INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

D8.1.3 – Power plant safety scenario
Part I: The Backup Power Supply (BPS) Case Study

WP8.1 – Power Plants
WP8 – Demonstrators

MODRIO (11004)

Version 1.0
Date 01/05/2016

Authors
Thuy NGUYEN EDF

Executive summary

This document is Part I of deliverable MODRIO D8.1.3 Power plant safety scenario. It describes in detail an example of property modelling based on a Backup Power Supply (*BPS*) system. This system is relatively simple in terms of physics but introduces relatively complex control logic. It is a realistic example of a reactive hybrid system, illustrating the modelling of timing issues and the use of discrete time domains.

The *BPS* case study aims at several goals. 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. To that end, it has been designed to be independent from any industrial sector. The second goal of the case study is to provide concrete examples of the use of the FORM-L, covering if not all, at least a large part of the language. The third goal is to illustrate various possible uses of modelling and simulation in systems engineering, in particular verification that the functional, timing and operational requirements specified for a given system are appropriate with respect to the environment of the system, and verification that overall design, and then detailed design, satisfy the specified requirements.

The methodology followed is the one proposed in [Thuy Nguyen, “MODRIO D2.1.2 Properties modelling method”.]. In particular, it suggests that separate models are developed for requirements specification, system specification, overall design and detailed design. Whereas the first three models are composed of property models developed early in the system engineering lifecycle, the last one is generally a behavioural model based on a precise knowledge of the characteristics and behaviour of the components constituting the system. In addition to models representing the system itself, models for the entities (technical systems or humans) constituting the environment of the system are used in order to provide a context for, and interact with, the models for the system under study. In practice, different models are developed at different stages of the system engineering lifecycle, by different persons who have different viewpoints, different backgrounds and different modelling constraints.

The property models for system requirements, system specification and system overall design are 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.]. The interfaces between property models or with behavioural models are based on the modelling architecture proposed by [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.]. As a perspective (i.e. after the end of MODRIO), it is planned to continue the work at EDF on this case study. The property models presented here will be complemented with a (Modelica) behavioural model representing the detailed design and the physical behaviour of the electrical and electromechanical components. Component failure modes will also be modelled using the multi-modelling concepts and languages proposed by WP4.

Summary

Acronyms

BPS	Backup Power Supply
CTL	Continuous Time Locator
	Computation Tree Logic
DG	Diesel Generator
FORM-L	FOrmal Requirements Modelling Language
LS	Load Sequencer
LTL	Linear Temporal Logic
MPS	Main Power Supply
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.
Reference Model	Top-level model for a system under study that serves as a reference for subsequent models
Requirement	Property that must be satisfied: it is the objective of simulation to verify that it is not **violated**
Surrogate Model	Property model for an entity that just satisfies the contracts to which that entity is a party.
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 I of deliverable MODRIO D8.1.3 *Power plant safety scenario*. It describes in detail an example of property modelling based on a Backup Power Supply (*BPS*) system. This system is relatively simple in terms of physics but introduces relatively complex control logic. It is a realistic example of a reactive hybrid system, illustrating the modelling of timing issues and the use of discrete time domains.

The methodology followed is the one proposed in [13]. In particular, it suggests that separate models are developed for requirements specification, system specification, overall design and detailed design. Whereas the first three models are composed of property models developed early in the system engineering lifecycle, the last one is generally a behavioural model based on a precise knowledge of the characteristics and behaviour of the components constituting the system. In addition to models representing the system itself, models for the entities (technical systems or humans) constituting the environment of the system are used in order to provide a context for, and interact with, the models for the system under study. In practice, different models are developed at different stages of the system engineering lifecycle, by different persons who have different viewpoints, different backgrounds and different modelling constraints.

The property models for system requirements, system specification and system overall design are developed based on the FOrmal Requirements Modelling Language (FORM-L) proposed in [14]. The interfaces between property models or with behavioural models are based on the modelling architecture proposed by [12]. As a perspective (i.e. after the end of MODRIO), it is planned to continue the work at EDF on this case study. The property models presented here will be complemented with a (Modelica) behavioural model representing the detailed design and the physical behaviour of the electrical and electromechanical components. Component failure modes will also be modelled using the multi-modelling concepts and languages proposed by WP4.

1.2. Objectives

The *BPS* (Back-up Power Supply) case study aims at several goals. 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. To that end, it has been designed to be independent from any industrial sector. The second goal of the case study is to provide concrete examples of the use of the FORM-L, covering if not all, at least a large part of the language.

The third goal is to illustrate the methodology proposed in [13] to support various systems engineering activities, in particular for verifying that the behavioural requirements specified for a given system are appropriate with respect to the environment and the operation of the system, and for verifying that a system specification, an overall design, and then a detailed design, satisfy the specified requirements.

Although it is a fictitious and does not describe a specific real-life system, the case study is representative of what could be required of a high-power backup electric power supply system. Also, though it is relatively simple, it is reasonably complete from a behavioural standpoint, with requirements covering the normal and failure situations (including probabilistic requirements regarding failures). Non-behavioural requirements such as volume, weight, ambient conditions, seismic conditions, electromagnetic compatibility, human factors, cyber security, practical aspects of maintenance and testing, installation on site, authorised noise levels, HVAC, power supply for the *BPS* are not covered.

1.3. Structure of the document

Section 2 presents an overview of the *BPS*, 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 an overview of the *BPS* system specification, and then provides a detailed presentation of the modelling of the system specification in FORM-L.

Section 4 presents an overview of the *BPS* overall design, and then provides a detailed presentation of the modelling of the overall design in FORM-L.

Section 5 presents the external features that need to be provided either by scenario models or by detailed design models if one wants to go further in the verification methodology (i.e. to test the models by simulation or to complete them with a detailed design).

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. *BPS* System Requirements

2.1. Brief Introduction to the *BPS*

The objective of the *BPS* is to provide electric power to a number of "clients" in case of loss of the Main Power Supply (*MPS*). Such clients could be important to safety (e.g., in an industrial installation or in a hospital) or could be essential for preventing unacceptable economic losses (e.g., in a manufacturing plant). Figure 2-1 places the *BPS* within its environment. In addition to the *MPS* and a human operator, three clients (*client1*, *client2* and *client3*) have been identified. Each has specific expectations with respect to the *BPS*.

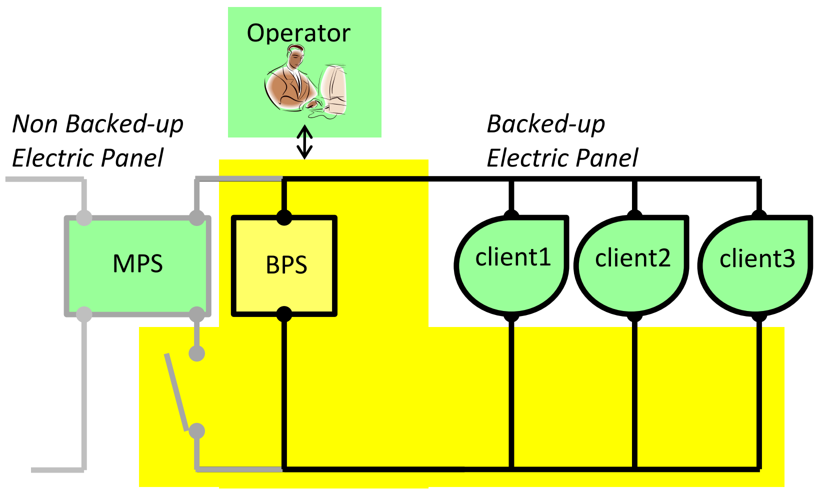


Figure 2-1
The BPS and its environment. The entities constituting the environment are shown in green. The BPS itself is shown on a yellow background.

2.2. *BPS* Non-Formal System Requirements

Each client has a certain potential for tolerating loss of electric power. That potential is quantitatively represented as a Real variable in range [0., 100.]. When the client is active and not powered, its potential is consumed. When the client is powered, its potential is replenished

pa The *BPS* should ensure that when the *MPS* is not available, no client tolerance potential reaches zero.

r1 The probability of not satisfying *pa* must be less than 10^{-3} per *BPS* demand.

Once activated, the *BPS* is put back to rest by a *Reset* signal issued by the operator. A *Reset* is valid if and only if it occurs after the *MPS* is available again, and at least 10 minutes after the *BPS* activation.

pb When a valid *Reset* is issued by the operator, if the *MPS* is not lost again, then the normal powering conditions (i.e., all clients powered by the *MPS*) should be restored within 10 seconds.

r2 The probability of not satisfying *pb* must be less than 10^{-3} per *BPS* demand.

The operator needs to be informed of *BPS* failures, so that he or she can take any necessary mitigating actions, as specified by operating procedures.

pc When *pa* or *pb* are violated, a failure signal should be sent to the operator.

r3 The probability of not sending a failure signal when necessary must be less than 10^{-3} per demand.

r4 The rate of spurious failure signal must be less than what it would be if there were a constant failure rate of 10^{-2} per year.

2.3. Modelling Organisation

The modelling organisation for the *BPS* system requirements is summarised in Figure 2-2. It is based on six object models:

- Reference object model *bpsReq* specifies the system requirements placed on the *BPS* (see Section 2.2 *BPS Non-Formal System Requirements*).
- Object model *mps* represents the *MPS*. This model could simply be a surrogate model (just satisfying the *mpsMonitoring* contract) or one among a set of generic scenario models (also complying with *mpsMonitoring*).

- Object model *op* represents the human operator. The same principles as for *mps* apply, mutatis mutandis.
- Class model *Client* specifies the general characteristics of a *BPS* client. It has three instances (*client1*, *client2* and *client3*) representing the three clients of the *BPS*. The same principles as for *mps* and *op* apply.

The *bpsReq* model has five contracts with the models of its environment entities:

- Contract *mpsMonitoring* with *mps*.
- Contract *hsi* (for Human-System Interface) with *op*.
- Contracts *bep1*, *bep2* and *bep3*, respectively with *client1*, *client2* and *client3*. They are instantiations of a standard contract named *Bep* (for Backup Electric Power).

Most of the *BPS* system requirements are specified in the contracts.

Note. The system names (e.g., *MPS*) are different from the model names (e.g., *mps*) to highlight the fact that a model is NOT a system, and to avoid ambiguities when explaining the modelling.

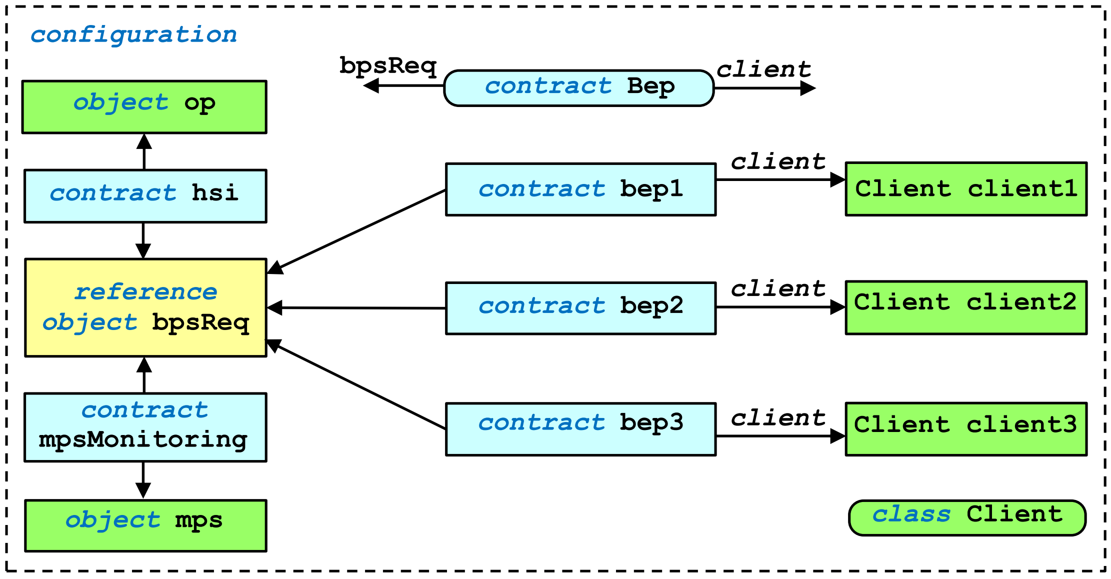


Figure 2-2
Overall organisation of the BPS system requirements modelling. The reference object model is shown in yellow, the other object models in green, and the contracts in blue. Classes and standard contracts are shown with round corners.

```
configuration BPS_REQ

  reference object model "*bpsReq*"  alias bpsReq; // *xxx* : file name

  object          "*mps*"           alias mps;
  object          "*op*"            alias op;
  class           "*Client*"        alias Client;
  property model  "*CLIENTS*"       alias CLIENTS; // Instances of Client

  contract        "*mpsMonitoring*" alias mpsMonitoring;
  contract        "*hsi*"           alias hsi;
  contract        "*Bep*"           alias Bep;  // Standard contract
  contract        bps1 = Bep(client1, powering1);
  contract        bps2 = Bep(client2, powering2);
  contract        bps3 = Bep(client3, powering3);

  end BPS_REQ;
```

2.4. The *mpsMonitoring* Contract

The property model for the *MPS* does not need to be a detailed model: any surrogate model satisfying the *mps* part of contract *mpsMonitoring* will be sufficient. In practice, one can chose one among various property models, each representing one generic test scenario of *MPS* behaviour.

```
contract mpsMonitoring

  party mps
    Voltage voltage;                 // at the terminals of the mps
    requirement v1 = check voltage isIn [[0.*v, 240.*v]];
    end mps;

  party bpsReq
    end bpsReq;

  end mpsMonitoring;
```

Contract *mpsMonitoring* can be read as follows:

- It is set between two parties: the *mps* and the *bpsReq*.
- There is only one deliverable in this contract: the *voltage* at the terminals of the *MPS*, and it is under the responsibility of the *mps* party. mps guarantees that *voltage* will be in the 0.-240. volt range.
- The *bpsReq* party has no obligation in this contract.

2.5. The *hsi* Contract

```
contract hsi

  party op
    // Reset button
    Boolean resetBtn;
    requirement b1 =
      after resetBtn becomes true  for 0.5*s check resetBtn;
    requirement b2 =
      after resetBtn becomes false for 0.5*s check not resetBtn;
    end op;

  party bpsReq
    event eFailure;        // BPS failure
    event eAlarm;          // Alarm signal sent to operator

    // Inform operator in case of BPS failure
    property pc = after eFailure within 200*ms check once eAlarm;
    requirement r3 =
      after eFailure check probability (pc.violated becomes true) < 10^-3;

    // No spurious failure signal
    property pd = before eFailure check no eAlarm;
    requirement r4 =
      check probability (pd.violated becomes true) < 1.-exp(-10^-2*time/y);
    end bpsReq;

  end hsi;
```

Contract *hsi* can be read as follows:

- It is set between two parties: the *op* model and the *bpsReq* model.
- Party *op* delivers a Boolean variable named *resetBtn* representing the state of a *Reset* button. The variable is *true* when the *Reset* button is pressed, and *false* otherwise. The *BPS* interprets the variable becoming *true* as a *Reset* request.

- Requirements *b1* and *b2* address the fact that no real system can detect an arbitrarily brief signal. Thus, the *Reset* button is in fact a device ensuring that the associated Boolean variable remains in the same *true* / *false* state for at least 0.5 second after a transition.
- Party *bpsReq* delivers two events: *eFailure* and *eAlarm*. *eFailure* occurs when the *BPS* fails to satisfy properties *pa* and / or *pb*. *eAlarm* is raised by the *BPS* as per requirements *r3* and *r4* (see Section 2.2 *BPS Non-Formal System Requirements*).

2.6. The *bep* Standard Contract

Each client has a contract with the *BPS*. As these contracts are similar, they are based on standard contract *Bep* (for *backup electric power*).

```
contract Bep (client, powering)

  party client
    Boolean active;
    Boolean powered;
    Real tolPot;
    end client;

  party bpsReq
    assumption c1 = check no active becomes false;
    event eVReset;
    Boolean powering;
    assumption c2 = during powering check powered;
    property pa = check tolPot > 0.;
    requirement r1 =
      after (active and not powered) becomes true
      check probability (pa.violated) < 10-3;
    property pb =
      after eVReset within 10*s check powered and not powering;
    requirement r2 = after eVReset check probability (pb.violated) < 10-3;
    end bpsReq;

  end Bep;
```

The contract can be read as follows:

- it has two parameters: *client* is the client, and *powering* is the variable that tells whether the client is powered by the *BPS*.
- It has two roles: *bpsReq* and a *client* (that needs to be powered when active, by the *bpsReq* or by any other source). The identity of the *client* needs to be specified at each *bep* instantiation.
- *active* is a Boolean deliverable of the *client* that indicates whether it is active and needs to be powered. *bpsReq* asumes (assumption *c1*) that once the *client* is *active*, it will remain so.
- *powered* is a Boolean deliverable of the *client* that indicates whether it is powered (by *bpsReq* or by any other source).
- *tolPot* is a Real deliverable of the *client* that determines its potential to tolerate loss of electric power (see Section 2.2 *BPS Non-Formal System Requirements*).
- *eVReset* is an event delivered by *bpsReq* indicating that the human operator has issued a valid *BPS* reset.
- *powering* is a Boolean delivered by *bpsReq* indicating whether it is providing power to the *client*. Assumption *c2* states that when *bpsReq* is *powering* the *client*, then *powered* must be true.

2.7. Clients

For the pupose of this case study, and at this stage of the system development process, a property model for *client1*, *client2* or *client3* just needs to provide the information necessary to instantiate standard contract *bep*:

- For *client1*, Boolean *active* is declared external and will be provided by subsequent models (design models in FORM-L or behavioural models in Modelica). For *client2* and *client3*, Boolean *active* is always true.
- For all three clients, Boolean *powered* and Real *tolPot* are declared external and will be provided by subsequent models.

```
class Client
   Boolean active;
   external Boolean powered;
   external Real tolPot
      start = 100.;
      end tolPot;
   end Client;

property model CLIENTS

   Client client1
      external Boolean active;
      contract bep1;
      end client1;

   Client client2
      active = true;
      contract bep2;
      end client2;

   Client client3
      active = true;
      contract bep3;
      end client3;

   end CLIENTS;
```

2.8. The *bpsReq* Reference Model

```
object model bpsReq

   contract mpsMonitoring;                                    // with mps
   external Boolean mpsAvailable;                             // for mpsMonit.
   guard g1 = when time = 0.*s check mpsAvailable;
   guard g2 = check count (mpsAvailable becomes false) ≤ 1;
```

```
  contract hsi;                                             // with op
  event eFailure =                                          // for hsi
    when (bep1.pa.violated or bep1.pb.violated
       or bep2.pa.violated or bep2.pb.violated
       or bep3.pa.violated or bep3.pb.violated) becomes true;
  external event eAlarm;                                    // for hsi

  event eVReset =                                           // for bep
    after (mpsAvailable becomes false + 10.*mn) and
    when (resetBtn and mpsAvailable) becomes true;
  external Boolean powering1;                               // for bep
  external Boolean powering2;                               // for bep
  external Boolean powering3;                               // for bep
  contract bep1;                                            // with client1
  contract bep2;                                            // with client2
  contract bep3;                                            // with client3

  // The final evaluation of the properties can be made only after the BPS
  // has attempted to restore normal powering conditions
  after eVReset + 15.*s then stop;

  end bpsReq;
```

The *bpsReq* model can be read as follows:

- Two guards specify the conditions of validity of the *bpsReq* model: *g1* states that the *MPS* should be initially available; *g2* states that the *MPS* should be lost only once.
- It cites the five contracts with the models for the *BPS* environment (*mpsMonitoring*, *hsi*, *bep1*, *bep2* and *bep3*). For the last three, it specifies the variable that provides the *powering* parameter (whether the *BPS* powers the *client*).
- Event *eFailure* is directly defined as a violation of *pa* or *pb*.
- Event *eVReset* is directly defined as a reset request at least 10 minutes after the *MPS* has become unavailable, and when it is available again.
- The other deliverables the *bpsReq* model is responsible for are declared external and will be defined by specification or design models.
- The last statement specifies that the final evaluation of the properties in the model can be made only after the *BPS* has attempted to restore normal powering conditions.

3. *BPS* System Specification

System specification is the first step in the definition of a solution that will meet the system requirements specified and modelled in Section 2 *BPS System Requirements*.

3.1. Brief Introduction to Systems Organisation

3.1.1. *BPS* and Clients Divisions

As the *BPS* and its clients are required to be continuously available over long periods (e.g., a year or more), to allow maintenance while in operation and to tolerate components failures, each is divided into two identical *divisions* (named *a* and *b*). At any given time, a client division is connected to one and only one *BPS* division. Switching from one *BPS* division to the other takes a very short time (in the order of a second) compared to the time interval between real demands to the *BPS* (in the order of once per year or per decade). Maintenance of a *BPS* division can be performed when all the client divisions are connected to the other division. Figure 3-1 presents this more detailed view of the *BPS* and of its environment.

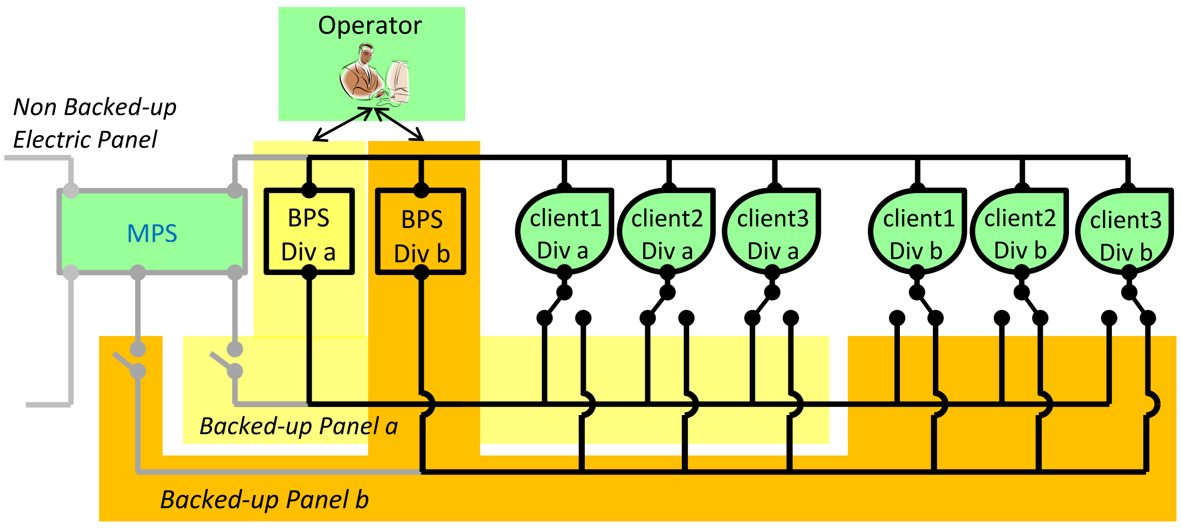


Figure 3-1
The BPS divisions and their environment. In the case shown, client divisions a are all connected to BPS division a, and client divisions b are all connected to BPS division b.

3.1.2. *BPS* States

When in operation, a *BPS* division can be in one of three main states:

- *Nominal* state: the *BPS* division is available and ready to perform its mission.
- *Test* state: as *MPS* loss is rare and the *BPS* is seldom active, periodic testing is necessary to ensure that when needed, the *BPS* division will indeed be able to perform its mission.
- *Maintenance* state: the *BPS* division is not in service, all clients are connected to the other division, and the *BPS* mission is ensured by the other division.

3.1.3. Operator

In addition to the reset button seen previously, the operator in the control room has:

- Two buttons, one for each *BPS* division, to signal the beginning of a maintenance session. In order to ensure their independence, the two divisions do not communicate. It is assumed that the operator will not put the two divisions into maintenance at the same time. It is also assumed that before one division is put into maintenance, field operators have connected all the clients to the other division.
- One button to signal the end of an on-going maintenance session.
- Two buttons, one for each *BPS* division, to order the start a periodic test.

3.1.4. Fault-Tolerance and Sensors

A *BPS* division is composed of a number of components of different types, including sensors. Sensors are relatively inexpensive but are subject to drift and failure. To further enhance fault-tolerance capability, the *BPS* specification requires that each division must be able to tolerate the failure of any single sensor.

3.1.5. Critical Components

A *BPS* division has components such as a diesel generator that are expensive and difficult to replace. They must be protected against damage whenever possible, in particular during periodic tests.

3.2. *BPS* Non Formal System Specification

As previously mentioned, the *BPS* is composed of two identical, redundant divisions. As they are fully independent from one another, the *BPS* system specification is expressed relative to each division. To avoid unnecessary complexity, each *BPS* division acts as if all client divisions were connected to it.

s1 When the *BPS* division is in maintenance, it must not be activated. (Otherwise, maintenance personnel could be harmed. The division might not be operable anyway.)

s2 In the absence of any *BPS* division component failure or in the presence of a single sensor failure, when the division is not in maintenance, and in case of loss of *MPS*:

a) If *client1* is active, its divisions must be powered within 25 seconds. Power must be provided until a valid reset from the operator.

b) The divisions of *client2* must be powered within 35 seconds. Power must be provided until a valid reset from the operator.

c) The divisions of *client3* must be powered within 45 seconds. Power must be provided until a valid reset from the operator.

s3 Probabilities of not satisfying requirements *s2* due to the *BPS* division components failures:

a) The probability of not powering one or more client divisions in the allocated time must be less than 10^{-3} per demand.

b) The *BPS* division failure rate while it powers its clients must be less than what it would be if there was a constant failure rate of 10^{-3} per day.

s4 In the absence of any *BPS* division component failure or in the presence of a single sensor failure, the division must not be spuriously activated.

s5 The probability of spuriously activating the *BPS* division must be lower than what it would be if there were a constant occurrence rate of spurious activation equal to 10^{-3} per year.

s6 When the *BPS* division is active:

a) If it mitigates a loss of the *MPS*, its protected components must be operated at any cost, even if this means damaging or eventually destroying them in the long run.

b) If it is operated for a periodic test, the risk of damaging its protected components must be minimised.

Once activated, the *BPS* division can be put back to rest by a *Reset* signal issued by the operator. A *Reset* is valid when and only when it occurs after the *MPS* is available again, and at least 10 minutes after the *BPS* activation.

s7 In the absence of any *BPS* division component failure or in the presence of a single sensor failure, when a valid *Reset* is issued by the operator, and if the *MPS* is not lost again:

a) The normal powering conditions for the client divisions (i.e., powered by the *MPS*) must be restored within 10 seconds.

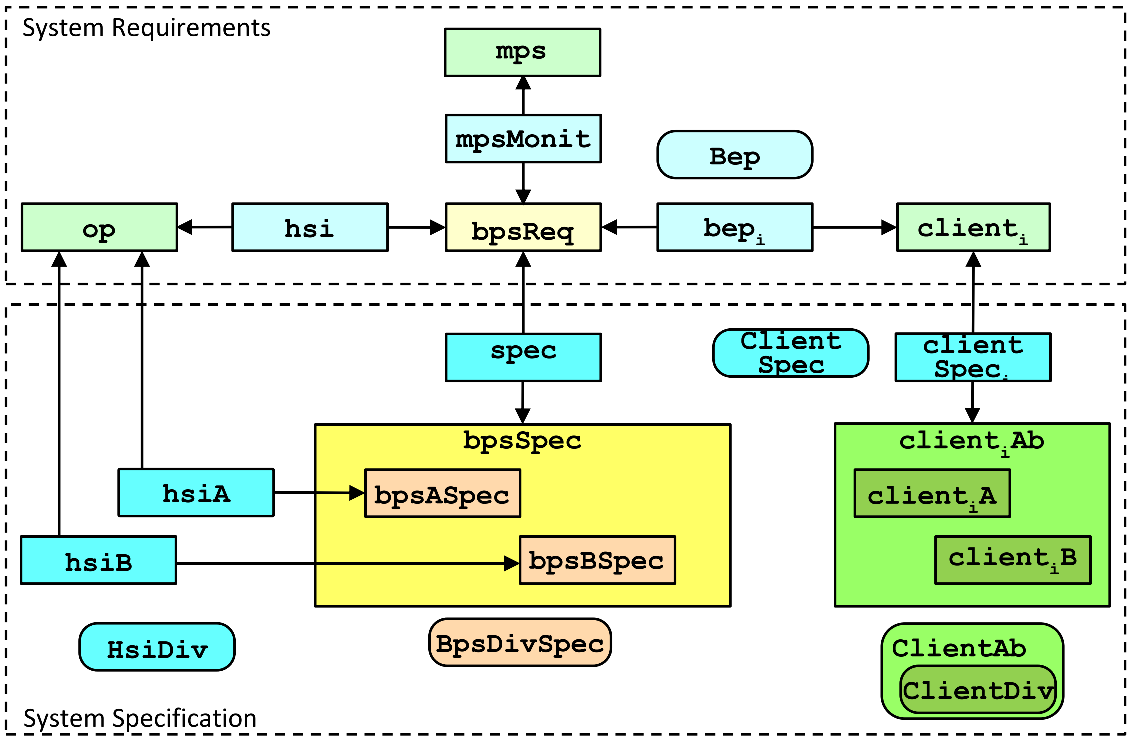
b) No client division should be unpowered for more than a total of 5 seconds after the *Reset*. (A brief depowering is necessary, for some *BPS* components must not be connected to the available *MPS*, otherwise, they could be seriously damaged.)

s8 The probability of not restoring the normal powering conditions for all the client divisions within 10 seconds after a valid *Reset* must be less than 10^{-3} per *BPS* demand.

psa If any client division cannot be powered as specified in *s2* and *s7*, a failure signal should be sent in the control room to inform the operator. (He or she can then take any necessary mitigating action specified in the operating procedures.)

s9 The probability of not sending a failure signal when required must be less than 10^{-3} per demand.

s10 The rate of spurious failure signal must be less than what it would be if there were a constant failure rate of 10^{-2} per year.



3.3. Modelling Organisation

The modelling for the *BPS* system specification is summarised in

Figure 3-2:

- Class *BpsDivSpec* models the system specification of a *BPS* division.
- Object *bpsSpec* specifies the *BPS* system specification, and models the *BPS* as two divisions (instances of *BpsDivSpec*, named *bpsASpec* and *bpsBSpec*) and six Boolean switches.
- Contract *spec* is set between *bpsSpec* and the reference model *bpsReq* that specifies the *BPS* system requirements.
- Generic contract *HsiDiv* models the mutual obligations between *op* (representing the operator) and a *BPS* division. It has two instances: *hsiA* (between *op* and *bpsASpec*) and hsiB (between *op* and *bpsBSpec*).
- Class *ClientDiv* models a *BPS* client division.
- Class *ClientAb* models a *BPS* client as two divisions, instances of *ClientDiv*.
- Generic contract *ClientSpec* models the mutual obligations between $client_x Ab$ and $client_x$. It has three instances: *client1Spec*, *client2Spec* and *client3Spec*.
- Object *client1Ab* is an instance of *ClientAb* and models *client1* as two divisions (*client1a* and *client1b*). Similarly, object *client2Ab* models *client2* as two divisions (*client2a* and *client2b*), and object *client3Ab* models *client3* as two divisions (*client3a* and *client3b*).

Figure 3-2
Overall organisation of the BPS system specification modelling. The models in the top half of this diagram are those presented in Section 2.3 Modelling Organisation for system requirements. The same colour convention applies.

```
configuration BPS_SPEC

  // For System Requirements
  ...

  // For System Specifications
  class          "*BpsDivSpec*"   alias BpsDivSpec;
  class          "*Component*"    alias Component;
  class          "*ProtectedComponent*"    alias ProtectedComponent;
  object         "*bpsSpec*"      alias bpsSpec;
  contract       "*spec*"         alias spec;

  class          "*ClientDiv*"    alias ClientDiv;
  class          "*ClientAb*"     alias ClientAb;
  contract       "*ClientSpec*"   alias ClientSpec;
  contract       client1Spec = ClientSpec(client1, clientAb1);
  contract       client2Spec = ClientSpec(client2, clientAb2);
  contract       client3Spec = ClientSpec(client3, clientAb3);
  property model "*CLIENTS_AB*"   alias CLIENTS_AB;

  contract       "*HsiDiv*"       alias HsiDiv;
  contract       hsiA = HsiDiv(bpsASpec, maintBtn1, testBtn1);
  contract       hsiB = HsiDiv(bpsBSpec, maintBtn2, testBtn2);

  end BPS_SPEC;
```

3.4. The *HsiDiv* Standard Contract

Through the instantiation of this standard contract, the *op* model provides 4 buttons to the *BPS* division given as the first argument: to start and end a maintenance session, to start a test and to reset. The *end maintenance* and the *reset* buttons are shared by the two *BPS* divisions. The *maintenance* and the *test* buttons are specific to each division, and are therefore also given as arguments to the standard contract.

```
contract HsiDiv
  (object bpsDiv, Boolean maintBtn, Boolean testBtn)

  party op
    Boolean maintBtn;
    requirement b3 =
      after maintBtn becomes true  for 0.5*s check maintBtn;
    requirement b4 =
      after maintBtn becomes false for 0.5*s check not maintBtn;

    Boolean endMaintBtn;
    requirement b5 =
      after endMaintBtn becomes true  for 0.5*s check endMaintBtn;
    requirement b6 =
      after endMaintBtn becomes false for 0.5*s check not endMaintBtn;

    Boolean testBtn;
    requirement b7 =
      after testBtn becomes true  for 0.5*s check testBtn;
    requirement b8 =
      after testBtn becomes false for 0.5*s check not testBtn;

    end op;

    Boolean resetBtn;
    requirement b1;
    requirement b2;

  party bpsDiv
    end bpsDiv;
```

```
   end HsiDiv;
```

3.5. The *op* Model

Values for the six buttons will be given by scenario models, and are therefore declared external. The eAlarm event is delivered by the *bpsReq* model (through contract *hsi*) and is also declared external.

```
object model op

   external Boolean maintBtn1;
   external Boolean maintBtn2;
   external Boolean endMaintBtn;
   external Boolean testBtn1;
   external Boolean testBtn2;
   external Boolean resetBtn;

   external event eAlarm;

   contract hsi;    // with bpsReq
   contract hsiA;   // with bpsASpec
   contract hsiB;   // with bpsBSpec

   end op;
```

3.6. The *ClientAb* and *ClientDiv* Classes

Class *ClientAb* models a client as two divisions. Features that are identical for both divisions are directly modelled in the *ClientAb* class:

- Whether the client is *active*.
- The parameters of the formula determining how fast the tolerance potential *tolPot* of a division is consumed or replenished: *kc* determines the linear rate at which the potential is consumed when the client is active and the division not powered, and *kr* the linear rate at which the potential is replenished when the division is powered.

Features that are different are modelled in embedded class *ClientDiv*:

- Each division is *powered* independently of the other.
- Consequently, its tolerance potential *tolPot* is also its own.

```
class ClientAb
   // Shared by the two divisions
   external Boolean active;
   specific constant Rate kc;
   specific constant Rate kr;

   class ClientDiv
      external Boolean powered;
      Real tolPot
         start = 100.;
         during not powered and active and tolPot > 0.
            then derivative(tolPot) = -kc;
         during powered and tolPot < 100. then derivative(tolPot) = kr;
         else fixed;
         end tolPot;
      end ClientDiv;

   ClientDiv divA;
   ClientDiv divB;

   // Consolidation across divisions
   Boolean powered = divA.powered or divB.powered;
   Real tolPot = largest {divA.tolPot, divB.tolPot};
   end ClientAb;
```

3.7. The *ClientSpec* Standard Contract

This standard contract specifies the mutual obligations between the model of a client viewed at the level of the *BPS* system requirements (where the client is viewed as a black box), and of the model of the same client viewed at the level of the *BPS* system specification (where the client divisions are identified).

```
contract ClientSpec (client, clientAb)

   party client
      Boolean active;
      requirement c1;
      end client;

   party clientAb
      Boolean powered;
      Real tolPot;
      end clientAb;

   end ClientSpec;
```

3.8. Clients

```
property model ClientsAB

   ClientAb client1Ab
      kc = 4./s;
      kr = 0.15/s;
      contract client1Spec;
      end client1Ab;

   ClientAB client2Ab
      kc = 2.5/s;
      kr = 0.3/s;
      contract client2Spec;
      end client2Ab;

   ClientAB client3Ab
      kc = 2./s;
      kr = 0.15/s;
      contract client3Spec;
      end client3Ab;

   ClientDiv{} clientDivs =
      {client1Ab.divA, client1Ab.divB,
       client2Ab.divA, client2Ab.divB,
       client3Ab.divA, client3Ab.divB};

   end ClientsAB;
```

3.9. The *Component* Class

The *BPS* specification has fault-tolerance requirements, even though the *BPS* components are yet unknown at specification time. Thus, components are represented by external sets of instances of class *Component*. For the purposes of the *bpsSpec* model, the only information that is needed on a component is whether it is in a failed state. This is represented by a Boolean attached to each component and declared as external to specify that for all components, its value is to be obtained later from the design. It is assumed that there are no components repairs during periods where the *BPS* is needed.

```
partial class Component
  external Boolean failed;
  assumption noRepair =
    during voltage < 190.*v check no failed becomes false;
  end Component;
```

3.10. The *ProtectedComponent* Class

Some components are of particular importance and difficult to replace. Thus, they are subject to particular care during operation and are modelled as instances of partial class *ProtectedComponent*, which is an extension of class *Component*. A protected component can be operated either in a *protected* mode (for example during tests, in which case it will be stopped if there is a risk of damaging it) or in a *workToDeath* mode (when the *BPS* is really needed, in which case the component will be operated even if there is a risk of damaging it).

```
partial class ProtectedComponent extends Component
  specific automaton mode = (protected, workToDeath);
  end ProtectedComponent;
```

3.11. The *BpsDivSpec* Class

This class models a *BPS* division.

```
class BpsDivSpec
```

3.11.1.Monitoring of the *MPS*

Inputs from the *MPS*

The only input from the *MPS* to be provided to the *BPS* division is the electric tension at the *MPS* terminals. It will be defined by each instance.

```
  specific Voltage voltage;
```

Guards

The *BPS* division applies a time filtering scheme that could last up to 3 seconds to determine whether the *MPS* is available. The following guard states that no demand should be placed on the *BPS* division in the 3 first seconds of an operational case, to give the time filtering scheme time to initialise. In real life, an *MPS* loss in this time period is also very unlikely, and as the overall installation has not had time to really operate, this is considered to have no damaging consequences anyway.

```
  guard g1 = until time = 3.*s check voltage > 215.*v;
```

Instantaneous *on* / *off* State

The main information regarding the *MPS* is whether it is available or not. The *BPS* division does not determine that itself: how that is done will be decided at a later stage by a *BPS* design. It just puts constraints on how the decision is made.

Based on *voltage*, the *BPS* division first determines the instantaneous *on* / *off* state of the *MPS*. That state is represented by a Boolean named *mpsOn* (that will be obtained at a later stage from the design). For legibility and to facilitate understanding, Boolean *mpsOff* is defined as the negation of *mpsOn*. *BpsDivSpec* specifies the constraints that apply to these Booleans (see Figure 3-3):

- Initially, the *MPS* is assumed to be *on*.
- It CAN be declared *off* when *voltage* gets below 190 volts.
- It MUST be declared *off* when *voltage* gets below 180 volts.
- It CAN be declared *on* again when *voltage* gets above 200 volts.
- It MUST be declared *on* again when *voltage* gets above 210 volts.

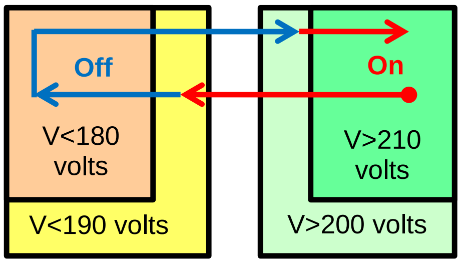


Figure 3-3
In requirements specification, boundaries must be specified with a certain "thickness" and hysteresis. The design must ensure that the transitions are indeed performed within the specified boundaries.

```
  external Boolean mpsOn;
  Boolean mpsOff = not mpsOn;
  assumption aMps1 = during voltage > 190*v check not mpsOff;
  assumption aMps2 = during voltage < 180*v check mpsOff;
  assumption aMps3 = during voltage < 200*v check not mpsOn;
  assumption aMps4 = during voltage > 210*v check mpsOn;
```

Time-filtred *available* / *unavailable* State

The *BPS* division should not be needlessly activated in case of short, transient voltage drops at the *MPS* terminals. Thus, whether the *MPS* is really available is represented by another external Boolean named *mpsAvailable* (also to be obtained from the design). The constraints specified by *BpsDivSpec* are based on a time-filtering scheme applied to *mpsOn* (see Figure 3-4):

- Initially, the *MPS* is assumed to be available.
- It CAN be declared unavailable when it has been off for more than 2 consecutive seconds.
- It MUST be declared unavailable when it has been off for more than 3 consecutive seconds.
- It CAN be declared available again when it has been on for more than 8 consecutive seconds.
- It MUST be declared available again when it has been on for more than 10 consecutive seconds.

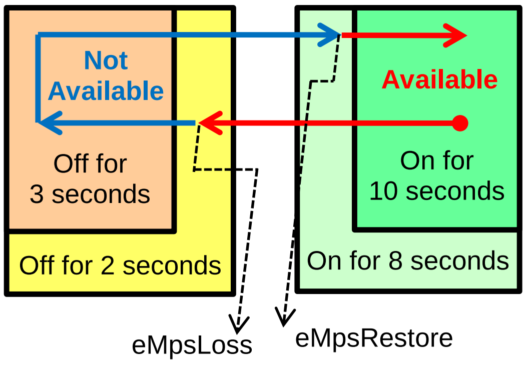


Figure 3-4
The Available / Unavailable transitions

For legibility and to facilitate understanding, Boolean *mpsUnavailable* is defined as the negation of *available*, and two events are introduced:

- Event *eMpsLoss* signals that the *MPS* has transited from *available* to *unavailable*.
- Event *eMpsRestore* signals that the *MPS* has transited from *unavailable* to *available*.

```
  external Boolean mpsAvailable;
  Boolean mpsUnavailable = not mpsAvailable;
  event eMpsLoss    = when mpsAvailable becomes false;
  event eMpsRestore = when mpsAvailable becomes true;
  private property off2s = duringAny  2*s check mpsOff;
  private property off3s = duringAny  3*s check mpsOff;
  private property on8s  = duringAny  8*s check mpsOn;
  private property on10s = duringAny 10*s check mpsOn;

  assumption aMps5 =
     during (mpsAvailable and not off2s) check no eMpsLoss;
  assumption aMps6 = during off3s check mpsUnavailable;
  assumption aMps7 =
     during (mpsUnavailable and not on8s) check no eMpsRestore;
  assumption aMps8 = during on10s check mpsAvailable;
```

3.11.2.Validation of Operator Requests

Operator Inputs

A *BPS* division receives four Boolean signals from the operator. They will be obtained by each instance through an instantiation of standard contract *hsiDiv*.

```
  external Boolean maintBtn;
  external Boolean endMBtn;
  external Boolean testBtn;
  external Boolean resetBtn;
```

Guards

The following guards state that the operator will not place a demand to the *BPS* division in the 3 first seconds of an operational case, to give the time filtering scheme time to initialise.

```
  guard g2 = until time = 3.*s check not maintBtn;
  guard g3 = until time = 3.*s check not endMBtn;
  guard g4 = until time = 3.*s check not testBtn;
  guard g5 = until time = 3.*s check not resetBtn;
```

The operator can issue orders to the *BPS* division, but the *BPS* division should verify that these orders are valid:

- A maintenance request is valid if and only if it is made when the *MPS* is available and the *BPS* is not active (i.e., it is not providing power to any of its clients).
- A test request is valid if and only if it is made when the *MPS* is available, the *BPS* is not active and no maintenance is being performed.
- A reset request is valid if and only if it is made when the *MPS* is available, the *BPS* is active, and at least 10 minutes after the *BPS* has started to operate.

```
  // For legibility
  event eMaint    = when maintBtn becomes true;
  event endMaint = when endMBtn  becomes true;
  event eTest     = when testBtn  becomes true;
  event eReset    = when resetBtn becomes true;

  // Defined later in the model
  automaton state = (normal, maintenance, test);
  event eBegins;      // When BPS becomes active
  Boolean needed;     // Whether the BPS is needed

  // Validating operator requests
  event eVMaint =
     (during mpsAvailable and normal and not needed) and eMaint;
  event eVTest =
     (during mpsAvailable and normal and not needed) and eTest;
  event eVReset =
     (during mpsAvailable) and (after eBegins + 10*mn) and eReset;
```

3.11.3.Operational States

The three states introduced in Section 3.1.2 *BPS States* and the associated transition criteria are represented by an automaton named *state*, as shown in Figure 3-5:

- The initial state is the *normal* state. In that state, the *BPS* stands ready to perform, or is performing, its main mission, which is to provide electric power to its clients should the *MPS* become unavailable.
- In the *test* state, the *BPS* division acts as if the *MPS* had become unavailable. The difference with a loss of *MPS* in the *normal* state is that in the *test* state, the *BPS* division operates its critical components in the *protection* mode, whereas in the *normal* state, it operates them in the *workToDeath* mode.
- In the *maintenance* state, the *BPS* division must remain inactive, even when the *MPS* is or becomes unavailable.

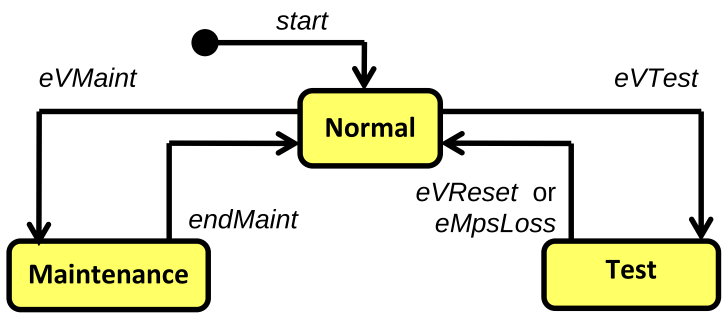


Figure 3-5
The main states of the BPS*, and the corresponding state transitions*

```
   automaton state = (normal, maintenance, test)
     start = normal;
     when normal and eVMaint then state = maintenance;
     when normal and eVTest then state = test;
     when maintenance and endMaint then state = normal;
     when test and (eMpsLoss or eVReset) then state = normal;
     else fixed;
     end state;
```

3.11.4.Fault-Tolerance

Some of the requirements of the *BPS* system specification express conditions like "*in the absence of any component failure*" or "*in the presence of at most a single sensor failure*". To that end, *BpsDivSpec* declares three sets of components:

- *sensors* is the set of all the sensors of the *BPS* division. Its precise membership is unknown to *BpsDivSpec*: the information will be obtained from the design.
- *nonSensors* is the set of all components of the *BPS* division that are not sensors. Like the previous set, its membership will be obtained from the design.
- *components* is the set of all the components of the *BPS* division. It is the union of the two previous sets.

```
   external Component{} nonSensors;
   external Component{} sensors;
   Component{} components = nonSensors union sensors;
```

Some of the *BPS* system specification requirements concern critical components that should be protected during tests. To that end, *BpsDivSpec* declares the set of such components:

```
   external ProtectedComponent{} protectedComponents;
```

Condition "*in the absence of any BPS component failure or in the presence of a single BPS sensor failure*" is represented by a Boolean named *tolerance*:

```
Boolean tolerance =
   (cardinal {c in nonSensors suchThat c.failed} = 0) and
   (cardinal {c in sensors    suchThat c.failed} ≤ 1);
```

3.11.5.Alarms

When due to component failures the *BPS* division cannot perform its main mission, it must send a signal to the operators in the control room. This signal is represented by an event named *eAlarm*. Its precise instants of occurrence are to be obtained from the design and it is therefore declared external.

```
external event eAlarm;
```

3.11.6.Activity

When in *normal* state, the *BPS* division can either be in standby (it then just monitors the availability of the *MPS* and the operator requests) or be active (it then performs whatever is necessary to provide electric power to its clients).

Boolean *needed* specifies when the *BPS* division needs to be active.

```
Boolean needed =
   after
      (not maintenance and (eMpsLoss or eVTest))
      or (mpsUnavailable and endMaint)
   until eVReset;
```

Boolean *active* specifies when the *BPS* division is indeed active. As it is determined by design, it is declared external.

```
external Boolean active;
```

To enhance clarity, two derived events signal when the *BPS* becomes, or ceases to be, active.

```
event eBegins = when active becomes true;
```

3.11.7.Powering Clients

Whether a *BPS* division is configured to provide power to one of the six client divisions is represented by six Booleans. They are external since they will be determined by the design: the system specification expresses constraints on when they should become true.

```
external Boolean powering1a;
external Boolean powering1b;
external Boolean powering2a;
external Boolean powering2b;
external Boolean powering3a;
external Boolean powering3b;
```

3.11.8.Specification

Specification *s1*

***s1*:** When the *BPS* division is under maintenance, it must not be activated.

This translates as:

```
requirement s1 = during maintenance check not active;
```

Specification *s2a*

s2a: In the absence of any *BPS* division component failure or in the presence of a single sensor failure, when the division is not in maintenance, and in case of loss of *MPS*, if *client1* is active:

- Its divisions must be powered within 25 seconds.
- Power must be provided until a valid reset from the operator.

Note. The grace times specified in the informal requirements specification for *s2a*, *s2b* and *s2c* include the time needed to confirm that the *mps* is indeed unavailable, which can take up to 3 seconds. Therefore, these grace times must be reduced by 3 seconds.

```
  private property p2a =
     during (needed and client1Active)  trunc 22*s
     check powering1a and powering1b;

  requirement s2a = during tolerance check not p2a.violated;
```

If *client1* is not active, it must not be powered.

```
  private property p2A =
     during (needed and not client1Active)
     check not (powering1a or powering1b);

  requirement s2A = during tolerance check not p2A.violated;
```

Specification *s2b*

s2b: In the absence of any *BPS* division component failure or in the presence of a single sensor failure, when the division is not in maintenance, and in case of loss of *the MPS*:

- The divisions of *client2* must be powered within 35 seconds.
- Power must be provided until a valid reset from the operator.

```
  private property p2b =
     during needed trunc 32*s
     check powering2a and powering2b;

  requirement s2b = during tolerance check not p2b.violated;
```

Specification s*2c*

s2c: In the absence of any *BPS* division component failure or in the presence of a single sensor failure, when the division is not in maintenance, and in case of loss of *the MPS*:

- The divisions of *client3* must be powered within 45 seconds.
- Power must be provided until a valid reset from the operator.

```
  private property p2c =
     during needed trunc 42*s
     check powering3a and powering3b;

  requirement s2C = during tolerance check not p2C.violated;
```

Specification *s3a*

s3a: The probability of not powering one or more client divisions in the allocated time (due to *BPS* component failures) must be less than 10^{-3} per demand.

```
  private property p3a =
     when (needed and client1.active) becomes true + 22.*s
     check powering1a and powering1b;

  private property p3b =
     when needed becomes true + 32.*s
     check powering2a and powering2b;

  private property p3c =
     when needed becomes true + 42.*s
```

```
    check powering3a and powering3b;

  requirement s3a = check probability
    ((p3a.violated or p3b.violated or p3c.violated) becomes true) < 10-3;
```

Specification *s3b*

s3b: The *BPS* division failure rate while it powers its clients must be less than what it would be if there was a constant failure rate of 10^{-3} per day.

```
  requirement s3b = check probability
    ((p2a.violated or p2b.violated or p2c.violated) becomes true)
      < 1-exp(-10-3 * duration(needed) /day);
```

Specification *s4*

s4: In the absence of any *BPS* division component failure or in the presence of a single sensor failure, the division must not be spuriously activated.

```
  private property p4 = during not needed check not active;
  requirement s4 = during tolerance check not p4.violated;
```

Specification *s5*

s5: The probability of spuriously activating the *BPS* division must be lower than what it would be if there were a constant occurrence rate of spurious activation equal to 10^{-3} per year.

```
  requirement s5 = check probability (p4.violated becomes true)
    < 1 - exp(-10-3 * duration(not needed) / year);
```

Specification *s6a*

s6a: When the *BPS* division is active to mitigate a loss of the *MPS*, its protected components must be operated at any cost, even if this means damaging or eventually destroying them in the long run.

Here again, the 200 milliseconds give time for the *BPS* to react:

```
  requirement s6a =
    during (normal and active) trunc 200*ms
    check AND{c.mode = workToDeath forAll c in protectedComponents};
```

Specification *s6b*

s6b: When the *BPS* division is operated for a periodic test, the risk of damaging its protected components must be minimised..

```
  requirement s6b =
    during (test and active) trunc 200*ms
    check AND{c.mode = protection forAll c in protectedComponents};
```

Specification *s7a*

s7a: In the absence of any *BPS* division component failure or in the presence of a single sensor failure, when a valid Reset is issued by the operator, and if the *MPS* is not lost again, the normal condition for powering the client divisions (i.e., powered by the *MPS*) must be restored within 10 seconds.

normalPower is an intermediate Boolean defining normal power conditions, where all the clients are powered by the *MPS* and the *BPS* is not active:

```
  private Boolean normalPower =
    mpsAvailable
      and (not active)
      and AND{d.powered forAll d in clientDivs};

  requirement s7a =
    during tolerance and after eVReset + 10*s
    check normalPower;
```

Specification *s7b*

s7b: In the absence of any BPS component failure or in the presence of a single BPS sensor failure, when a valid Reset is issued by the operator, no client division should be unpowered for more than a total of 5 seconds after the *Reset*.

```
  property p7b =
     after eVReset until normalPower
     check AND{inPDuration(not d.powered) < 5*s forAll d in clientDivs};

  requirement s7b = during tolerance check not p7b.violated;
```

Specification *s8*

s8: The probability of not restoring the normal powering conditions for all the client divisions within 10 seconds after a valid Reset must be less than 10^{-3} per *BPS* demand.

```
  requirement s8 =
     after eVReset within 10.*s
     check probability (p7a.violated becomes true) < 10⁻³;
```

Specification *s9*

psa: If any client cannot be powered as specified in *s2* and *s7*, a failure signal should be sent in the control room to inform the operator.

```
  property psa =
     after (
        (  s2a.violated
        or s2b.violated
        or s2c.violated
        or s7.violated) becomes true) within 200.*ms
     check once eAlarm;
```

s9 The probability of not sending a failure signal when required must be less than 10^{-3} per demand.

```
  requirement s9 = check probability (psa.violated becomes true) < 10⁻³;
```

Specification *s10*

s10: That failure signal should not be raised spuriously. The rate of spurious failure signal must be less than what it would be if there was a constant failure rate of 10^{-2} per year.

```
  property p10 =
     during not(
        (  s2a.violated
        or s2b.violated
        or s2c.violated
        or s7.violated) becomes true)
     check no eAlarm;

  end BpsDivSpec;
```

3.12. The *bpsSpec* Object

This object models the *BPS* as two divisions (*bpsASpec* and *bpsBSpec*) and six switches, each connecting a client division to one and only one *BPS* division.

```
object bpsSpec
```

Contracts

By this contract declaration, the *bpsSpec* model acknowledges the contract that binds it to the *bpsReq* model: the *bpsReq* model is the reference model, and the *bpsSpec* model is a solution model.

```
  contract spec;
```

Switches

The six switches connecting the six client divisions to a *BPS* division are represented by a Boolean vector: when a switch is true, then the corresponding client division is connected to the *BPSa* division; else it is connected to the *BPSb* division. The normal situation is that *BPSa* services one division of each client, and *BPSb* the other division. When one *BPS* division is set in maintenance mode, then the client divisions it normally services are switched to the other *BPS* division. They are retunned to it when it is no longer in maintenance.

```
Boolean[6] switch
   start[1, 3, 5] = true;
   start[2, 4, 6] = false;
   when bpsASpec.maintenance becomes true  then switch[1, 3, 5] = false;
   when bpsASpec.maintenance becomes false then switch[1, 3, 5] = true;
   when bpsBSpec.maintenance becomes true  then switch[2, 4, 6] = true;
   when bpsASpec.maintenance becomes false then switch[2, 4, 6] = false;
   else fixed;
   end switch;
```

BPS Divisions

```
BpsDivSpec bpsASpec
   contract hsiA;
   contract odsA;
   voltage = bpsSpec.voltage;
   active =
      (if switch[1] then client1a.active else false) or
      (if switch[2] then client1b.active else false);
   end bpsASpec;

BpsDivSpec bpsBSpec
   contract hsiB;
   contract odsB;
   voltage = bpsSpec.voltage;
   active =
      (if switch[1] then false else client1a.active) or
      (if switch[2] then false else client1b.active);
   end bpsBSpec;
```

Deliverables to *bpsReq*

The *bpsReq* model expects 4 deliverables from any solution model: the three powering Booleans and the *eAlarm* event. The following consolidates the powering determinations made separately by *bpsASpec* and *bpsBSpec* concerning the client divisions they service.

```
Boolean powering1a = // whether division a of client 1 is powered by BPS
   if switch[1] then bpsASpec.powering1a else bpsBSpec.powering1a;
Boolean powering1b = // whether division b of client 1 is powered by BPS
   if switch[2] then bpsASpec.powering1b else bpsBSpec.powering1b;
Boolean powering1 =  // client 1 is powered if any of its division is
   powering1a or powering1b;

Boolean powering2a =
   if switch[3] then bpsASpec.powering2a else bpsBSpec.powering2a;
Boolean powering2b =
   if switch[4] then bpsASpec.powering2b else bpsBSpec.powering2b;
Boolean powering2 =
   powering2a or powering2b;

Boolean powering3a =
   if switch[5] then bpsASpec.powering3a else bpsBSpec.powering3a;
Boolean powering3b =
   if switch[6] then bpsASpec.powering3b else bpsBSpec.powering3b;
Boolean powering3 =
   powering3a or powering3b;

// An alarm is signalled if any of the BPS divisions detects a failure
event eAlarm = bpsASpec.eAlarm or bpsBSpec.eAlarm;
```

```
  end bpsSpec;
```

3.13. The *spec* Contract

The *BPS* system specification is modelled as a contract named *spec* between *bpsReq* and *bpsSpec*. Here, the contract just declares the mutual deliverables and the requirements that *bpsSpec* guarantees to *bpsReq*: the definitions are provided by the *bpsSpec* model. Specifications *6a* and *6b* are not listed, because the are not required by *bpsReq* but have been added by *bpsSpec*. For a configuration where *bpsReq* is declared as a reference model and *bpsSpec* as a solution model, a test case generator will consider *bpsSpec*'s requirements as assumptions to emulate an implementation consistent with the model.

```
contract spec

  party bpsReq
    Voltage voltage;                // at the terminals of the mps
    end bpsReq;

  party bpsSpec
    // Deliverables
    Boolean powering1;
    Boolean powering2;
    Boolean powering3;
    event eAlarm;

    // Guarantees
    requirement bpsASpec.s1;
    requirement bpsBSpec.s1;
    requirement bpsASpec.s2a;
    requirement bpsBSpec.s2a;
    requirement bpsASpec.s2A;
    requirement bpsBSpec.s2A;
    requirement bpsASpec.s2b;
    requirement bpsBSpec.s2b;
    requirement bpsASpec.s2c;
    requirement bpsBSpec.s2c;
    requirement bpsASpec.s3a;
    requirement bpsBSpec.s3a;
    requirement bpsASpec.s3b;
    requirement bpsBSpec.s3b;
    requirement bpsASpec.s4;
    requirement bpsBSpec.s4;
    requirement bpsASpec.s5;
    requirement bpsBSpec.s5;
    requirement bpsASpec.s7a;
    requirement bpsBSpec.s7a;
    requirement bpsASpec.s7b;
    requirement bpsBSpec.s7b;
    requirement bpsASpec.s8;
    requirement bpsBSpec.s8;
    requirement bpsASpec.s9;
    requirement bpsBSpec.s9;
    requirement bpsASpec.s10;
    requirement bpsBSpec.s10;
    end bpsSpec;
  end spec;
```

4. *BPS* Overall Design

4.1. Brief Introduction to the Overall Design

The Overall Design:

- Identifies the main components of a *BPS* division.
- Specifies the interactions between these components.
- States the assumptions made regarding the behaviour of each component.

Diesel Generator *DG*

As the level of power required by the clients is in the megawatts range and the duration of the backing-up could last up to several days, a *BPS* division is based on a Diesel Generator, or *DG*. This generator has a number of constraints. In particular, if all clients divisions were connected simultaneously, it would be overloaded and stall. In addition, when electric power is restored to a given component, there is an important, transient call for current and power (see Figure 4-1). Thus, much like a conventional car engine needs a gearbox, the *BPS* division and its *DG* need an active control system to ensure a progressive and orderly increase of requested power (hence the presence of circuit breakers).

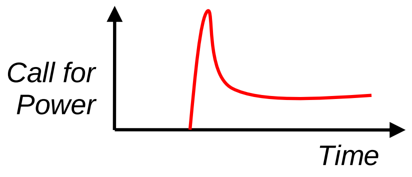


Figure 4-1
Transient call for current when electric power is supplied to an electric component.

Other *BPS* Components

Figure 4-2 presents the overall design of a *BPS* division and its main components. It introduces a number of design decisions compared with Figure 3-1. In particular:

- The *DG* is the only protected component.
- Each client division has a circuit breaker named *brk$_i$*, where i = *1a*, *1b*, *1c*, *2a*, *2b* and *2c*. It can isolate the client division from the *Backed-Up Electric Panel*.
- The breakers are controlled by a digital control system called the *Load Sequencer*, or *LS*. Its role is to send control signals to the *DG* and the various circuit breakers, based on information provided by the sensors and the human operators. It also sends a failure signal to the operators when necessary.
- The *MPS* has a circuit breaker named *brkMps*. The breaker is controlled by the *LS* and can isolate the *MPS* from the *Backed-Up Electric Panel*.
- The *DG* has a circuit breaker named *brkDg*. The breaker is controlled by the *LS* and can isolate the *DG* from the *Backed-Up Electric Panel*.
- In addition to the circuit breakers of the *BPS* division, the two divisions of *client1* have each a second circuit breaker named respectively *brk1* and *brk2* that are controlled by *client1* itself. They are closed when *client1* is active, independently of the *BPS*.
- Sensors have been added. In Figure 4-2, they are represented as circles. All provide Boolean outputs:

- *nbV1* and *nbV2* are two redundant voltmeters that determine whether the tension at the terminals of the *Non-Backed-Up Electric Panel* is *on* or *off*.
- *bV1* and *bV2* are two redundant voltmeters that determine whether the tension at the terminals of the *Backed-Up Electric Panel* is *on* or *off*.
- *dgV1* and *dgV2* are two redundant voltmeters that determine whether the tension at the terminals of the *DG* is *on* or *off*.
- *dgF1* and *dgF2* are two redundant frequency meters that determine whether the current frequency at the terminals of the *DG* is appropriate or not.
- *mpsPos1* and *mpsPos2* are two redundant position sensors signalling the *open* / *close* position of the *MPS* breaker.
- *dgPos1* and *dgPos2* are two redundant position sensors signalling the *open* / *close* position of the *DG* breaker.
- *brk1Pos* is a position sensor signalling the *open* / *close* position of the *brk1* breaker.
- *brk2Pos* is a position sensor signalling the *open* / *close* position of the *brk2* breaker.

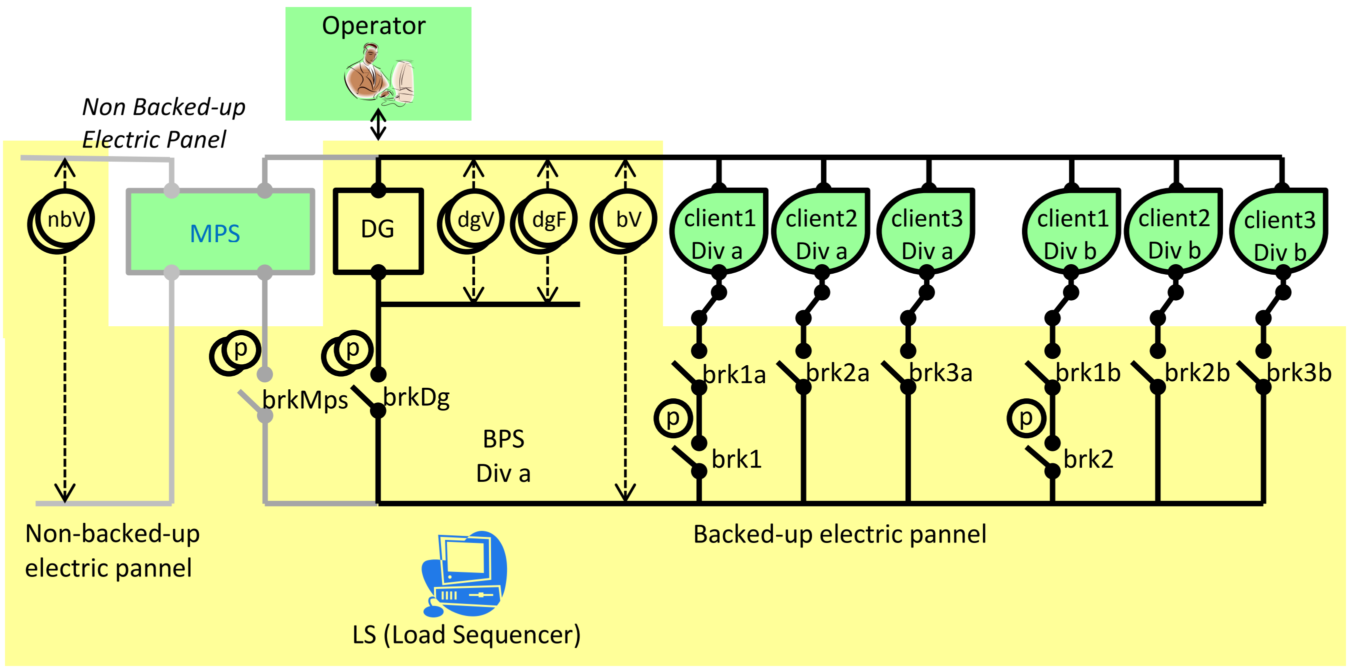


Figure 4-2
Overall Design of a BPS division.

Note. This overall design may not comply with all the system requirements and the system specification specified in Sections 2 *BPS System Requirements* and 3 *BPS System Specification*. Indeed, it is very likely that it does not comply with some of the fault tolerance requirements. It will be up to the verification to identify discrepancies, and to the engineering team to decide whether it is the requirements, the specification or the design that needs to be amended, possibly on the basis of a cost-benefit analysis.

Overall Design Assumptions

The Overall Design presented here "allocates" the system specification of Section 4 to the identified components. This is done in the form of requirements specified for each component. However, at that stage, the choices of products to be used for the components are not made yet: this will be decided by the Detailed Design. Therefore, the precise characteristics of these components (such as the start-up time of the *DG*, the accuracy and response time of sensors, the response time of breakers) are not known yet.

The following pattern is used to model these requirements and unknowns:

```
// Characteristics of a Component
external constant Real realValue;        // Decided by Detailed Design
constant Real maxValue = ...;            // Value allocated by ODS
requirement a = realValue ≤ maxValue;
```

4.2. Modelling Organisation of the *Overall Design*

Following the decision made in the previous Section, the overall design of the BPS is composed of two identical divisions. Since the relationships between these divisions and the complete BPS have already been specified and modelled in the previous Section, the overall design (*ODS*) presented here concerns a single BPS division. It is mainly organised into class definitions, and Figure 4-4 shows their extension hierarchy) that are intantiated for each of the divisions. Figure 4-3 shows the main objects modelling the overall design:

- Classes *Component* and *ProtectedComponent* (an extension of *Component*) have already been introduced.
- Class *Breaker*, an extension of *Component*, models the electric circuit breakers.
- Class *MSensor*, an extension of *Component*, models the measurement sensors, and is itself extended by classes *Voltmeter* and *FreqMeter*.
- Class *PSensor*, an extension of *Component*, models the sensors informing on the open / close position of electric circuit breakers.
- Class *Dg*, an extension of *ProtectedComponent*, models the diesel generator that produces backup electricity for the *BPS* division.
- Class *Ls* models the synchronous, digital control system that orchestrates the actions of the *BPS* division.
- Class *BpsDivOds* models the overall design of a *BPS* division, based on the preceding classes.OIt has two instances: *bpsAOds* and *bpsBOds*.
- Standard contract *Ods* models the mutual obligations between an instance of *BpsDivSpec* (which models the specification of a *BPS* division) and the corresponding instance of *BpsDivOds*. It is instantiated into *odsA* (between *bpsASpec* and *bpsAOds*) and *odsB* (between *bpsBSpec* and *bpsBOds*).
- Class *Step* models additional information regarding a client division. Indeed, each client is itself in a parallel design process to the *BPS*, and new information is made available at each step. The six client divisions are modelled as a single *Step* vector named *step* and shared by the two *BPS* divisions.

Note. A client division is called a *step*, due to the fact that a *BPS* division will address them sequentially.

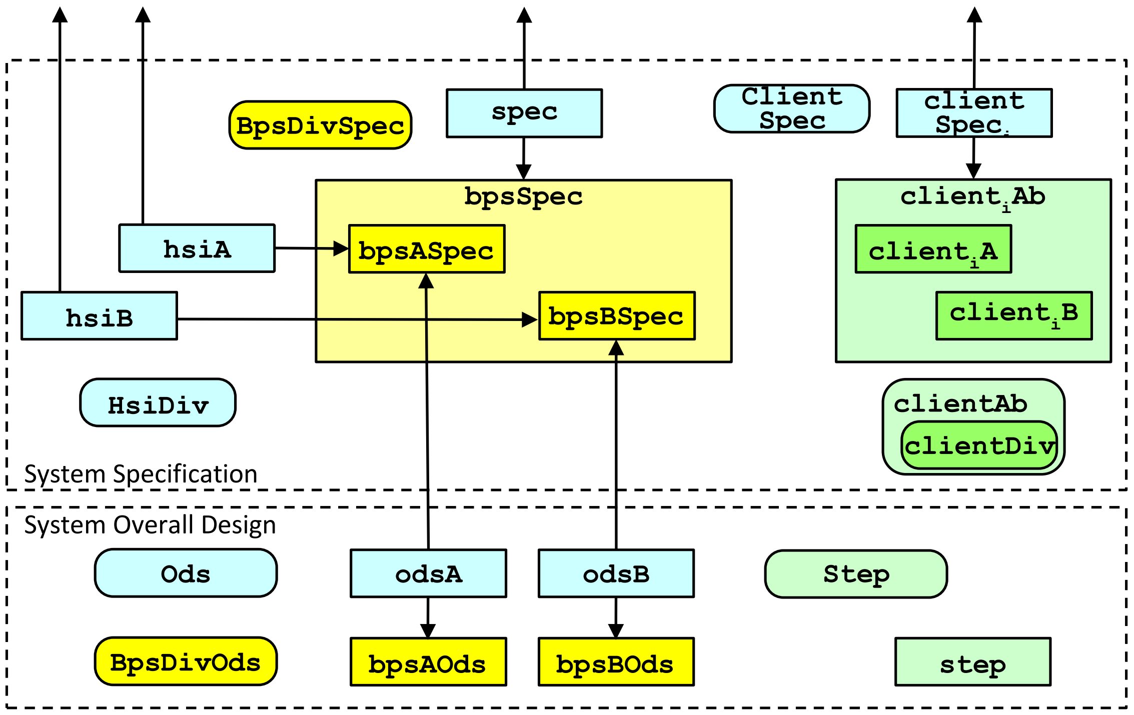


Figure 4-3
The two objects bpsAOds and bpsBOds model the overall design of the BPS disivsions. Object step models the clients divisions from the point of view of the overall design of the BPS.

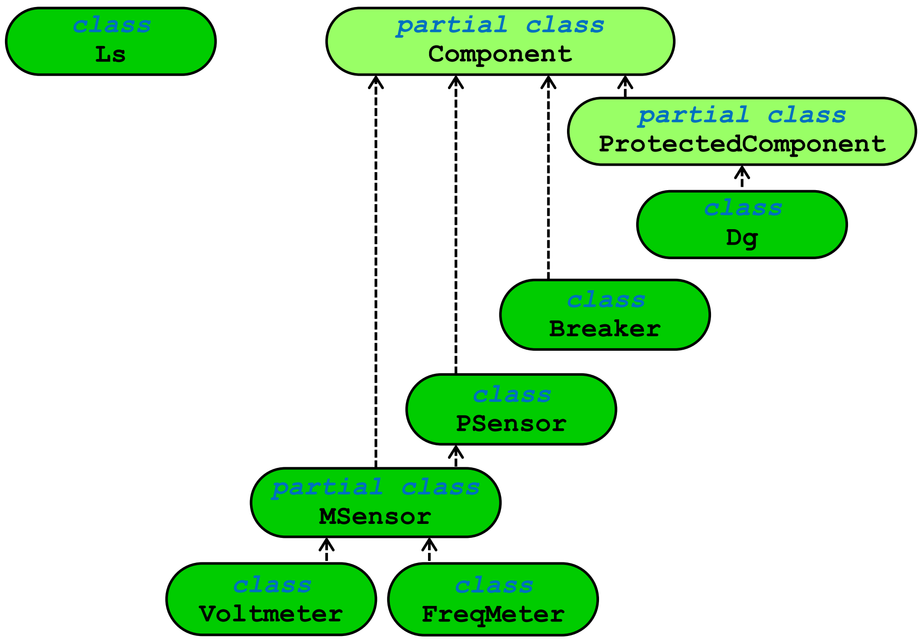


Figure 4-4
Extension hierarchy of the classes used in the overall design of a BPS division.

```
configuration BPS_ODS

  // For System Requirements
  ...

  // For System Specifications
  ...

  // For System Overall Design
  class          "*Step*"        alias Step;
  class          "*Breaker*"     alias Breaker;
  class          "*MSensor*"     alias MSensor;
  class          "*Voltmeter*"   alias Voltmeter;
  class          "*FreqMeter*"   alias FreqMeter;
  class          "*PSensor*"     alias PSensor;
  class          "*Dg*"          alias Dg;
  class          "*Ls*"          alias Ls;
  class          "*BpsDivOds*"   alias BpsDivOds;

  property mode  "*BpsOds*"      alias BpsOds;
  contract       "*Ods*"         alias Ods;
  contract       odsA = Ods(bpsASpec, bpsAOds);
  contract       odsB = Ods(bpsBSpec, bpsBOds);

  end BPS_SPEC;
```

4.3. The *Breaker* Class

```
class Breaker extends Component
```

A *BPS* circuit breaker can receive two Boolean signals, one to open (*openCmd*) and one to close (*closeCmd*).

```
  Boolean openCmd;
  Boolean closeCmd;
```

Opening starts when *openCmd* becomes true. This is represented by event *eOpenCmd*. Closing starts when *closeCmd* becomes true. This is represented by event *eCloseCmd*.

```
  event eOpenCmd  = when openCmd becomes true;
  event eCloseCmd = when closeCmd becomes true;
```

The open / close signals are in fact issued by the *LS*. It is assumed that there are no contradictions:

```
  assumption noContradiction = no (eOpenCmd and eCloseCmd);
```

A real circuit breaker cannot open or close instantly. The maximum delay for a given breaker (*realTMax*) depends on the specific product that will be chosen. That choice is made by detailed design and is not known yet at the time of overall design. The overall design just puts an upper bound value (*tMax*) for *realTMax*. That value is not necessarily the same for all the BPS breakers.

```
  specific constant Duration tMax;
  external constant Duration realTMax;             // From Detailed Design
  requirement a1 = realTMax isIn ]]0*s, tMax]];    // Overall Design Req.
```

The two following events represent the real instants when the breaker finishes opening or closing. They are to be provided by the detailed design behavioural model and are therefore declared external:

```
  external event eOpeningCompleted;
  external event eClosingCompleted;
```

A circuit breaker can be in one of four states: *open*, *closed*, *opening* and *closing*. This is represented by automaton *state*. The initial state depends on the instance: therefore, it is declared **specific**:

```
  automaton state = (open, closed, opening, closing)
    start = specific;
    when state ≠ open and eOpenCmd then opening;
    when eOpeningCompleted then open;
    when state ≠ closed and eCloseCmd then closing;
    when eClosingCompleted then closed;
    else fixed;
    end state;
```

The purpose of the following is to state formally the meaning of *tMax*: after an open / close order (events *eOpenCmd* / *eCloseCmd*), the breaker will complete the action (events *eOpeningCompleted* / *eClosingCompleted*) within a *tMax* delay, provided that in the meantime it does not receive a counter command:

```
  requirement openInTime =
    after eOpenCmd within tMax check eOpeningCompleted or eCloseCmd;
  requirement closeInTime =
    after eCloseCmd within tMax check eClosingCompleted or eOpenCmd;

  end Breaker;
```

4.4. The *MSensor* Class

This class represents the measurement sensors (voltmeters and frequency meters) used in the *BPS* division overall design. It is later extended by classes *Voltmeter* and *FreqMeter*.

```
partial class MSensor extends Component
```

input is the true analog signal to be measured. It will be provided by a behavioural or stimulation model.

```
  external Real input;
```

on is the output of the sensor. It is a Boolean signal stating the on / off state of the *input*. It will also be provided by the behavioural or stimulation model.

```
  external Boolean on;
```

lB1, *lB2*, *uB1* and *uB2* determine whether the output *on* must be true or not (see Figure 4-5). They are declared but their values will be specified by *Voltmeter* and *FreqMeter*:

```
  constant Real lB1;
  constant Real lB2;
  constant Real uB1;
  constant Real uB2;
  requirement consistency = (lB1 < lB2) and (lB2 < uB1) and (uB1 < uB2);
```

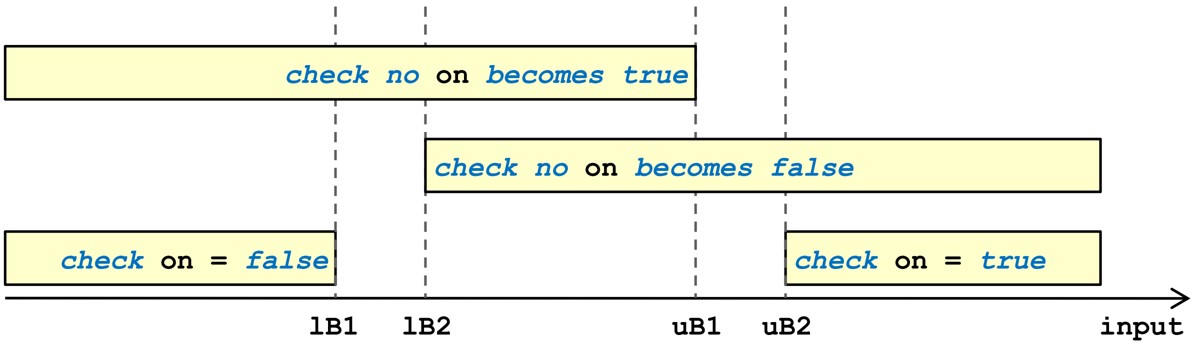


Figure 4-5
Constraints on output value

Measurement sensors need a time delay to adjust the output to the input. The real duration of that delay (*realTMax*) is declared *external* and will be provided by the detailed design model. The overall design just specifies an upper bound (*tMax*). This upper bound is specific to each type of *MSensor* and will be specified in the extension models:

```
   specific constant Duration tMax;
   external constant Duration realTMax;            // From Detailed Design
   requirement a1 = realTMax isIn ]]0*s, tMax]];
```

Measurement sensors have also a limited accuracy. However, this is taken into account in the determination of *lB1*, *lB2*, *uB1* and *uB2* and does not need to appear explicitly in the model.

Finally, the following specifies the overall design requirements regarding the output of the sensor:

```
   requirement r1 = during input > uB2 trunc tMax check on;
   requirement r2 = during input < lB1 trunc tMax check not on;;
   requirement r3 = during input > lB2 trunc tMax check no on becomes false;
   requirement r4 = during input < uB1 trunc tMax check no on becomes true;

   end MSensor;
```

4.5. The *Voltmeter* Class

All the voltmeters have the same features:

```
class Voltmeter extends MSensor

   lB1 = 180*v; lB2 = 190*v;
   uB1 = 200*v; uB2 = 210*v;
   tMax = 20*ms;            // Assumed by ODS

   end Voltmeter;
```

4.6. The *FreqMeter* Class

All the frequency meters have the same features:

```
class FreqMeter extends MSensor

   lB1 = 47*hz; lB2 = 48*hz;
   uB1 = 48*hz; uB2 = 49*hz;
   tMax = 100*ms;             // Assumed by ODS

   end FreqMeter;
```

4.7. The *PSensor* Class

```
class PSensor  extends Component
```

brk is the circuit breaker associated with the position sensor: i.e., the sensor determines the *open* / *close* state of *brk*. It is specific since each position sensor is associated to a specific breaker:

```
   specific associated Breaker brk;
```

open is the output of the sensor. It is a Boolean signal stating the Open / Close state of *brk*. It is declared external because it will be provided by the behavioural model of the detailed design:

```
   external Boolean open;
```

Position sensors need a time delay to adjust the output to the input. The real duration of that delay (*realTMax*) is declared external and will be provided by the detailed design model. The overall design just specifies an upper bound (*tMax*):

```
   specific constant Duration tMax;
   external constant Duration realTMax;          // From Detailed Design
   requirement a1 = realTMax isIn ]]0*s, tMax]];
```

Finally, the following specifies the overall design requirements regarding the output of the sensor:

```
   requirement r1 = during brk.state.closed trunc tMax check not open;
   requirement r2 = during brk.state.open   trunc tMax check open;
```

```
  end PSensor;
```

4.8. The *Step* Class

In the same way as the overall design of the BPS has added new information on the BPS, the overall design of the clients have added information on each client, in particular regarding the electric power required by each client division. In addition, the BPS overall design has assigned them a priority order: when required and allowed, the highest priority client division is reloaded first.

Class

```
class Step
```

Each *step* embeds a circuit breaker named *brk* that is used for shedding (disconnection from the Backed-Up Electric Panel) or reloading (reconnection to the Backed-Up Electric Panel) as illustrated by Figure 4-1. The shedding and reloading orders are issued by the *LS*. The *LS* has a unique Boolean signal to shed all the *Steps* (*ls.shedAll*), but it has a specific Boolean signal to reload each individual *step*. Therefore, the reloading signal (*reload*) is declared as specific.

```
  specific Boolean reload;
  Breaker brk
    state.start = closed;
    tMax = 500*ms;              // Assumed by ODS
    openCmd = ls.shedAll;
    closeCmd = reload;
    end Brk;
```

Events *eShed* and *eReload* mark the instants when the *LS* orders the *step* to open or close its breaker. The events occur when the signals become true:

```
  event eShed = when LS.shedAll becomes true;
  event eReload = when reload becomes true;
```

The *LS* property model requires that shedding and reloading signals do not become true at the same instant, but nonetheless, the *Step* class also states a requirement to this end:

```
  requirement noContradiction = no (eShed and eReload);
```

A *step* may be required or not (*step1a* and *step1b* are required when and only when *client1* is active; the other *steps* are always required). The corresponding condition will be specified separately for each instance:

```
  specific Boolean required;
```

For a *step* to be *powered*, several conditions need to be satisfied:

- Its breaker must be closed (*brk.closed*).
- To be powered by the *MPS*, the *MPS* must be available (*MPS.available*) and its own breaker must be closed (*brkMps.closed*).
- To be powered by the *BPS*, the *DG* must be ready (*DG.ready*) and its own breaker must be closed (*brkDg.closed*).

```
  Boolean powered =
    brk.closed and
    ((MPS.available and brkMps.closed) or (DG.ready and brkDg.closed));

  Boolean poweredByBPS = brk.closed and DG.ready and brkDg.closed;
```

The demand of a *step* for *BPS* electric power is a function of time (see Figure 4-6). The effective demand is represented by a Real function named *demand*. It will be provided at a later stage by a behavioural model. For overall design verification purposes, one just puts constraints on this demand:

- When the *step* is not active, the demand is null.

- When it is just reloaded, it generates a spike demand that could last up to, but no more than, a given duration (*settleTime*) and that could rise up to, but no more than, a given value (*initialDemand*).
- After *settleTime*, the demand is at most *stabilisedDemand*.

initialDemand, *settleTime* and *stabilisedDemand* are specific to each *step*:

```
   external Real demand;                          // Demand for power
   assumption a1 = demand ≥ 0.*w;

   specific constant Real initialDemand;
   assumption a2 = initialDemand > 0.*w;

   specific constant Real stabilisedDemand;
   assumption a3 = stabilisedDemand > 0.*w;
   assumption a4 = stabilisedDemand < initialDemand;

   constant Duration maxSettleTime = 4.*s;    // Assumed by ODS
   constant specific Duration settleTime;
   assumption a5 = settleTime > 0.*s;
   assumption a6 = settleTime ≤ maxSettleTime;

   Real demandToBPS
      during poweredByBPS then demand;
      else 0.*w;
      end demandToBPS;

   assumption a7 =
      after poweredByBPS becomes true for maxSettleTime
      check demand ≤ initialDemand;

   assumption a8 =
      during poweredByBPS trunc maxSettleTime
      check demand ≤ stabilisedDemand;

   end Step;
```

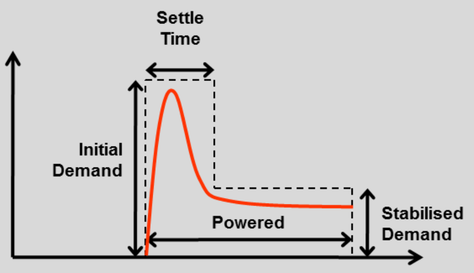


Figure 4-6
Assumptions on the demand for power of a step

Instances

The 9 *steps* are declared as a single 9 element vector. The priority order is given by the order in the vector, the first element having the highest priority.

```
Step step[6]
   step[1, 2].required = client1.active;
   step[3, 4].required = client2.active;
   step[5, 6].required = client3.active;
   step[1..6].initialDemand = 500.*kw;
   step[1..6].settleTime = 4.*s;
   step[1..6].stabilisedDemand = 70.*kw;
   step[1].Reload = ls.reload1;
   step[2].Reload = ls.reload2;
   step[3].Reload = ls.reload3;
   step[4].Reload = ls.reload4;
   step[5].Reload = ls.reload5;
   step[6].Reload = ls.reload6;
   end step;
```

4.9. The *DG* (Diesel Generator) Class

```
class dg  extends ProtectedComponent
```

Start, Stop, Ready, Stall

The *DG* receives two Boolean start / stop signals from the *LS*:

- When *ls.startDg* becomes true, the engine starts. This is represented by event *eStart*.
- When *ls.stopDg* becomes true, the engine stops. This is represented by event *eStop*.

```
   event eStart = when ls.startDG becomes true;
   event eStop = when ls.stopDG becomes true;
```

The *ls* model requires that both signals do not become true at the same instant, but nonetheless, the *dg* model also states a requirement to this end (just in case):

```
   requirement noContradiction = no (eStart and eStop);
```

The *LS* assesses the readiness of the *DG* to provide power to the required *steps*, based on the voltmeters and the frequency meters at the *DG* terminals. This assessment is represented by the *ls.eDgReady* event. This event is internal to the *ls* model: i.e., it is not one of the output signals of the LS. Therefore, it cannot be used to trigger actual, physical action on any *BPS* component: it can be used only for verification. Event *eReady* marks the instant when the *LS* considers the *DG* to be ready:

```
   event eReady = ls.eDgReady;
```

When the DG is overloaded, it can stall. This is represented by an external event that will be signalled by the behavioural model:

```
   external event eStall;
```

DG States

The *DG* can be in one of three states:

- In state *stopped*, it does not function at all. This is the initial state.
- In state *starting*, it is functioning, but the tension and frequency at its terminals are not yet appropriate.
- In state *ready*, it is functioning and fully operational.

The state transitions are triggered by events *eStart*, *eReady*, *eStop* and *eStall*.

```
   automaton state = (stopped, starting, ready)
      start = stopped;
      when stopped and eStart then starting;
      when eReady then ready;
      when eStop or eStall  then stopped;
      else fixed;
      end State;
```

When the *DG* is not ready, its breaker must be open:

```
   requirement brkOpen = during not ready check brkDg.open;
```

Startup Time

When the *DG* is ordered to start, it needs a certain time to get ready and to provide the right tension and frequency at its terminals. Since at the time of overall design no actual product has been chosen yet for the *DG*, the maximum delay *realTMax* is still unknown. The overall design just puts an upper bound *tMax* on this value. The purpose of the following is to declare the value of *tMax* and to ensure that the *realTmax* of the chosen product will be less than *tMax*:

```
constant Duration tMax = 5.*s;          // Value assumed by ODS
external constant Duration realTMax;    // From detailed design
assumption a1 = realTMax > 0.*s;
assumption a2 = realTmax ≤ tMax;
```

The meaning of *tMax* is as follows: after a start order (event *eStart*), the *DG* will be ready (event *eReady*) within a *tMax* delay, provided that in the meantime it does not receive a stop order (event *eStop*):

```
requirement r1 = after eStart within tMax check (eReady or eStop);
```

Electric Load

The *load* of the *DG* is a Real function that is the sum of the demands for power from the 6 *Step*s. When the *DG* is stopped or its breaker is open, its load is null:

```
Real load
  during ((starting or ready) and brkDg.closed)
    then SUM {s.demandToBPS forAll s in step};
  else 0.*w;
  end load;
```

The *DG* has a maximum load *realMaxLoad* beyond which it may stall (event *eStall*) and cease to provide power. Since this value is still unknown, a lower bound *maxLoad* is assumed:

```
constant Power maxLoad = 1200.*kw;             // Value assumed by ODS
external constant Power realMaxLoad;
assumption a3 = realMaxLoad ≥ maxLoad;
assumption a4 = during load ≤ maxLoad check no eStall;
```

The overall design requires that the *DG* shall never be used at more than 80% of the maximum load:

```
requirement r2 = load ≤ 0.80 * maxLoad;
```

DG Mode

As specified by the *mode* automaton inherited from *ProtectedComponent*, the *DG* can run on one of two modes: *protection* or *workToDeath*. It switches between the two based on Boolean signal *dgProtect* sent by the *LS*: when the signal is true, the *DG* runs in the *protection* mode; otherwise, the *DG* runs in the *workToDeath* mode. The following provides a closed definition for the automaton:

```
automaton mode = (workToDeath, protection)
  start = workToDeath;
  when workToDeath and (ls.dgProtect becomes true) then protection;
  when protection and (ls.dgProtect becomes false) then workToDeath;
  else fixed;
  end mode;
```

When the *DG* is not required to operate, its breaker MUST be opened when the Backed-Up Panel is powered by the *MPS*, otherwise the *DG* could be damaged or destroyed.

```
requirement r3 =
  during mpsAvailable and brkMps.closed check brkDg.open;

end dg;
```

4.10. The *Ls* (Load Sequencer) Class

The *LS* is a digital control system in charge of monitoring the *BPS* division sensors, of receiving events issued by the operator, and of sending control signals to the other *BPS* division components (*DG*, circuit breakers).

```
class Ls
```

4.10.1.Time Domain

The *LS* is a synchronous digital control system that has its own discrete time domain named *tdLS*: the *LS* is the only object in that time domain. The time domain is periodic, with a period of 50 milliseconds. Its phase is random (see Figure 4-7).

```
  in timeDomain tdLs
    constant Duration cycleTime = 50.*ms;
    private fixed Duration phase = random (CycleTime);
    clock = (every CycleTime) + phase;
    end tdLs;
```

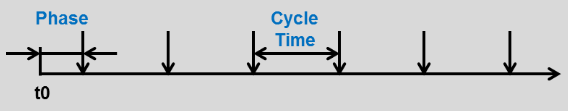


Figure 4-7
The tdLs time domain

4.10.2.Main States

The main *LS* state automaton is shown in Figure 4-8.

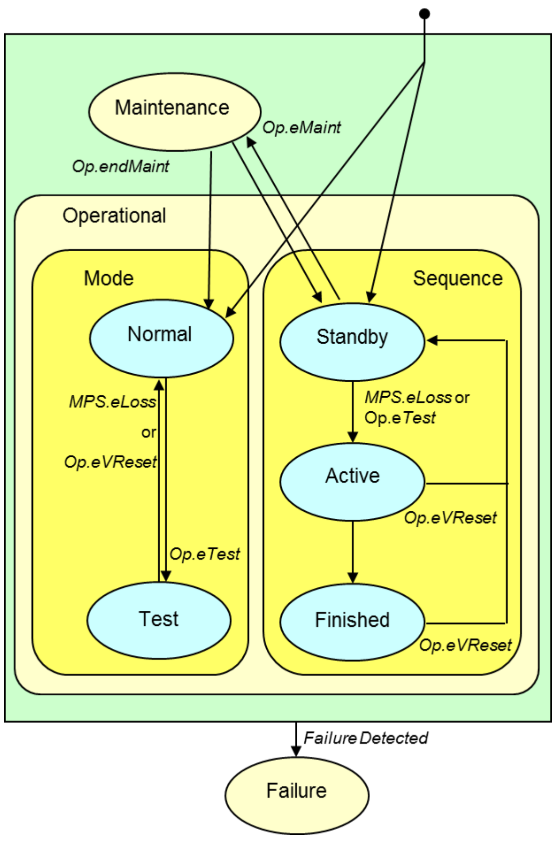


Figure 4-8

The LS main state automata

It is composed of three states:

- In the *operational* state, the *LS* is ready to perform, or is performing, its mission. This state is the initial state and is the Cartesian product of two sub-automata: *sequence* and *mode*:
  - The *sequence* sub-automaton indicates at which sequencing stage the *LS* is. It is composed of three sub-states:
    - In *standby*, no action is necessary yet, but the *LS* is ready to act.
    - In *active*, sequencing is being performed so that the required *steps* are powered by the *DG*.
    - In *finished*, all *steps* are powered by the *DG* and the *LS* waits for a valid reset signal to restore normal power conditions where the *steps* are powered by the *MPS*.
  - The *mode* sub-automaton is composed of two sub-states (*normal* and *test*) indicating in which operating mode (respectively *workToDeath* and *protection*) the *LS* should run the *DG* when it is needed. If the *MPS* becomes unavailable during a test, the *LS* continues sequencing, but the operating mode of the *DG* is switched to *worktoDeath*.
- The *maintenance* state of the *LS* corresponds to the *maintenance* state of the *BPS* division.
- The *failure* state is taken when the *LS* detects a failure preventing the *BPS* division from performing its mission.

This is modelled as follows:

```
automaton sequence = (standby, active, finished);
automaton mode = (normal, test);
automaton state = (operational = sequence & mode, maintenance, failure)
  start = operational(standby, normal);
  // State transitions will be specified in a scattered manner
  // throughout the Ls class
  end state;
```

4.10.3.Inputs

All the inputs and outputs of the *LS* are Boolean signals. As seen in previous sections, it is the role of the sensors to convert analog tension or frequency signals into Boolean signals.

One needs to remember that the *LS* is in its own discrete time domain and that all the inputs are issued from the continuous time domain. Therefore, the continuous input signals are "sampled" and transformed by the *LS* prior to any utilisation, based on the following pattern:

```
//Boolean lsName = clock and inputName;
```

In principle, it should not be necessary to explicitly state these interfaces, since a FORM-L compiler would be able to do it automatically. However, it is done so here, just in case. The input signals and their *LS* translations are:

Input Name	***LS* Name**	**Description**
bV1.on *bV2.on*	*bV1* *bV2*	Redundant on / off signals for the tension at the terminals of the Backed-Up Panel
nbV1.on *nbV2.on*	*nbV1* *nbV2*	Redundant on / off signals for the tension at the terminals of the Non-Backed-Up Panel
dgV1.on *dgV2.on*	*dgV1* *dgV2*	Redundant on / off signals for the tension at the terminals of the *DG*
dgF1.on, dgF2.on	*dgF1* *dgF2*	Redundant on / off signals for the frequency at the terminals of the *DG*

brk1MpsPos.open *brk2MpsPos.open*	*brkMps1Open* *brkMps2Open*	Redundant open / closed signals for the position of the *MPS* circuit breaker
brk1DgPos.open *brk2DgPos.open*	*brkDg1Open* *brkDg2Open*	Redundant open / closed signals for the position of the *DG* circuit breaker
brk1Pos.open	*brk1Open*	open / closed signal position for the specific breaker of *step[1]*
brk2Pos.open	*brk2Open*	open / closed signal position for the specific circuit breaker of *step[2]*
maintBtn	*maint*	Operator request for maintenance
endMaintBtn	*endMaint*	Operator signal informing the *LS* of the end of maintenance
testBtn	*pTest*	Operator request for a periodic test
resetBtn	*reset*	Operator request to reset the *BPS*

```
  Boolean bV1          = clock and bV1.on;
  Boolean bV2          = clock and bV2.on;
  Boolean nbV1         = clock and nbV1.on;
  Boolean nbV2         = clock and nbV2.on;
  Boolean dgV1         = clock and dgV1.on;
  Boolean dgV2         = clock and dgV2.on;
  Boolean dgF1         = clock and dgF1.on;
  Boolean dgF2         = clock and dgF2.on;
  Boolean brk1MpsOpen  = clock and brk1MpsPos.open;
  Boolean brk2MpsOpen  = clock and brk2MpsPos.open;
  Boolean brk1DgOpen   = clock and brk1DgPos.open;
  Boolean brk2DgOpen   = clock and brk2DgPos.open;
  Boolean brk1Open     = clock and brk1Pos.open;
  Boolean brk2Open     = clock and brk2Pos.open;
  Boolean maint        = clock and maintBtn;
  Boolean endMaint     = clock and endMaintBtn;
  Boolean pTest        = clock and testBtn;
  Boolean reset        = clock and resetBtn;
```

4.10.4.Outputs

The outputs of the *LS* are modelled as Boolean signals to reflect the effective interfaces of the actual system. Initially, all are set to **false**. They are:

dgProtect	Sets the *DG* mode of operation (true: *protection*; false: *workToDeath*)
startDG	Starts the *DG* when it becomes true
stopDG	Stops the *DG* when it becomes true
openBrkMps	Opens the *MPS* breaker when it becomes true
closeBrkMps	Closes the *MPS* breaker when it becomes true
openBrkDg	Opens the *DG* breaker when it becomes true
closeBrkDg	Closes the *DG* breaker when it becomes true
shedAll	Sheds all the *steps* when it becomes true
reload1	Reloads *step1* when it becomes true

reload2	Reloads *step2* when it becomes true
reload3	Reloads *step3* when it becomes true
reload4	Reloads *step4* when it becomes true
reload5	Reloads *step5* when it becomes true
reload6	Reloads *step6* when it becomes true
failure	When true, informs the human operators that the main *BPS* mission has failed

The first output signal is solely under the control of the *mode* automaton. It is completely defined (i.e., in a closed manner) as follows:

```
Boolean dgProtect
   start = false;
   when state becomes operational(current, normal) then false;
   when state becomes operational(current, test) then true;
   else fixed;
   end dgProtect;
```

Note that no delay is specified here, since the *mode* state transitions are already delayed by one clock tick of the *LS* discrete time domain.

The other outputs are under the control of the *sequence* automaton. They all start as false, and reset to false when the automaton enters the *Standby* state (note again that no delay is specified here, since the *sequence* state transitions are also already delayed by 1 clock tick of the *LS* discrete time domain). All except the reload signals are defined in a closed manner and are associated with an event: they are set to true one clock tick after their associated event is raised. The reload signals have an open definition: in particular, they will be directly set to true in the course of the reloading sequence (see Section 4.10.6 *State Transitions and Actions*) without the need for an associated event.

```
event eStartDg;

Boolean startDg
   start = false;
   when eStartDg + clock(1) then true;
   when state becomes operational(standby, current) then false;
   when state becomes maintenance then false;
   else fixed;
   end startDg;

event eStopDg;
Boolean stopDg
   start = false;
   when eStopDg + clock(1) then true;
   when state becomes operational(standby, current) then false;
   when state becomes maintenance then false;
   else fixed;
   end StopDg;

event eOpenBrkMps;
Boolean openBrkMps
   start = false;
   when eOpenBrkMps + clock(1) then true;
   when state becomes operational(standby, current) then false;
   when state becomes maintenance then false;
   else fixed;
   end openBrkMps;

event eCloseBrkMps;
```

```
  Boolean closeBrkMps
     start = false;
     when eCloseBrkMps + clock(1) then true;
     when state becomes operational(standby, current) then false;
     when state becomes maintenance then false;
     else fixed;
     end closeBrkMps;

  event eOpenBrkDg;
  Boolean openBrkDg
     start = false;
     when eOpenBrkDg + clock(1) then true;
     when state becomes operational(standby, current) then false;
     when state becomes maintenance then false;
     else fixed;
     end openBrkDg;

  event eCloseBrkDg;
  Boolean closeBrkDg
     start = false;
     when eCloseBrkDg + clock(1) then true;
     when state becomes operational(standby, current) then false;
     when state becomes maintenance then false;
     else fixed;
     end closeBrkDg;

  event eShedAll;
  Boolean shedAll
     start = false;
     when eShedAll + clock(1) then true;
     when state becomes operational(standby, current) then false;
     when state becomes maintenance then false;
     else fixed;
     end shedAll;

  Boolean reload1
     start = false;
     when state becomes operational(standby, current) then false;
     when state becomes maintenance then false;
     end reload1;

  Boolean reload2
     start = false;
     when state becomes operational(standby, current) then false;
     when state becomes maintenance then false;
     end reload2;

  Boolean reload3
     start = false;
     when state becomes operational(standby, current) then false;
     when state becomes maintenance then false;
     end reload3;

  Boolean reload4
     start = false;
     when state becomes operational(standby, current) then false;
     when state becomes maintenance then false;
     end reload4;

  Boolean reload5
     start = false;
     when state becomes operational(standby, current) then false;
```

```
    when state becomes maintenance then false;
    end reload5;

  Boolean reload6
    start = false;
    when state becomes operational(standby, current) then false;
    when state becomes maintenance then false;
    end reload6;

  event eFailure;
  Boolean failure
    start = false;
    when eFailure + clock(1) then true;
    when state becomes operational(standby, current) then false;
    when state becomes maintenance then false;
    else fixed;
    end failure;

  Boolean {} sequencingOutputs = {
    startDg, stopDg,
    openBrkMps, closeBrkMps,
    openBrkDg, closeBrkDg,
    shedAll,
    reload1, reload2, reload3, reload4, reload5, reload6,
    failure};
```

4.10.5.Main Events and Conditions

The 18 inputs signals of the *LS* are not all independent from one another. For the *LS*, they can be compounded and abstracted into 9 main events and conditions that will be used later on to complete the requirements specification:

Name	**Type**	**Description**
eMaint	event	Start a maintenance session
eEndMaint	event	End an on-going maintenance session
eTest	event	Start a periodic test
client1Active	Boolean	*Client1* is active
eBrkMpsOpen	event	*MPS* breaker is open
eBrkDgClosed	event	*DG* breaker is closed
eMpsLoss	event	Loss of *MPS*
eDgReady	event	*DG* is ready
eVReset	event	Valid reset to restore normal power conditions

Some of these events and conditions have been defined in the other property models.

Also, some features declared external by other property models are defined by the *LS*.

Start a Maintenance Session

End an On-going Maintenance Session

Start a Periodic Test

The *start maintenance*, *end maintenance* and *test* signals issued by the operator:

```
  event eMaint = when maint becomes true;
  event eEndMaint = when endMaint becomes true;
  event eTest = when pTest becomes true;
```

Client1 is Active

The information is provided by the sensors monitoring the positions of the specific breakers of the two *client1* divisions (*brk1* and *brk2*): client1 is considered to be active when either of the specific breakers is closed.

```
  condition client1Active = when (not brk1Open) or (not brk2Open);
```

MPS Breaker is Open

The *LS* determines the position of the *MPS* breaker based on the inputs from the two redundant position sensors:

```
  Boolean brkMpsOpen = brk1MpsOpen or brk2MpsOpen
  Boolean brkMpsClosed = not brkMpsOpen;
  event eBrkMpsOpen   = when brkMpsOpen becomes true;
  event eBrkMpsClosed = when brkMpsOpen becomes false;
```

DG Breaker is Closed

The *LS* also determines the position of the *DG* breaker based on the inputs from two redundant position sensors, but with a different logic:

```
  Boolean brkDgOpen = brk1DgOpen or brk2DgOpen
  Boolean brkDgClosed = not brkDgOpen;
  event eBrkDgOpen   = when brkDgOpen becomes true;
  event eBrkDgClosed =  when brkDgOpen becomes false;
```

MPS On / Off

The *LS* can make a decision regarding the *on* / *off* state and the *Availability* of the *MPS* based on the inputs from the four voltmeters at the Backed-Up and Non-Backed-Up Panels terminals.

A significant part of the *on* / *off* decision is made by the voltmeters. (Indeed, had there been only one voltmeter, the complete decision would have been made by that voltmeter.) One role of the *LS* on that issue is to decide when to rely on the voltmeters of the Backed-Up Panel (*bV1* and *bV2*), and when to rely on those of the Non-Backed-Up Panel (*nbV1* and *nbV2*). Normally, it relies on the first pair. However, if the *MPS* breaker is open (during a test), these voltmeters are disconnected from the *MPS* and the *LS* will then need to rely on the second pair.

Another role of the *LS* is to manage redundancy. When it uses the first pair (which is most of the time), it is preferable not to fail to activate the *BPS* when it is needed. Thus, any one of the pair must be sufficient to declare the tension *off*. When it uses the second pair (i.e., during a test), the consequences of a single voltmeter failure are much less important (the *DG* will run in *Protection* mode instead of *WorkToDeath*). The overall *BPS* design minimises the risk of damaging the *DG* and requires that both voltmeters agree before declaring the tension *off*.

```
  Boolean mpsOn =  (brkMpsClosed and bV1 and bV2) or
                   (brkMpsOpen and (nbV1 or nbV1));
  Boolean mpsOff = not mpsOn;
```

MPS Availability - *MPS* Loss

The available / unavailable state of the *MPS* is decided by the *LS* based on the following criteria:

- The *LS* declares the *MPS* to be *unavailable* when it sees it *off* during 2.5 consecutive seconds.
- It declares the *MPS* to be back again and *available* when it sees it *on* during 9 consecutive seconds.

```
   private property offEnough = duringAny 2.5*s check mpsOff;
   private property onEnough  = duringAny  9.*s check mpsOn;
   Boolean mpsAvailable
      start = true;
      when (mpsAvailable and offEnough.satisfiedNow) becomes true
         then false;
      when (not mpsAvailable and onEnough.satisfiedNow) becomes true
         then true;
      else fixed;
      end mpsAvailable;
   Boolean mpsUnavailabble = not mpsAvailable;
```

For its own purposes, the *LS* marks the loss of the *MPS* by the following event:

```
   event eMpsLoss = when mpsAvailable becomes false;
```

DG Ready

When the *DG* becomes necessary, it will be started by the *LS*. The *DG* needs some time to be fully ready. It is the *LS* that makes the decision that the *DG* is ready, based on the tension and frequency at the *DG* terminals: both tension and frequency must be appropriate.

```
   event eDgReady = when ((dgV1 or dgV2) and (dgF1 or dgF2)) becomes true;
```

Valid Reset

The reset signal needs a little more processing than the other operator signals, since in the following, only valid reset signals will be taken into consideration. Deciding whether a reset signal is valid is based on whether the *MPS* is available and also whether the *BPS* is active:

```
   event eReset = when resetBtn becomes true;
   event eVReset =
      (during mpsAvailable)
      and (after state becomes operational(active, current) trunc 10*mn)
      and eReset;
```

One can note that the last expression is close to the one given in Section 4. There are a few differences though:

- The beginning of the *BPS* division activity is marked by a transition to *operational*(*active*; *current*).
- The *eReset* event here is the one in the tdLS time domain.

4.10.6.State Transitions and Actions

In this Section, the states specified in Section 4.10.2 *Main States* are analysed in terms of reaction to the 9 events and conditions of Section 4.10.5 *Main Events and Conditions*. The analysis of each state follows the same pattern:

- It begins with the actions that must be performed unconditionally upon entry to the state.
- Next are the requirements that must be satisfied while in the state.
- Next is the list of events and conditions that are taken into consideration in that state for the subsequent actions. The others are ignored. The actions include transition to other states.

Standby State

Upon entry to this state, all outputs except *dgProtect* are already set to false (see Section 4.10.4 *Outputs*). The following requirement states that they must be maintained at that value all the time the *LS* is in that state.

```
   requirement ls1 = during operational(standby, current)
      check AND {not s forAll s in sequencingOutputs};
```

In this state, the *LS* reacts to:

- *eMpsLoss*
- *eTest*
- *eMaint*

Upon receiving a maintenance request (eMaint), the *LS* enters the *Maintenance* state. The +clock(1) is a grace period that gives time for the *LS* to react to its inputs:

```
   when (operational(standby; current) and eMaint) + clock(1)
      then state = maintenance;
```

Upon receiving an *MPS* loss event (*eMpsLoss*), the *LS* enters the *operational(active; normal)* state, unless it receives at the same time a maintenance request. The maintenance request has a higher priority than the *MPS* loss event.

```
   when
      (operational(standby; current) and
      eMpsLoss and not eMaint) + clock(1)
      then state = operational(active, normal);
```

Upon receiving a test request (*eTest*), the *LS* enters the *operational(active; test)* state, unless it receives at the same time a maintenance request or an *MPS* loss event:

```
   when
      (operational(standby; current) and
      eTest and not eMpsLoss and not eMaint) + clock(1)
      then state = operational(active, test);
```

Active State

Upon entry to this state, the *LS* must also issue a sequence of orders. The beginning of that sequence is shown in Figure 4-9. *shedAllTmax* is the maximum time for the *step* breakers to open, plus a half second margin.

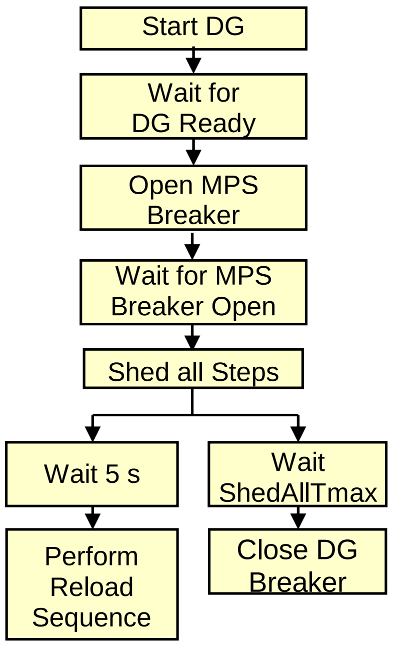


Figure 4-9
Beginning of the LS sequencing

```
  event eReloadSequence;
  Duration shedAllTMax = 500.*ms + LARGEST{ s.brk.tMax forAll s in step};
  when state becomes operational(active, current) then sequence
    raise eStartDG;
    wait eDgReady then raise eOpenBrkMps;
    wait eBrkMpsOpen then raise eShedAll;
    simultaneous
      wait (shedAllTmax + 0.5*s) then raise eCloseBrkDg;
      wait 5.*s then raise eReloadSequence;
      end;
    end;
```

The *steps* reloading sequence must comply with the following constraints:

- Only the required *steps* are to be reloaded. The fact that a *step* is required or not is modelled by its *required* attribute (see Section 4.8 *The Step* Class).
- Required *steps* must be reloaded one by one, following their priority order. That order is determined by the order in the *step* vector, the first element having the highest priority.
- There must be at least *maxSettleTime* between two consecutive reloadings.
- Each required *step* must be reloaded as soon as it is possible.
- The first required *step* must be reloaded immediately at the beginning of the reloading sequence.

The following event marks all the instants where a reloading occurred:

```
  event eReloads = OR {s.eReload forAll s in step};
```

The following condition covers the time periods where no reloading should occur:

```
  Boolean noReload =
    (until eReloadSequence) or
    (after eReloads for maxSettleTime) or
    (after eVReset);
  requirement during noReload check no eReloads;
```

The following subset contains the indexes of all the required *steps* not reloaded yet:

```
  Integer{} candidates =
    {i forAll i in 1..6
      suchThat step[i].required and not step[i].brk.closed};
```

Finally:

```
  when ((not noReload) and (cardinal candidates)>0) becomes true + clock(1)
    then step[SMALLEST(candidates)].reload = true;
```

One clock tick after all the *steps* are reloaded, the *LS* enters the *finished* state:

```
  when ((cardinal candidates) becomes 0) + clock(1)
    then state = operational(finished, current);
```

In the *active* state, the *LS* also reacts to the *eVReset* event: when it occurs, it executes the reset sequence (see next Section and Figure 4-10).

Finished State - Resetting the *ODS* Model

In this state, the *LS* just waits for a valid reset event (*eVReset*). It then performs the reset sequence (see Figure 4-10).

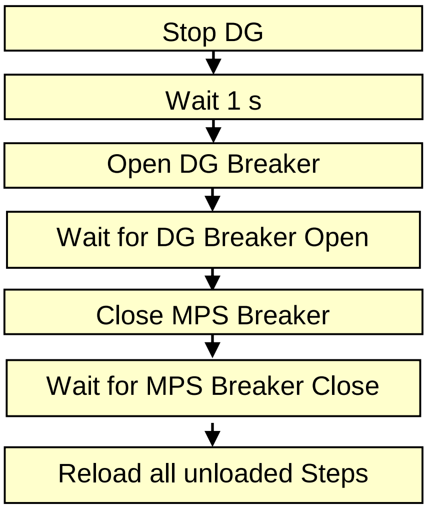


Figure 4-10
Reset Sequence

```
  when
     (operational(active; current) or operational(finished, current))
       and eVReset then
    sequence
       raise eStopDG;
       wait 1.*s then raise eOpenBrkDg;
       wait eBrkDgOpen then raise eCloseBrkMps;
       wait eBrkMpsClosed + clock(1) then
         forAll s in step then s.reload = true;
       end;
```

Test State

If in *test* mode, the *LS* returns to *normal* mode in case of loss of *MPS*:

```
  when (operational(current, test) and eMpsLoss) + clock(1)
    then state = operational(current, normal);
```

Maintenance State

In the *maintenance* state, the *LS* returns to the *operational*(*standby*, *normal*) state upon event *eEndMaint*:

```
  when (maintenance and eEndMaint) + clock(1)
    then state = operational(standby, normal);
```

Failure State

This state is taken when the *LS* detects failures of other *BPS* components that prevent the *BPS* from performing its main mission, i.e., the timely reloading of the required *steps*. In case of a failure of the *LS* itself, the design includes a watchdog external to the LS that monitors its well-being, that raises the failure signal when necessary. This mechanism is not modelled here.

The *LS* can only detect failures of the *MPS* and *DG* breakers, and of the *DG* itself, i.e., when they fail to open, close or be ready in a timely manner, or when they open, close or stop spuriously.

Failure to open, or close, or be ready, in a timely manner:

```
  when ((eOpenBrkMps + brkMps.tMax) and brkMpsClosed) + clock(1)
    then state = failure;
  when ((eCloseBrkMps + brkMps.tMax) and brkMpsOpen) + clock(1)
    then state = failure;
  when ((eOpenBrkDg + brkDg.tMax) and brkDgClosed) + clock(1)
    then state = failure;
  when ((eCloseBrkDg + brkDg.tMax) and brkDgOpen) + clock(1)
    then state = failure;
  when ((eStartDG + dg.tMax) and not dg.ready) + clock(1)
    then state = failure;
```

Spurious opening, closing or stopping:

```
  private property noOpenBrkMpsOrder =
    duringAny brkMps.tMax check no eOpenBrkMps;
  when eBrkMpsOpen and noOpenBrkMpsOrder.satisfiedNow + clock(1)
    then state = failure;

  private property noCloseBrkMpsOrder =
    duringAny brkMps.tMax check no eCloseBrkMps;
  when eBrkMpsClose and noCloseBrkMpsOrder.satisfiedNow + clock(1)
    then state = failure;

  private property noOpenBrkDgOrder =
    duringAny brkDg.tMax check no eOpenBrkDg;
  when eBrkDgOpen and noOpenBrkDgOrder.satisfiedNow + clock(1)
    then state = failure;

  private property noCloseBrkDgOrder =
    duringAny brkDg.tMax check no eCloseBrkDg;
  when eBrkDgClose and noCloseBrkDgOrder.satisfiedNow + clock(1)
    then state = failure;

  private property noStopDgOrder =
    duringAny dg.tMax check no eStopDg;
  when dg.ready becomes false and noSopDgOrder.satisfiedNow + clock(1)
    then state = failure;
```

When entering the *failure* state, event *efailure* is raised to set the *failure* output signal:

```
  when state becomes failure then raise eFailure;
```

```
  end Ls;
```

4.11. The *BpsDivOds* Class

This class models the overall design of a *BPS* division.

```
class BpsDivOds
```

Load Sequencer

Each division has one load sequencer:

```
  Ls ls;
```

Diesel Generator

Each division also has one diesel generator:

```
  Dg dg;
```

Breakers

The following breaker is associated with the *MPS*. The instantiation specifies the values of the features declared as *specific*:

```
Breaker brkMps
   state.start = closed;
   tMax = 800*ms;                 // Assumed by ODS
   openCmd = ls.openBrkMps;
   closeCmd = ls.closeBrkMps;
   end brkMps;
```

The following breaker is associated with the *DG*:

```
Breaker brkDg
   state.start = open;
   tMax = 800*ms;                 // Assumed by ODS
   openCmd = ls.openBrkDg;
   closeCmd = ls.closeBrkDg;
   end brkDg;
```

The breakers associated with the client divisions are declared as part of the divisions.

Voltmeters

```
Voltmeter bV1;     // For the Backed-up panel
Voltmeter bV2;
Voltmeter nbV1;    // For the Non-Backed-up panel
Voltmeter nbV2;
Voltmeter dgV1;    // For the Diesel Generator
Voltmeter dgV2;
```

FreqMeters

```
FreqMeter dgF1;       // Redundant frequency meters for the DG
FreqMeter dgF2;
```

Breaker Position Sensors

```
MPensor brk1Pos
   brk = brk1;
   tMax = 20.*ms;
   end brk1Pos;

MPensor brk2Pos
   brk = brk2;
   tMax = 20.*ms;
   end brk2Pos;

MPensor mps1Pos
   brk = brkMps;
   tMax = 20.*ms;
   end mps1Pos;

MPensor mps2Pos
   brk = brkMps;
   tMax = 20.*ms;
   end mps2Pos;

MPensor dg1Pos
   brk = brkDg;
   tMax = 20.*ms;
   end dg1Pos;
```

```
  MPensor dg2Pos
    brk = brkDg;
    tMax = 20.*ms;
    end dg2Pos;
```

Obligation from the Ods Contract

```
  Boolean active =
    ls.operational(active, current) or ls.operational(finished, current);
  Boolean mpsOn = ls.mpsOn;
  Boolean mpsAvailable = ls.mpsAvailable;
  Boolean powering1a = dg.ready and brkDg.closed and client1a.brk.closed;
  Boolean powering1b = dg.ready and brkDg.closed and client1b.brk.closed;
  Boolean powering2a = dg.ready and brkDg.closed and client2a.brk.closed;
  Boolean powering2b = dg.ready and brkDg.closed and client2b.brk.closed;
  Boolean powering3a = dg.ready and brkDg.closed and client3a.brk.closed;
  Boolean powering3b = dg.ready and brkDg.closed and client3b.brk.closed;
  event eFailure = ls.eFailure;

  Component {} sensors = {
    bV1, bV2,
    nbV1, nbV2,
    dgV1, dgV2,
    dgF1, dgF2,
    brk1Pos, brk2Pos,
    mps1Pos, mps2Pos;
    dg1Pos, dg2Pos};

  Component {} nonSensors = {brkMps, brkDg, {s.brk forAll s in step}, dg};

  ProtectedComponent {} protectedComponents = {dg};
```

```
  end BpsDivOds;
```

4.12. *The Ods Standard Contract*

The *Ods* standard contract models the mutual obligations between the model of a *BPS* division stating its system specification (an instance of class *BpsDivSpec*) and the corresponding overall design model (an instance of class *BpsDivOds*).

```
contract Ods (BpsDivSpec divSpec, BpsDivOds divOds)

  party divSpec
    Boolean maintBtn;
    Boolean endMaintBtn;
    Boolean testBtn;
    Boolean resetBtn;
    Boolean client1Active;
    Voltage voltage;
    end divSpec;

  party divOds
    Boolean active;
    Boolean mpsOn;
    Boolean mpsAvailable;
    Boolean powering1a;
    Boolean powering1b;
    Boolean powering2a;
    Boolean powering2b;
    Boolean powering3a;
    Boolean powering3b;
    event eFailure;
    Component {} sensors;
    Component {} nonSensors;
    ProtectedComponent {} protectedComponents;
    end divOds;

  end Ods;
```

4.13. The Models for the Overall Design of the Two *BPS* Divisions

```
property model BpsOds

  BpsDivOds bpsAOds
    contract odsA;
    end bpsAOds;

  BpsDivOds bpsBOds
    contract odsB;
    end bpsBOds;

  end BpsOds;
```

5. To be Provided by Scenarios or Detailed Design

A number of external features need to be provided either by scenario models or by detailed design models:

```
- event Dg.eStall                        // For diesel generator
- constant Duration Dg.realTMax          // For diesel generator
- constant Real Dg.realMaxLoad           // For diesel generator
- constant Duration Breaker.realTMax     // For each breaker
- event Breaker.eOpeningCompleted        // For each breaker
- event Breaker.eClosingCompleted        // For each breaker
- Real MSensor.input                     // For each measurement sensor
- Boolean MSensor.on                     // For each measurement sensor
- constant Duration MSensor.realTMax     // For each measurement sensor
- Boolean PSensor.open                   // For each position sensor
- constant Duration PSensor.realTMax     // For each position sensor
- Real Step.demand                       // For each Step
```

6. Conclusion

This case study is representative of reactive systems that require significant control functions (generally provided by digital systems). As a perspective (after the end of MODRIO), it is planned to continue the work at EDF on this case study. The property models presented here will be complemented with a (Modelica) behavioural model representing the detailed design and the physical behaviour of the electrical and electromechanical components. Component failure modes will also be modelled using the multi-modelling concepts and languages proposed by WP4.

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

[14] 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.

[15] Jardin A., Thuy Nguyen, “MODRIO D8.1.3 Part II: The Intermediate Cooling System (SRI) Case Study”.

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

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