FORM-L: A MODELICA Extension for Properties Modelling Illustrated on a Practical Example

Thuy Nguyen
EDF R&D
6, quai Watier F-78401 Chatou Cedex, France

Abstract

As systems engineering methodologies for complex systems make increasing use of modelling and simulation techniques, it has become important to extend the MODELICA language to also cover requirements, and more generally, properties modelling. The ITEA2 MODRIO project is currently developing an extension for that very purpose: the FORM-L language (FOrmal Requirements Modelling Language). This paper presents an overview of the FORM-L concepts, and illustrates them with examples based on a practical case study, the Backup Power Supply (BPS) system.

Keywords: physical modelling; requirement modelling; systems engineering; methodology

1 Introduction

Systems engineering methodologies for complex systems increasingly rely on, or could benefit from, modelling and simulation. For MODELICA to support activities such as functional validation of system requirements, design verification against requirements, testing, dysfunctional analyses and verification of operational procedures, the ITEA2 MODRIO project is developing extensions to the language. One of them concerns formal requirements and properties modelling, and is called FORM-L (FOrmal Requirements Modelling Language). This paper presents the main concepts underlying FORM-L, and illustrates them with examples taken from a MODRIO case study, the Backup Poser Supply (BPS) system.

Section 2 presents the main objectives assigned to FORM-L. Section 3 introduces briefly the BPS case study in oder to provide a background context for the examples given in the floowing sections. Section 4 presents how FORM-L considers *functions*, *constants* and *fixed* variables. Section 5 introduces the notions of *condition* and *event*. Section 6 presents the notions of *properties*, *requirements*, *assumptions* and *guards*. Section 7 presents the notion of *time locator*, continuous or discrete. Section 8 presents how FORM-L views *sets* and *arrays*. Lastly, Section 9 presents how *actions* are modelled in FORM-L.

2 FORM-L Overview

2.1 Motivation

Paper *Innovative Modelling Architecture for the Verification of Design against System requirements* presents how one of the methodologies developed by MODRIO (to verify the design of a system against its requirements) is supported by FORM-L.

This includes in particular a clear separation of models serving different purposes in the systems engineering lifecycle or the support to systems operation. In particular, there should be a well-identified model that clearly and formally specifies:

- The boundaries of the system under study.
- The interactions of the system with its environment (including human operators), including any assumptions made regarding this environment.
- The system requirements, including functional and timing requirements (concerning the interactions with the environment) and system operational requirements (including quality of service and fault-tolerance, and operational constraints aiming for example at reducing wear and tear of system components).

2.2 Main Notions

This is supported by the FORM-L with the notions of *property*, *requirement*, *assumption*, and *external* information (to be supplied either by other MODELICA models or by engineering databases).

The expression of a requirement or a desirable property needs to address four basic issues: WHAT, WHERE, WHEN (cf. EuroSysLib WP7.1 Property Modelling) and HOW WELL.

WHAT states what needs to be achieved or what must be avoided. This is expressed in FORM-L with the notions of *condition* (that must be satisfied), *event* (that must or must not occur), *function* and *action*.

WHERE states where in the system the WHAT needs to be achieved. This can be expressed in FORM-L by the explicit naming of objects, but also with the notion of *set*, in particular of set resulting from queries. Indeed, at the time system requirements are specified, early in the system lifecycle, the names, number, types, characteristics and locations of the objects concerned are often not known yet. They are determined at a later stage, the information being usually stored in one or more engineering databases.

WHEN states when the WHAT needs to be achieved. Possibly complex temporal logic is often needed when considering reactive systems such as the BPS. Such logic can be expressed in FORM-L using *continuous* or *discrete time locators*. FORM-L also includes the notions of *finite state automaton*, *statechart* and *time domain* (the latter being extremely useful, or even necessary, when considering hybrid systems).

HOW WELL states how well the WHAT needs to be achieved (as real life systems are bound to have failures). This is expressed in FORM-L using probabilistic properties. Extensive work is being done in the framework of MODRIO on stochastic issues and multi-mode modelling: FORM-L only addresses what concerns properties.

2.3 Readability

It is not sure that all aspects of requirements specification (in particular complex temporal logic) can be represented graphically without risks of misinterpretation from the part of readers, or even authors of models. Therefore, the clarity of the FORM-L language textual syntax is important, as the language is mainly intended to be used by application specialists rather than modelling experts. The syntax that is proposed here voluntarily includes significant amounts of "syntactic sugar" for this very reason.

3 A Brief Introduction to the BPS Example

3.1 BPS Objective

The objective of the BPS (Backup Power Supply) is to provide electric power to electrical components (e.g., pumps and valves in a thermohydraulic industrial installation) that are considered essential, in case of loss of the Main Power Supply (MPS). Such components could be important to safety (e.g., in an industrial installation or in a hospital) or could be required to prevent unacceptable economic losses (in a semiconductor fab).

Figure 1 presents the overall organisation of the BPS from the standpoint of the requirements specifiers. Figure 2 presents a system architecture developed by designers as a possible answer tot he requirements. The objective of the system requirements model and of the architecture model is to support the verification that the architecture indeed meets the requirements.

3.2 BPS Principles

As the levels of power required by the backed-up sets of components are assumed to be very high, and the duration of the backing-up to last up to several days, the BPS is based on a Back-Up Generator, or BUG. This generator has a number of constraints. In particular, it cannot power instantaneously all necessary components: if all 'client' components were connected simultaneously to the BUG, the BUG could be overloaded and stall, due to the fact that when electric power is restored to a given component, there is an important, transient call for current and power (see figure 3). Thus, much like a conventional car engine needs a gearbox, the BPS and its BUG need an active control system to ensure a progressive and orderly increase of requested power (hence the presence of circuit breakers).

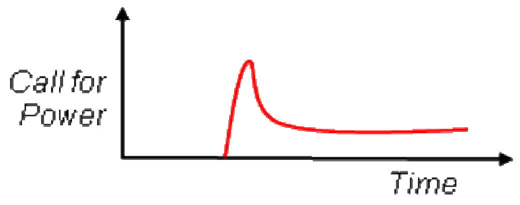

Figure 3: Transient call for current when electric power is supplied to an electric component

Also, not all components powered by the backed-up electric panel have the same needs and functional roles. In the case study, six sets of backed-up electric components (SBC) have been identified:

- *SBC1* groups components that implement Context Specific Actions (*CSA*). Such actions are not always needed. However, if and when they are, then Set1 must be powered back within 20 seconds.
- *SBC2*, *SBC3*, *SBC4* and *SBC5* are redundant sets of components, and it is sufficient to power any two of them within 40 seconds.

- *SBC6* must be powered back within 60 seconds.

When in operation, the BPS can be in one of three main states:

- *Nominal* state: the BPS is available and ready to perform its missions.
- *Test* state: as *MPS* loss is rare and the *BPS* is seldom required, periodic testing is necessary to ensure that when needed, the *BPS* will indeed be able to perform its missions.
- *Maintenance* state: the *BPS* is under repair and is not able to perform its main mission, which is to provide electric power to the backed-up electric panel.

.

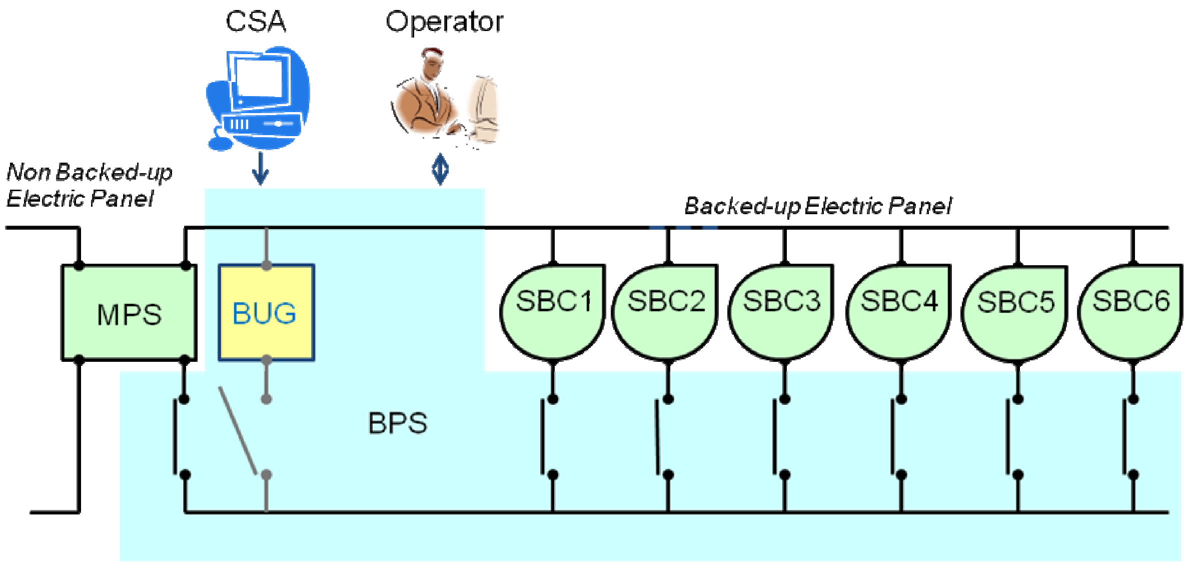

Figure 1: BPS and its environment as viewed by system requirements specifiers. The overall configuration corresponds to the case where the MPS is available

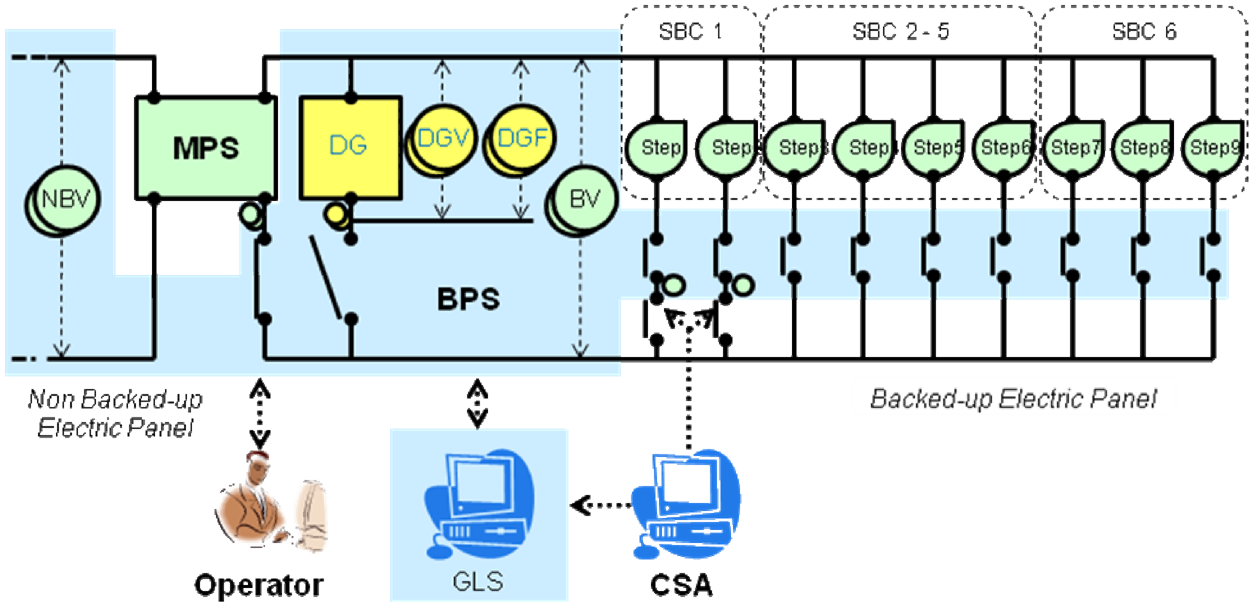

Figure 2: BPS and its environment as viewed by the system designers. Note the introduction of sensors (voltmeters, frequency meters, breaker position sensors), of the digital control system GLS (Generator Load Sequencer), of the splitting of certain SBCs into smaller Steps.

3.3 The BPS Models

The BPS case study considers 4 models (see figure 4):

- *BPS.REQ* is a property model specifying the functional and quality of service requirements applicable to the *BPS*. This property model also describes the environment of the BPS, and the interactions between the BPS and that environment.
- *BPS.ADS* is a property model describing an architectural design for the *BPS*. In particular, it identifies the components constituting the *BPS*, and puts requirements on these. One objective is to verify that this design will indeed satisfy the requirements stated by *BPS.REQ*.
- *BPS.ENV* is a behavioural model that simulates the system environment of the *BPS*, including operators' actions.
- *BPS.BEV* is a behavioural model that simulates the design specified by *BPS.ADS*.

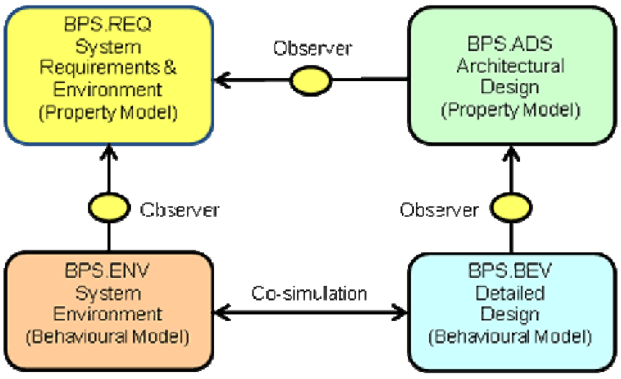

Figure 4: The MODELICA models for the BPS

4 Functions, Constants and Fixed Values

4.1 Functions

Functions are features the value of which depend on time. In addition to functions already supported by current MODELICA, FORM-L has a few additional types:

- *Conditions* are combinations of Boolean and temporal logic. Their values can be *true*, *false* or *undefined*.
- *Finite state automata* are functions the values of which are in an enumerated set (see section 10 Discrete States - Finite State Automata).
- Probabilities are functions the values of which are real in the [0., 1.] range (see section 13 Probabilistic Properties).

4.2 Constants

Qualifier **`constant`** may be used to specify that a feature does not vary in time and has the same value for all simulation runs. This is not absolutely necessary (the FORM-L compiler should be able to detect that automatically) but may clarify authors' intentions.

```
constant real pi = 3.1416;
```

4.3 Fixed Values

Qualifier **`fixed`** may be used to specify that a feature has a value that is determined at the beginning of a run and does not vary during that run. However, it may be different in different runs. Here again, this is mainly to clarify authors' intentions.

```
constant duration CycleTime = ms50;
fixed duration Phase = random (0.0, Cy-
cleTime);
```

The GLS (Generator Load Sequencer, see Figure 2) is a digital, synchronous control system inroduced by the architectural design. It operates in a discrete time domain, the *CycleTime* of which is 50 milliseconds (`ms50`). The *Phase* of the time domain is a random value that does not change once the system has started, but that will be different when the system is restarted.

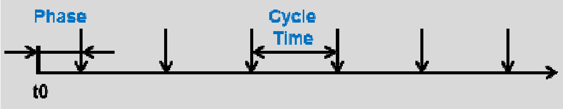

Figure 5: The GLS time domain

5 Conditions and Events

5.1 Conditions

Conditions determine the evaluation of Boolean expressions restricted to time periods specified by *continuous time locators* (or CTLs). The general rule is that they are *true* when not in the time period, and take the value of the Boolean expression when in the time period. However, there are cases where they are undefined (see example below).
Conditions are used to specify so-called *condition-based properties*. The difference between the two is that properties express something that is desirable, required or assumed, whereas conditions are just buiding blocks to express properties, as it is often preferable to break the expression of complex properties into simpler, intermediate expressions.

```
condition C = duringAny s4 check Off;
```

This condition is used to specify when the MPS can or must be declared unavailable. In particular, to avoid activating the BPS for short MPS losses, it must have been off for at least 4 consecutive seconds (`s4`).

Condition *C* is evaluated at the end of each 4 second time window (**`duringAny`** `s4`): it is *true* if *Off* had been *true* during the complete time window. It is *false* otherwise. It is *undefined* during the first 4 seconds of the simulation run, since no 4 second time window has elapsed yet.

5.2 Events

An *`event`* characterises the occurrence of one or more facts that have no duration: each of these facts is an occurrence of the event. Events are used to specify so-called event-based properties.

```
external Boolean EndMaint;

event endMaint =
  when EndMaint becomes true;
```

The external Boolean *Endmaint* represents the time-continuous signal issued by one of the buttons at the disposal of the human operator. This signal is simulated by the behavioural model of the BPS environment. Transition from *false* to *true* (**`becomes true`**) denotes the end of the on-going maintenance operation.

6 Properties, Requirements, Assumptions, Guards

There are two types of properties: condition-based properties, and event-based properties. Two Boolean functions are attached to each property:

- *Violated* is initially *false*. It becomes *true* at the first instant where the property is violated, and remains so until the end of the simulation run.
- *Evaluated* is also initially *false*. It becomes *true* at the first instant where the *Violated* / *notViolated* status can no longer be modified in the course of the simulation run, and remains so until the end of the simulation run.

6.1 Condition-Based Properties

Like a condition, a condition-based property specifies a Boolean expression and a CTL: the Boolean expression should be *true* during the CTL.

A three-valued function (*satisfied*, *notSatisfied*, *notApplicable*) is attached to each condition-based property. During the CTL, the property is *satisfied* when the Boolean expression is *true*, and *notSatisfied* otherwise. It is *notApplicable* at time instants not covered by the CTL or when the combination of CTL and Boolean expression is *undefined*. It becomes *Violated* at the first instant where it is *notSatisfied*.

```
property P2ce =
  after (BPSNeeded becomes true)
    within s60
  check SBC[6].Powered becomes true;
```

This property expresses the need to start providing electric power to the 6th SBC (`SBC[6].Powered` **`becomes true`**) at most 60 seconds after the BPS has been declared as needed.

6.2 Event-Based Properties

An event-based property specifies a constraint on the number of occurrences of an event during a given time locator.

A three-valued function (*belowLimits*, *withinLimits*, *aboveLimits*) is attached to each event-based property, indicating whther the number of occurrences is below, within or above the limits specified. The property becomes *Violated* at the first instant where it is *aboveLimits*.

```
property P2 = {P2a; P2b; P2c};

required property R7 =
  until (or{P in P2 | P.Violated})
    becomes true
  check no eFailure;
```

`P2` is the set of properties stating when SBC1 (`P2a`), SBC2-5 (`P2b`) and SBC6 (`P2c`) must be powered by the BPS. When any one of the SBCs cannot be powered back within the allocated time, a failure signal (`eFailure`) must be sent to the operator.

Property `R7` expresses the requirement that this signal must not be sent (**`no`** `eFailure`) spuriously, i.e., before any one of the `P2` properties is violated.

6.3 Requirements,

A *requirement* is a property that MUST be satisfied: it is the objective of simulation to verify that it is not *Violated*.

6.4 Assumptions

An *assumption* is a property that is supposed to be satisfied: simulation scenarios assume / ensure that it is satisfied. Assumptions are usually made with respect to the environment or the boundaries of the system under study. They can also be made on system aspects not fully determined yet, in preliminary stages of design.

```
assumed property
  during MPS.Unavailable
  check no eMaint;
```

This assumption concerns the human operator's actions. It states that when the MPS is not available, the operator will not launch a maintenance session (**no** eMaint).

6.5 Guards

A guard is a property that states the conditions that must be satisfied for a model to be valid. It is to be used for multi-modelling, when several models are available for the same system or components, each corresponding to specific situations.

7 Time Locators

There are two types of time locators in FORM-L: continuous time locators (CTLs) and discrete time locators (DTLs).

7.1 Basic Continuous Time Locators (CTLs)

A CTL specifies one or more time periods. Time periods have a duration and usually have a position in time (see Figure 6).

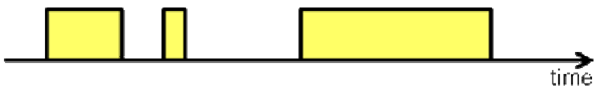

Figure 6: Time intervals

durationAny duration

This defines a sliding time window, i.e., any time period of a given `duration` (see Figure 7 and example in Section 5.1).

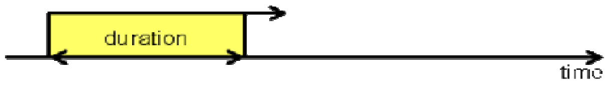

*Figure 7: **duringAny** duration*

Sliding time windows are a very distinct type of CTL and cannot always be used as the other types of CTL.

during condition

The time periods defined are those where `condition` is ***true*** (see figure 8).

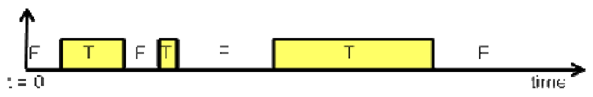

*Figure 8: **during** condition*

```
property P4 =
  during not(BPSNeeded)      // CTL
  check not(Active);
```

In this example, the CTL is specified in line 2. The property states that when the BPS is not needed, it should not be activate.

***after* event**

The time periods defined begin with each occurrence of `event` and last until the end of the simulation run (see figure 9). There are as many periods as there are event occurrences: if there are more than one occurrence, they will overlap.

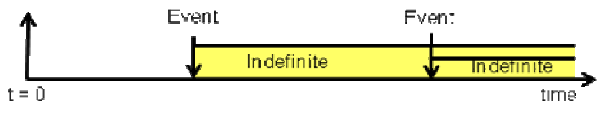

*Figure 9: **after** event*

```
event eMustAbortM =
  (after eMaint within s60)
  and MPS.eLoss;
```

In this example, the meanings of the named events and conditions are as follows:

- `eMustAbortM` is an event that signals that a BPS maintenance request previously raised by the operator needs to be cancelled.
- `eMaint` is an event raised by the operator signalling that maintenance will be performed on the BPS.
- `MPS.eLoss` is an event signalling that the MPS has been lost, and thus that the BPS is needed.

This event occurs when the MPS is lost 60 seconds or less after a maintenance request has been issued. In such cases, the maintenance request is expected to be aborted (see example below).

after event for duration

after event within duration

The time periods defined begin with each occurrence of `event` and last for the specified `duration` (see figure 10).

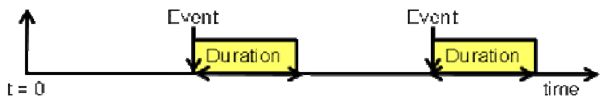

*Figure 10: **after** event **for** duration*

Both syntaxes have the same meaning, but whereas the expression with ***for*** is mainly used for condition-based properties (the condition must be true ***for*** a certain time period), the expression with ***within*** is mainly intended for event-based properties (an event must occur ***within*** a certain time period).

```
assumed property
   after eMustAbortM within s5
   check endMaint;
```

This statement assumes that within 5 seconds (`s5`) after an occurrence of `eMustAbortM` (the event signalling that a BPS maintenance request should be cancelled), the effective cancellation event (`endMaint`) is indeed raised by the operator.

`after event1 untilNext event2`

The time periods defined begin with each occurrence of *event1* and last until the first strictly following occurrence of *event2* (see figure 11). There are as many intervals as there are occurrences of *event1*.

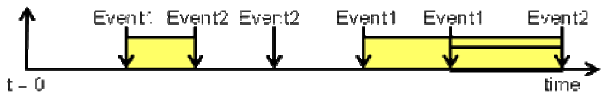

*Figure 11: **after** event1 **untilNext** event2*

```
condition Running =
   after eStart untilNext eStop;
```

This statement concerns the backup generator of the BPS. The generator can signal several events, including `eStart` (it has started) and `eStop` (it has stopped). The statement defines the time periods where the generator is `Running`.

`until event`

The time intervals defined all start at the beginning of the simulation run. There is one time interval per occurrence of `event`, and it ends with the occurrence (see figure 12).

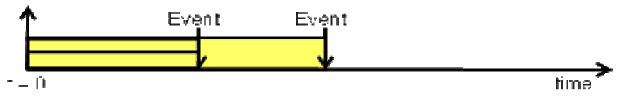

*Figure 12: **until** event*

```
condition BPSNeeded =
   (MPS.Unavailable or after Op.eVTest)
   and not(Maintenance)
   and until Op.eVReset;
```

This statement specifies when the BPS is needed. It is a time period that begins when the MPS has become unavailable (`MPS.Unavailable`) or when the operator has initiated a periodic test (`Op.eVTest`), under the condition that the *BPS* is not under maintenance (**`not`**`(Maintenance)`). It ends when the operator issues a valid reset (`Op.eVReset`):.

***`every`* `duration1` *`for`* `duration2`**

This expression defines periodic time intervals (see figure 13). `duration2` should be shorter than `duration1`.

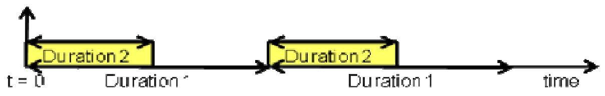

*Figure 13: **every** Duration1 **for** Duration2*

```
required property
   every s10 for s2 check eCheck;
```

This statement requires that periodically, every 10 seconds (`s10`), event *eCheck* should occur in the first 2 seconds (`s2`).

7.2 Combining / Transforming CTLs

The FORM-L offers various means for deriving new CTLs from already existing ones. These are summarised in figure 14. The only type of CTL that cannot be used in these expressions are sliding time windows: they can be neither transformed nor combined directly, but conditions based on sliding time windows can be used (with keyword **when**) to define CTLs that can be combined and transformed.

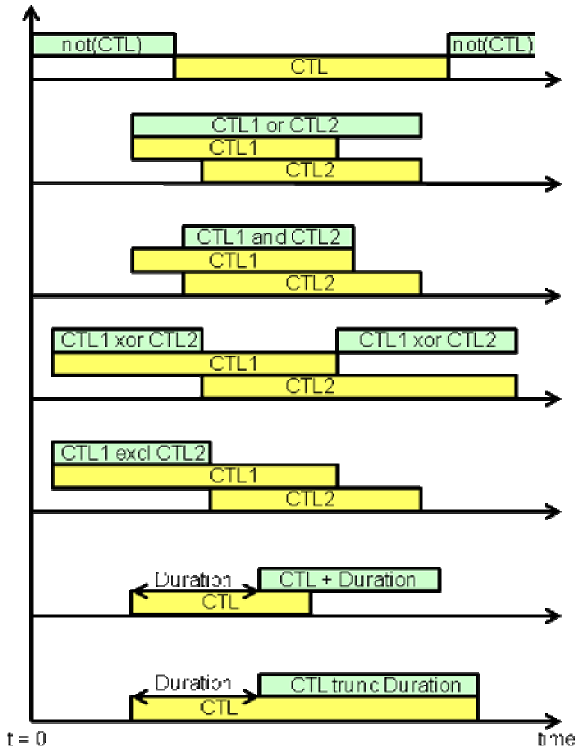

Figure 14: Deriving new CTLs from existing ones

7.3 Basic Discrete Time Locators (DTLs)

A DTL defines one or more positions in time and has no notion of duration.

`when condition becomes true`

`when condition becomes false`

`when condition changes`

The DTL has a time position for each instant where `condition` becomes *true*, *false* or changes value (see Figues 15 and example of Section 6.1). In a discrete time domain, `condition` changes value when and only when its value at a given instant is different from its value at the preceding instant.

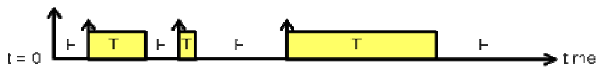

Figure 15a: when condition becomes true

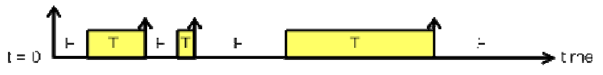

Figure 15b: when condition becomes false

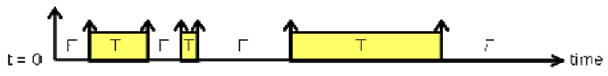

Figure 15c: when condition changes

`when fsa becomes state`

`when fsa leaves state`

`when fsa changes`

`when integer changes`

DTLs can be defined when a finite state automation (`fsa`) enters or leaves a given discrete `state` or changes state. They can also be defined when an `integer` function changes value.

`when event`

This expression simply denotes the DTL associated with an `event`.

`every duration`

This expression defines a periodic DTL, the period of which is `duration` (see figure 16).

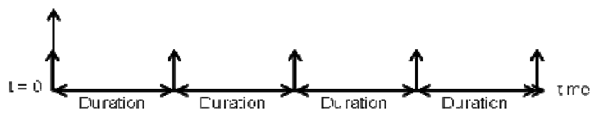

Figure 16: ***every*** *duration*

```
constant duration CycleTime = ms50;
private fixed duration Phase =
  random(0.0, CycleTime);
dtl = (every CycleTime) + Phase;
```

This example defines the discrete time domain associated with the GLS, which has a synchronous design: it sees its environment and acts upon it only at the instants defined by the specified `dtl`. The period is defined by `CycleTime` (50 milliseconds). The time domain has a random phase defined by `Phase`.

7.4 Combining / Transforming DTLs

FORM-L offers various means for deriving new DTLs from already existing ones. These are summarised in figure 17.

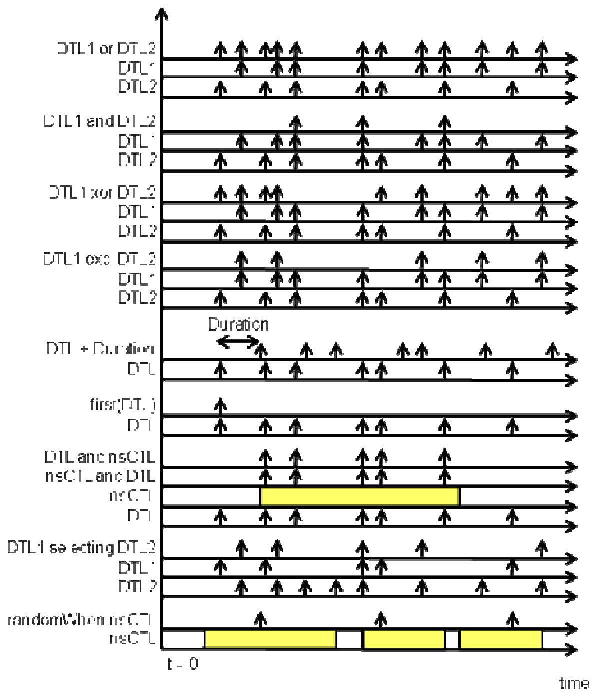

Figure 17: Deriving new DTLs from existing ones or from CTLs

8 Sets, Subsets and Arrays

Sets are an essential ingredient of FORM-L, particularly when requirements and assumptions are stated at very early stages of a project, where the precise identifications and numbers of the objects constituting the system are yet unknown.

Sets are first divided into two main categories: *static sets* (the membership of which does not vary in time) and *dynamic sets.*

Static sets are either *enumerated sets* (their members are listed individually) or *queried sets* (their membership is defined in intention based on constraints on static attributes, and is obtained at the beginning of a simulation run through a query to an engineering database).

Dynamic sets are always subsets of static sets. Their membership is defined based on constraints involving dynamic attributes.

All sets are ordered, either explicitly or implicitly. In this sense, they are close to single dimension arrays (vectors) of the current MODELICA language.

However, there is one significant difference: an array contains its members and an object cannot belong to two different arrays; a set references its members, and an object may belong to several sets.

FORM-L has set operators (such as union or intersection), arithmetic operators (such as sum, minimum or maximum) and Boolean operators (such as and or or). It also provide a cardinal function that calculates the number of members in a set and that allows the expression of universal or existential quantifiers.

9 Actions

FORM-L provides the notion of action to support the modelling of designs.

9.1 Elementary Actions

Elementary actions belong to one of two types:

- Assignments to functions.
- Raising of events.

They are composed of two parts: an optional delay part (specified by a time locator) and an action specification (the effective event raising or function assignment).

9.2 Composite Actions

Composite actions regroup two or more actions that need to be performed in a coordinated manner at instants specified by a CTL or during time periods specified by a DTL. There are two types of coordination:

- Sequences, where the member actions are performed one after the other.
- Simultaneous actions, where all actions are performed at the instants specified by a DTL.
- Non-ordered actions, where the actions are performed at unspecified instants within the time periods specified by a CTL.

```
when State becomes Operational(Active;)
  then sequence
    raise eStartDG;
    wait eDGReady
      then raise eOpenMPSBrk;
    wait eMPSBrkOpen
      then raise eShedAll;
    do
      wait s1 then raise eCloseDGBrk;
      wait s5 then raise eReload;
    end;
  end;
```

This composite action specifies the elementary actions to be implemented by the GLS when the BPS is needed (see Figure 17).

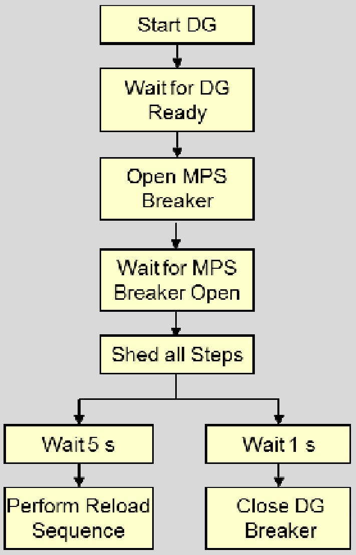

Figure 17: Beginning of the GLS sequencing

10 Probabilistic Properties

The MODRIO project is developing extensive stochastic modelling capabilities for MODELICA. These will not be addressed here: for requirements modelling, FORM-L needs only two simple notions.

10.1 Probability Function

The first notion introduced by FORM-L is a function of time: its value at a given instant is a Real number in the [0., 1.] range that represents the probability that a specified event occurred at least once before that instant.

```
property P3 =
  during not(BPSNeeded)
  check not(Active);

required property R3 =
  when SingleSensorFailure then P3;

required property R9 =
  probabilityFunction
    (P3.Violated becomes true)
      < 1-exp(-Y*time);
```

Property P3 states that the BPS should not be spuriously activated when it is not needed. Requirement R3 states that when there are no BPS component failures, or at most one sensor failure, then P3 must be satisfied. Requirement R9 puts a limit on the probability of spurious activation of the BPS (due to BPS components failures). This probability must be lower than what it would be if there was a constant occurrence rate of spurious activation, equal to Y.

10.2 Probability

The second second notion introduced by FORM-L is a Real constant or fixed value (not a function of time) in the [0., 1.] range that states the probability of an event occurring at least once during a given time period (possibly during the whole simulation run). This could be used for example to specify conditional probabilities, with the time period representing the condition. Application of conditional probabilities include probabilities of failure on demand or probabilities of common-cause failure.

```
property P2ce =
  after (BPSNeeded becomes true)
    within s60
  check SBC[6].Powered becomes true;

property P2e = {P2ae; P2be; P2ce};

required property R3a =
  probability (
    (card{P in P2e | P.Violated} == 1)
       becomes true)
    < 5*10-3;
```

Property `P2ce` states that SBC6 (`SBC[6]`) should start to be powered within 60 seconds after the BPS is declared needed. There are similar properties for the other SBCs (`P2ae` and `P2be`). They are grouped in `P2e` which is the set of properties regarding the timeliness of providing power to the SBCs.

Requirement `R3a` specifies the maximum probability of not satisfying any single one of these properties.

11 Conclusions

A number of tasks remains to be achieved, mainly:

- Finalizing the FORM-L concrete syntax, in order to bring it closer to the current MODELICA syntax when this is not at the cost of clarity and conciseness.
- Deciding on an implementation path in the existing tools environments. It is likely that integration with other existing tool environments (in particular with Computer Aided Design and Product Life Management environments and their data bases).
- Developing the methodological aspects to better support systems engineering and systems operation activities.

12 Acknowledgments

This work is partially supported by DGCIS within the ITEA 2 MODRIO project.

References

[1] Schamai W., Buffoni L., Bouskela D., Fritzson P., ‘Automatic Model Composition using Bindings in Modelica’, Modelica 2014 conference proceedings, Lund, 2014.

[2] Bouskela D., El Hefni B., ‘A physical solution for solving the zero-flow singularity in static thermalhydraulics mixing models’, Modelica 2014 conference proceedings, Lund, 2014.

[3] Bouskela D., Jardin A., Nguyen T. 'Innovative Modelling Architecture for Design Verification against System requirements', , Modelica 2014 conference proceedings, Lund, 2014.