSH_req)

   Remark: when applying algorithm given in §3.4.5 for R.y = Pump_R.cavitate, the instance binding pump1 ← { CoolingSystem_B1.p1, CoolingSystem_B1.m1, CoolingSystem_A1.pump1 } should be used for pump1, where CoolingSystem_A1.pump1 is automatically added to the binding (by considering that pump1 is bound to itself). The same consideration applies to pump2 and pump3.
4. Expand R2 for all p in pumps = { pump1, pump2, pump3 }.

   In the following, elements of R2 are denoted { R2.pump1, R2.pump2, R2.pump3 }.

   R2.pump1 = if pump1.inOperation then not (pump1.cavitate) else 'undefined'
   = if Obs_PumpStarted_1 (CoolingSystem_B1.m1.V) then
   not (Obs_PumpCavitating (CoolingSystem_B1.p1.Pin,
   CoolingSystem_B1.p1.q,
   CoolingSystem_A1.pump1.NPSH_req))

   else 'undefined'

   R2.pump2 = if pump2.inOperation then not (pump2.cavitate) else 'undefined'
   = if Obs_PumpStarted_1 (CoolingSystem_B1.m2.V) then
   not (Obs_PumpCavitating (CoolingSystem_B1.p2.Pin,

CoolingSystem_B1.p2.q,
CoolingSystem_A1.pump2.NPSH_req))

else ‘undefined’

R2.pump3 = if pump3.inOperation then not (pump3.cavitate) else ‘undefined’
= if Obs_PumpStarted_2 (CoolingSystem_B1.p3.Cm) then
not (Obs_PumpCavitating (CoolingSystem_B1.p3.Pin,
CoolingSystem_B1.p3.q,
CoolingSystem_A1.pump3.NPSH_req))

else ‘undefined’

5. Compute the accumulated value of R2 over set pumps = { pump1, pump2, pump3 } at one instant t.

   The accumulated value of R2 at instant t over set pumps is denoted req2(t).

   The value of req2(t) is:

   a. ‘true’ when at least one element in pumps evaluates to ‘true’ and none evaluates to ‘false’.
   b. ‘false’ when at least one element in pumps evaluates to ‘false’.
   c. ‘undefined’ when all elements in pumps evaluate to ‘undefined’.

6. The value of (Req. 2) is given by the accumulated value of req2(t) over a simulation run.

   The accumulated value of req2(t) over a simulation run is denoted Req2.

   The value of Req2 is:

   a. ‘satisfied’ if req2(t) evaluates to ‘true’ at at least one instant t and never evaluates to ‘false’.
   b. ’violated’ if req2(t) evaluates to ‘false’ at at least one instant t.
   c. ‘undecided’ if req2(t) always evaluates to ‘undefined’.

A possible Modelica verification model for (Req. 2) is given below.

```
model VerificationModel_Req2          // Verification model for requirement (Req. 2)

  import BehavLib;                    // Import library of behavioral models
  import ObsLib;                      // Import library of observation operators

  BehavLib.CoolingSystem_B1  Behav;   // Unique instance of the behavioral model
                                      // of the cooling system

  parameter Real pump1_NPSH_req[:, 2]={ /* Curve to be interpolated */ };
  parameter Real pump2_NPSH_req[:, 2]={ /* Curve to be interpolated */ };
  parameter Real pump3_NPSH_req[:, 2]={ /* Curve to be interpolated */ };

  parameter Integer nb_pumps = 3;     //  Number of pumps

  Boolean3 R2[nb_pumps];              //  Vector { R2.pump1, R2.pump2, R2.pump3 }
  Boolean3 req2;                      // Value of requirement (Req. 2) at instant t
  Boolean3 Req2;                      //  Value of requirement (Req. 2) at end of simulation

  Boolean3 pumpStarted[nb_pumps];     //  Intermediate vector
  Boolean3 pumpCavitating[nb_pumps]; //  Intermediate vector

equation

  pumpStarted[1] = ObsLib.Obs_PumpStarted_1 (Behav.m1.V);
  pumpStarted[2] = ObsLib.Obs_PumpStarted_1 (Behav.m2.V);
```

```
  pumpStarted[3] = ObsLib.Obs_PumpStarted_2 (Behav.p3.Cm);

  pumpCavitating[1] = ObsLib.Obs_PumpCavitating (Behav.p1.Pin, Behav.p1.q,
                                                 pump1_NPSH_req);
  pumpCavitating[2] = ObsLib.Obs_PumpCavitating (Behav.p2.Pin, Behav.p2.q,
                                                 pump2_NPSH_req);
  pumpCavitating[3] = ObsLib.Obs_PumpCavitating (Behav.p3.Pin,  Behav.p3.q,
                                                 pump3_NPSH_req);

  for i in 1:nb_pumps loop
    R2[i].def = pumpStarted[i].def and pumpCavitating[i].def and pumpStarted [i].val;
    R2[i].val = not (pumpCavitating[i].val);
  end for;

  req2.def = (R2[1].def or R2[2].def or R2[3].def);
  req2.val = ((if R2[1].def then R2[1].val else true) and
              (if R2[2].def then R2[2].val else true) and
              (if R2[3].def then R2[3].val else true));

  Req2 = integ(req2);

end VerificationModel_Req2;
```

The behavioral model and the observation operators are imported as black boxes into the verification model. However the NPSH curves must be individually retrieved from the architectural model.

In the calls to the observation operators, CoolingSystem_B1 must be replaced by the name of the unique instance of class CoolingSystem_B1. The actual name given to this unique instance can be freely chosen and is Behav in the model above.

The value of the number of pumps nb_pumps must be fixed to the number of elements in CoolingSystem_R.pumps for the binding

CoolingSystem_R.pumps ← { CoolingSystem_A1.Pump_A } .

The 3-valued Boolean logic

y = if cond then not (u) else ‘undefined’;

is translated as such in standard Modelica:

y.def = cond.def and u.def and cond.val;
y.val = not (u.val);

The accumulated value of R from R1 and R2 is computed as such in standard Modelica:

R.def = R1.def or R2.def;
R.val = (if R1.def then R1.val else true) and (if R2.def then R2.val else true);

R evaluates to ‘true’ if R1 or R2 evaluate to ‘true’ and R1 and R2 do not evaluate to ‘false’. R evaluates to ‘false’ if R1 or R2 evaluate to ‘false’. R evaluates to undefined if R1 and R2 evaluate to ‘undefined’.

The function y = integ(u) (which is in fact a Modelica block as it has an internal state) computes the accumulated value of u from the initial instant t0 to the current instant t of the simulation. y is computed according to the following algorithm:

a. At instant t0-, y is initialized to ‘undefined’.
b. If at instant t-, u evaluates to ‘false’, then y evaluates to ‘false’ for all instants after t+ (including t+).
c. If at instant t-, u evaluates to ‘true’ and y is ‘undefined’, then y evaluates to ‘true’ at t+.

t^- denotes instant t just before the computation of integ. t^+ denotes instant t just after the computation of integ.

At the end of the simulation run:

a. If Req2 is 'true', then (Req. 2) is satisfied.
b. If Req2 is 'false', then (Req. 2) is violated.
c. If Req2 is 'undefined', then (Req. 2) is undecided.

In order to execute the tests, test scenarios must be provided as inputs to the verification model. Possible inputs are boundary conditions on the behavioral model, design data on the architectural model, hypothesis on the requirements model as shown in Figure 10. The method to generate test scenarios is not discussed here. Also, the initial state of the behavioral model should be consistent with the inputs. The way to compute a consistent initial state from the inputs is not discussed here.

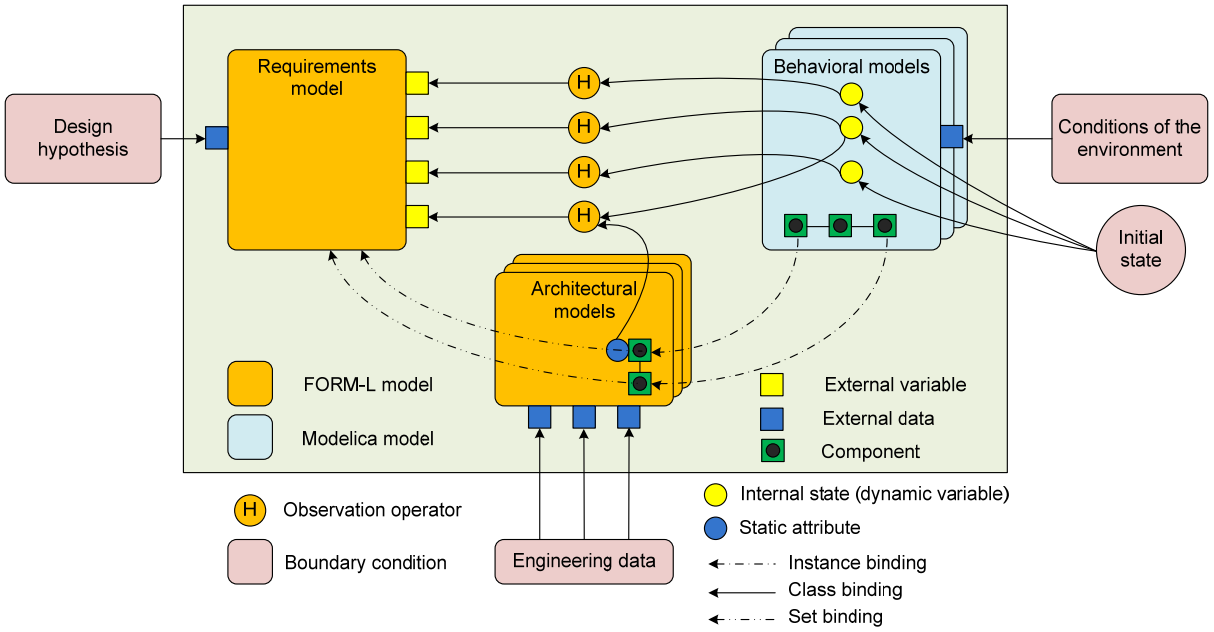


Figure 10: complete configuration with the boundary conditions and the initial state

4. How FORM-L complements Modelica

4.1. Introduction

The purpose of this chapter is to explain how FORM-L complements Modelica for the purpose of design verification against system's requirements.

FORM-L complements Modelica on 4 fundamental aspects:

1. FORM-L expresses constraints.
2. FORM-L uses time locators.
3. FORM-L uses spatial locators.
4. FORM-L uses 3-valued logic.

To explain how FORM-L complements Modelica, a formal approach is used that is independent of the syntaxes of both languages. Mathematical notations are used, in particular from the set theory that is central to FORM-L. The precise definition of Modelica can be found in [1] and a specification of FORM-L can be found in [9].

4.2. FORM-L expresses constraints

FORM-L uses constraint equations on the system states defined by the Modelica equations.

The Modelica equations are used to define the evolution in time of the system physical states (the states that are solution of the equations).The objective of FORM-L is not to define the physical states, but the constraints on those states to ensure the proper operation of the physical system. In FORM-L, the states are referenced as external quantities (i.e. quantities that are computed outside of the FORM-L model), which are computed by the Modelica behavioral model.

For a given operating mode, Modelica typically expresses the evolution equations in the form of DAEs:

$$\begin{cases} \dot{x}_m = f_m(x_m, a_m, u, t) \\ 0 = g_m(x_m, a_m, u) \\ m^+ = fsa(m^-, x_{m^-}, a_{m^-}, t) \end{cases} \quad (4.1)$$

x_m denotes the differential states and a_m denotes the algebraic states in the system for mode m, m denoting a discrete value among the finite set of possible modes. f_m and g_m are continuous and differentiable functions. fsa denotes the finite state automaton that gives the value m^+ of the mode just after the event that causes the mode switching from the value m^- of the mode just before the event. u denotes the inputs. t denotes the time. A new proposal to implement DAE systems in Modelica with mode switching involving changes in the structure of the state vector is given in [15].

If the system is static, then the size of x_m is zero and there is only one mode, hence Eq. (4.1) reduces to $0 = g(a,u)$, g being the static model.

FORM-L typically expresses constraints in the form

$$\begin{cases} c_m(y_m, t) < 0 \\ m^+ = fsa(m^-, y_{m^-}, t) \end{cases} \quad (4.2)$$

Note that the fsa used in Eq. (4.2) that defines the required transitions between operating modes is not necessarily the same as the one used in Eq. (4.1). The purpose of Eq. (4.1) is to compute the states x_m and a_m, whereas the purpose of Eq. (4.2) is to define the proper operating domain for x_m and a_m. However, in general, Eq. (4.2) does not involve directly x_m and a_m but quantities y_m that are derived from x_m and a_m. As already stated in §3.3, this is due to the fact that requirements are always expressed on measurable quantities or quantities that are meaningful to the operation engineer, and that the quantities that compose x_m and a_m do not always have those characteristics. So it is necessary to transform x_m and a_m into such quantities before expressing the constraints.

The relationship between y_m and x_m and a_m is given by the so-called observation operator H_{y_m} that depends on the quantity y_m to be observed:

$$\begin{cases} y_m = H_{y_m}(x_m, a_m) \\ P_{y_m}(x_m, a_m) < 0 \end{cases} \quad (4.3)$$

H_{y_m} is always an explicit function of x_m and a_m, so the values of y_m never influence the values of x_m and a_m. Therefore Eq. (4.3) does not constrain the values of x_m and a_m.

As stressed in §3.3, depending on the state of the physical system, observation operators cannot always be evaluated. P_{y_m} is the condition on the state that ensures that the observation operator H_{y_m} may be evaluated.

4.3. FORM-L uses time locators

FORM-L uses DTLs (discrete time locators) and CTLs (continuous time locators) to define the time periods when requirements should be evaluated.

DTLs are sets of events. An event defines occurrences of a fact that has no duration. A DTL is denoted Δ_D.

CTLs are sets of continuous time periods. A CTL is denoted Δ_C. A continuous time period is bounded by two events: the start and the end of the continuous time period.

An object which may be a DTL or a CTL is denoted Δ_{DC}.

Hence $\tau \in \Delta_D$ is an event, $\tau \in \Delta_C$ is a time interval and $\tau \in \Delta_{DC}$ is an event or a time interval.

To ease notations, $t \in \Delta_{DC}$ denotes an instant belonging to τ with $\tau \in \Delta_{DC}$.

The CTL that represents the time length of a simulation run is denoted Δ_U. $\Delta_U = \left[t_0, t_f\right]$ where t_0 and t_f are resp. the initial and final instants of the simulation run.

4.4. FORM-L uses spatial locators

FORM-L uses spatial locators to define the sets of objects that are subject to the requirements.

In the sequel, Σ represents the system subject to the requirements. It is mathematically defined as the set that contains all the objects and sub-systems in the system that are involved in the requirements. Therefore $e \in \Sigma$ denotes an object or a sub-system of Σ. Σ_A represents a possible architectural model for Σ and $\mathrm{M}(\Sigma_A)$ represents a possible behavioral model for Σ_A. $\overline{\mathrm{M}}(\Sigma)$ represents the set of possible modes for Σ and $\overline{\mathrm{M}}(\mathrm{M}(\Sigma_A))$ represents the set of possible modes for $\mathrm{M}(\Sigma_A)$. Notice that in general $\overline{\mathrm{M}}(\mathrm{M}(\Sigma_A)) \neq \overline{\mathrm{M}}(\Sigma)$, which means that the modes of the behavioral model may be different from the modes defined for the requirement model.

Each element e of Σ is defined by its attributes, which are the quantities that characterize element e for the requirements. There are two kinds of attributes: (1) static attributes, whose values are fixed by design choices, and (2) dynamic attributes, whose values depend on time and can only be computed from the behavioral model equations. Static attributes are denoted $e.p$. Dynamic attributes correspond to the observed quantities defined by Eq. (4.3) and are therefore denoted $e.y_m$ where $m \in \overline{\mathrm{M}}(\Sigma)$.

Similarly, each element e of Σ_A is defined by its attributes.

Each element $\hat{e}$ of $\mathrm{M}(\Sigma_A)$ is defined by its variables and the equations that constrain them, as in Eq. (4.1):

$$\begin{cases} \hat{e}.\dot{x}_m = \hat{e}.f_m(\hat{e}.x_m, \hat{e}.a_m, u, t) \\ 0 = \hat{e}.g_m(\hat{e}.x_m, \hat{e}.a_m, u) \\ \hat{e}.m^+ = \hat{e}.fsa(\hat{e}.m^-, \hat{e}.x_{m^-}, \hat{e}.a_{m^-}) \end{cases} \quad (4.4)$$

where $m \in \overline{\mathrm{M}}(\mathrm{M}(\Sigma_A))$ and where the dot notation means that the variable or function on the right-hand side of the dot belongs to the element on the left-hand side of the dot.

Model $\mathrm{M}(\Sigma_A)$ is constructed by connecting together the elements $\hat{e}$ of $\mathrm{M}(\Sigma_A)$. Formally, connections can be expressed as additional equations:

$$0 = cx(\hat{e}_1.x_{m_1}, \hat{e}_1.a_{m_1}, \hat{e}_2.x_{m_2}, \hat{e}_2.a_{m_2}, \ldots, \hat{e}_n.x_{m_n}, \hat{e}_n.a_{m_n}) \quad (4.5)$$

To be solved, the resulting system of equations combining one Eq. (4.4) for each element and one Eq. (4.5) for each connection should be square, i.e. there should be as many equations as there are unknowns to be solved. This means that the Modelica components used in the model should be complete in the sense that when connecting the components together to form the model, the resulting system of equations should be square. Hence, Modelica operates on instances of completely defined objects.

On the contrary, the system of equations given by Eq. (4.2) can have any number of equations. So it is possible to express requirements on partially defined objects, and extend the list of requirements as the design of the system becomes more and more detailed. For the same reason, it is also possible to express the same requirement on a collection of objects of variable size N, with N depending on some static or dynamic condition. These sets are constructed in intention, i.e. by giving the properties that the objects should fulfill to be members of the set.

$$S(t) = \{e \in \Sigma | P(e,t)) = true\} \quad (4.6)$$

Eq. (4.6) states that set S is constructed by selecting all elements in Σ that satisfy the spatial locator P which is a logical condition involving attributes of elements e. Notice that the number of elements in set S varies dynamically on time as $P(e,t)$ depends on time.

P may be decomposed into two parts: a static part denoted P_S that involves only static attributes, and a dynamic part denoted P_T that involves only dynamic attributes.

$$P_S(e) = [c(e.p) < 0] \quad (4.7)$$

$$P_T(e,t) = [c_m(e.y_m, t) < 0,\ m^+ = fsa(m^-, e.y_{m^-}, t)] \quad (4.8)$$

$$P(e,t) = [P_S(e) \wedge P_T(e,t)] \quad (4.9)$$

where the notation $P = [<\exp r>]$ means that P represents the expression inside the brackets and $\wedge$ denotes the *and* operator.

In equations (4.7) through (4.9), e may denote a single element or a vector of elements if several elements are involved in the same property.

The condition to be verified is denoted as:

$$P_C(e,t) = [c'_m(e.y_m, t) < 0,\ m^+ = fsa(m^-, e.y_{m^-}, t)] \quad (4.10)$$

In Eq. (4.10) the prime notation has been used to distinguish the condition c'_m to be verified from the condition c_m used to construct the spatial locator.

Constraints $P_C(e,t)$ can be collectively applied to all elements in set S to form requirement R.

$$R(t) = \left[\forall e \in S(t),\ P_C(e,t) = true\right] \tag{4.11}$$

It is interesting to note that the quantifier $\forall$ (any) is used in Eq. (4.11). Using FORM-L, it is therefore possible to express requirements that involve quantifiers. Other quantifiers can be used as well to express that a property should be satisfied by at least, or at most one or more elements in set S.

$$\exists(\geq n)\, e \in S \,|\, P_C(e,t) = true \tag{4.12}$$

$$\exists(\leq n)\, e \in S \,|\, P_C(e,t) = true \tag{4.13}$$

Eq. (4.12) states that there exists at least n elements of S that satisfy P_C.

Eq. (4.13) states that there exists at most n elements of S that satisfy P_C.

Note that as Σ is always finite, Eq. (4.11) is equivalent to

$$R(t) = \left[\exists(= card(S))\, e \in S \,|\, P_C(e,t) = true\right] \tag{4.14}$$

where $card(S)$ is the number of elements in S and $\exists(=n)$ means that there exist exactly n elements in the set that satisfy the required property.

Hence, the quantifiers that are needed to express all types of requirements are $\exists(\geq n)$, $\exists(\leq n)$ and $\exists(=n)$. $\exists$ is a shorthand notation for $\exists(\geq 1)$. $\forall$ is a shorthand notation for $\exists(= card(.))$.

4.5. FORM-L is linked to Modelica using bindings

FORM-L dynamic external variables are evaluated using bindings to Modelica.

As already emphasized, set S introduced in Eq. (4.6) is defined in intention. That means that $card(S)$ can be computed only when properties $P(e,t)$ may be evaluated.

To evaluate $P_S(e)$ (cf. Eq. (4.7)), S must be replaced by the set S_A that contains the elements of the architectural model Σ_A that represent the design choices for S. As the properties to be fulfilled by the elements of S_A are the same as the properties to be fulfilled by the elements of S, S is constructed by replacing Σ by Σ_A in Eq. (4.6):

$$S_A(t) = \left\{e \in \Sigma_A \,|\, P(e,t) = true\right\} \tag{4.15}$$

To evaluate $P_T(e,t)$ (cf. Eq. (4.8)), the values of the dynamic attributes $e.y_m$ of elements e must be computed using the behavioral model $\mathrm{M}(\Sigma_A)$ chosen to represent the dynamic behavior of Σ_A. As for Eq. (4.3), the values of the dynamic attributes are given by:

$$e.y_m = H_{y_m}(\hat{e}.x_m, \hat{e}.a_m) \tag{4.16}$$

where $\hat{e}$ is the model component or the vector of model components in $\mathrm{M}(\Sigma_A)$ chosen to represent the dynamic behavior of element e in Σ_A.

Notice that in Eq. (4.16), H_{y_m} does not depend neither on e nor on $\hat{e}$. It only depends on y_m. Therefore the same observation operator can be used for a given quantity to be observed, whatever

elements e and $\hat{e}$. It is therefore possible to build libraries of observation operators that can be reused.

The value of $P_T(e,t)$ depends on the choice of $\mathrm{M}(\Sigma_A)$ and is denoted $\hat{P}_T(e,t)$. However, if $\mathrm{M}_1(\Sigma_A)$ and $\mathrm{M}_2(\Sigma_A)$ are two different correct behavioral models for Σ_A, then the values of $P_T(e,t)$ should be very close to each other for both models. That is why, and for the sake of simplicity, the notation $\hat{P}_T(e,t)$ does not recall the choice for $\mathrm{M}(\Sigma_A)$.

As for Eq. (4.3), $\hat{P}_T(e,t)$ must be completed by the condition P_{y_m} that ensures that $e.y_m$ may be evaluated. Notice that P_{y_m} depends on y_m, does not depend on e, and its evaluation depends on $\mathrm{M}(\Sigma_A)$.

$$\hat{P}_T(e,t) = \left[P_{y_m}(\hat{e}.x_m, \hat{e}.a_m) < 0 \ \wedge P_T(e,t)\right] \tag{4.17}$$

Similarly, the value of $P_C(e,t)$ corresponding to $\mathrm{M}(\Sigma_A)$ is denoted $\hat{P}_C(e,t)$ must be completed by the condition P_{y_m} that ensures that $e.y_m$ may be evaluated.

$$\hat{P}_C(e,t) = \left[P_{y_m}(\hat{e}.x_m, \hat{e}.a_m) < 0 \ \wedge P_C(e,t)\right] \tag{4.18}$$

Then $P(e,t)$, $S_A(t)$ and $R(t)$ may be evaluated:

$$\hat{P}(e,t) = \left[P_S(e) \wedge \hat{P}_T(e,t)\right] \tag{4.19}$$

$$\hat{S}_A(t) = \left\{e \in \Sigma \middle| \hat{P}(e,t)) = true\right\} \tag{4.20}$$

$$\hat{R}(t) = \left[\forall e \in \hat{S}_A(t),\ \hat{P}_C(e,t) = true\right] \tag{4.21}$$

In Eq. (4.21), quantifier $\forall$ may be replaced by any of the other quantifiers $\exists(\geq n)$, $\exists(\leq n)$ and $\exists(=n)$.

The role of the bindings is to associate (1) S_A to S, (2) $\hat{e}$ to e, and (3) y_m, x_m, a_m to H to form the equation $y_m = H_{y_m}(x_m, a_m)$.

The relation between S_A and S corresponds to the operator *bind_set* introduced in §3.4.2. It is denoted $S \leftarrow S_A$ to recall that all elements of S_A are assigned to S.

The relation between $\hat{e}$ and e corresponds to the operator *bind_instance* introduced in §3.4.4. It is denoted $e \leftarrow \{\hat{e}\}$ to recall that several $\hat{e}$ may be bound to one e.

The relations between y_m and H_{y_m} on the one hand, and x_m, a_m and H on the other hand to form the equation $y_m = H_{y_m}(x_m, a_m)$ correspond respectively to the operators *bind_variable* and *bind_input* introduced in §3.4.3. They are respectively denoted $y_m \leftarrow \left[H_{y_m}\right]$ and $H_{y_m} \leftarrow (x_m, a_m)$.

4.6. FORM-L uses 3-valued logic

FORM-L uses 3-valued logic: *true*, *false* and *undefined*.

The value of requirement $R(t)$ is given by Eq. (4.21). In the following equation, $\hat{R}(t)$ is re-expressed showing explicitly the time locator Δ_{DC} for R:

$$\hat{R}(t) = \left[\forall t \in \Delta_{DC} \wedge \forall e \in \hat{S}_A(t),\ \hat{P}_C(e,t) = true\right] \tag{4.22}$$

In Eq. (4.22), quantifier $\forall$ in front of e may be replaced by any of the other quantifiers $\exists(\geq n)$, $\exists(\leq n)$ and $\exists(=n)$.

For instants $t \notin \Delta_{DC}$, $\hat{P}_C(e,t)$ is not evaluated and $\hat{R}(t)$ is undefined. Otherwise, when $\hat{P}_C(e,t)$ may be evaluated, $\hat{R}(t)$ evaluates to true or false.

The value of $\hat{R}(t)$ is the accumulated value of $\hat{P}_C(e,t)$ over set $\hat{S}_A(t)$. This is similar to integration in space, where the classical integration over a continuous variable is replaced by a discrete accumulated value over a 3-valued Boolean variable.

$$\hat{R}(t) = \int_{\forall e \in \hat{S}_A(t)} \hat{P}_C(e,t)\ de \tag{4.23}$$

In Eq. (4.23), quantifier $\forall$ in front of e may be replaced by any of the other quantifiers $\exists(\geq n)$, $\exists(\leq n)$ and $\exists(=n)$.

If for a given instant $t \in \Delta_{DC}$ $\hat{P}_C(e,t)$ is false for at least one element $e \in \hat{S}_A(t)$, then $\hat{R}(t)$ is false. Else if for a given instant $t \in \Delta_{DC}$ $\hat{P}_C(e,t)$ is undefined for all elements $e \in \hat{S}_A(t)$, then $\hat{R}(t)$ is undefined.

Else $\hat{R}(t)$ is true, i.e. if for a given instant $t \in \Delta_{DC}$ $\hat{P}_C(e,t)$ is true for at least one element $e \in \hat{S}_A(t)$ and is not false for all elements $e \in \hat{S}_A(t)$.

$$t \notin \Delta_{DC} \Rightarrow \hat{R}(t) = undefined \tag{4.24}$$

$$t \in \Delta_{DC} \wedge \exists e \in \hat{S}_A(t) \left| \hat{P}_C(e,t) = false \Rightarrow \hat{R}(t) = false \right. \tag{4.25}$$

$$t \in \Delta_{DC} \wedge \forall e \in \hat{S}_A(t),\ \hat{P}_C(e,t) = undefined \Rightarrow \hat{R}(t) = undefined \tag{4.26}$$

$$t \in \Delta_{DC} \wedge \exists e \in \hat{S}_A(t) \left| \hat{P}_C(e,t) = true \wedge \forall e \in \hat{S}_A(t),\ \hat{P}_C(e,t) \neq false \Rightarrow \hat{R}(t) = true \right. \tag{4.27}$$

Equations (4.25) through (4.27) should be adapted to the other possible quantifiers $\exists(\geq n)$, $\exists(\leq n)$ and $\exists(=n)$ in front of e in Eq. (4.23).

$\hat{R}(t)$ is the value of requirement R at instant t. This value is obtained while simulating the model $\mathrm{M}(\Sigma_A)$, so it depends on the past values of the system state $\hat{e}.x_m$ for all elements $e \in \hat{S}_A$. If the fsa in Eq. (4.4) does not feature any stochastic transitions, then the initial state $\hat{e}.x_m(t=0)$ determines all future states $\hat{e}.x_m(t)$ and $\hat{R}(t)$ depends only on the initial state $\hat{e}.x_m(t=0)$. But if the fsa features stochastic transitions, then $\hat{R}(t)$ depends on other past states than the initial state.

In the following, a trajectory is defined as the set in time of all values $\hat{e}.x_m(t)$ for all elements $e \in \hat{S}_A(t)$ during a simulation run between initial instant t_0 and final instant t_f:

$$T_A(t_0,t_f) = \left\{\forall t \in \left[t_0,t_f\right] \wedge \forall e \in \hat{S}_A(t),\ \hat{e}.x_m(t)\right\} \tag{4.28}$$

To emphasize that $\hat{R}$ is constrained by the trajectory of the system, $\hat{R}$ is expressed as a function of x instead of t, where x represents the state $\hat{e}.x_m(t)$ for all elements $e \in \hat{S}_A(t)$:

$$\begin{aligned} \hat{R}(t) &\equiv \hat{R}(x(t)) \\ x(t) &\equiv \left\{\ \hat{e}.x_m(t)\right\}_{e\in\hat{S}_A} \end{aligned} \tag{4.29}$$

Then the value of requirement R over trajectory T_A is defined as the accumulated value of $\hat{R}(x)$ over that trajectory:

$$\hat{R}(T_A) = \int_{T_A} \hat{R}(x)\ dx \tag{4.30}$$

Eq. (4.30) shows that the value of requirement R depends on the trajectory T_A of the system for a given architecture Σ_A.

The computation of $\hat{R}(T_A)$ from the values of $\hat{R}(x)$ as given by Eq. (4.30) follows the same rules as the computation of $\hat{R}(t)$ from $\hat{P}_C(e,t)$ (cf. equations (4.25) through (4.27)):

$$\exists t \in \left[t_0,t_f\right] \left| \hat{R}(t) = false \Rightarrow\ \hat{R}(T_A) = false \right. \tag{4.31}$$

$$\forall t \in \left[t_0,t_f\right],\ \hat{R}(t) = undefined \Rightarrow \hat{R}(T_A) = undefined \tag{4.32}$$

$$\exists t \in \left[t_0,t_f\right] \left| \hat{R}(t) = true\ \wedge \forall t \in \left[t_0,t_f\right],\ \hat{R}(t) \neq false \Rightarrow\ \hat{R}(T_A) = true \right. \tag{4.33}$$

For a given trajectory T_A :

- R is *satisfied* for trajectory T_A if $\hat{R}(T_A)$ is true.
- R is *violated* for trajectory T_A if $\hat{R}(T_A)$ is false.
- R is not *violated* for trajectory T_A if $\hat{R}(T_A)$ is true or undefined.
- R is *undecided* for trajectory T_A if $\hat{R}(T_A)$ is undefined.

More globally, for a given architecture Σ_A :

- R is *satisfied* for architecture Σ_A if $\hat{R}(T_A)$ is true for all possible trajectories T_A. This is of course impossible to verify when the number of possible trajectories is infinite.
- R is *violated* for architecture Σ_A if $\hat{R}(T_A)$ is false for at least one trajectory T_A.
- R is *not violated* for architecture Σ_A if no trajectory T_A could be produced such as $\hat{R}(T_A)$ is false.

Hence:

- R is *violated* for trajectory T_A implies that R is *violated* for architecture Σ_A.
- R is *not violated* for architecture Σ_A is equivalent to R *satisfied* for architecture Σ_A only in

the very special case where the number of trajectories is finite (systems having only deterministic transitions between a finite number of discrete states).

So in practice, the best proof achievable to show compliance of design vs. requirement R is the non-violation of requirement R. To increase confidence in the proof, a measure of the test coverage should be added, in particular to detect unfavorable cases where non-violation of R is obtained from trajectory computations that yield frequent undecided evaluations. Test coverage metrics are not discussed here.

5. Conclusion and future work

A new modeling architecture has been presented to support a model-based verification method for complex physical systems such as power plants. It is technologically based on the open source language Modelica and a new extension of this language called FORM-L. However, the principles of the method do not depend on those languages, so may be adapted to other technologies.

This new modeling architecture enables the separate modeling of the system's requirements on the one hand and the system's dynamic behavior on the other. The architectural model that captures the design choices for the system and the observation operators that translate physical quantities into functional ones provide the bridges between the two models, and enable to generate automatically the verification model. The system design can then be tested against the system requirements by simulating the behavioral model with the requirements model as an automatic observer to detect possible violations of the requirements in the course of the simulation. Automatic testing enables to increase the test coverage, and potentially solve the problem of overcoming the possible combinatorial explosion of operating modes to be explored.

It is of course impossible to systematically explore all possible situations, as their number is infinite. However, in order to explore all potential situations of interest, whose number may still be very high, it is necessary to have well planned experience scenarios. It would be of great interest to use the requirements model to automatically produce those scenarios which optimize to the extent practical the test coverage. This is a direction for future work.

In principle, two solutions are possible regarding the implementation of FORM-L: FORM-L as an extension of Modelica or as a separate language from Modelica. The first solution has been chosen at the MODRIO WP2 workshop in Munich held on June 13, 2013.

References

[1] Modelica Association "Modelica® - A Unified Object-Oriented Language for Systems Modeling - Language Specification - Version 3.3," May 2012.
Available: https://www.modelica.org/documents/ModelicaSpec33.pdf

[2] OMG, SysML. Available: http://www.omgsysml.org/

[3] Object Constraint Language (OCL), ISO/IEC 19507, May 2012.

[4] Feiler P. H., Gluch D. P. and Hudak J., "The Architecture Analysis & Design Language (AADL): An Introduction," Technical Note CMU/SEI-2006-TN-011, Carnegie Mellon University, 2006. Available: http://www.sei.cmu.edu/reports/06tn011.pdf

[5] Yang Z., Hu K., Ma D., Bodeveix J-P., Pi L., "From AADL to Timed Abstract State Machine: A Certified Model Transformation," J. of Systems and Software, Elsevier, 2014, pp.20.

[6] Schamai W., "Model-Based Verification of Dynamic System Behavior against Requirements," Dissertation Thesis No. 1547, Linköping University, October 2013.

[7] Bruel G. and Jardin A., "OPENPROD Project - WP6 - I&C functional validation based on the modelling of requirements and properties: evaluation of ModelicaML," EDF R&D Technical Report No. H-P1A-2012-03040-EN, Feb. 2013.

[8] Baier C. and Katoen J.-P., "Principles of Model Checking," The MIT Press, 2008. Available: http://is.ifmo.ru/books/_principles_of_ model_checking.pdf

[9] Nguyen T., "MODRIO D2.1.1 – Modelica Extensions for Properties Modelling - FOrmal Requirements Modelling LAnguage (FORM-L)," MODRIO deliverable D2.1.1, internal EDF R&D report H-P1A-2014-00535-EN, to be released.

[10] Nguyen T., "FORM-L: A Modelica Extension for Properties Modelling, Illustrated on a Practical Example," 10th international Modelica conference proceedings, Lund, March 2014.

[11] Bouskela D., Nguyen T. and Jardin A., "Novel Open Source Modelling Architecture for the Design Verification against System Requirements using Simulation," 57th ISA POWID Division Symposium, conference proceedings, Scottsdale, June 2014.

[12] Coppolani P., "La chaudière des réacteurs à eau sous pression," INSTN, EDP Sciences, 2004.

[13] El Hefni B., Bouskela D. and Lebreton G., "Dynamic modelling of a combined cycle power plant with ThermoSysPro," 8th international Modelica conference proceedings, Dresden, March 2011.

[14] Wylie E. B. and Streeter V. L., "Fluid Transients in Systems," Prentice Hall, 1993.

[15] Elmqvist H., Mattsson S. E. and Otter M., "Modelica extensions for Multi-Mode DAE Systems," 10th international Modelica conference proceedings, Lund, March 2014.

[16] Jardin A., "WP2 workshop, MUNICON, Munich, June 13th, 2013," MODRIO meeting minutes, June 2013.