D8.1.3 – Power plant safety scenario Part II: The Intermediate Cooling System (SRI) Case Study

WP8.1 – Power Plants
WP8 – Demonstrators

MODRIO (11004)

Version 1.0

Date 02/05/2016

Authors

Audrey JARDIN EDF

Thuy NGUYEN EDF

Executive summary

This document is Part II of deliverable MODRIO D8.1.3 *Power Plant Safety Scenario*. It describes an example of property modelling based on an Intermediate Cooling System (*SRI*, in French *Système de Refroidissement Intermédiaire*). This system is much simpler in terms of control logic than the Backup Power Supply (*BPS*) presented in [Thuy Nguyen, "MODRIO D8.1.3 Part I - The Backup Power Supply (BPS) Case Study".], but has more physical aspects.

The main objective of the SRI case study is to illustrate an example with more physics than in the BPS example and to show how the FORM-L methodology can be (to some extent) translated in a Modelica environment.

The methodology followed is a much simplified version of the one proposed in [Thuy Nguyen, "MODRIO D2.1.2 Properties modelling method".]. The modelling is done with three separate models in conformance with the architecture proposed in [Bouskela D., "MODRIO D2.1.1 Part II - Modeling Architecture for the Verification of Requirements", also available as EDF technical report, H-P1C-2014-15188-EN, 2015.]: the first model is a property model expressing the system requirements, the second model is a Modelica architecture model describing the system overall design and the third model is a Modelica behavioural model representing the system detailed design.

The property model for system requirements is initially developed based on the FOrmal Requirements Modelling Language (FORM-L) proposed in [Thuy Nguyen, "MODRIO D2.1.1 Part III - FOrmal Requirements Modelling Language (FORM-L)", also available as EDF technical report, H-P1A-2014-00550-EN.].

Then the translation in the Modelica environment uses results presented in [Otter M. et al., "MODRIO D2.1.1 Part IV – Modelica for properties modelling". In particular the system requirements expressed in the property model are implemented in Modelica using the *Modelica_Requirements* library, and then bound to the architecture and the behavioural models thanks to discussions conducted about model bindings (or mappings) to produce the overall verification model.

Finally this verification model is simulated to check whether requirements are violated, satisfied or even not tested.

As a perspective (after the end of MODRIO), it is planned to continue the work at EDF on this case study. Some aspects will be completed in particular to refine the system requirements model and test more simulation scenarios. Even if promising, the *Modelica_Requirements* will also be improved (enhancements of the automation of bindings, handling of sets and overlapping time periods, possibility to express requirements with probabilities…) and the development of a standalone FORM-L compiler will be invested to target other environments than the ones based on Modelica.

Summary

Acronyms

BPS	Backup Power Supply
CTL	Continuous Time Locator
	Computation Tree Logic
FORM-L	FOrmal Requirements Modelling Language
LTL	Linear Temporal Logic
SRI	Intermediate Cooling System
WP	Work Package

Glossary

Assumption	Property that is supposed to be satisfied: simulation scenarios assume / ensure that it is satisfied.
Conditional Probability	***Real*** value (not a function) in the [0., 1.] range that states the probability of an event occurring at least once during a specified time period (a CTL). The time period specifies the condition
Continuous Time Domain	Time domain where time is perceived continuously. There is only one continuous time domain
Continuous Time Locator	Expression that specifies one or more time intervals
Contract	Property model specifying the mutual obligations of two or more parties.
Discrete Time Domain	Time domain where time is perceived not continuously but at specific, defined instants. Different discrete time domains may specify different sets of instants
Discrete Time Locator	Expression that specifies one or more time instants
Event	Named expression that characterises the occurrences of a fact that has no duration
Event-Based Property	An event-based property states that an event does or does not occur at a specified time locator, possibly with a count constraint
Finite State Automaton	Function of time the values of which belong to an enumerated set
Guard	Property that states the conditions that must be satisfied for a model to be valid
Probability	Function of ***time***, the result of which is a **real** value in the [0., 1.] range. Its value represents the probability that a given ***event*** has already occurred at least once
Property	Expression of something that is desirable, expected or required.
Requirement	Property that must be satisfied: it is the objective of simulation to verify that it is not **violated**
Time Domain	Part of a model where objects have the same perception of time
Time Domain Interface	Part of a model where events and variables in one time domain is made perceptible by objects belonging to another time domain
Time Instant	Position in time that has no duration
Time Interval	Period of time that has a duration (as opposed to an event, which is instantaneous and has no duration)
Time Locator	See *Continuous Time Locator* and *Discrete Time Locator*

1. Introduction

1.1. Context

This document is Part II of deliverable MODRIO D8.1.3 *Power Plant Safety Scenario*. It describes an example of property modelling based on an Intermediate Cooling System (*SRI*, in French *Système de Refroidissement Intermédiaire*). This system is much simpler in terms of control logic than the Backup Power Supply (*BPS*) presented in [16], but has more physical aspects.

The methodology followed is a much simplified version of the one proposed in [14]. The modelling is done with three separate models in conformance with the architecture proposed in [12]: the first model is a property model expressing the system requirements, the second model is a Modelica architecture model describing the system overall design and the third model is a Modelica behavioural model representing the system detailed design.

The property model for system requirements is initially developed based on the FOrmal Requirements Modelling Language (FORM-L) proposed in [24]. Then the translation in the Modelica environment uses results presented in [13]. In particular the system requirements expressed in the property model are implemented in Modelica using the *Modelica_Requirements* library, and then bound to the architecture and the behavioural models thanks to discussions conducted about model bindings (or mappings) to produce the overall verification model. Finally this verification model is simulated to check whether requirements are violated, satisfied or even not tested.

As a perspective (after the end of MODRIO), it is planned to continue the work at EDF on this case study. Some aspects will be completed in particular by refining the system requirements model and by testing more simulation scenarios. Even if promising, the *Modelica_Requirements* will also be improved (enhancements of the automation of bindings, handling of sets and overlapping time periods, possibility to express requirements with probabilities…) and the development of a standalone FORM-L compiler will be invested to target other environments than the ones based on Modelica.

1.2. Objectives

The *SRI* case study has two main objectives. The first one is to serve as a self-consistent public case study that can be freely communicated and easily understood even by readers who did not participate in the MODRIO project. The second goal of the case study is to provide a concrete example with more physics than in the BPS example and to show how the FORM-L methodology can be (to some extent) translated in a Modelica environment.

1.3. Structure of the document

Section 2 presents an overview of the *SRI*, of its environment and of its system requirements. It then provides a detailed presentation of the modelling of the system requirements in FORM-L.

Section 3 presents the translation of the system requirements modelled in FORM-L into Modelica, using the *Modelica_Requirements* library.

Section 4 presents an overview of the SRI overall design and how such an architecture model has been coded in Modelica.

Section 5 presents an overview of the SRI detailed design and how such behavioural model has been coded in Modelica, using the ThermoSysPro library, an EDF open source library to model energy system and power plants [19].

Section 6 shows how the three different separate models (requirements, architecture and behavioural) have been connected to produce verification models and run some simulations to check whether the requirements are violated or not.

1.4. Notation

FORM-L statements are written in a **`bold Courier New`** font.

FORM-L keywords are in ***blue italics***.

The names of property models, of classes and types begin with an upper-case letter.

The names of objects and variables begin with a lower-case letter.

Event names begin with an 'e'.

Names for desirable properties begin with a 'p', whereas names for requirements begin with an 'r'.

The names of configurations are in upper-case.

The use of a predefined FORM-L library defining physical constants of various types is assumed. For example:

- `1*s` 1 second
- `10*s` 10 seconds
- `1*h` 1 hour
- `100*ms` 100 milliseconds
- `1000*kw` 1000 kilowatts

2. *SRI* System Requirements in FORM-L

2.1. The *SRI* and its Environment

The *SRI* is a power plant system that is used to evacuate the heat produced by a number of "clients". These clients are other plant systems that produce heat when in operation, and that need to be cooled using clean, demineralised water: they cannot be cooled by water coming directly from the cold source (e.g., the sea, a lake or a river), which could damage them. Figure 2-1 places the *SRI* within its environment. In addition to a source of cold water (*SEN*), a source of demineralised water (*SED*), a sewer collecting leaks (*SEK*) and a human operator, three clients (*client1*, *client2* and *client3*) have been identified. Each has specific expectations with respect to the *SRI*.

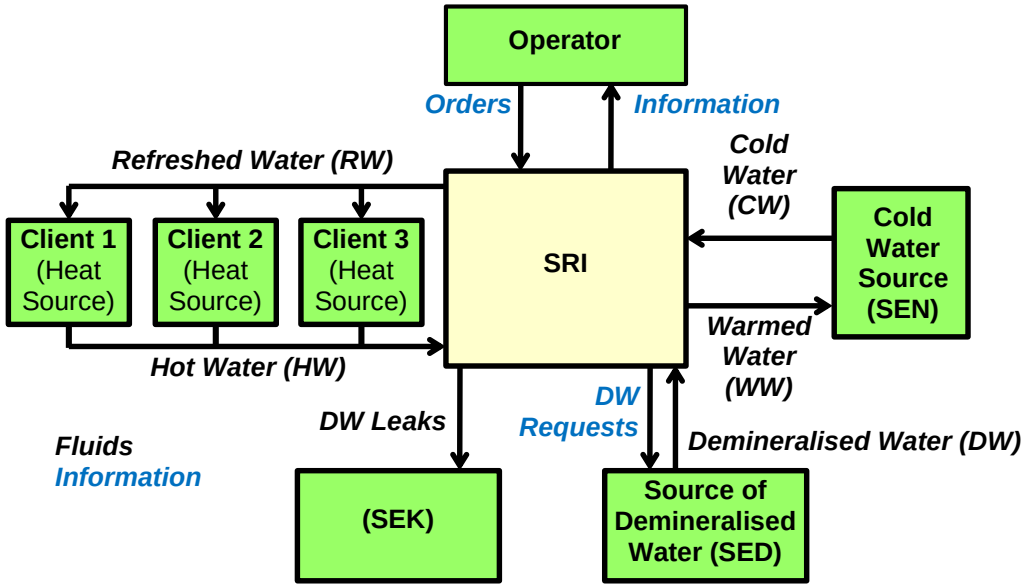

Figure 2-1
The SRI and its environment. The entities constituting the environment are shown in green. The BPS itself is shown in a yellow box.

For the purpose of this demonstrator, only a subset of the SRI requirements have been modelled. In particular, failures have not been taken into consideration.

2.2. Modelling Organisation

The modelling organisation for the *SRI* system requirements is summarised in Figure 2-2. It is based on:

- Object model *sri*, which represents the *SRI*.
- Object model *sen*, which represents the source of cold water *SEN*.
- Object model *sed*, which represents the source of demineralised water *SED*.
- Object model *sek*, which represents *SEK*, the sewer collecting water leaks.
- Object model *op*, which represents the human operator.
- Class model *Client* specifies the general characteristics of an *SRI* client. It has three instances (*client1*, *client2* and *client3*) representing the three clients of the *SRI*.

In addition, there are the following contracts:

- Contract *hsi* between *SRI* and the human operator.
- Contract *coldW* between the *SRI* and the *SEN*.
- Contract *deminW* between the *SRI* and the *SED*.
- Contracts *cool1*, *cool2* and *cool3* between the SRI and its three clients. These contracts are based on standard contract *Cool*.

The *sen*, *sed*, *sek*, *client1*, *client2* and *client3* models could simply be surrogate models or generic scenario models: they just need to satisfy the assumptions of the *SRI* regarding the real world entity they represent, as expressed in the contracts.

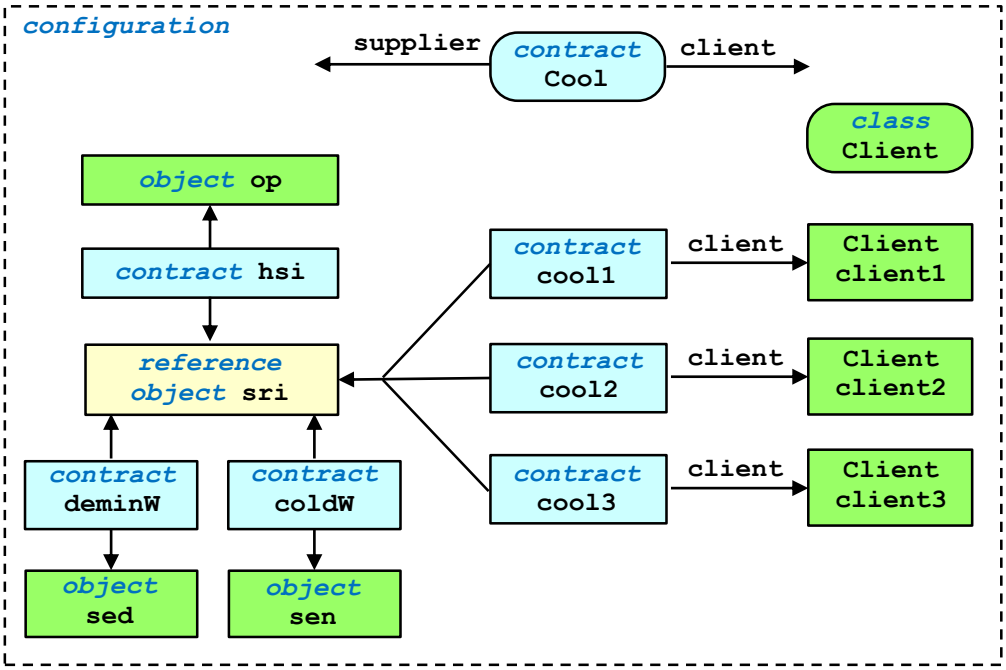

Figure 2-2
Overall organisation of the SRI system requirements modelling. The reference object model is shown in yellow, the other object models in green, and the contracts in blue. Classes and standard contracts are shown with round corners.

2.3. Class *Client*, *SRI* Clients, *Cool* Standard Contract

When in operation, a client produces heat that must be removed. To this end, it has a heat exchanger and an isolation valve. The valve is closed when the temperature of the cooling water (*RW*) sent by the SRI is not adequate. It is under the responsibility of the client, not of the *SRI*. This is illustrated in Figure 2-3.

Class *Client*

For what concerns the *SRI*, the features of a client are:

- *powerMax*: maximum thermal power generated by the client when it is in operation.
- *temp*: the internal client temperature.
- *tMax*: maximum internal temperature allowed for the client. It is the *SRI*'s objective to maintain the client's *temp* below that temperature.
- *tRWMin*: minimum acceptable temperature of the cooling water (below that temperature, the client closes its valve). For clients 1 and 2, *tRWMin* = 15°C, but for client 3, *tRWMin* = 7°C.
- *flow*: the flow of demineralised cooling water through the client's heat exchanger.
- *flowMax*: maximum acceptable cooling water flow through the client's heat exchanger. Beyond that value, the client's heat exchanger might be damaged. Keeping the flow under that value is the responsibility of the client.
- *delay*: time during which the client is in operation and needs to be cooled after the plant has stopped. For client 1, *delay* = 0 s, but for clients 2 and 3, *delay* = 20 mn.
- A client also has a mass and a specific heat capacity determining temperature increase when the heat produced is not evacuated, but they are not used in the requirements model.

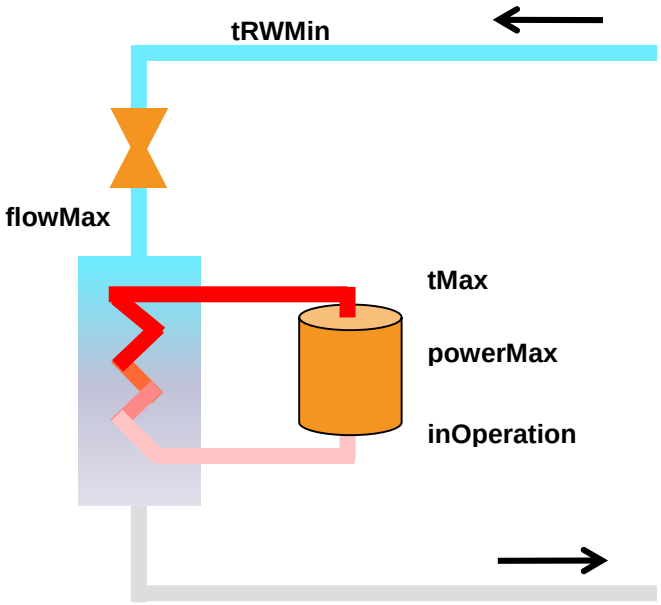

Figure 2-3
A client as viewed by the *SRI*.

```
class Client

  specific constant Power powerMax;
  specific constant Temperature tMax;
  specific constant Temperature tRWMin;
  specific constant Flow flowMax;
  specific constant Duration delay;

  external Boolean open;          // State of client's valve
  external Temperature temp;      // to be provided by the behavioural model
  external Flow flow;             // through a binding

  end Client;
```

Standard Contract *Cool*

```
contract Cool (Client client)

  party client
    // Internal temperature of client
    Temperature temp;
    constant Temperature tMax;
    requirement cl1 = check tMax isIn [50.*C, 80.*C];
    assumption cl2 = check temp < tMax;   // Requirement placed on the sri

    // Water flow through client's heat exchanger
    Flow flow;
    constant Flow flowMax;
    requirement cl3 = check flowMax > 0.; // Assumptions for the sri
    requirement cl4 = check flow isIn [0., flowMax];

    // Minimum temperature of cooling water
    Boolean open;                         // State of client's valve
    constant Temperature tRWMin;
    requirement cl5 = check tRWMin isIn [7.*C, 15.*C];
    requirement cl6 = during tRW < (tRWMin-0.1*C) check not open;
    requirement cl7 = during tRW > tRWMin check open;

    constant Power powerMax;              // Max power generated by client
    constant Duration delay;              // Operation after shutdown
    requirement cl8 = check delay isIn [0., 20.*mn];

    end client;

  party sri
    Temperature tRW;                      // Temp of water sent to client
    automaton state =
      (stopped, startupCold, startupWarm, normalOp, stopping);
    requirement cl9 =
      during startupWarm or normalOp or stopping check tRW > 15.*C;
    end supplier;

  end Cool;
```

The Three *clients*

```
Client client1

  powerMax = 20.*kW;
  tMax = 65.*C;
  tRWMin = 15.*C;
  flowMax = 100.*l/mn;
  delay = 0.*s;

  end client1;
```

```
Client client2

   powerMax = 25.*kW;
   tMax = 65.*C;
   tRWMin = 15.*C;
   flowMax = 130.*l/mn;
   delay = 20.*mn;

   end client2;

Client client3

   powerMax = 40.*kW;
   tMax = 55.*C;
   tRWMin = 7.*C;
   flowMax = 200.*l/mn;
   delay = 20.*mn;

   end client3;
```

2.4. Operator, *hsi* Contract

The operator can send two events to the *SRI*:

- *ePlantStartUp* signals that the power plant is to be started up and that the *SRI* must get active.
- *ePlantShutDown* signals that the plant is to be shut down.

In this simplified case study, the *SRI* has no obligation with respect to the operator.

```
contract hsi
   party op
      event ePlantSartUp;
      event ePlantShutDown;
   end op;

   party sri
   end sri;

end hsi;

object op                              // Surrogate model
   external event ePlantSartUp;
   external event ePlantShutDown;
   end op;
```

2.5. Source of Cold Water (*SEN*), *coldW* Contract

It is assumed that the temperature of the cold water is always above freezing and below a site-dependent value called *tCWMax*. Typically, *tCWMax* = 27.2°C.

Due to environmental regulation, the increase in temperature of the warmed water sent back to the cold source must not exceed a site-dependent value called *deltaTMax*. Typically, *deltaTMax* = 5°C.

The water of the cold source (a river, a lake or the sea) must not contaminate the demineralised cooling water. This is done by ensuring that the pressure in the demineralised cooling water circuit (*pDW*) is higher than the pressure in the cold water circuit (*pCW*). Typically, the minimum difference in pressure is 0.25 bars.

```
contract coldW

   party sen
      Temperature tCW;                          // Temp of cold water
      constant Temperature tCWMax = 27.2*C;     // Max temp of cold water
      constant Temperature deltaTMax = 5.*C;    // Max temp increase
```

```
    end sen;

  party sri
    Temperature tWW;                              // Temp of water back to SEN
    Pressure pCW;                                 // Cold water circuit
    Pressure pDW;                                 // Demin water circuit
    assumption sen1 =
      during not (stopped or stopping) check tCW isIn [0.*C, tCWMax];
    requirement sen2 =
      during not (stopped or stopping) check tWW < (tCW + deltaTMax);
    requirement sen3 =
      during startupWarm or normalOp or stopping
      check pDW > (pCW + 0.25*b);
   end sri;

  end coldW;

object sen                                 // Surrogate model
  external Temperature tCW;
  end sen;
```

2.6. Source of Demineralised Water (*SED*), *deminW* Contract

On request of the *SRI* (modelled in the form of a flow request named *dWDemand*), the *SED* must ensure a flow of demineralised water *dWFlow* that is equal to the requested flow, as long as the demand is below or equal to a maximum value *dWFlowMax*.

```
contract deminW

  party sed
    Flow dWFlow;                     // Flow of dWater really provided by SED
    requirement sed1 =
      during dWDemand ≤ dWFlowMax check dWFlow = dWDemand;
    requirement sed2 =
      during dWDemand > dWFlowMax check dWFlow = dWFlowMax;
    end sed;

  party sri
    Flow dWDemand;                   // Flow of dWater requested by SRI
    constant Flow dWFlowMax;         // Maximum possible request by SRI
    requirement sed2 = dWDemand isIn [0.*l/s, dWFlowMax];
    end sri;

  end deminW;

object sed                           // Surrogate model
  external dWFlow;
  end sen;
```

2.7. *SRI*

Operation

The *SRI* is one system among many in a power plant. It starts operating when the plant is started-up. The corresponding signal is sent to the *SRI* by the operator. A number of conditions need to be satisfied before the *SRI* can start cooling its clients, and this might take some time. Grace times are specified for the *SRI*, first for cooling some clients, and then for cooling all its clients.

A client can operate only when the SRI is cooling it, and the plant cannot be in normal operation if any of the *SRI* clients is not in operation. If the SRI fails to cool any of its clients (e.g., due to pump failure), the plant must be stopped. Then, a failure signal must be sent to the operator, who will then take any necessary action.

Most clients stop operating immediately when the plant is shut down. However, some must continue operating up to 20 mn after plant shut-down. The SRI must cool them while they are operating, even in case of pumps failure or loss of electric power.

States

The *SRI* can be in one of five operational states (see Figure 2-4):

- *stopped*: it is not operational (e.g., when the plant is shut down) and is subject to no requirements.
- *startupCold*: it is operating, but its demineralised water is too cold to cool any client, or the pressure in the demineralised water circuit is not high enough. It must first warm and pressurize the demineralised water circuit.
- *startupWarm*: it is operational, is already cooling some clients, but its water is still too cold for others. It has to continue warming the water.
- *normalOp*: it is fully operational and cooling all its clients.
- *stopping*: it is shutting down, all its active components (i.e., components that need electric power to operate) are stopped, but it still has to cool some clients using passive components only (i.e., components that do not need electric power to operate).

The SRI must not remain in the *startupCold* state for more than 1 hour. This includes the time necessary to obtain a sufficient amount of demineralised water from the *SED*, to warm that water to a temperature acceptable by some clients, and to raise the pressure of the demineralised water circuit to a sufficient level.

It must not remain in the *startupWarm* state more than 30 mn. This includes the time necessary to warm the demineralised water to a temperature acceptable to all clients.

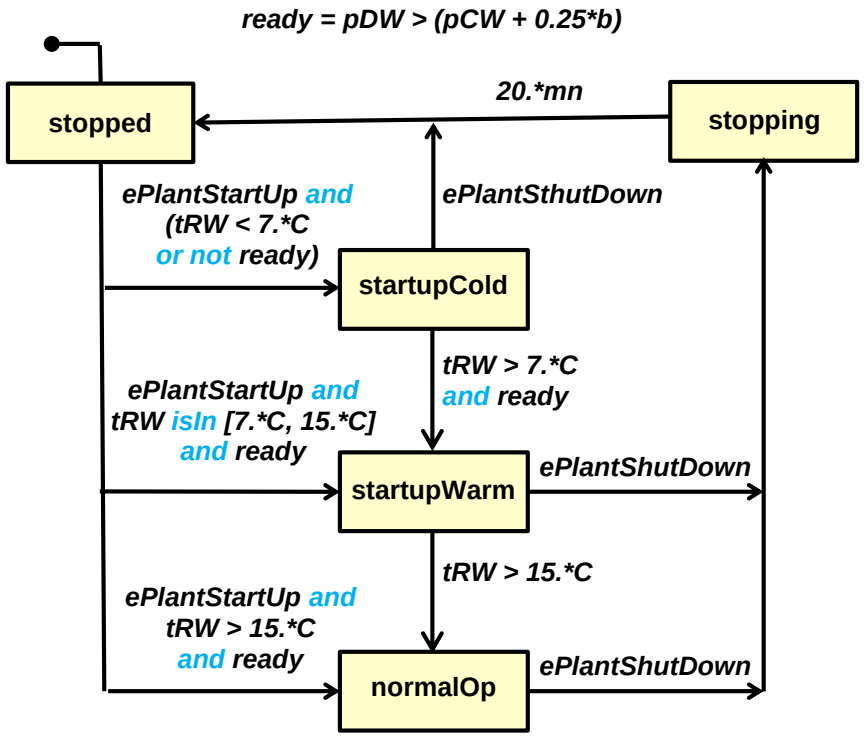

Figure 2-4
The SRI states

Class *ActiveComponent*

This class is used to represent all active *SRI* components that need electric power to operate. They should be off when the *SRI* is stopping.

```
partial class ActiveComponent
  external Boolean on;
  end ActiveComponent;
```

The *sri* Model

```
object sri

  contract hsi;                 // with operator
  contract coldW;               // with sen (source of cold water)
  contract deminW;              // with sed (source of demineralised water)
  contract cool1;               // with client1
  contract cool2;               // with client2
  contract cool3;               // with client3
  external Temperature tRW;     // Refreshed water sent to client

  external ActiveComponent{} aComponents;
  external Temperature tWW;     // Warmed water sent back to sen
  external Pressure pCW;        // Cold water circuit
  external Pressure pDW;        // Demineralised water circuit

  Boolean ready = pDW > (pCW + 0.25*b);

  automaton state = (stopped, startupCold, startupWarm, normalOp, stopping)
    start = stopped;
    when stopped and ePlantStartUp and (tRW < 7.*C or not ready)
      then startupCold;
    when stopped and ePlantStartUp and tRW isIn [7.*C, 15.*C] and ready
      then startupWarm;
    when stopped and ePlantStartUp and tRW > 15.*C and ready
      then normalOp;
    when startupCold and ePlantShutDown then stopped;
    when startupCold and tRW > 7.*C and ready then startupWarm;
    when startupWarm and ePlantShutDown then stopping;
    when startupWarm and tRW > 15.*C then normalOp;
    when normalOp and ePlantShutDown then stopping;
    when stopping becomes true + 20.*mn then stopped;
    end state;

  requirement startup1 =
    after startupCold becomes true within 1.*h
    check (startupWarm or normalOp or stopping or stopped) becomes true;

  requirement startup2 =
    after startupWarm becomes true within 30*mn
    check (normalOp or stopping or stopped) becomes true;

  requirement stopping1 =
    during stopping
    forAll c in aComponents
    check not c.on;

  end sri;
```

3. SRI System Requirements in Modelica (partial translation)

A Modelica translation of the SRI system requirements depicted in the previous section has been tested using the *Modelica_Requirements* library. This library has been developed within the framework of MODRIO deliverable D2.1.1 [13] but not all the facets of FORM-L are supported yet.

Some simplifications have hence been made. In particular:

- The three clients have been gathered into only one equivalent “big” client with no requirements on its valve’s state or on the duration it should be cooled.
- Failure are not addressed.
- Need for cooling some client even after the plant has stopped is not taken into consideration.
- Requirements on the SRI *stopping*, *startupCold* and *startupWarm* states have not been modelled.
- Since the notion of contracts does not exist in the library, the assumptions made by the *SRI* and the requirements it must satisfy that result from contracts to which the *sri* model is a party are directly imported in the *sri* or the *client* requirement model.
- Since assumptions and requirements are implemented in the *Modelica_Requirements* library with the same kinds of blocks. They have been distinguished only graphically by adding some comments in the models.

3.1. Client requirements model

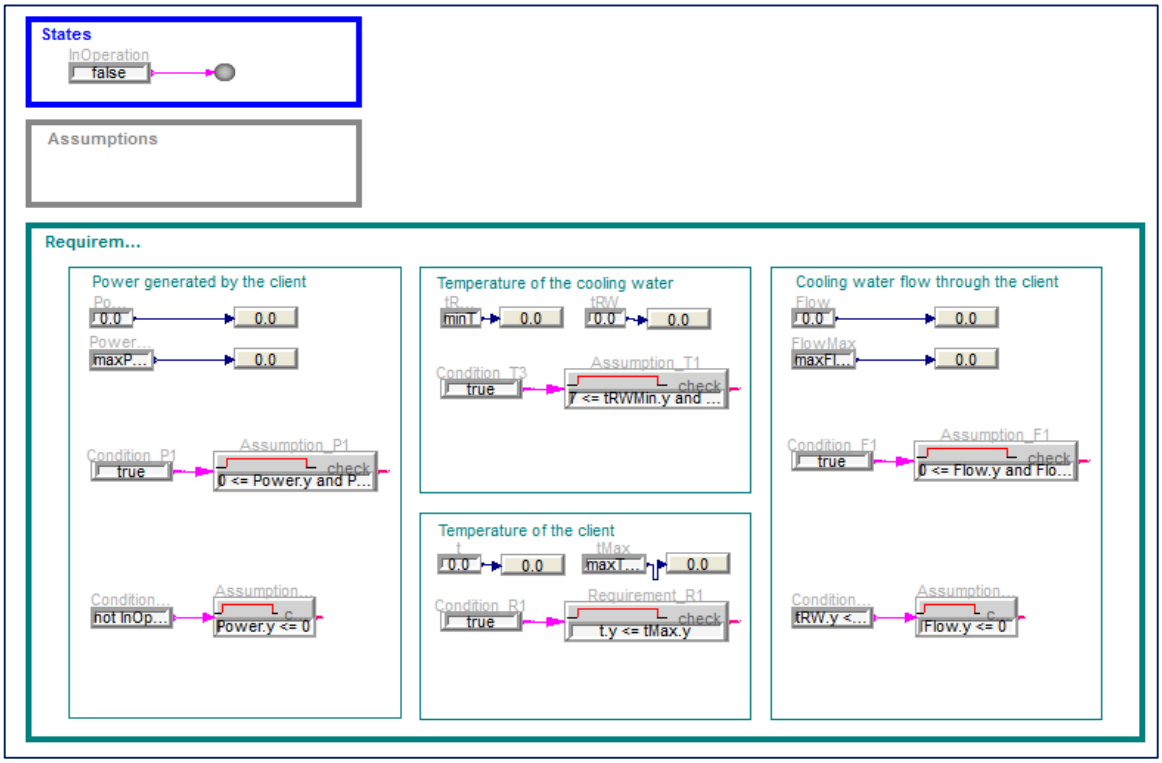

Figure 5: Partial implementation of the client *requirements model using the Modelica_Requirements library*

3.2. SRI requirement model

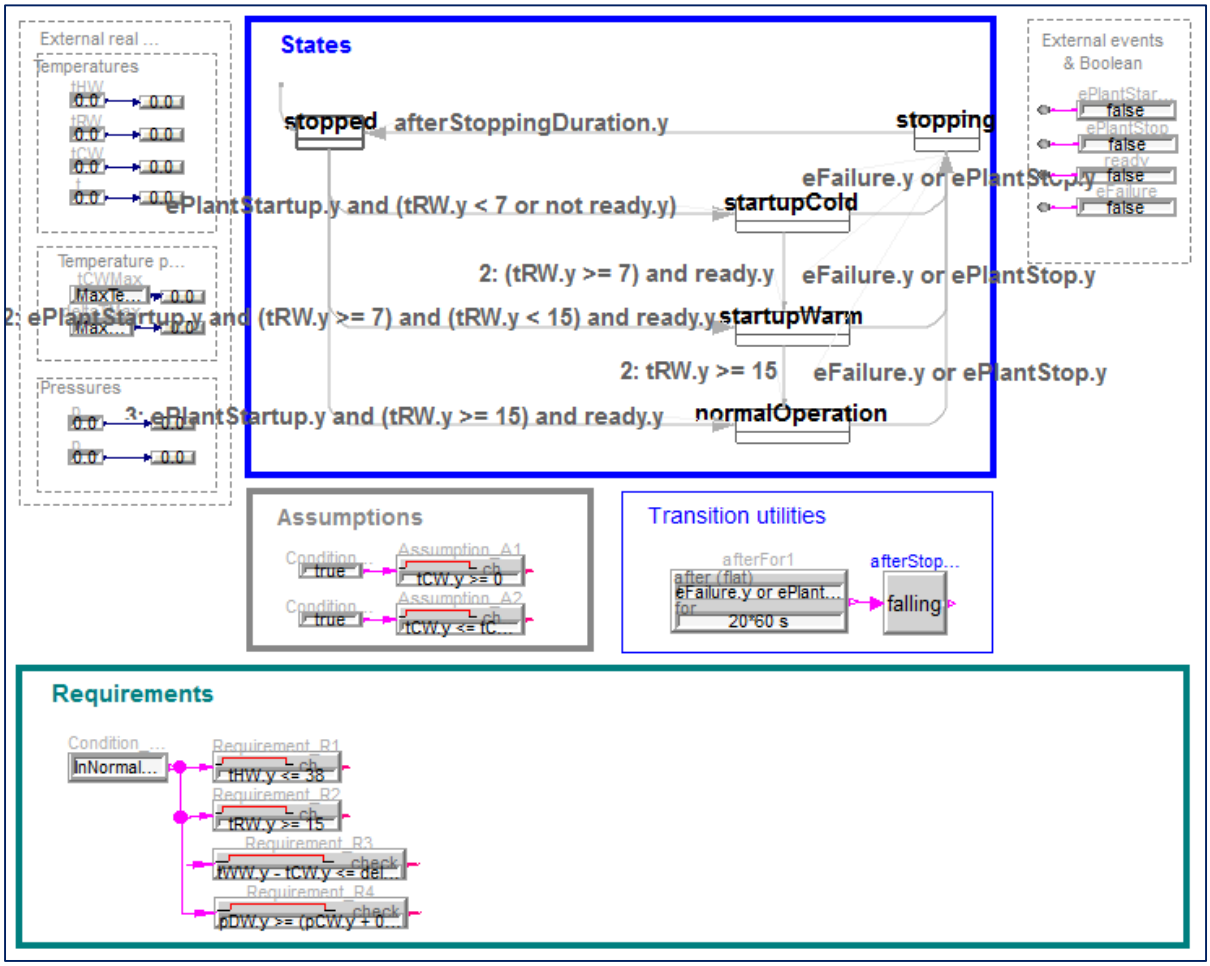

Figure 6: Partial implementation of the sri *requirements model using the Modelica_Requirements library*

At the bottom of the *sri* requirements model, one can hence find the Modelica translation of the following statements:

- Temperature of the cooling water (sent to the clients) should be maintained above 15°C.
- Temperature of the hot water (returned to the clients) should not exceed 38°C.
- Pressure in the demineralized water circuit should be higher than the pressure in the cold water circuit by at least 0.25 bar.
- Increase in the temperature of the warmed water sent back to the cold source should not exceed 5°C.

4. SRI Overall Design: Architectural Modelling in Modelica

Once the system requirements has been stated, the next step is for designers to determine one or more architectures. An architecture is expressed in terms of components constituting the system, and of assumptions placed on the components and their interactions. In the first design stages, the components are themselves considered as black boxes, and the modeling can be made with FORM-L.

In the case of the *SRI*, the architecture determines whether pumps are used, and if so, their number (for example to comply with fault-tolerance and availability requirements) as shown in Figure 7.

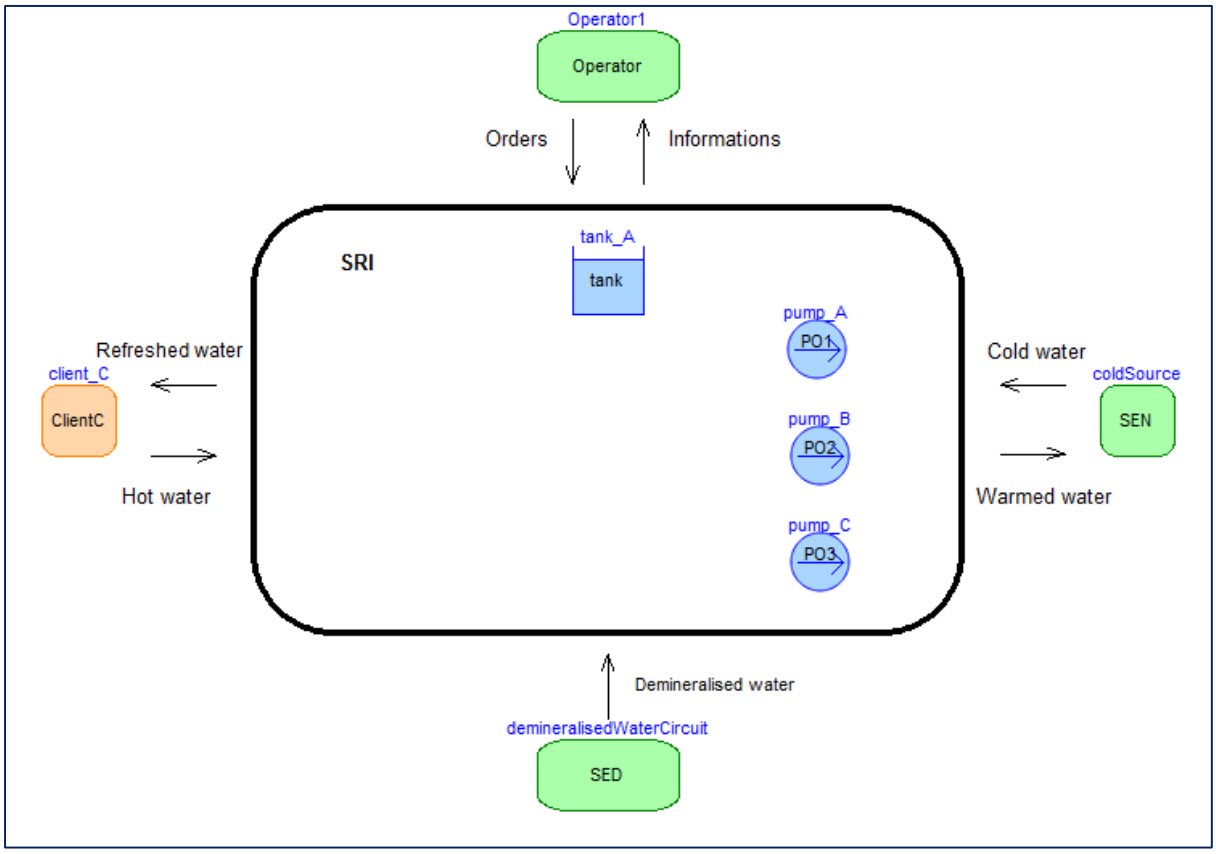

Figure 7: Possible sri *architectural model (overall design) coded in pure Modelica*

Since such architecture model simply lists the different components with their own characteristics (parameters), it can be easily translated from FORM-L to pure Modelica as shown on the example of a pump architecture model in Figure 8.

```
block Pump_A "Architectural model of a pump"
  extends CoolingSystem_SRI.Architectures.Lib.PartialArchitecture;

  parameter String id;
  parameter String component_type = "Pump";
  parameter String subtype;

  parameter Modelica.SIunits.Conversions.NonSIunits.Pressure_bar NPSH_req = 2;
  input Boolean cavitate = false;
  input Boolean inOperation = true;

  a
end Pump_A;
```

Figure 8: Example of a pump *architecture model coded in Modelica*

Please note that this kind of architecture model does not describe the behaviour of the different components in details and are also sometimes denoted by overall design model.

5. SRI Detailed Design: Behavioural Modelling in Modelica

At some point in design refinement, it will be possible to represent the design by a behavioural model, e.g. based on Modelica and/or block diagrams. This model will depict the physics of the system and hence precise the system detailed design.

In the case of the *SRI*, such a behavioural model has been developed in Modelica using ThermoSysPro, an EDF open source library which is daily used to model energy systems and power plants [19].

A picture of this model can be seen in Figure 9.

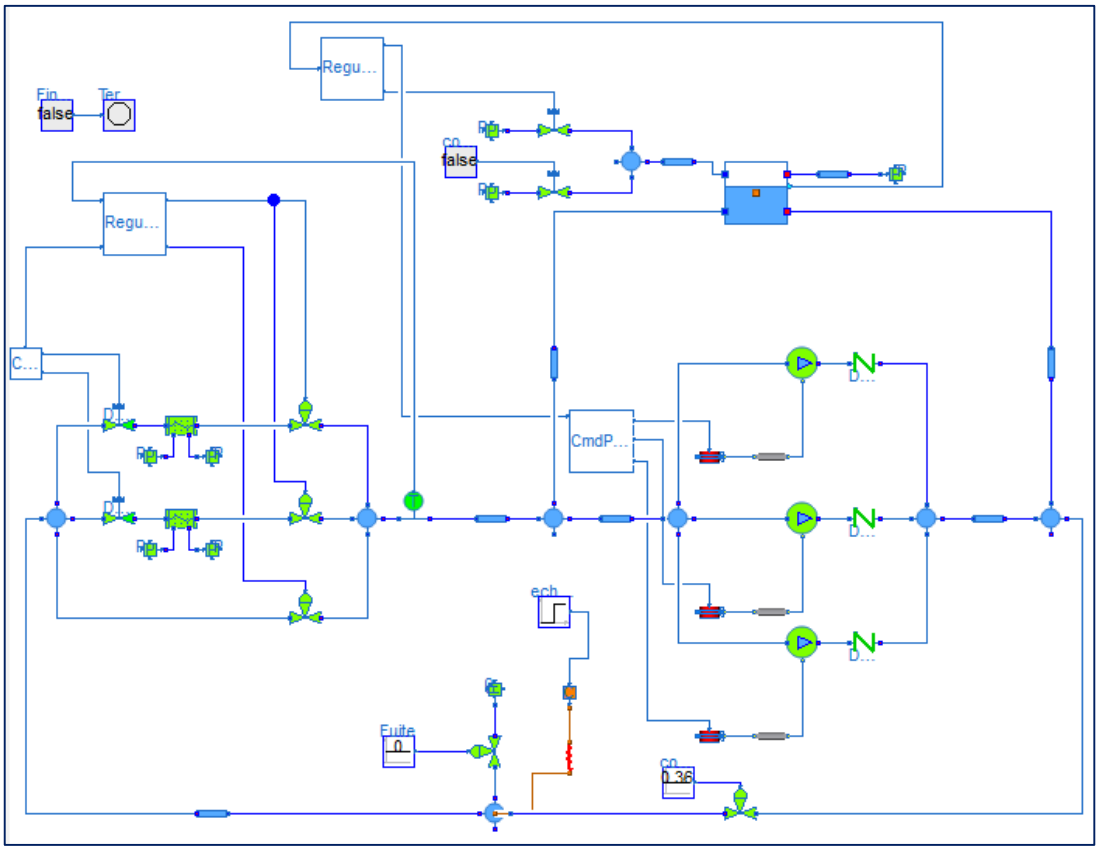

Figure 9: SRI behavioural model (detailed design) using the ThermoSysPro library

6. Verification Models and Tests Scenarios

6.1. Generation of the Verification Model(s)

Once the different kinds of model have been developed (potentially by different engineering teams), the question is how to connect these models to produce an overall verification model that could be simulated.

Indeed, to be very concise, according to the FORM-L concepts, a requirement can be expressed under the form:

[Where][When][What is the condition to be satisfied][How well]

To automatically check such a requirement by simulation, each of its attributes should be evaluated during the simulation run.

As explained in the architecture recalled in Figure 10:

- The [where] as well as other static attributes (e.g. some components characteristics) are provided by the architecture model;
- While the other attributes which are dynamic are provided by the behavioral model (or the test scenario when the behavioral model does not exist).

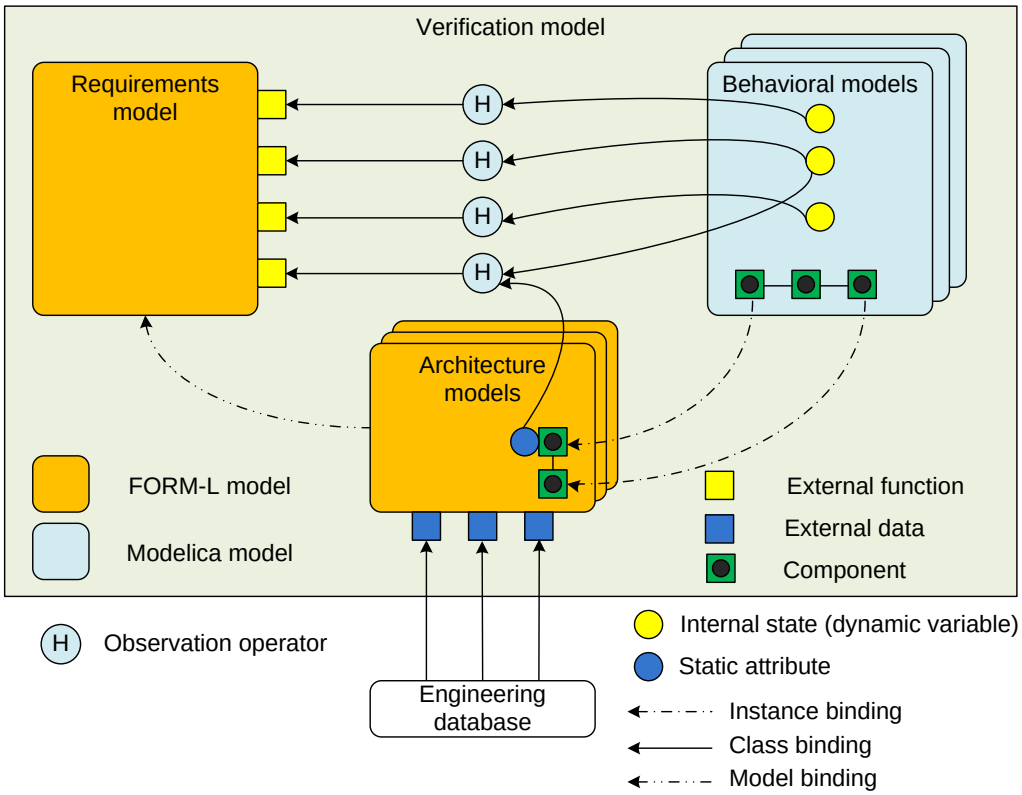

Figure 10: Recall of the Modelling Architecture for the Verification of Formal requirements (extract from [12])

Since models are usually developed by different experts, the designer in charge of the verification task should hence be capable to bind the different kinds of models in a convenient way. The process should be convenient in the sense that:

- The initial models should not been modified (to avoid drift into a situation where some parallel developments are no more validated and maintained by the right experts)
- The process cannot be done fully manually otherwise it will be very time consuming and error prone for industrial systems (even for not so large systems, it is rather easy to deal with many requirements to be instantiated on many components according to many different operation states).

Several proposals have been made to automate a little bit this task of bindings (see [24] for more details).

Here only the scripting approach proposed in the *createCheck* function of the *Modelica_Requirements* library has been tested.

6.2. Test Scenarios

Once a verification model has been obtained, some test scenarios should be defined to run some simulations and state whether the requirements are violated or not.

In the case of the SRI, the temperature of the cold source is of particular importance. Thus, test scenarios may be used to generate cases with very cold CW (near freezing) and client operating at low power, or CW at or near its maximum expected (27.2°C), and clients are operating at full power.

6.3. Verification Model to test the System Requirements

In the very preliminary stage of the design cycle, once the system requirements are specified and formally modelled, simulation can be used to verify that the model correctly represents what is intended. As no architecture or behavioural model are available yet, simulation can be made using only the requirement model with a test scenario (manually set) consistent with the assumptions.

Although it has not been tested yet, a random test case generator can also be used to explore more scenarios. In that case, it is even possible to guide the test case generator to produce particular test cases of interest by specifying test scenarios in the form of additional assumptions.

6.4. Verification Model to test the Overall Design

Latter in the design cycle, an architecture model will be available and bound to the requirements model to check by simulation whether the requirements are violated or not (and hence to validate the overall design choice represented in the architecture model).

In the case of the SRI, the architecture model will provide for instance information such as the effective sets of pumps (here according to Figure 7: pumps = {PO1, PO2, PO3}).

Let us assume that we have an additional requirement which states that, for each operating pump, they should not cavitate (see Figure 11 for its implementation using *Modelica_Requirements*).

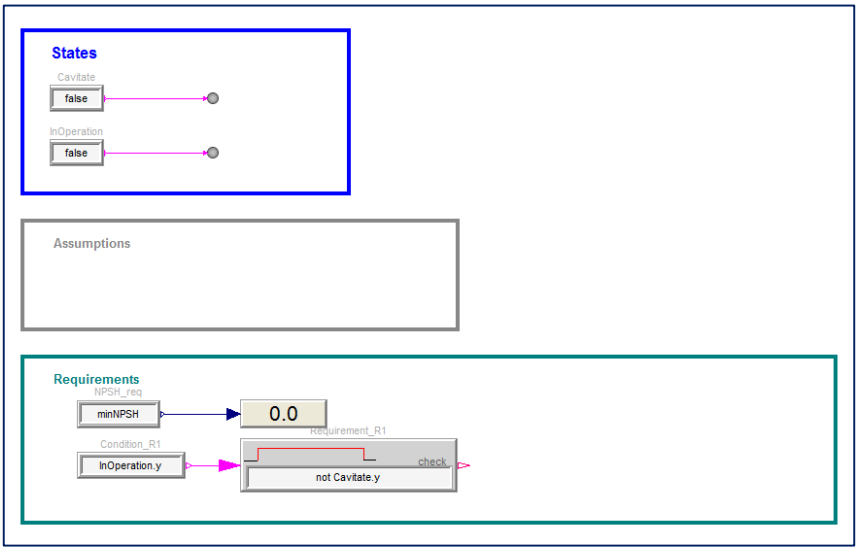

Figure 11: Implementation of the pump *requirements model using the Modelica_Requirements library*

Creation of the verification model

In this case, the pump requirement model should be automatically instantiated (and parameterized) for each pump of the architecture model, naming for PO1, PO2 and PO3.

This is what has been successfully tested by adjusting the *createCheck* function of the *Modelica_Requirements* library (see Figure 12).

```
function instantiateAndParameterizeCoolingSystemRequirements_R1A2
  "Instantiate and parameterize requirements R1 according to architecture A2"

algorithm
  //  Bindings between the studied Requirements and the studied Architecture model
  CoolingSystem_SRI.VerificationModels.Lib.createCheck_Disp(
    targetName="CoolingSystem_SRI.Architectures.CoolingSystem_A2",
    checkTargetName="CoolingSystemRequirements_R1A2",
    stopTime=2000,
    tolerance=1e-6,
    bindings={
      Modelica_Requirements.Bindings.Binding(
        checkName=  "CoolingSystem_SRI.Requirements.CoolingSystemRequirements_R1",
        instances=  {""},
        modifier=   "tHW.y=tHW,
                     tRW.y=tRW,
                     tCW.y=tCW,
                     tWW.y=tWW,
                     MaxTempCW=coldSource.tCWMax,
                     MaxDeltaT=coldSource.deltaTMax,
                     pDW.y=demineralisedWaterCircuit.pDW,
                     pCW.y=coldSource.pCW,
                     ePlantStartup.y=true,
                     ePlantStop.y=false,
                     ready.y=true,
                     eFailure.y=false"),
      Modelica_Requirements.Bindings.Binding(
        checkName=  "CoolingSystem_SRI.Requirements.ClientRequirements",
        instances=  {""},
        modifier=   "maxPower=client_C.maxPower,
                     minT=client_C.minT,
                     maxTclient=client_C.maxTclient,
                     maxFlow=client_C.maxFlow,
                     tRW.y=tRW,
                     Flow.y=client_C.Flow,
                     Power.y=client_C.Power,
                     t.y=client_C.Temp,
                     InOperation.y=client_C.InOperation"),
      Modelica_Requirements.Bindings.Binding(
        checkName=  "CoolingSystem_SRI.Requirements.PumpRequirements",
        instances=  {"pump_A","pump_B","pump_C"},
        modifier=   "Cavitate.y=$.cavitate,
                     InOperation.y=$.inOperation,
```

Figure 12: Modelica function used to generate the verification model to test the SRI Overall Design

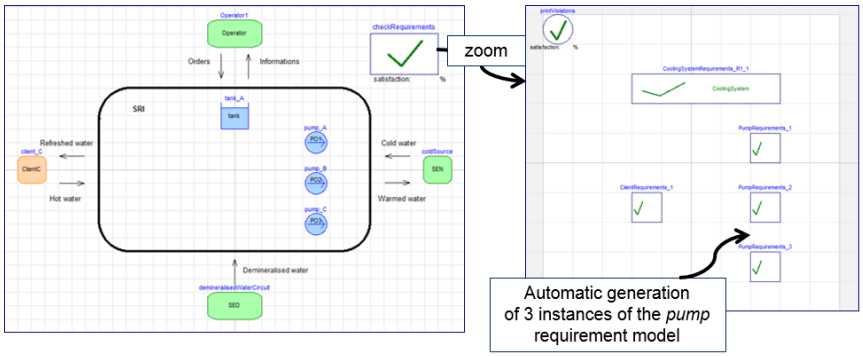

Figure 13: Verification model automatically generated to test the SRI Overall Design

Simulation of the verification model

An architecture is verified in the same manner as the requirements model: with test scenarios set manually or with the help of a generator. In that case of a random generator, it generates components behaviors consistent with the assumptions placed on each of them. The difference is that co-simulation

of the requirements and the architectural models can be used to automatically verify whether the architecture violates requirements. Thus, a large number of test cases can be performed, and the test case generator can be guided to ensure adequate test coverage. This facilitates the in-depth examination (including dysfunctional analyses, such as failure modes effects and criticality analysis) of many different architectures.

6.5. Verification Model to test the Detailed Design

Again, binding and co-simulation can be used to automatize the verification that the initial requirements are satisfied. Here to test the Detailed Design, the three different kinds of models will be utilized: the requirements, the architecture and the behavioural models.

Creation of the verification model

In this case, the different requirements models will be automatically instantiated (and parameterized) according to both the architecture model (Figure 7) and the behavioural model (Figure 9).

This is what has been successfully tested by adjusting the *createCheck* function of the *Modelica_Requirements* library (see Figure 14 for more details)

```
function createCheckCoolingSystem_R1A2B1
  "Generate model CheckCoolingSystem to check requirements R1 according to architecture A2 and behavior B1"

algorithm
  //  Bindings between the Requirements model generated accordingly to the Architecture model and the studied
  CoolingSystem_SRI.VerificationModels.Lib.createCheck_Disp(
    targetName="CoolingSystem_SRI.Behaviors.SRI_v3",
    checkTargetName="CheckCoolingSystemRequirements_R1A2B1",
    stopTime=2000,
    tolerance=1e-6,
    bindings={
      Modelica_Requirements.Bindings.Binding(
        checkName=  "CoolingSystem_SRI.VerificationModels.Temp.CoolingSystemRequirements_R1A2",
        instances=  {""},
        modifier=   "tHW=Modelica.SIunits.Conversions.to_degC(Flux1.C2.T),
                     tRW=Modelica.SIunits.Conversions.to_degC(VolumeD3.T),
                     tCW=Modelica.SIunits.Conversions.to_degC(EchangeurAPlaques1D1.Tef),
                     tWW=Modelica.SIunits.Conversions.to_degC(EchangeurAPlaques1D1.Tsf),
                     pDW=Modelica.SIunits.Conversions.to_bar(EchangeurAPlaques1D1.Sc.P),
                     pCW=Modelica.SIunits.Conversions.to_bar(EchangeurAPlaques1D1.Sf.P),
                     ePlantStartup=true,
                     ePlantStop=false,
                     ready=true,
                     eFailure=false,
                     client_C.Flow=50e-3,
                     client_C.Power=-SourceC1.C[1].W,
                     client_C.Temp=Modelica.SIunits.Conversions.to_degC(SourceC1.C[1].T),
                     client_C.InOperation=if time < echelon.startTime then false else true,
                     pump_A.cavitate=CoolingSystem_SRI.ObservationOperators.Obs_PumpCavitating(PompeCentrifuge
                     pump_A.inOperation=Moteur1.marche.signal,
                     pump_B.cavitate=CoolingSystem_SRI.ObservationOperators.Obs_PumpCavitating(PompeCentrifuge
                     pump_B.inOperation=Moteur2.marche.signal,
                     pump_C.cavitate=CoolingSystem_SRI.ObservationOperators.Obs_PumpCavitating(PompeCentrifuge
                     pump_C.inOperation=Moteur3.marche.signal")});
```

Possible use of observation operators to « convert » functional states into physical states

Figure 14: Modelica function used to generate the verification model to test the SRI Detailed Design

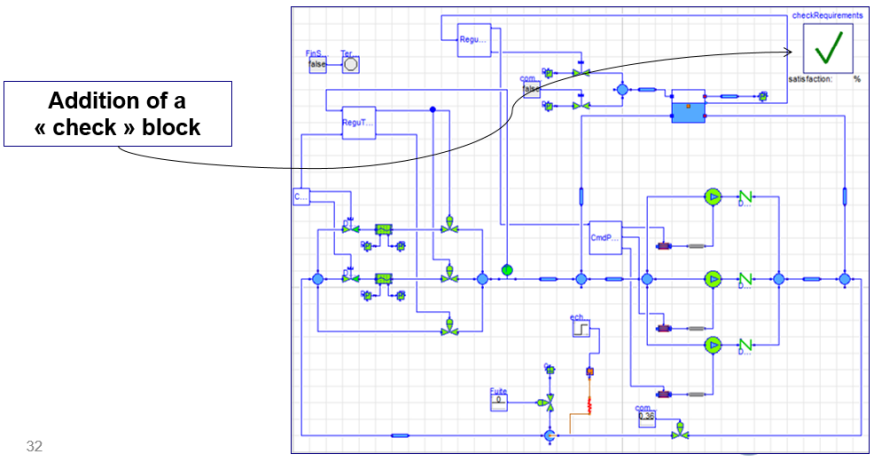

Figure 15: Verification model automatically generated to test the SRI Detailed Design

Simulation of the verification model

Once again the verification model can be simulated with test scenarios set manually or with the help of a generator.

Due to time constraints, only one scenario has been tested here but it has permitted to check that the verification model automatically generated can be simulated well.

7. Conclusion

This case study is much simpler in terms of control logic than the Backup Power Supply (*BPS*) presented in [16], but has more physical aspects. As a perspective (after the end of MODRIO), it is planned to continue the work at EDF on this case study. Some aspects will be completed in particular to refine the system requirements model and test more simulation scenarios. Even if promising, the *Modelica_Requirements* will also be improved (enhancements of the automation of bindings, handling of sets and overlapping time periods, possibility to express requirements with probabilities…) and the development of a standalone FORM-L compiler will be invested to target other environments than the ones based on Modelica.

8. References

[1] Fritzson P., “Principles of Object-Oriented Modelling and Simulation with Modelica”, IEEE Press, 2003.

[2] Harel D., “Statecharts: A Visual Formalism For Complex Systems”, in Science of Computer Programming, vol. 8, pp. 231-274, 1987.

[3] Schamai W., “Model-Based Verification of Dynamic System Behavior Against Requirements”, Dissertation No. 1547, Linköping University, 2013.

[4] Rapin N., “ARTiMon: Outil de Monitoring de Propriétés”.

[5] SPEEDS: Contract Specification Language (CSL)

[6] Jardin A., “EuroSysLib sWP7.1 - Properties Modelling”, EDF technical report, H-P1C-2011-00913-EN, 2011.

[7] Duriaud Y., “EuroSysLib – Deliverable 7.1b – Etude des Langages de Spécifications de Propriétés”, EDF technical report, H-P1A-2007-01326-FR, 2007.

[8] Duriaud Y., “EuroSysLib – Deliverable 7.1a – Collected needs within EDF”, EDF technical report,

H-P1A-2007-01712-FR, 2008.

[9] Marcon F., “EuroSysLib WP7.1 – Synthesis of needs within EDF”, EDF technical report, 2009.

[10] Thomas E., “EuroSysLib WP7.1 – Dysfunctional: Use Cases and User Requirements”, Dassault Aviation technical report, DGT116083, 2008.

[11] Thomas E., Chastanet L., “EuroSysLib WP7.1 – Dysfunctional: Use Cases and User Requirements”, Dassault Aviation technical report, DGT116083B, 2009.

[12] Bouskela D., “MODRIO D2.1.1 Part II - Modeling Architecture for the Verification of Requirements”, also available as EDF technical report, H-P1C-2014-15188-EN, 2015.

[13] Otter M. et al., “MODRIO D2.1.1 Part IV – Modelica for properties modelling”.

[14] Thuy Nguyen, “MODRIO D2.1.2 Properties modelling method”.

[15] Thuy Nguyen, “MODRIO D2.1.1 Part III - FOrmal Requirements Modelling Language (FORM-L)”, also available as EDF technical report, H-P1A-2014-00550-EN.

[16] Thuy Nguyen, “MODRIO D8.1.3 Part I - The Backup Power Supply (BPS) Case Study”.

[17] Feiler P., Gluch D., Hudak J., “The Architecture Analysis & Design Language (AADL): An Introduction”, Software Engineering Institute, 2006.

[18] Selic B., Gérard S., “Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE”, Elsevier, 2014.

[19] El Hefni B., Bouskela D., Lebreton G., “Dynamic modelling of a combined cycle power plant with ThermoSysPro“, Proceedings in the 8th International Modelica Conference, 2011.