The BASAALT
(Behaviour Analysis & Simulation All Along systems Life Time)
systems engineering method

and FORM-L
(FOrmal Requirements Modelling Language),
its supporting language

Thuy NGUYEN

Executive Summary

This document presents the principles of the BASAALT (Behaviour Analysis & Simulation All Along systems Life Time) systems engineering method, which has been designed to support the engineering of large socio-technical and cyber-physical systems all along their life time. These systems, such as nuclear power plants and their many plant systems, combine multi-physics, computation, networking and human actions. The method may also be applied to the engineering of systems of systems, i.e., systems composed of multiple interacting socio-technical and cyber-physical systems owned, developed and operated by different organisations, such as wide area energy systems and smart grids.

BASAALT is based on the study of a wide variety of concrete cases, including hydraulic cooling systems, HVAC (Heating, Ventilation and Air Conditioning) systems, backup electric power supplies, smart urban heat and electricity networks, and ECSs (Environmental Control Systems) in commercial aircrafts. It integrates semi-formal aspects which may sometimes be imprecise or subjective, with formal, rigorous and objective aspects.

Semi-formal aspects are used during initial conceptual phases, and also to document and organise the complex and sometimes subjective chains of reasoning that justify the WHY of key engineering decisions and also the legitimacy of assumptions made. However, they offer very limited tool-support. The main objective of the formal and rigorous aspects of BASAALT is to complement the semi-formal aspects to support systems engineering with simulation (and in favourable cases with comprehensive formal analysis) all along systems lifecycle to help master complexity and to coordinate (just as necessary) the many teams, disciplines, organisations and stakeholders involved, which often have difficulties understanding one another.

Thus, in addition to BASAALT, this document presents the notions, the syntax and examples of FORM-L (FOrmal Requirements Modelling Language), the modelling language supporting the formal and some semi-formal aspects of BASAALT. It also briefly presents the main notions and principles of justification frameworks (which capture and organise complex chains of reasoning) and also the principles of MESKAAL (Maintenance of Engineering and Safety Knowledge on systems All Along their Life time).

FORM-L creation has been decided after a long but unsuccessful search for existing formal constraint specification languages answering to the needs of the initial phases of physical systems engineering: a number constraint languages were found, but they address mainly cyber aspects. It has been specifically designed to express constraints and requirements on dynamical phenomena in a rigorous and unambiguous manner. Such requirements address functionality (what the system must do and must not do, depending on the situations the system may face, including failure situations), performance (e.g., response times and accuracy) and quality of service (e.g., fault-tolerance and limits to failure probabilities). It also allows the expression of assumptions, in particular regarding the environment of the system and human actions.

Modelling in FORM-L is very different from, and complementary to, modelling in Modelica or with functional diagrams. Whereas a Modelica model or a functional diagram is a deterministic model which, given initial and boundary conditions, allows only one behaviour, a FORM-L model is a constraint model that allows many possible solutions and behaviours. Such a model may be developed at very early stages of the systems engineering process, when there is no design solution yet. FORM-L constraint models may also be used to support the first design steps, where subsystems are identified and system requirements are allocated to subsystems. Particularly when subsystems are to be subcontracted, it is essential to make sure beforehand that the combined subsystems requirements specifications collectively satisfy the system requirements.

With suitable tools such as StimuLus, such modelling enables simulation for large numbers of alternative solutions and scenarios. In favourable cases, it may even enable exhaustive formal analysis. This could be used to validate the adequacy of system requirements (are they suitable for all the situations the system may face?), to verify system overall design and detailed design, and to support activities such as failure analyses.

The FORM-L syntax presented here is designed to facilitate the use of the language by application domain experts: it is they who ultimately will develop, or at least verify, the models. It can be translated into other languages such as StimuLus, Modelica, Scade or Figaro to benefit from their tools.

Synthèse

Ce document présente les principes de la méthode d'ingénierie système BASAALT (Behaviour Analysis & Simulation All Along systems Life Time) conçue pour assister l'ingénierie des grands systèmes sociotechniques et cyber-physiques tout au long de leur cycle de vie. Ces systèmes, comme par exemple les centrales nucléaires et leurs nombreux sous-systèmes, intègrent physique, informatique, réseaux de communication et actions humaines. La méthode peut aussi être appliquée à l'ingénierie des systèmes de systèmes, c'est à dire de systèmes composés de systèmes sociotechniques et cyber-physiques interagissant mais possédés, développés et exploités par des organisations différentes, comme par exemple les systèmes énergétiques et les réseaux dits intelligents (smart grids).

BASAALT résulte de l'étude d'une variété de cas concrets, tels que des systèmes hydrauliques de refroidissement, des systèmes de ventilation et de climatisation (HVAC), des alimentations électriques de secours, des réseaux urbains intelligents de chaleur et d'électricité, et des ECS (Environmental Control Systems) de l'aéronautique. Elle intègre aspects semi-formels parfois imprécis ou subjectifs, et aspects formels rigoureux et objectifs.

Les aspects semi-formels sont utilisés d'une part lors des phases conceptuelles initiales, et d'autre part pour expliciter et structurer les raisonnements compliqués et parfois subjectifs expliquant le POURQUOI des choix effectués et justifiant la validité des hypothèses faites. Cependant, ils ne permettent qu'un support outillé limité. Un objectif fort des aspects formels et rigoureux de BASAALT est de compléter ces aspects semi-formels afin de permettre une forte assistance par la simulation (et dans les cas favorables, par l'analyse formelle exhaustive) tout au long du cycle de vie des systèmes afin d'aider à en maîtriser la complexité et à coordonner au plus juste des équipes, disciplines, organisations et parties prenantes ayant souvent du mal à se comprendre.

Ce document présente également les concepts, la syntaxe et des exemples de FORM-L (FOrmal Requirements Modelling Language), le langage de modélisation qui supporte les aspects formels et certains aspects semi-formels de BASAALT. Il présente aussi brièvement les structures de justification (justification frameworks) permettant d'expliciter et structurer les raisonnements humains, ainsi que les principes de MESKAAL (Maintenance of Engineering and Safety Knowledge on a system All Along its Life time).

La création de FORM-L a été décidée après une longue recherche infructueuse de langages formels de spécification de contraintes adaptés aux phases initiales de l'ingénierie de systèmes physiques, les quelques langages de contraintes existant ne traitant que les aspects cyber. Il a été spécifiquement conçu pour exprimer de façon rigoureuse et non ambiguë des contraintes et des exigences sur les phénomènes dynamiques, notamment sur les fonctionnalités (ce que le système doit faire et ne pas faire, selon la situation, y compris les situations de défaillance), la performance (comme les temps de réponse ou la précision) et la qualité de service (comme la tolérance aux pannes et les probabilités de défaillance). Il permet aussi d'exprimer des hypothèses, concernant par exemple l'environnement du système et les actions humaines.

La modélisation en FORM-L est très différente mais complémentaire de la modélisation en Modelica ou en diagrammes fonctionnels. Alors qu'un modèle Modelica ou un diagramme fonctionnel est un modèle déterministe qui, à partir de conditions initiales et aux limites, conduit à un comportement unique, un modèle FORM-L est un modèle de contraintes autorisant de multiples solutions et comportements. En conséquence, un tel modèle peut être développé très tôt dans le processus d'ingénierie, quand aucune solution n'a encore été choisie. Les modèles de contrainte en FORM-L peuvent aussi être utilisés dans les premières étapes de conception, lorsqu'on identifie les sous-systèmes et qu'on leur répartit et alloue les exigences du système. En particulier lorsque des sous-systèmes sont sous-traités, il est essentiel de s'assurer que l'ensemble des exigences sur les différents sous-systèmes vont collectivement satisfaire les exigences du système.

Avec des outils tels que StimuLus, une telle modélisation autorise la simulation d'un grand nombre de cas pour la validation des exigences (sont-elles adaptées à toutes les situations auxquelles le système aura à faire face ?) et la vérification des solutions architecturales, des solutions détaillées ainsi que des effets des défaillances.

La syntaxe de FORM-L présentée ici vise à faciliter l'utilisation du langage par des experts applicatifs : ce sont eux qui, *in fine*, vont développer ou au moins vérifier les modèles. Elle peut être traduite vers d'autres langages comme StimuLus, Modelica, Scade ou Figaro pour bénéficier de leurs outils.

Table of Contents

1 Introduction

1.1 Context & Objective of the Document

The systems engineering approach and the modelling framework presented in this document were initially developed in the framework of the European ITEA programme, first with project EuroSysLib (EUROpean leadership in SYStem modelling and simulation through advanced modelica LIBraries, 2007-2010), and then with project MODRIO (Model Driven Physical Systems Operation, 2012-2015). The principles of the justification framework also presented in this document were developed in the framework of the European EURATOM programme, with project HARMONICS (Harmonised Assessment of Reliability of MOdern Nuclear Instrumentation and Control Software, 2011-2015).

The research effort continued at EDF after MODRIO and HARMONICS. This document presents the principles of BASAALT (Behaviour Analysis & Simulation All Along systems Life Time), a model-based systems engineering (MBSE) approach assisted by simulation and analysis, and the notions and syntax of the FORM-L language (FOrmal Requirements Modelling Language) which supports the formal and some semi-formal aspects of BASAALT. Both are illustrated with examples. MESKAAL (Management of Engineering and Safety Knowledge about systems All Along their Life time) is a by-product of BASAALT, FORM-L and the justification framework.

1.2 Rationale

At the beginning of the MODRIO project, the plan was first to list the desirable features for a rigorous language suitable for the modelling, simulation and formal analysis of large and complex *socio-technical systems* (i.e., systems integrating human actions and technological aspects), *cyber-physical systems* (i.e., systems integrating physics, computing and networking) and large, distributed *systems of systems* (i.e., dynamic sets of multiple, more or less independent but interacting systems) all along their life cycle, then to perform a survey of the existing languages, and finally to select the most appropriate one. Unfortunately, although the survey identified several languages suitable for *cyber* systems (i.e., systems based exclusively on digital electronics, computing and networking), none was found that also covered the *human* and *physical* aspects (see [7] and [8]).

Thus, based on the in-depth study of a wide variety of concrete cases from actual engineering projects (for example, see [21], [22], [23], [24], [25] and [26]), including hydraulic cooling systems, HVAC (Heating, Ventilation and Air Conditioning) systems, backup electric power supplies, smart urban heat and electricity networks, and ECSs (Environmental Control Systems) in commercial aircrafts, the list of desirable features gradually matured into the BASAALT approach and the FORM-L language.

As the engineering of large and complex systems covers a very wide spectrum that would be difficult and inconvenient to address with a single method and a single language, BASAALT and FORM-L are centred on and deal with *dynamic phenomena*, even though they can address, with limited capability and ease, other aspects such as geometry and topology[1]. They have four main objectives:

- To support the rigorous and unambiguous modelling of *constraints* on dynamic phenomena, with *variables* (representing continuous or discrete system states), *events* (representing the occurrence of facts the duration of which is neglected), *properties* (such as *assumptions*, *objectives* and *requirements*), *sets* (enabling definitions *in intention* and first order logic) and *objects* (representing real world entities such as systems, subsystems, human agents, passive structures or the physical environment).
- To support the engineering of systems all along their life cycle, with step-by-step *extension* and *refinement* to keep track, as engineering progresses, of what is gradually known and decided about a system, its environment and its constituents.
- To support the coordination necessary to the engineering of large and complex systems, with *contracts* (formal, engineered interfaces between two or more objects), *encroachments* (formal, undesired effects an object may have on another one due to failures and other abnormal conditions), *bindings* (formal interfaces between models that have been developed independently from one another or of different natures, including non-FORM-L models) and *main objects* or *main classes* (representing the systems of interest of a given FORM-L model), which together enable *modular modelling*, *models composition*, *focus* and *abstraction* (concentrating on what is necessary to a given engineering activity and setting aside non essential details).

[1] A future extension is contemplated to facilitate the addressing of geometric and topological aspects in 3D, 2D and 1D spaces.

- To support informative traceability between, and to facilitate the understanding of the rationales of, engineering decisions, with the notions of *objective*, *extension*, *refinement*, *concretisation*, *substitution* and *justification*.

1.3 Structure of the document

The BASAALT principles are presented in Section 3 and illustrated with an extensive case study in Section 4. The subsequent sections present FORM-L, the modelling language supporting BASAALT. As its notions are highly interdependent, Section 5 provides an overview of the language that is probably sufficient for most casual readers. Readers interested in the details of the language can find them in Sections 6 to 20.

1.4 What Changed?

Compared to the first edition [10] of the document issued in May 2021, most changes are purely editorial, to facilitate understanding and improve clarity and precision. The method and the language themselves did not change significantly, except that:

- A model no longer *refines* other models: it *extends* them. This is to have consistent definitions for *extension* (which creates a new item) and *refinement* (which does not).
- *Spatial locators* are now named *selectors*, to prepare a possible language evolution with true spatial locators in 3D, 2D or 1D spaces.
- *Model configurations* are now named *mock-ups*, to be consistent with the terms used in the FMI (Functional Mock-Up Interface) standard [11].
- *Side effects* are now named *encroachments* to highlight their invasive nature.
- As a party to a contract or an encroachment may be a class (standing for all its instances), standard contracts and standard side effects / encroachments are of little use and have been removed.
- Pre-defined property class ***Expectation*** has been removed: in a *contract*, the *expectations* of one party with respect to the others are now expressed with ***Assumptions***.
- Attribute ***values*** has been introduced, grouping all the ***value*** of an object (its own ***value*** if it is a valued object, and the ***value*** of all its direct and indirect valued features).
- To facilitate understanding, property operators ***wOr***, ***sOr*** and ***pAnd*** have been renamed ***weakOr3***, ***strongOr3*** and ***and3***.
- Set operators ***orAll***, ***andAll***, ***wOrAll***, ***sOrAll*** and ***pAndAll*** have been renamed ***OR***, ***AND***, ***weakOR3***, ***strongOR3*** and ***AND3***.
- Set of set operators ***unionAll*** and ***interAll*** have been renamed ***UNION*** and ***INTER***.
- Operator ***changes*** (which applies to the ***value*** of a valued object) has been completed with operators ***mutates*** (which applies to the ***values*** of an object), ***CHANGES*** (which applies to the ***value*** of each member of a set of valued objects) and ***MUTATES*** (which applies to the ***values*** of each member of a set of objects).
- The syntax of basic duration-based *Continuous Temporal Locators* (*CTL*) has been modified for improved clarity.
- The syntax for *Sliding Temporal Locators* (*STL*) has been modified to eliminate the need for specific keywords.
- *Iterations* have been simplified: since they can be placed under the scope of a temporal locator specifying a Boolean condition or a terminating event, only indefinite iterations (with keyword ***repeat***) and iterations for a specified number of times (with keyword ***iterate***) are necessary.
- In *selections*, keyword ***remaining*** has been replaced by **true** (in the case of deterministic Boolean selections) and **1** (in the case of probabilistic selections).
- *Domains* are now *sets of values* (with a **${ }** notation).
- A new determiner has been added: conceptual items are now marked by determiner **^**, whereas determiner **~** now marks items that are not intended to be formalised.

- Keyword ***redeclared*** is used when some aspects of an existing declaration are no longer valid and need to be modified.
- All FORM-L items (including instructions) may be named, so that they can be referred to within the justification framework.
- The notion of *type* is now merged with the notion of *class*. What used to be *behavioural items* are now *objects*. *Variables*, *events*, *sets* and *properties* are now *valued objects* and are treated on an equal footing with what used to be called *objects*. What used to be called *objects* are now *non-valued objects*. Multiple inheritance now applies to all defined classes.

2 Notations

2.1 Notation for Examples

For easy identification, illustrative examples are given within boxes with a light grey background. FORM-L models are written **`using this font`**. Comments are written with a normal font.

Names follow the *CamelCase* convention, where names based on multiple words are written without separators, the beginning of each word (possibly except the first one) being marked by a single capital letter. Model and template names begin with a capital letter, whereas instance names generally begin with a lowercase letter. For example:

- ***`AcVoltage`*** **`mpsVoltage`** where ***`AcVoltage`*** is a class (i.e., a template), and **`mpsVoltage`** a variable (i.e., an instance)
- **`BpsIntroduction`** the name of a model

By convention, event names begin with an *e*. Also, the use of a predefined FORM-L standard *library* defining physical *quantity classes* and *units* is assumed. For example:

- **`1*s`** 1 second
- **`10*V`** 10 volts (constants representing standard units sometimes begin with a capital letter to conform to the International Standard notation)
- **`100*ms`** 100 milliseconds
- **`1000*kW`** 1000 kilowatts

To facilitate legibility and understanding, FORM-L examples are shown with syntactic highlighting as can be provided by syntactic editors such as Notepad++ [13]:

- General purpose keywords are in ***`deep blue italics`***: ***`begin end`***
- Keywords specifying the nature of a FORM-L item and the classes defined by the standard library are in ***`deep green italics`***: ***`Property`***, ***`Class`***, ***`Boolean`***, ***`Mass`***
- Keywords related to *temporal locators* are in ***`light blue italics`***: ***`when`***, ***`otherwise`***
- Keywords related to *selectors* are in ***`pink italics`***: ***`for all`***, ***`such that`***
- Keywords related to *actions* (assignments, event signals, constraints and object deletions) are in ***`brown italics`***: ***`define`***, ***`is`***, ***`ensure`***
- Named operators are in ***`light green italics`***: ***`and`***
- *Attribute* names are in ***`grey italics`***: ***`next`***, ***`derivative`***, ***`rate`***
- Constants from the standard library, predefined values, number and string literals are in **`red`**: **`5`**, **`s`**, **`A`**, **`"string"`**, **`true`**, **`me`**, **`deferred`**

```
AcVoltage mpsVoltage begin
   ensure value in [0, 240]*V;
   ensure derivative in [-100, 100]*V/s;
end mpsVoltage;
```

2.2 Notation for the Syntax

Also for easy identification, the FORM-L syntax is specified within boxes with a light-yellow background, in an *Xtext*-like format [6]. *Xtext* is an Eclipse Foundation framework for implementing domain-specific languages (*DSL*s): it is used to produce a compiler (an ongoing work) that translates FORM-L into other, tool-supported *DSL*s such as StimuLus [15], Modelica [1] or Figaro.

In addition to the formal specification of a *DSL* syntax, the actual *Xtext* format includes elements that support the automatic generation of *Abstract Syntax Trees* (*AST*). These elements have been removed to facilitate legibility and understanding. Also, to be 'compilable', a formal syntax specification (also called a *grammar*) needs to follow rules that sometimes make it difficult to read. Thus, the FORM-L grammar provided here is not the real *Xtext* grammar but a simplified, more easily understandable one.

A grammar is expressed in terms of *syntactic elements*, which can be either *terminal tokens* or *syntactic terms*, using a *BNF*-like (Backus-Naur Form) notation.

There are two sorts of terminal tokens: *named tokens* and *literal tokens*. Named token are named syntactic elements that are not further defined and are in capital letters (e.g., **NAME**). Literal tokens are red character strings enclosed in quotes (e.g., 'begin' or ';').

Syntactic terms are shown in *italics*. They are named and specified in terms of:

- Sequence of other syntactic elements.

```
ObjectDeclaration:
   class NAME ';' ;
```

Syntactic term *ObjectDeclaration* specifies that an object declaration is composed of a class identification (itself specified by syntactic term *class*) followed by a **NAME** and then by a semi-column.

- Alternatives, denoted with operator |.

```
visibility:
   'public' | 'private' ;
```

A *visibility* is specified either with keyword public or with keyword private.

- Unordered groups, denoted with operator &.

```
VariableDeclaration:
   (visibility & variability) Class NAME ';' ;
```

A *VariableDeclaration* begins with a visibility (itself specified by syntactic term *visibility*) and a variability (itself specified by syntactic term *variability*), in any order.

- *Cardinalities*. There are four possible cardinalities: exactly one (no operator), one or none (operator ?), zero or more (operator *), and one or more (operator +).

```
EventDeclaration:
   'Event' determiner? NAME ';'                                    ;
```

An *EventDeclaration* may specify an optional *determiner*.

```
TypeDeclaration:
   'Class' NAME ('extends' path (',' path)*)? ';'                  ;
```

A *TypeDeclaration* is composed of keyword Class followed by a **NAME** (the name of the class being declared), by an optional part beginning with keyword extends followed by a list of comma-separated *path* (pathnames to the classes being extended), and then by a semi-column.

Note. A grammar defines only a superset of the actual syntax of a language and does not necessarily filter out all forbidden sentences. For example, it generally does not specify that names used must be declared, that declarations and definitions must not be contradictory, or that one cannot assign a real value to a Boolean variable. Indeed, a trade-off must be found between the simplicity of the grammar and its selectivity, and most often (and this is the case here), one choses simplicity. Thus, the grammar needs to be complemented with compilation-time verifications. The depth of these verifications depends on the rigour of the compiler. It is generally preferable to aim at a high level of rigour so that as much as possible modelling errors are signalled at compilation time rather than during simulation or analysis.

3 BASAALT, a Systems Engineering Approach

3.1 Overview

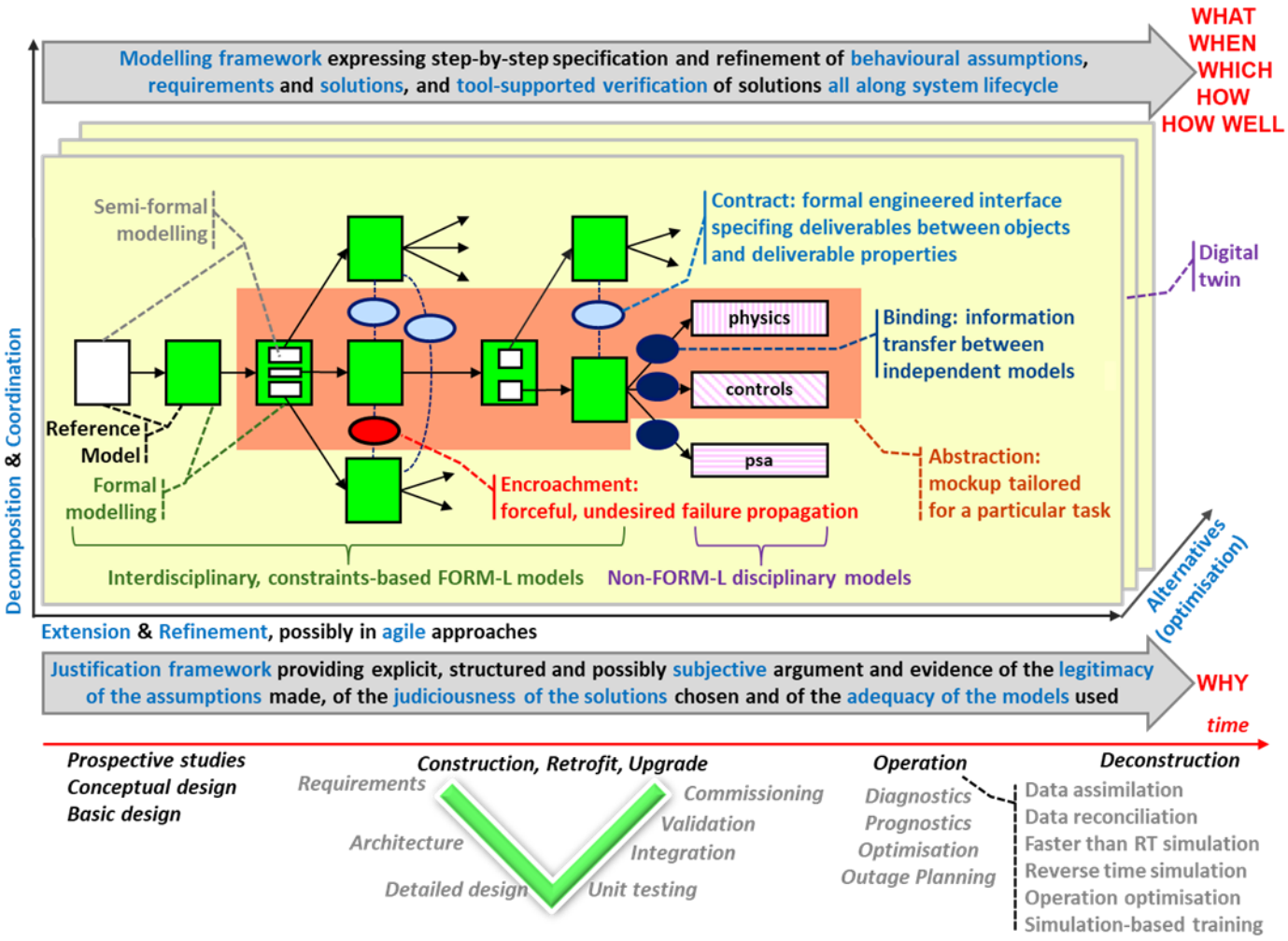


Figure 3-1
Summary of BASAALT main principles. Rectangles with a border represent models. White rectangles are semi-formal FORM-L models, whereas green ones are formal FORM-L models. Hatched rectangles are non-FORM-L models. Ellipses are interface items.

Figure 3-1 summarises the main BASAALT principles:

- Modelling modularity is necessary when addressing large and complex systems. In BASAALT and FORM-L, it is obtained, not with models of different natures (e.g., goals, responsibility, objects, structure, functions, ...) like in some other system modelling languages such as KAOS [4] or SysML [14], but with models that keep track of engineering progress, enable teams coordination, and express system decomposition.
- Initial studies may be based on natural language and on *semi-formal models*.
- As engineering progresses, top-down, step-by-step model *extension* (i.e., creation of new models providing additional information) may be used to formalise natural language statements and to gradually capture new engineering decisions.
- *Formal modelling* enables *tool-aided analysis and simulation* in support of system development but also of system operation.
- Formal modelling of object and class interfaces with *contracts* and *encroachments* facilitates just-as-necessary and tool-aided coordination of teams, disciplines, organisations and stakeholders.

- Contracts and encroachments also facilitate complexity management by enabling the composition of *bespoke mock-ups* (i.e., assemblies of models constituting simulable and analysable wholes) selectively focused on the (sub)systems and issues of interest of particular engineering activities, making abstraction of unnecessary details.
- Thus, for BASAALT, a *digital twin* is not a single model, but a set of interrelated or co-simulable models kept up-to-date with respect to the actual system, and from which one can extract different mock-ups answering to the needs of different engineering activities.
- Tool-aided analysis and simulation facilitate the exploration of alternatives in the search of *optimal solutions*, possibly in *agile* approaches.
- *Bindings* enable the bottom-up integration of pre-existing, off-the-shelf solution models and of discipline-specific, non-FORM-L detailed solution models.
- This *modelling framework* provides *objective* information regarding the WHAT, WHEN, WHICH, HOW and HOW WELL of system, its environment and its constituents.
- It is complemented by a *justification framework* that provides a structured means to explicitly *justify*, possibly with *subjective* reasoning and / or with information out of the scope of the modelling framework (e.g., historical data, regulations, standards, …), WHY the engineering organisation(s) think that the assumptions made are legitimate, the solutions chosen are judicious, the models used are appropriate, and the verifications performed are sufficient.

3.2 Requirements Engineering

Experience from all industrial sectors shows that even for highly critical systems, defects in requirements specification are a significant cause of failure to meet expectations, with sometimes wholly unacceptable consequences in terms of delay, cost, damage to property or the environment, and even loss of life. For the nuclear industry, see [17], [18] and [19]. See also . Thus, *requirements engineering* is an essential and vital engineering activity.

In BASAALT, it is not confined to a separate engineering phase, but is on the contrary performed recursively at least all along development and often during construction, operation and decommissioning. That is particularly true in the case of large and complex socio-technical and cyber-physical systems, for which many subsystems are systems of their own: then, the design of a system at a given level implies specifying requirements for its subsystems and components (see Figure 3-2).

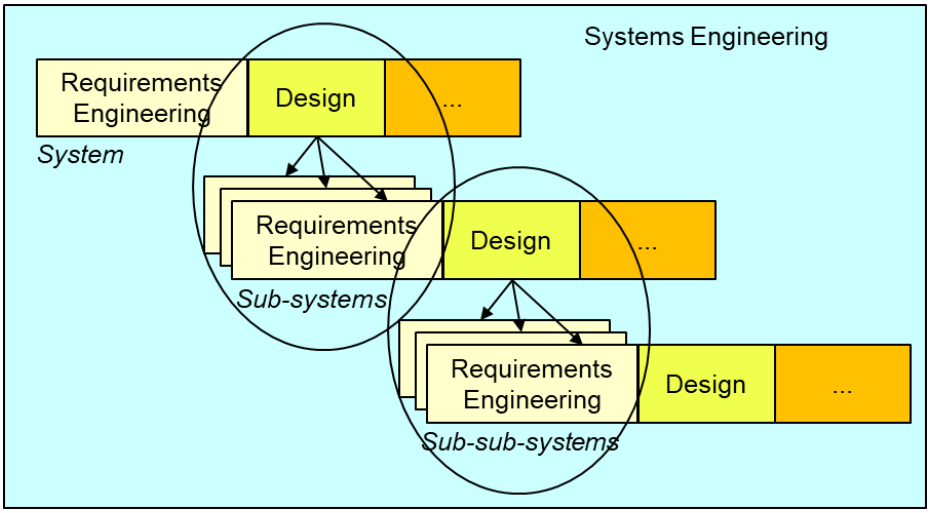


Figure 3-2
Requirements engineering permeates the system life time.

As systems engineering is concerned with the complete life cycle of a system, requirements engineering needs to take account of their possible consequences at each stage of the life cycle, and thus of *system owners'* standpoint:

- system, not of its operation. They receive specified requirements as an input, and look for possible defects essentially as impediments to their own work.
- Owners, on the contrary, are responsible for the overall system over its complete life cycle, and it is they who have to elicit and specify the top-level requirements and make sure that the lower-level requirements will not have unacceptable consequences at any stage of the life cycle.

From an owner point of view, seven main classes of defects may affect specified requirements, some of which could lead to unacceptable consequences:

- *Inadequacy*, where, in some situations, what is specified is woefully inappropriate (see the *Cranbrook Manoeuvre* example in Section 4.4), or where what is necessary in some situations is not specified (silence). BASAALT and FORM-L address this issue with extensive simulation and / or formal verification. Formal *assumptions* may be used to model the set of situations the system of interest may face: test case generators and simulators like StimuLus [15] can generate automatically any number of different operational cases to explore that set of situations. Formal *requirements* express what the system must do and not do, and can be automatically verified for satisfaction or violation in each considered operational case.
- *Ambiguity*, where different persons concerned could make different interpretations of what is specified. Even though formal modelling does provide some help, it is not sufficient in and by itself. In particular, most formal languages may still leave ambiguities in how the individual terms of a formal expression relate to entities or values of the real world. Also, some persons may have an incorrect understanding of the formal language used and may misinterpret some expressions. BASAALT and FORM-L address these issues with the notions of *determiner* and *quantity* (as defined by the SI international standard of units [12]) and with simulation (which helps understand the true semantics of a given formal expression).
- *Apathy*, where what is specified makes no difference between what is genuinely needed and is the reason of being of the system, and what is only barely tolerable in exceptional situations. BASAALT and FORM-L address this issue with *probabilistic requirements*, where what is only barely tolerable must have a low probability of occurrence.
- *Over-ambition*, where what is specified might be interesting but is not essential and could lead to excessive complexity, higher costs, longer delays and greater risks of errors (in design, construction, operation and / or maintenance). BASAALT and FORM-L address this issue with step-by-step modelling and verification of solutions at reasonable cost and effort: one can thus explore the space of possible solutions and possibly realise early in the engineering process that no simple solution can be found.
- *Over-specification*, where what is specified is not the real objective but one possible solution, not necessarily the best and simplest, and worse, not necessarily fully satisfying the real objective. BASAALT and FORM-L address this issue by encouraging the specification of top-level objectives and requirements. These may need to be expressed in terms of immaterial, *intangible* concepts that can be neither observed nor measured, but that represent the true intentions.
- *Intangibility*, where what is specified is based on immaterial, abstract concepts, with no concrete, verifiable acceptance criteria. BASAALT and FORM-L address this issue with a specific *determiner* associated with intangible objectives and requirements, and by checking that these are eventually *concretised* with verifiable requirements based on tangible and measurable elements.
- *Infeasibility*, where what is specified is not achievable or where satisfaction of some specified requirements necessarily implies violation of others (contradiction). Like for over-ambition, BASAALT and FORM-L address this issue with step-by-step modelling and verification of solutions to explore the space of possible solutions and possibly realise early in the engineering process that no solution can be found. In favourable cases, the case generator can signal contradictions.

In practice, the requirements engineering process for a given (sub)system always starts with imperfect requirements suffering some of the aforementioned defects: not only that is inevitable, but often, that is desirable (see *Figure 3-3*). One objective of the requirements engineering process is to gradually correct these defects, but also to keep trace the process, since that may provide useful insights. To do so, one needs to consider the individual and collective meaning of specified requirements, addressing not just form and appearance, but also intentions and semantics.

For large and complex systems, purely manual verification is ineffective and insufficient: much like for software, one rarely, if ever, detects all defects just by inspecting design documentation and source code or models. Residual defects in requirements specification are then revealed late in system life cycle, often during validation, commissioning, or even worse, during operation.

Advanced tool-supported verification with simulation or formal verification needs requirements to be formally specified, i.e., modelled in languages (such as FORM-L) with rigorously defined syntax and semantics that are also understandable by all concerned persons with minimal training.

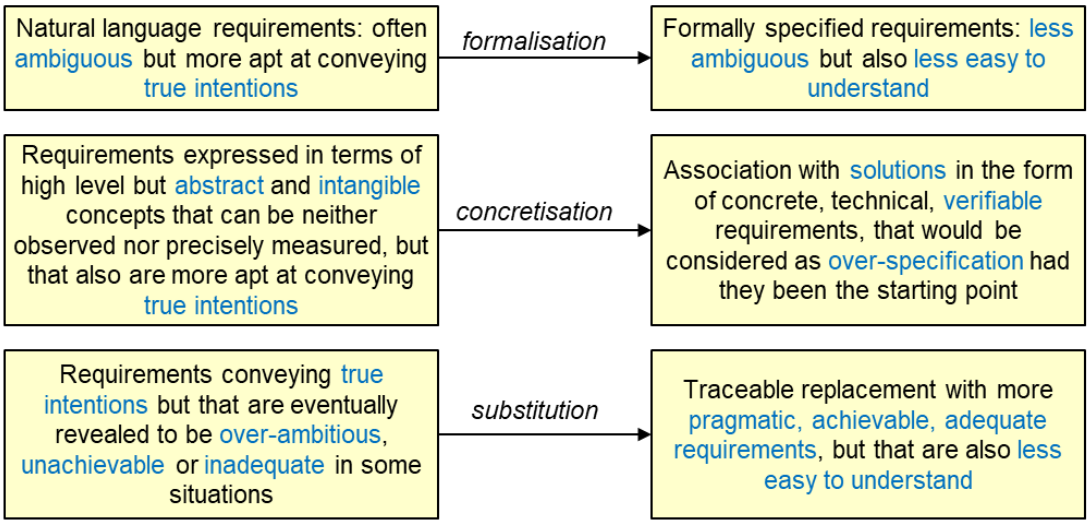


Figure 3-3
It is sometimes worthwhile to begin the requirements engineering process with imperfect but informative requirements.

Figure 3-4 shows as an example how the defects of requirement **repower** from the *BPS* case study presented in Section 4 are gradually corrected.

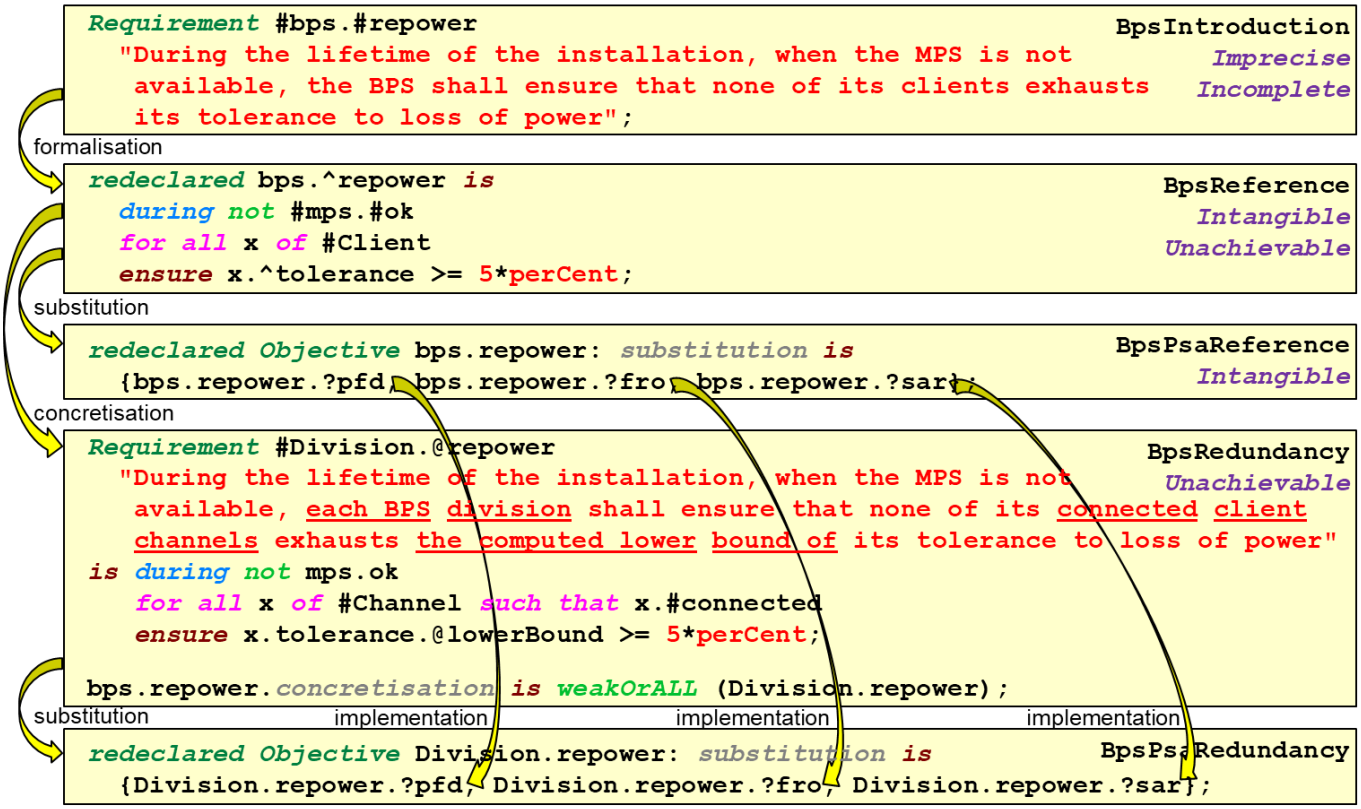


Figure 3-4
Each box contains an excerpt from a specific model named in the upper-right corner of the box. The excerpts follow the gradual improvement of requirement **repower**. *Just below the model name are the defects that affect the requirement specification within the box.*

4 The *BPS* Example

To make the BASAALT approach and its principles more concrete, it is illustrated with a practical example, the *Backup electric Power Supply* (*BPS*) system.

Note. This was the first case study in the development of BASAALT and FORM-L. The initial system of interest was the computer-based control system of the *BPS*, called the *Load Sequencer* (*LS*): the objective was to make sure that the functional and timing requirements for the *LS* were adequate and complete with respect to the requirements, design and operational conditions of its parent system, the *BPS*. However, it became rapidly clear that to do so, a deep understanding of the human and physical aspects of the *BPS* was necessary: the *BPS* then became the primary system of interest, and BASAALT and FORM-L were both extended to address the engineering of socio-technical and cyber-physical systems.

4.1 Initial Conceptual Studies, Semi-Formal Modelling

During initial studies, stakeholders try to make their mind about, and agree on, the reasons of being, the boundaries, the nature and the top-level objectives of a system of interest, mostly in a non-formal or semi-formal manner where the objects involved are identified and assumptions, objectives and requirements are expressed in natural language and informal diagrams.

The *BPS* is part of an overall installation (e.g., a power plant, a hospital or a factory). The example starts with the *BPS* though in theory, systems engineering should start with the installation, since the engineering of the *BPS* is performed in the framework and as a part of the engineering of the installation.

That simply shows that systems engineering in general, and BASAALT and FORM-L in particular, may be applied selectively on parts of an overall system. This particularly important when introducing the approach in an organisation or a project: one may try them on limited, critical parts before a more general application.

The installation will be in continuous operation for several decades. It has three kinds of electric panels:

- Uninterruptible electric panels power mission critical systems that must never lose electric power under pain of dire consequences.
- Interruptible backed-up electric panels power mission critical systems that can tolerate absence of electric power, but only for limited time. As these are also essential for preventing unacceptable consequences, they must be repowered before their tolerance is completely consumed.
- Non-backed-up electric panels power non-essential systems that may lose electric power without serious consequences.

The main objective of the *BPS* is, in case of loss of the *Main electric Power Supply* (the *MPS*), to provide electric power, in the megawatt range and for days, weeks or months if necessary, to an interruptible backed-up electric panel, to which are connected a number of other systems of the installation (the *clients* of the *BPS*).

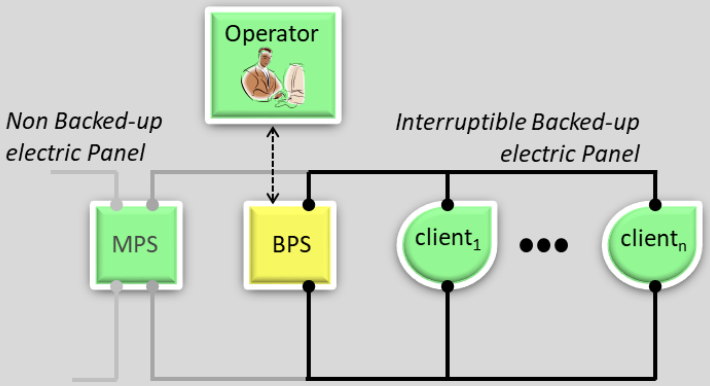


Figure 4-1
The BPS as a black box in its environment.

Figure 4-1 places the *BPS* within its operational environment: the *MPS*, a human operator, and its clients (*client*$_1$ to *client*$_n$). Clients identity and precise number will be known only at a later stage of the engineering of the installation. Each client will have specific expectations with respect to the *BPS* (in particular, the electric power it needs and the maximum delay before its tolerance is completely consumed) but these are also not known yet at this very preliminary stage.

Initial studies identified three top-level functional requirements for the *BPS*:

- During the lifetime of the installation, when the *MPS* is not available, the *BPS* shall ensure that none of its clients exhausts its tolerance to loss of power.
- While the *BPS* powers its clients, at operator's request and if the *MPS* is available, the *BPS* shall ensure that all its clients are again powered by the *MPS*.
- When the *BPS* detects failures that could prevent it from satisfying the two previous requirements, it shall send an alarm to the operator.

The BASAALT approach starts by capturing these in a first introductory, semi-formal FORM-L model.

```
Model BpsIntroduction "Initial, semi-formal model for the BPS" begin
  external Object #mps      "Main electric Power Supply";
  external Object #operator "Human operator of the installation";
  external Class  #Client   "Any system powered by the backed-up panel";

  main Object #bps "Backup electric Power Supply" begin
    Requirement #repower
     "During the lifetime of the installation, when the MPS is not
      available, the BPS shall ensure that none of its clients exhausts
      its tolerance to loss of power";

    Requirement #reset
     "When the BPS powers its clients, at operator's request and
      if the MPS is available, the BPS shall ensure that all
      clients are again powered by the MPS";

    Requirement #alarm
     "When the BPS detects failures that could prevent it from satisfying
      'repower' or 'reset', it shall send an alarm to the operator";
  end bps;
end BpsIntroduction;
```

As one can see, each entity of Figure 4-1 is represented in this first model:

- The *BPS*, the *MPS* and the operator are each represented by an ***Object***.
- The *BPS* clients are represented by the instances of class ***Client***. No *BPS* clients are declared yet because their identity and number are still unknown.

The **bps** object is declared as the ***main*** object: this formally identifies the *BPS* (which it represents) as the system of interest of the model.

The **mps** and **operator** objects and class **Client** are declared ***external***: this means that they do not represent and are not part of the system of interest, and that what follows needs and can and will include only what concerns the system of interest.

The three top-level functional requirements assigned to the *BPS* by initial studies are modelled as FORM-L ***Requirements*** attached to **bps** and named respectively **repower**, **reset** and **alarm**. Attaching a *property* (requirements being a particular class of property) to an object means that this object is responsible for that property.

Each modelling item may have an attached natural language description. At this stage, all items are just declared with their natural language description. If that description is stored in a database (e.g., an engineering or a requirements database), it may be obtained through an appropriate reference link (not shown here) instead of through copy-and-paste.

The **#** signs are *determiners* indicating that what they are attached to represents and models actual, physical aspects of the real world. Thus, **bps**, **mps**, **operator** and **Client** represent actual, physical entities of the real world, and **repower**, **reset** and **alarm** are requirements on actual, physical behaviours. (See Section 7.3 for more information on determiners.)

4.2 Tool-Aided Verification, Formal Models

At this very preliminary stage, verification that the model is adequate and represents the intentions of its authors must rely on classical techniques such as reviews and inspections. This should be relatively easy since the model is small and simple (18 lines). However, BASAALT aims at supporting the complete system life cycle: as engineering progresses, more information and details will be added, to

the point that for systems worthy of the name, these essentially manual techniques become prohibitively labour-intensive, time consuming, difficult, expensive and ineffective. Tool-aided verification would be a precious help, but as the model is mostly semi-formal, that help is necessarily very limited.

In-depth tool-aided model verification can be based on two main principles:

- *Formal analysis* examines ALL possible *legitimate operational cases* (i.e., all cases consistent with the specified assumptions and definitions of the model) to determine whether a solution complies with specified requirements. This is the most powerful verification method, but even with appropriate models, it is not always applicable due to theoretical and practical limits of analysis methods and tools. It is also not applicable to the very first model since there is no other model that could serve as a reference.
- *Simulation* examines individual operational cases, one by one. To reach a high level of confidence that a model is adequate or that a solution complies with specified requirements, large numbers of legitimate cases are usually necessary. Tool support consists in the automatic generation of legitimate cases of interest, in the automatic computation for each examined case of system behaviour and of its effects on its environment, and in the verification that in each case system behaviour and effects on the environment are as expected and comply with all requirements.

Tools supporting these principles need models with rigorously defined syntax and semantics, i.e., *formal models*. In particular, to automatically verify that a behaviour complies with requirements, the latter need to be specified formally.

Also, one needs to formally specify the criteria determining whether a case is legitimate: in BASAALT and FORM-L, this is done with *assumptions*. Assumptions are as essential to BASAALT as requirements. Whereas requirements usually apply to the system of interest, assumptions often apply to its environment or to contemplated solutions. The adequacy of the requirements specified for the system of interest can be meaningfully assessed only in the framework of the assumptions made on its environment: incorrect assumptions on the environment could lead to inadequate system requirements. Lastly, assumptions placed by the system of interest on some if its components, or on a technical system or human agent in its environment, must be considered by that component, technical system or human agent as a requirement: thus, assumptions and requirements are the two faces of the same coin.

It is important that formal models correctly represent the intentions of their authors and that they can be inspected and verified by concerned stakeholders and engineers, not all of whom are modelling or formal methods experts. Thus, formal models must not only have precise and unambiguous syntax and semantics, they must also be legible and understandable with minimal training to whoever participates in the engineering process. To that end, FORM-L has a syntax close to natural language. It is presented here in an English variant, but variants close to other natural languages could be defined so that models authors can use the variant of their choice and that a model written in one variant can be automatically translated into another variant.

Formalisation of the **`BpsIntroduction`** model may be done either directly within the model or in an *extension model*. Choosing one or the other option is mainly a matter of taste. Here, it is done in an extension model named **`BpsReference1`**. An extension model contains all the information of the models it extends, and then provides additional, consistent information (i.e., not contradicting information in the models being extended).

```
Model BpsReference1 "First, imperfect, top-level formal model for the BPS"
  extends BpsIntroduction
begin
  Library  Standard  "Commonly used quantities, units and functions";

  Duration !lifetime "Operational life time of installation" is 60*year;
  when t0 + lifetime signal eoc;  // Nothing happens after end of lifetime

  refined bps;
  refined mps;
  refined operator;
  refined Client;
end BpsReference1;
```

For convenience and to facilitate legibility and understanding, a FORM-L model may be organised as a series of *statements*. This first statement is a *declaration* of the **`BpsReference1`** model that provides an overview *definition* of the model, in the form of *embedded statements*. The first embedded statement declares the **`Standard`** library, which defines the quantities of the SI international system of units [12]. This way, physical quantities are not represented by mere dimensionless numbers, and one is prevented

from adding a length to a duration. **Standard** also defines constants for commonly used units, so that when one adds **1*foot** to **1*meter**, one does not obtain **2*foot** or **2*meter**, but **1.3048*meter** or **4.28084*foot**.

The second embedded statement defines the lifetime of the installation, which is represented by a ***Duration*** (a quantity class defined by the **Standard** library) named **lifetime**, the value of which is 60 **year** (**year** is a constant of class ***Duration*** and is also defined by the **Standard** library). The **!** sign attached to **lifetime** is a determiner indicating that it is or will be completely pre-determined during design, i.e., it will have exactly the same behaviour in all operational cases.

Each operational case has two particular *instants*: **t0** is the instant before which nothing happens (i.e., no changes to any value and no occurrence of any event), and **eoc** (for *end of case*) is the instant after which nothing new happens. **t0** is modelled as a predefined *event* with a single *occurrence*, and cannot be modified by a model. **eoc** is modelled as a predeclared[2] event, the single occurrence of which may be explicitly signalled by a model as is the case here with the third embedded statement (***when*** **t0 + lifetime** ***signal*** **eoc**).

As **BpsReference1** will provide definitions for **mps**, **operator**, **Client** and **bps** (which were just declared by **BpsIntroduction**), it announces a *refinement* for each of them (e.g., ***refined*** **bps**). Such announcements are informative and not mandatory.

```
mps begin
  AcVoltage #voltage "Electric tension at the MPS terminals" begin
    value is deferred;
    Assumption range is ensure value in [0, 260]*V;
  end voltage;
end mps;
```

This statement is a definition block that refines the **mps** object. It specifies that **mps** has a state variable named **voltage** and of class ***AcVoltage***. The **#** determiner means that **voltage** represents and models an actual and measurable *physical state*, not an *information* like what a sensor would produce.

deferred means that the ***value*** of **voltage** will be defined later in the engineering process, in some future extension model. However, **voltage** is assumed to always be in the 0-260 volts range (***Assumption*** **range**).

This example shows that in FORM-L, one can specify precise values (as in the case of **lifetime**), but also constraints on values (as in the case of **voltage**). In this sense, FORM-L models may be but are not necessarily *deterministic*. Most dynamic phenomena models like finite element models (for physics), Modelica [1] models (for multi-physics) or Scade [16] models (for controls) are fully deterministic: given initial and boundary conditions, they determine a single, unique behaviour. FORM-L models may be, and most usually are, *constraint* models specifying *envelopes* of possible behaviours (see Figure 4-2).

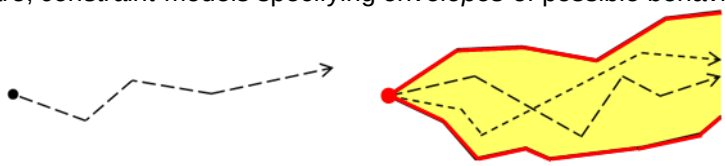


Figure 4-2
Modelling of a specific behaviour vs modelling of an envelope *of expected or acceptable behaviours.*

This is particularly useful in the initial stages of a system life cycle, where one needs to model the full range of external conditions the system may face, and also to avoid over-specification. When looking for optimal solutions, over-specification often means that better solutions are likely to be overlooked. Also, the specification of solutions is often more complex (and thus more difficult to verify and more error-prone) than the specification of the problem itself, and it may happen that a specified solution does not fully solve the real but unstated problem.

The notion of envelope is also important when dealing with the *uncertainties* and the *unknowns* inherent to early conceptual studies, human behaviour and physical systems (due to their analogue nature, noise and random failures).

[2] In FORM-L, a declaration states the existence and nature of an item, whereas a definition specifies its value, behaviour or contents.

It applies to values and timings, but also when a property (e.g., an assumption, objective or requirement) needs to be expressed with respect to information, components or subsystems that are not precisely known yet.

For example, one might require that "*the BPS shall perform its missions even in the presence of any single failed component* " at a time when its composition in terms of components is unknown.

FORM-L has the notion of *set* to represent parts of the system that are not known in enough detail yet to be listed in *extension* (i.e., by individually naming each of their members), and that are just defined in *intention*.

For example, in the scope of an expression, **`Client`** is the set of all named instances of class ***`Client`***.

```
operator begin
  Event @eReset "Reset command" is deferred;
end operator;
```

The **`operator`** refinement models the reset commands issued by the human operator with an ***`Event`*** named **`eReset`**. The **`@`** mark is another determiner indicating that **`eReset`** represents a *control information* (not an actual state of the physical system). It also implies that **`operator`** represents not just a human agent but also the human-system interface device or system that transforms actual human actions (which would have a **`#`** determiner) into control information: the modelling implicitly shows that that device or system is not considered a part of the *BPS*.

The distinction brought by **`#`** and **`@`** is important: when addressing cyber-physical systems, one needs to take account of the fact that sensors (which transform **`#`** objects into **`@`** objects) and actuators (which transform **`@`** objects into **`#`** objects) are not perfect and introduce discrepancies and uncertainties due to limited accuracy, response time, miscalibration, drift and failure.

```
Client begin
  common constant AcVoltage !minimumVoltage
   "Electric tension above which all clients are fully operational and do
    not consume their tolerance level, and below which they may consider
    they are not powered"
    is deferred;

  Boolean #powered
    "Power is always provided through the BPS.
     It may be actually produced either by the MPS or by the BPS";

  Percentage ^tolerance "Instantaneous level of tolerance to loss of power"
  begin
    value is deferred;
    Assumption range is ensure value in [0, 100]*perCent;
    Assumption replenish
      "When powered, a client shall replenish its tolerance level"
      is during powered
         ensure derivative >= 1*perCent/mn or value > 99*perCent;
  end tolerance;
end Client;
```

Constant **`minimumVoltage`** is the electric tension above which it is assumed that clients can operate as required and do not consume their tolerance level, and on the contrary can replenish it. Since its value is the same for all clients, it is declared ***`common`***. The **`!`** determiner means that although it is **`deferred`** for now, it must and will be fully determined during design and will not depend on operation.

Each instance of class **`Client`** has a state variable named **`powered`** indicating whether it receives electric power, and another one named **`tolerance`** representing its instantaneous tolerance to loss of power. As it is expressed as a ***`Percentage`*** of the maximum tolerance level, the ***`value`*** of **`tolerance`** is assumed to be in range 0-100% (***`Assumption`*** **`range`**). It is also assumed to be replenished when **`powered`** is true (***`Assumption`*** **`replenish`**). Its **`^`** determiner means that it represents an abstract, *conceptual state* that is not observable, measurable and computable: it may only be used to define other conceptual objects, in particular conceptual properties.

In the definition of **`replenish`**, one can note that *attributes* ***`derivative`*** and ***`value`*** are not those of **`replenish`** (which, being a *property*, does not have such attributes) but of **`tolerance`**.

Eventually, conceptual properties will need to be *concretised*, i.e., each will need to be associated, through their `concretisation` attribute, with a *stronger* concrete property (with an **#** or **@** determiner). A property is in one of three possible states: `untested`, `satisfaction` or `violation`. *Stronger* means that when the concrete property is in `satisfaction`, the conceptual one must also be. Simulation or analysis will issue a warning when that is not the case. The transpose is not necessarily true: the `violation` of the concrete property does not necessarily mean the `violation` of the conceptual one.

```
bps begin
  // ---- repower ------------------------------------------- repower ----
  mps: Boolean #ok is voltage >= Client.minimumVoltage;

  redeclared ^repower "Clients' tolerance shall never fall below 5%" is
    during not mps.ok                    // Temporal locator
    for all x of Client                  // Selector
    ensure x.tolerance >= 5*perCent;     // Constraint
```

Here, **ok** is a feature of object **mps**, but for convenience and because it serves only the purposes of object **bps**, its declaration and definition are embedded within **bps**. Its declaration specifies its determiner explicitly, though it could have been automatically inferred from its definition: as **mps.voltage** has a **#** determiner and **Client.minimumVoltage** has a **!** determiner, **mps.ok** must have a **#** determiner.

When that makes sense, it is generally preferable to specify determiners explicitly at least once, as they are an indication of what model authors have in mind. Static analysis tools can then check whether each explicitly specified determiner is consistent with the one inferred automatically, and thus help reveal modelling errors.

The formalisation of **repower** shows the three parts of a formal property definition:

- The third part (`ensure x.tolerance >= 5*perCent`) is the *constraint* to be satisfied. Keyword `ensure` indicates that it is an *invariant constraint*: such constraints need to be satisfied all along the specified temporal locator.
- The first part (`during not mps.ok`) is a *temporal locator* specifying when the constraint applies. Temporal locators are optional: absence means `always`.
- The second part (`for all x of Client`) is a *selector*. Selectors are used when *sets* are involved and specify which sets and set members are concerned with the constraint (here, all the instances of class **Client**).

As it adds a point not in the initial natural language description (the 5% margin), it also provides a natural language complement (`"Clients' tolerance shall never fall below 5%"`).

The initial declaration of **repower** in **BpsIntroduction** associated it with a **#** determiner, indicating that it must be satisfied in the real, physical world. However, when formalising it, one realises that it intrinsically depends on **tolerance**, which is an abstract object. Thus, **repower** can no longer be considered as a physical requirement, and `redeclared` **^repower** (which modifies the initial declaration within the limits authorised by FORM-L) acknowledges that it is also abstract and intangible.

Similarly, the determiner of **repower** can be inferred from its definition: **mps.ok** and set **Client** both have a **#** determiner, but as **x.tolerance** has a **^** determiner, **repower** must have a **^** determiner. Had **repower** not been redeclared, or had `redeclared` **@repower** been specified, a verification tool would signal the discrepancy.

When a property applies to a set as is the case of **repower**, it is first evaluated separately for each selected member of the set. Then, with `for all`, it is in `violation` if it is for any selected member, in `satisfaction` if it is for all selected members (there must be at least one), and is `untested` otherwise. With `for some`, it is in `satisfaction` if it is for any selected member, in `violation` if it is for all selected members (again, there must be at least one), and is `untested` otherwise.

```
  // ---- reset ------------------------------------------- reset ----
  Client begin
    Boolean #poweredByBps
      "Whether the client receives power produced by the BPS" is deferred;
```

```
    Boolean #powered
      "Whether the client receives power (always through the BPS).
       Power may be produced either by the MPS or the BPS":
        during poweredByBps define value is true
      otherwise
        during not mps.ok and not poweredByBps define value is false
      otherwise value is deferred;
  end Client;

  Event @eVReset "Reset commands are valid only if the MPS is available" is
    operator.eReset while mps.ok;

  #reset
    "Clients should be powered by the MPS within 10 s after
     a valid reset, unless the MPS is lost again"
  is after eVReset within 10*s
     for all x of Client
     achieve (x.powered and not x.poweredByBps) or not mps.ok;
```

The formalisation of **reset** introduces two Boolean features for class **Client**: one that is **true** when and only when the client is actually powered (**#powered**), and one that is **true** when and only when the client is actually powered with power produced by the *BPS* (**#poweredByBps**).

Keyword ***achieve*** indicates that the constraint is an *achievement constraint*. Such a constraint is not required to be satisfied all along the specified temporal locator: it may not be satisfied at the beginning but needs to become so at some point during the temporal locator.

Since **eVReset** has an **@** determiner, and **Client.powered** and **Client.poweredByBps** both have a **#** determiner, there is no natural implicit choice of determiner for **reset**. Here, a **#** determiner was explicitly specified to indicate that each client must be actually powered by the *MPS*.

```
  // ---- alarm ------------------------------------------------ alarm ----
  Event @eFailure "When failure is detected"              is deferred;
  Event @eAlarm   "Signal sent to operator when eFailure" is deferred;
  alarm "with a 200 ms time margin" is
    after eFailure within 200*ms
    achieve eAlarm;
end bps;
```

The **@** determiners indicate that **eFailure** and **eAlarm** represent control information. For **eFailure**, that means that occurrences do not represent actual failures but detection (whether correct or spurious) of failures. Had the natural language description of **alarm** specified **"When failures could prevent ..."** instead of **"When the BPS detects failures that could prevent ..."**, then **eFailure** would have been given a **#** determiner.

For **eAlarm**, the **@** determiner means that **bps** does not include the human-system interface device or system that transforms occurrences into actual signals perceptible by a human being.

One can note the introduction of elements not in the initial natural language descriptions: the notion of minimum voltage for clients, the 5% margin on tolerance levels for **repower**, the 10 second grace time for **reset**, the 200-millisecond grace time for **alarm**. Indeed, these are necessary, for no system can react instantly and has perfect precision. Also, the formal refinement of **reset** takes account of the fact that the *MPS* could be lost again during the time necessary to switch the clients back to the *MPS*.

4.3 Engineered Interfaces, Contracts

Even if it correctly reflects its authors intentions regarding the *BPS*, the **BpsReference1** model as shown in the previous section is not completely satisfactory from a BASAALT perspective. When dealing with systems interacting with various elements of their environment or being composed of multiple subsystems, it is essential to have well-identified and clearly specified interfaces between objects. In BASAALT and FORM-L, *engineered interfaces* (i.e., desired, designed interfaces, as opposed to undesired *encroachments* caused by exceptional conditions, failures and accidents, but also side effects during normal operation, which will be seen later) are specified with *contracts*.

Contracts are an essential BASAALT ingredient. They are used to coordinate, just as necessary, the many individuals, teams, disciplines, organisations and stakeholders contributing to the engineering of

the system. They are also used as a means for abstraction and focus, and thus for complexity management. Here is now a more satisfactory version of the model, named **BpsReference2**.

```
Model BpsReference2
  "Improved, but still imperfect, top-level formal model for the BPS"
  extends BpsIntroduction
begin
  Library  Standard  "Commonly used quantities, units and functions";

  Duration !lifetime "Operational life time of installation" is 60*year;
  when t0 + lifetime signal eoc;  // Nothing happens after end of lifetime

  refined  bps;

  Contract mainPower    (mps);
  Contract hsi          (bps, operator);
  Contract backupPower  (bps, Client);
end BpsReference2;
```

Now, the model no longer refines **mps**, **operator** and **Client**: instead, it declares and then defines the contracts they have with **bps**. That is a typical modelling pattern for external objects and classes: they may be defined solely by their contracts with the object or class of interest.

A contract declaration lists its *parties*, i.e., the objects or classes involved. Contract **mainPower** has object **mps** as its only party: that means **mps** offers the interface to all the other objects and classes of the model, provided they *acknowledge* it. Contract **hsi** (the human-system interface) has two parties: object **bps** and object **operator**. Contract **backupPower** has two parties, object **bps** and class ***Client***: that means that **bps** has that contract with each instance of ***Client***.

```
mainPower begin
  party mps begin
    AcVoltage #voltage "Electric tension at the MPS terminals" begin
      value is deferred;
      Guarantee range is ensure value in [0, 260]*V;
    end voltage;
  end mps;
end mainPower;
```

A contract definition specifies the *obligations* of each party, i.e., the *deliverables* it owes to the other parties, and the *guarantees* it ensures on these deliverables. The obligations of **mps** as per the **mainPower** contract are those specified in the first version of the model. Here, the model author has chosen to not just declare the obligations of **mps**, but to also to define them.

voltage.range is no longer an ***Assumption*** but a ***Guarantee***. A ***Guarantee***, like an ***Assumption*** or a ***Requirement***, is a particular class of FORM-L ***Property***: it is ensured by one party (here, **mps**) and can be considered by the other parties as an ***Assumption***.

This is a common and convenient modelling pattern in FORM-L: first, some objects (other than the one representing the system of interest) guarantee specific behavioural envelopes. Analysis or simulation tools treat these guarantees as assumptions and help verify whether operational cases consistent with these envelopes satisfy the requirements placed on the system of interest. If so, one can proceed a step further, where a solution for the objects bearing the guarantees is designed: that solution now considers (and redefines) the guarantees as requirements.

```
hsi begin
  party operator begin
    Event @eReset "Reset command" is deferred;
  end operator;

  party bps begin
    Event @eAlarm "Signal sent to operator when a failure is detected";
    Requirement @alarm;
  end bps;
end hsi;
```

The **hsi** contract has two parties, each with a dedicated section. The **operator** section states that it owes **bps** a single deliverable, **eReset**, as was specified in the first version of the model. Since **operator** is external and is not refined, **eReset** is not just declared, it is also defined (***is deferred***).

The **bps** section states that it owes **operator** deliverable **eAlarm**. **bps** also informs **operator** that it will ensure requirement **alarm**. As **bps** is refined by the model, whether its obligations are defined in the contract or in the **bps** refinement is a matter of taste. Here, they are just declared in the contract and are defined in the **bps** refinement.

```
backupPower begin
  party Client begin
    common constant AcVoltage !minimumVoltage
     "Electric tension above which all clients are fully operational and
      do not consume their tolerance level, and below which they may
      consider they are not powered"
      is deferred;

    Percentage ^tolerance
     "Instantaneous level of tolerance to loss of power"
    begin
      value is deferred;
      Guarantee range is ensure value in [0, 100]*perCent;
      Guarantee replenish
        "When powered, a client shall replenish its tolerance level"
        is during powered
           ensure derivative >= 1*perCent/mn or value > 99*perCent;
    end tolerance;
  end Client;

  party bps begin
    Client: Boolean #powered
      "Power is always provided through the BPS.
       It may be actually produced either by the MPS or by the BPS";
    Requirement ^repower;
    Requirement #reset;
  end bps;
end backupPower;
```

The **backupPower** contract is between **bps** and each instance of **Client**. The obligations are those specified in the first version of the model.

```
bps begin
  // Acknowledgment of contracts
  Contract mainPower (mps);
  Contract backupPower (me, Client);
  Contract hsi (me, operator);

  // ---- repower ------------------------------------------- repower ----
  mps: Boolean #ok is voltage >= Client.minimumVoltage;

  redeclared ^repower "Clients' tolerance shall never fall below 5%" is
    during not mps.ok
    for all x of Client
    ensure x.tolerance >= 5*perCent;

  // ---- reset --------------------------------------------- reset ----
  Client begin
    Boolean #poweredByBps
     "Whether the client has power actually produced by the BPS"
      is deferred;

    Boolean #powered
     "whiter the client has power (always through the BPS).
      Power may be actually produced either by the MPS or by the BPS":
        during poweredByBps define value is true
      otherwise
```

```
        during not mps.ok and not poweredByBps define value is false
      otherwise value is deferred;
  end Client;

  Event @eVReset "Reset commands are valid only if the MPS is available" is
    operator.eReset while mps.ok;

  reset
   "Clients should be powered by the MPS within 10 s after
    a valid reset, unless the MPS is lost again"
  is after eVReset within 10*s
     for all x of Client
     achieve (x.powered and not x.poweredByBps) or not mps.ok;

  // ---- alarm ------------------------------------------------ alarm ----
  Event @eFailure "When failure is detected" is deferred;
  eAlarm is deferred;
  alarm "with a 200 ms time margin" is
    after eFailure within 200*ms
    achieve eAlarm;
end bps;
```

This refinement of **bps** is very similar to the first version. The main difference is that **bps** needs now to *acknowledge* (i.e., declare) the contracts to which it is a party to make sure that model authors are aware of them and have accepted them.

One can see here another case where an item (object **bps**) provides a definition block for another one (class **Client**, with the declaration and definition of features **powered** and **poweredByBps**).

4.4 Top-Level Formal Models, Reference Models

BpsReference2 is a fully formal, tool-friendly model that can be simulated, with manually defined or with automatically generated operational cases. As manual case definition is tedious, cumbersome and labour-intensive, automatic generation (e.g., with tools like StimuLus) is often the preferred approach. For generation tools to produce *legitimate* but also *useful* cases of interest, one generally needs to provide a *scenario* model as will be seen in the next section. However, for **BpsReference2**, no other tool-friendly, higher-level model is available: the behaviour and outcome of each simulated case need to be assessed manually. Thus, it is preferable that this top-level formal model be as simple as possible and concentrates on what is absolutely essential (i.e., the reason of being of the system of interest), leaving less important aspects to subsequent engineering phases and models.

Other requirements could have been specified in **BpsIntroduction** and **BpsReference2**, such as the single failure criterion or freedom from spurious activation, but as they were deemed of lesser importance, they will be introduced later in the engineering process, in some extension model.

Formal extension models developed in subsequent engineering phases can be verified with full tool support against the assumptions and requirements of the formal models they extend, including against those of the top-level formal model: this is why it is called the *reference model*.

The correctness and adequacy of the reference model is of extreme importance since it will be the touchstone for all subsequent engineering decisions. Here are a few BASAALT recommendations:

- To focus attention on needs to be satisfied and on objectives and requirements rather than on means and solutions, the reference model should consider the system of interest as a black box and should not make or imply any unnecessary choice of solution, including in terms of technical requirements.
- It should place the system within its environment: external entities that interact with it or have an influence or expectations on it should appear in the model. This could include other technical systems, passive structures, human agents or the physical environment (providing ambient and / or seismic conditions for example). To that end, model authors need to have a clear understanding of the boundaries of the system. In the identification of these external entities, it is again important to consider the system as a black box. Later, as engineering progresses, the environment may be gradually extended, since particular solutions may involve or be affected by additional external entities.

- The reference model should identify the different *situations* the system may face. Situations may arise from system states (normal or abnormal), from the states of environment entities (again, normal or abnormal) and from the operational goals assigned to the system at any given instant, and which may vary in time. This is particularly important, since some requirements or assumptions may be appropriate in some situations, but not in others or at some transitions between situations.

Real, catastrophic accidents have been caused by requirements that were wholly inappropriate in situations not foreseen or not taken into full consideration. The *Cranbrook Manoeuvre* [20] is one such accident. It occurred in February 1978 at Cranbrook International Airport, British Columbia, Canada.

- Spurious deployment of aircraft thrust reversers during flight having previously caused several catastrophic accidents, a requirement was specified whereby reversers must be de-energised when the aircraft is airborne.
- Due to very light traffic load, Cranbrook airport was an uncontrolled airport, which means air traffic control was done remotely from Calgary.
- Up to one meter of snow had fallen in Cranbrook, and more was still coming down. When an airliner announced Calgary its impending arrival, a snowplough was sent to clear the runway.
- Having taken a shorter route than expected, the aircraft landed earlier than estimated by air traffic control. When the wheels touched ground, the reversers were deployed. A few instants later, while still decelerating, the pilots saw the snowplough on the runway: they had not seen it at first due to poor visibility conditions.
- They immediately ordered the stowing of the reversers, pushed the throttles to maximum power and took off. When the wheels left ground, one reverser was fully stowed, but not the other, and because it was now de-energised, stowing could not be completed.
- Though the aircraft managed to clear the snowplough, aerodynamic pressure redeployed the reverser. As the pilots did not have time to perform all what was necessary, the aircraft crashed, killing 42 of the 49 people on board.

The accident was caused by a combination of factors, but one of them was that although the requirement regarding reversers de-energising during flight was appropriate in most situations, it was woefully inadequate in that one, where the aircraft was configured for landing and pilots abruptly changed the operational goal from *landing* to *take off*.

- Then the reference model should identify the flow of deliverables between the system and its environment entities, possibly depending on situation. That needs to be done very carefully: even when the system of interest is considered as a black box, there is a potential for over-specification if one identifies deliverables that represent technical solutions rather than the reason of being of the system. To avoid this, one may need to rely on conceptual deliverables.

For the *BPS*, one top-level state variable is the **`^tolerance`** of each client to loss of power.

Similarly, as the reason of being of a cooling system is to ensure that its 'clients' are maintained in a given temperature range, a top-level state variable would be the temperature of each client, not the features of a cooling fluid (which will appear at a later engineering stage as a technical solution).

- Lastly, the reference model should specify the guarantees and requirements regarding these deliverables, also possibly depending on situation.

4.5 Generic Scenario Models

Even though **`BpsReference2`** is fully formal, it cannot be meaningfully simulated yet since key information is still missing:

- No clients have been declared and defined.
- Several variables (**`mps.voltage`**, **`operator.eReset`**, **`Client.minimumVoltage`**, **`Client.tolerance`**, **`Client.powered`**, **`Client.poweredByBps`**) and events (**`bps.eFailure`**, **`bps.eAlarm`**) are **`deferred`** and do not have an operational definition. A random case generator would produce completely unconstrained sequences, which are unlikely to constitute cases of interest, and no useful conclusions could be drawn from them.

BASAALT addresses this issue with a *scenario model* (see *Figure 4-3*).

```
Model BpsReferenceScenario extends BpsReference2 begin
  refined mps.voltage;
  refined operator.eReset;
  refined Client.minimumVoltage is 190*V;
  Client  client1;
  Client  client2;
  refined bps;
end BpsReferenceScenario;
```

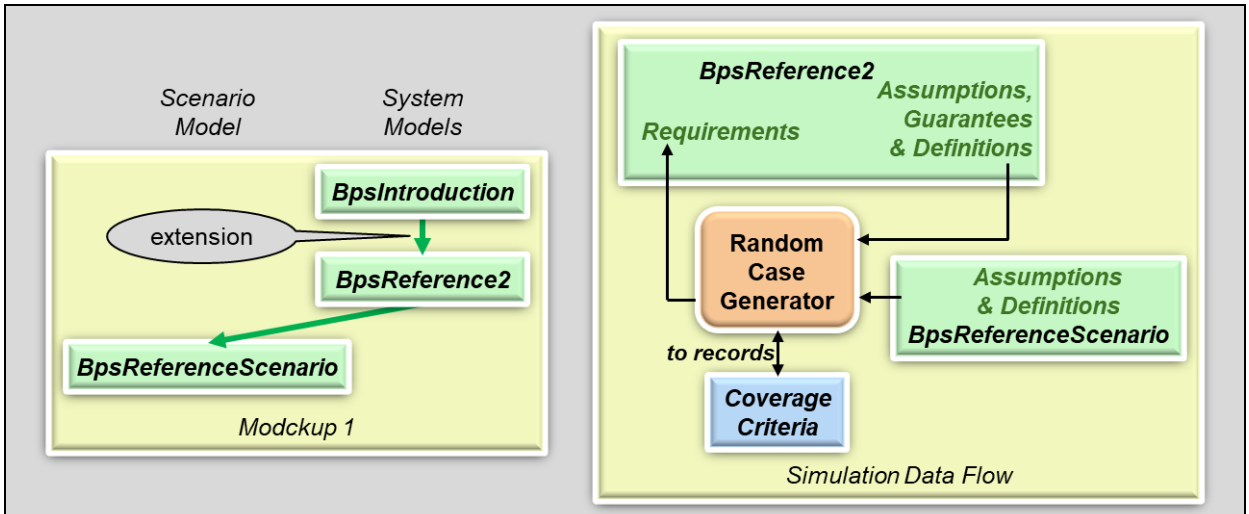


Figure 4-3
Mock-up and simulation data flow for **`BpsReference2`**.

```
mps.voltage begin
  private Event eMicro "Micro cuts or false hopes": rate is 3/mn;

  from t0 repeat sequence
    during random (1, 10)*mn Assumption is
      from eMicro during random (0.1, 4)*s  // Micro cut
        ensure value in [  0, 260]*V and derivative in [-1000, 1000]*V/s
      otherwise                             // Normal tension
        ensure value in [210, 260]*V and derivative in [  -10,   10]*V/s;

    during random (0.1, 10)*s Assumption is // Transition
        ensure value in [  0, 260]*V and derivative in [-1000, 1000]*V/s;

    during random (5, 60)*mn Assumption is
      from eMicro during random (0.1, 4)*s  // False hope
        ensure value in [  0, 260]*V and derivative in [-1000, 1000]*V/s
      otherwise                             // Absence of tension
        ensure value in [  0, 150]*V and derivative in [ -100,  100]*V/s;
  end sequence;
end mps.voltage;
```

Completely random generation for **`mps.voltage`** would produce noise of 260 volts amplitude, which would be useless. The ***`repeat sequence`*** iteratively generates the envelope shown in *Figure 4-4*, where one considers that **`mps.voltage`** is in one of three possible states of highly variable duration:

- When tension is *normal*, its *`value`* is between 210 and 260 volts and is continuous (its *`derivative`* is limited to a narrow range), but it may be subject to micro cuts.
- During *absence of tension*, its *`value`* is below 150 volts with possibly significant variations (its *`derivative`* is in a wider range), but it may be subject to false hopes.
- During *transitions*, micro cuts or false hopes, its *`value`* may take the full range allowed by **`BpsReference2`** (i.e., 0 to 260 volts) and is subject to brutal variations (its *`derivative`* is in very wide range).

As one can see in this example, assumptions may be unnamed.

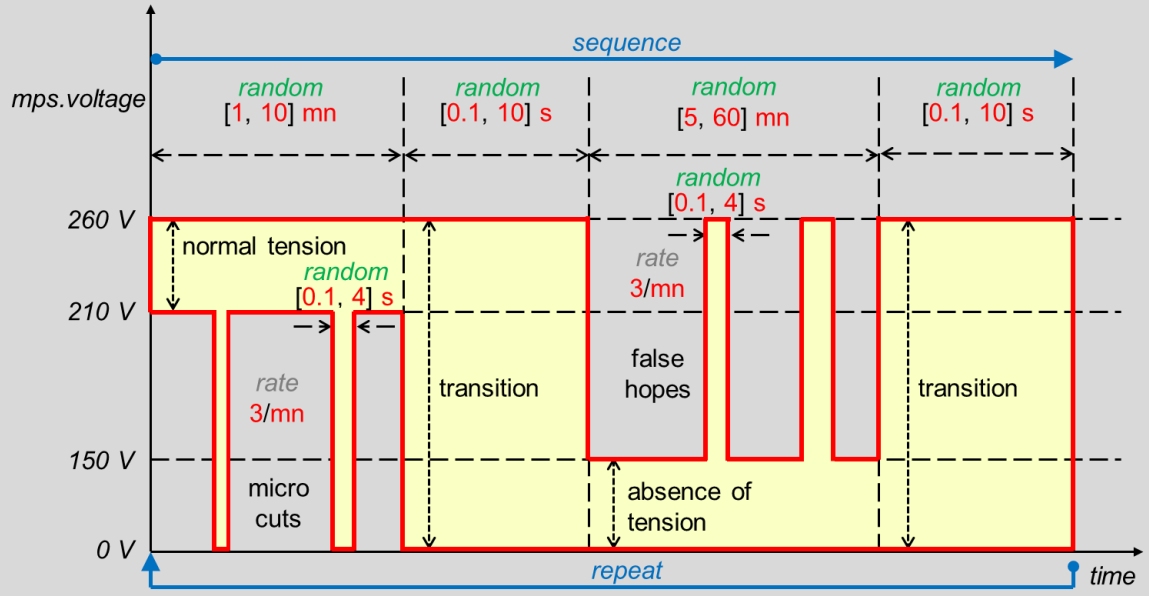


Figure 4-4
Scenario envelope for **mps.voltage**.

```
operator.eReset begin
  Assumption notTooOften is
    after occurrence during 20*s
    ensure no occurrence;

  Assumption notTooRare is
    after t0 or occurrence within 2*h
    achieve occurrence;
end operator.eReset;
```

The scenario states that there is at least 20 seconds and at most 2 hours between two successive transitions of **resetCmd** to **true**. In this example, one can see two different kinds of constraints: with keyword ***ensure***, the constraint must be satisfied at all instants specified by the temporal locator; with keyword ***achieve***, the constraint must become satisfied (and remain so) in the course of each time period specified by the temporal locator.

```
client1 "Test client" begin
  tolerance begin
    when t0 define value is 100*perCent;

    Assumption consumption is
      during not powered
      ensure derivative in [0, -2.5]*perCent/s  or value = 0*perCent;

    Assumption replenishment is
      during powered
      ensure derivative in [2,   10]*perCent/mn or value = 100*perCent;
  end tolerance;
end client1;
```

The scenario declares two clients, **client1** and **client2**. It states that for **client1**, **tolerance** is initially at 100%, consumed at at most 2.5% per second when not powered, and replenished at at least 2% per minute and at most 10% per minute when powered. A similar model is used for **client2**, albeit with different maximum consumption and replenishment rates.

```
bps begin
  private Boolean #required "When BPS is required to operate" is
    from mps.ok becomes false
    until operator.resetCmd becomes true while mps.ok;
```

```
  Client begin
    poweredByBps begin
      when t0 define value is false;

      // Spurious BPS actuation
      (value becomes true while not required).rate is 0.1/h;

      Assumption "Legitimate Bps actuation, could be too late" is
        after mps.ok becomes false within 3*mn
        achieve value becomes true;

      Assumption "BPS failure during operation" is
        after value becomes true within 45*mn
        achieve value becomes false;

      Assumption "BPS reset, could be too late" is
        after required becomes false within 20*s
        achieve value becomes false;
    end poweredByBps;

    powered is (mps.ok and not required) or poweredByBps;
  end Client;

  eFailure.rate is 1/h;
  Assumption "Alarm, but may be too late" is
    after eFailure within 500*ms
    achieve once eAlarm;
end bps;
```

The scenario considers that the *BPS* is **required** from the instant the *MPS* ceases to be available and until the **operator** issues a valid reset, and states that it can take up to 3 minutes to repower a client and up to 20 seconds to return it to the *MPS*. Powering by *BPS* could stop even in the absence of reset, due to failure (with a *rate* of 0.1 per hour).

Regarding **alarm**, the assumption states that **eAlarm** occurs within a 500 ms delay, so that there will be cases where **alarm** is satisfied and cases where it will not. A credible definition is also given to **eFailure**.

The definitions and assumptions of a scenario do not need to be exactly those of real life or those needed for detailed solution models. Here, although it is relatively simple, it allows cases where **repower**, **reset** and **alarm** are satisfied, and others where they are violated, so that model authors can check whether that agrees with their intentions.

When very different envelopes need to be considered, one can define several simple scenario models instead of a unique but complicated one.

Simulation helped realise that when a client has a too low level of tolerance when an *MPS* loss occurs, the *BPS* does not have enough time to satisfy its **repower** requirement. Thus, a new version of the reference model (named **BpsReference**) has the following corrections:

- In the **backupPower** contract, the **bps** party declares and delivers **power** to each client.
```
Client: Power #power begin
  value is deferred;
  during powered ensure power > 0*W
  otherwise ensure power = 0*W;
end power;
```
- The definition of **tolerance** in the **backupPower** contract offers one additional guarantee concerning the initial value:
```
Guarantee init is when t0 ensure value > 99*perCent;
```
- The definition of **reset** in the **bps** object has an additional condition, which is that the **tolerance** of all clients must be close to its maximum before any client is returned to the *MPS*:

```
  Event ^eVReset "Valid reset command" is
    operator.eResetCmd while mps.Ok
                          and AND (Client.tolerance >= 99*perCent);
  redeclared ^reset is
    after eVReset within 10*s
    for all x of Client
    achieve (x.powered and not x.poweredByBps) or not mps.Ok;
```

- Also, as it now depends on `Client.tolerance` which has a `^` determiner, **`reset`** is redeclared to have a `^` determiner.

4.6 Coordination of Different Disciplines

A complex system can be defined as a system that cannot be fully understood and mastered in all necessary aspects and to the necessary level of detail by any single individual or from the standpoint of any single engineering discipline. On the contrary, it needs the cooperation and coordination of many teams, disciplines, organisations and stakeholders, covering a wide range of topics, from so-called "hard sciences" (such as physics, materials, civil engineering and computer and software engineering) to "softer sciences" (such as psychology and sociology). Different participants may also be responsible for different parts of the system. Coordinating them all is an essential but difficult task. In particular, as each may have their own notions, terms and methods, they may, and often do, have difficulties understanding one another. Also, too little coordination rapidly leads to chaos, but too much could lead to paralysis. The essence of systems engineering is thus the art of coordinating all participants *just as necessary*.

To this end, BASAALT and FORM-L rely on *modelling modularity* and *model composition*, whereby each participant can develop separate but interrelated, compatible and interoperable models representing their view points on the system, its components and its environment, and where multiple models representing all necessary viewpoints can be assembled into self-standing *mock-ups* tailored to support particular engineering activities. The mutual consistency of all the models constituting a mock-up could be verified with extensive tool support.

As *BPS* failures may have dire, unacceptable consequences, its dependability is of utmost importance. Thus, a dependability team is involved very early in its engineering. **`BpsReference`** is their main input, but they consider that the model needs to be extended: as component failures are doomed to occur, no human-made system can perfectly achieve the three requirements (**`repower`**, **`reset`** and **`alarm`**) as they are specified. Consequently, they developed a model named **`BpsPsaReference`** that extends **`BpsReference`**. (*Psa* stands for *Probabilistic Safety Assessment*.)

```
Model BpsPsaReference "Top-level dependability model for the BPS"
  extends BpsReference
begin
  refined bps;
end BpsPsaReference;
```

```
bps begin
  // Repower failures
  redeclared Objective repower begin
    Requirement ?°pfd       // Probability of Failure on Demand
     "The pfd of 'repower' shall be less than 1e-4";
    Requirement ?fro        // Failure Rate in Operation
     "The failure rate while powering clients shall be less than 1e-4/h";
    Requirement ?sar        // Spurious Actuation Rate
     "The spurious repower rate shall be less than 1e-3/year";
    substitution is {°pfd, fro, sar};
  end repower;

  // Reset failures
  redeclared Objective reset begin
    Requirement ?°pfd
     "The pfd of 'reset' shall be less than 1e-4";
    Requirement ?sar
     "The spurious reset rate shall be less than 1e-3/year";
    substitution is {°pfd, sar};
  end reset;
```

```
  // Alarm failures
  redeclared Objective alarm begin
    Requirement ?°pfd
     "The probability of failure to alarm shall be less than 1e-3/demand";
    Requirement ?sar
     "The rate of spurious alarms shall be less than 1e-2/year";
    substitution is {°pfd, sar};
  end alarm;
```

BpsPsaReference considers that the three requirements of **BpsReference** are not achievable and can no longer be considered as requirements. Thus, it starts by stating that they are now mere objectives (e.g., *redeclared Objective* **repower**). Each is then substituted with achievable probabilistic dependability requirements (e.g., *substitution is* **{°pfd, fro, sar}**), which are at first captured in natural language.

Note. As *pfd* is a keyword representing a property attribute, it cannot be used to name a requirement. Thus, here, requirements also related to probabilities of failure on demand are named **°pfd**.

In BASAALT and FORM-L, there is a clear distinction between an *objective* and a *requirement*: an objective is a property that is desired but not necessarily achievable, whereas a requirement is a property that MUST be achieved. From a tool perspective, the difference is that formal analysis or simulation will not flag the violation of an objective as an unacceptable outcome requiring solutions to be modified.

The **?** determiner indicates that a value is an *a posteriori* value, i.e., that it can be neither determined *a priori* (during design) nor evaluated during individual operational cases: on the contrary, to determine its value, one needs to assess the outcomes of large sets of simulated cases. This is the case of expressions involving attributes *pfd* and *rate*.

```
  Boolean ^needed "Whether BPS action is needed"
    is from mps.ok becomes false until (eVReset + 10*s);

  repower begin
    °pfd is
      from needed becomes true during 2*mn
      ensure repower.pfd < 1e-4;

    fro is
      during needed except first 2*mn
      ensure repower.eViolation.rate < 1e-4/h;

    Event eSpurious is deferred;
    sar is ensure eSpurious.rate < 1e-3/year;
  end repower;
```

```
  reset begin
    °pfd is ensure reset.pfd < 1e-4;

    Event eSpurious is deferred;
    sar is ensure eSpurious.rate < 1e-3/year;
  end reset;
```

```
  alarm begin
    °pfd is ensure alarm.pfd < 1e-3;

    Objective noSpurious is
      during not (from eFailure for 1*mn)
      ensure no eAlarm;
    sar is ensure noSpurious.eViolation.rate < 1e-2/year;
  end alarm;
end bps;
```

Now, the natural language PSA requirements are formalised. Here, one can see examples of attributes (such as variable **repower.***pfd* and event such as **repower.***eViolation*) that represent the dynamics of a property (here, **repower**) and that are used to specify other properties (such as **repower.°pfd** and **repower.fro**).

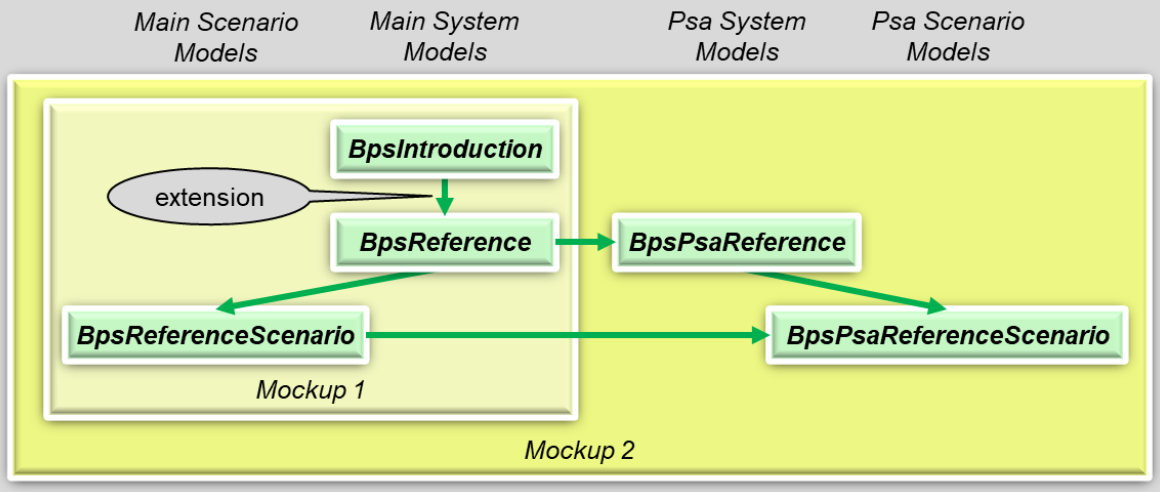


Figure 4-5
Mock-up to verify **`BpsPsaReference`**.

Like **`BpsReference`**, **`BpsPsaReference`** will need a scenario model (here, **`BpsPsaReference-Scenario`**), which could be an extension of both **`BpsPsaReference`** and **`BpsReferenceScenario`** that would define failure probabilities (see *Figure 4-5*).

Like the PSA team, other disciplinary teams could provide their own viewpoints by extending **`BpsReference`** or **`BpsPsaReference`**.

4.7 Non-Engineered Interactions, Encroachments

To highlight the fact that failure propagation and some side effects in normal operation are undesired, non-engineered but sometimes unavoidable, they are not modelled with contracts but with *encroachments*. A FORM-L encroachment is between two parties: the *origin* that causes an effect and the *target* that suffers the effect. Both may be individual objects or classes.

Only the origin has an active role: it may force the behaviour of some of the target's variables, events and sets during the encroachment, with precedence with respect to their definitions by the target.

4.8 Abstraction, Focus, Complexity Management, Mock-Ups

As engineering progresses, each new step adds new solution models, new discipline-specific models and new scenario models. When dealing with systems as large and complex as industrial plants, power grids, modern cars or commercial aircrafts, this could (and will) result in such a large number of models that all together, they would be so voluminous that they would be impossible to analyse or simulate even with the best of tools and the most powerful of computers. The BASAALT solution to this issue is based on model composition with the notion of bespoke *mock-ups*, as shown in Figure 3-1 and Figure 4-5.

A mock-up is a self-sufficient set of models tied together by extension and bindings (see Section 4.11 *Integration of Existing Solutions in Bottom-Up Approaches*) that can be simulated and / or analysed. Self-sufficient means that all objects **`deferred`** by a model are given a definition somewhere in the mock-up. A mock-up could be tailored to the specific needs of a particular engineering activity (possibly with tool assistance) by selecting the models focused on the objects and issues of interest for that activity.

Contracts and encroachments help limit mock-ups to just what is necessary: objects and classes that are not the objects of interest for the activity may be declared ***`external`*** and be represented only by their contracts and encroachments with the object or class of interest.

The *BPS* clients are systems of their own such as cooling systems, ventilation systems or systems monitoring critical parameters of the overall installation. They have their own requirements and their own possibly complex design. Engineering activities addressing the *BPS* can make abstraction of that: as clients are not their objects of interest, they can be represented solely by their contracts with **`bps`**.

4.9 Step-By-Step Solutions in Top-Down Approaches

The engineering of a system worthy of the name is not done in a single stroke: on the contrary, it is a step-by-step process, where at each step, new engineering decisions are made in addition to all those

made at earlier steps; where some decisions made at earlier steps may be reconsidered; and where new details regarding the system are revealed or decided. To cover systems life cycle, models need to take account of this progressive process. In BASAALT and FORM-L, it is supported by the notions of *extension* and *refinement*.

Although refinement and extension have similarities (they both add new information to some base items), there is an important difference: an extension creates a new item, and for classes, it corresponds to the classical notion of class inheritance in object-oriented approaches; a refinement modifies an item *in situ*. For example, when a class is refined, all its instances benefit from the new information, including instances that existed before the refinement. This is not the case with extension.

Having specified and validated the top-level requirements in **`BpsReference`** and **`BpsPsaReference`**, one can now proceed to design (see Figure 4-6).

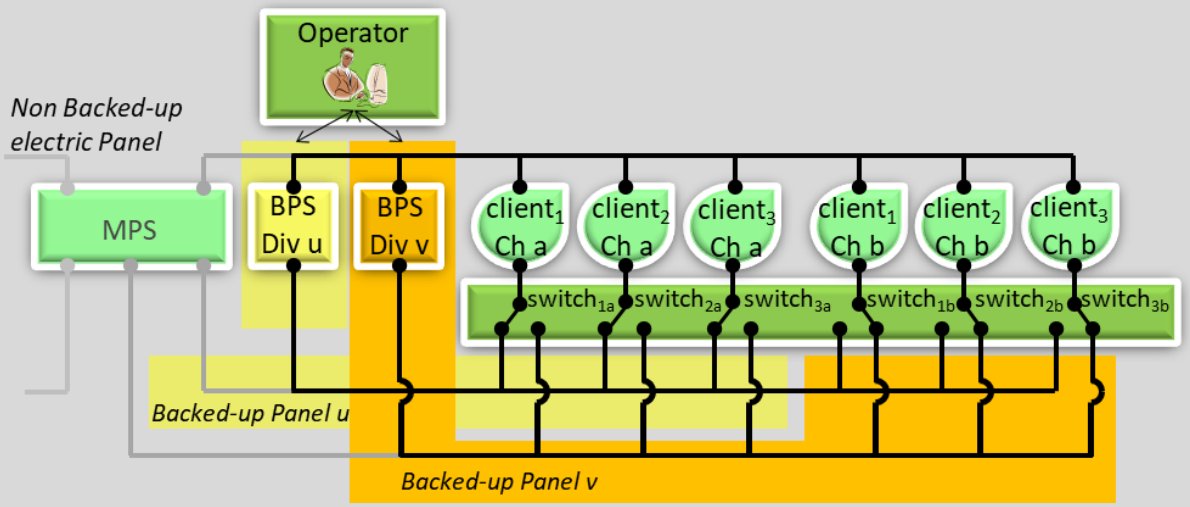


Figure 4-6
Outline of the first design step: redundancy. The switches are operated by the operator.

To simplify the example, several decisions that could have been made in separate steps are here merged into a single one:

- As the lifetime of the overall installation and of the *BPS* is very long, and as *BPS* dependability requirements are stringent, this first design step introduces redundancy. The *BPS* is now composed of two identical, independent ***`Division`*** named **`u`** and **`v`**. Independence means that no failure in one division can propagate to the other: this places strong constraints on engineering disciplines such as buildings architecture, civil engineering, piping, cabling and ducting.
- Now, the set of *BPS* clients (**`client`**$_1$, **`client`**$_2$ and **`client`**$_3$) and their expectations with respect to the *BPS* (electric power needed and maximum consumption and minimum replenishment rates of their tolerance level) are known. Also, **`client`**$_2$ and **`client`**$_3$ operate continuously, whereas **`client`**$_1$ operates on demand, i.e., only when conditions in the overall installation require it.
- As the *BPS* clients are themselves highly critical systems (otherwise they would not need back-up power), they are also redundant: each client is composed of two identical, independent ***`Channel`***[3] named **`a`** and **`b`**. One channel is sufficient to perform a client's mission.
- The backed-up electric panel of the *BPS* is also composed of two redundant panels, each being associated with one *BPS* division. Each client channel has a **`switch`** that connects it to one and only one of the redundant panels. The switches are managed by the operator and are not part of the *BPS*.
- When one *BPS* division needs to be maintained, it is isolated and put out-of-service: the *BPS* requirements must then be ensured by the remaining division. In particular, all channels must be connected to its panel.
- Also, as *BPS* demands are rare, periodic testing is necessary to limit the likelihood that random, unrevealed defects prevent the divisions from performing their missions when required.

[3] Divisions and channels are identical notions. Two different terms are used so that a division is necessarily of the *BPS*, and a channel is necessarily of a client.

These design decisions are formalised in a new model named **BpsRedundancy** that extends **BpsReference**. This model also introduces new requirements and assumptions deriving from these design decisions, but these are attached to the divisions and the channels.

Contrary to the previous models, this one and the subsequent ones will not be presented in full detail: only aspects of BASAALT and FORM-L not seen in the previous models will be discussed.

In an overview statement, the main features of the refined **bps** and **Client** are declared to provide a meaningful context to the subsequent statements. In particular, one can note that ***Client*** now has six *parameters*. These will be applied to the channels of **Client** instances: the first three define a computable lower bound for the channel's **tolerance** to loss of power (see *Figure 4-7*), whereas the last three define a computable upper bound for the channel's call for power (see *Figure 4-8*). Three clients are then declared, each with its own set of parameter values.

```
Model BpsRedundancy extends BpsReference begin
  refined bps begin
    Class #Division;
      Automaton
      [ standby     "Monitoring MPS, ready to power channels if necessary"
      , active      "Powering channels with own power after an MPS loss"
      , test        "Performing a periodic test"
      , maintenance "Being repaired and unavailable"
      , failure     "In failure and unavailable"
      ] #state;
    end Division;

    // The BPS has two redundant divisions u and v
    Division u;
    Division v;

    // Single Failure Criterion (fault tolerance)
    Requirement repower.#sfc
      "'repower' shall meet the single failure criterion";
    Requirement reset.#sfc
      "'reset'   shall meet the single failure criterion";
  end bps;

  refined Client
  ( constant Rate     !kcMax      "Maximum tolerance consumption rate"
  , constant Rate     !krMin      "Minimum tolerance replenishment rate"
  , constant Duration !latency    "Time for replenishment to be effective"
  , constant Power    !initial    "Initial call for power"
  , constant Rate     !stabilised "Stabilised call for power"
  , constant Duration !settleTime "Duration of initial call for power"
  ) begin
    Class #Channel;

    // A client has two redundant channels a and b
    Channel a;
    Channel b;
  end Client;

  Percentage pC is 1*perCent;
  //       kcMax      krMin      latency   initial   stabilised   settleTime
  Client (2.5*pC/s, 10*pC/mn, 5*s,      12*kW,    4.0*kW,      4*s) client1;
  Client (4.0*pC/s,  8*pC/mn, 8*s,       8*kW,    2.5*kW,      3*s) client2;
  Client (5.0*pC/s,  6*pC/mn, 2*s,      15*kW,    5.0*kW,      3*s) client3;

  // Contract between each BPS division and each client channel
  Contract backupPowerDiv (bps.Division, Client.Channel);

  Class @PushButton extends Boolean "Boolean HSI input device";
  refined hsi;
end BpsRedundancy;
```

With redundancy, one can see that **bps** has now two new requirements: **repower.sfc** and **reset.sfc** (*sfc* stand for *Single Failure Criterion*, a type of fault-tolerance).

One can also see the declaration of ***PushButton***, a defined class of variable based on ***Boolean***, and of **backupPowerDiv**, a contract between two classes, i.e., between each instance of each class.

PushButton models the Boolean output of the devices used by the operator to send commands to the *BPS*. Its objective is to make sure that these commands are detectable: if they remain **true** for too brief an instant, a technical system could miss them. Also, they must not remain frozen on **true** otherwise new commands would be impossible. Thus, ***PushButton*** is an extension of ***Boolean*** with requirements guaranteeing that instances remain **true** for at least 0.5 second and at most 2 seconds.

In the refined **hsi**, the **operator** delivers six ***PushButton*** that, like **eReset**, are control information (**@** determiner):

- One to place ***Division*** **u** in maintenance, and another one for ***Division*** **v**.
- One to signal the end of any on-going maintenance.
- One to initiate a periodic test of ***Division*** **u**, and another one for ***Division*** **v**.
- One to signal **eReset** and also to end any on-going periodic test.

Lastly, **operator** delivers a ***Boolean*** **#switch** for each channel, specifying whether the channel is connected to ***Division*** **u** or ***Division*** **v**.

In addition to these new deliverables, **operator** offers **bps** new guarantees:

- At most one division at a time will be placed in maintenance or test.
- Prior to placing a division in maintenance, all channels will be connected to the other division.
- Switches are not manipulated when the *BPS* is active, unless one division signals an alarm during a real demand (i.e., not during a test): then the operator will transfer the channels connected to the failed division to the other division if it is itself neither in maintenance nor in failure.

operator does not need to guarantee that it will not place a division in maintenance or test during a real demand: that will be enforced by **bps** itself.

bps defines the **eAlarm** event as the union of two new deliverables: events **u.eAlarm** and **v.eAlarm**, i.e., the separate alarms from each division.

In this model, **operator** is refined instead of being solely defined by its **hsi** contract, because as refined **hsi** introduces many new deliverables and guarantees, its authors have chosen to have it as an overview, with declarations only: definitions are provided by **operator** and **bps**.

Many other systems of the overall installation place their own demands on the operator, and new demands may arise as engineering progresses. At any engineering step, by collecting all the contracts to which **operator** is a party, human factors engineers can determine whether the operator always has the information and time necessary to make appropriate decisions and take necessary actions.

They can also determine whether there are situations where workload is too high, whether the operator should in fact be an operation team and how work should be allocated between team members. Also, analysis or simulation at installation level may be used to verify that the procedures that the operator are required to apply are adequately specified and appropriate.

A cost team could then estimate operators related costs and determine whether they are acceptable: if not, redesign might be necessary.

The dependability team could include human errors in their models, and estimate the possible effects and criticality of incorrect application of procedures, and even of incorrect choices of procedure.

Class ***Channel*** is a party to **backupPowerDiv**. As per this contract, each channel provides each division the following deliverables:

- Whether it is **@required** to operate. It is not when in maintenance; otherwise, it is if and only if the client to which it belongs is.
- Its own **^tolerance** (see *Figure 4-7*). As this is conceptual and not actually measured during operation, the channel also delivers three pre-determined constants that together define a lower bound (**@lowerBound**) to **tolerance**: **!kcMax** is the maximum consumption rate when the channel

is required to operate but not powered; **`!krMin`** is the minimum replenishment rate when the channel is powered; **`!latency`** is the time needed after repowering for the replenishment mechanism to be effective.

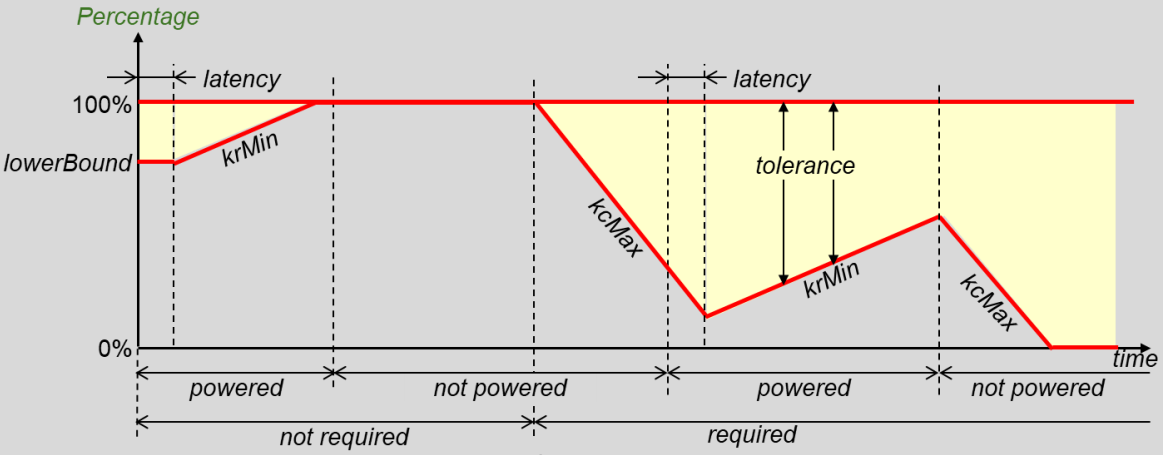


Figure 4-7

The yellow area is an envelope for the **`tolerance`** *of a channel.*

- Its instantaneous call for *BPS* power (**`^powerCall`**, see *Figure 4-8*). As this is also not actually measured during operation, the channel delivers three other pre-determined constants that together define an upper bound (**`@upperBound`**) to **`powerCall`**: when it is not required to operate or not powered by the *BPS*, that upper bound is naught; when it is powered by the *BPS* while required to operate, the laws of electricity tell us that there will first be an important spike in the call for power (up to **`!initial`**), and that after a certain time (**`!settleTime`**), the call for power will stabilise below a lower value (**`!stabilised`**).

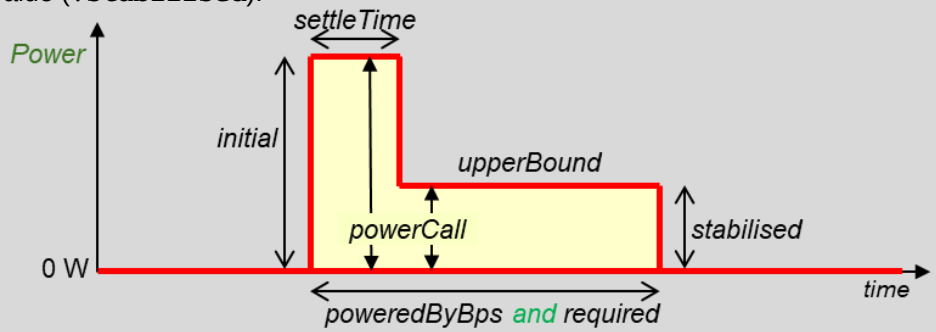


Figure 4-8

The yellow area is an envelope for call for the **`powerCall`** *of a channel.*

As per the contract, each channel guarantees that its **`tolerance`** and **`powerCall`** always remain within their respective envelopes.

In return, each division ensures its own **`repower`** and **`reset`** requirements. These are similar to **`repower`** and **`reset`** at **`bps`** level, but with two important differences: first, they apply only to connected channels; second, they do not refer to channels' **`tolerance`** (the exact value of which during operation is not known) but to their **`lowerBound`** (the value of which is fully determined): thus, they can have a **`#`** determiner and are a *`concretisation`* of **`bps.repower`** and **`bps.reset`**.

```
bps.repower.concretisation is u.repower  weakOr3 v.repower;
bps.reset  .concretisation is u.reset    weakOr3 v.reset  ;
```

The ***`weakOr3`*** operator returns a property that is in *`satisfaction`* when and only when any of its arguments is (even when the other is in *`violation`*), and in *`violation`* when both arguments are.

Also, to satisfy **`repower`**, each division delivers a **`#power`** and a **`#voltage`** (actual, physical electric power and voltage) to each channel. Their value is naught when the division does not power the channel, and delivered **`power`** is guaranteed to be equal to the channel's **`powerCall`** when the division does power the channel. The exact value of **`powerCall`** during operation is also not measured, but the division guarantees that it is able to deliver up to **`upperBound`**.

In addition to defining class **`Division`** and its two instances **`u`** and **`v`**, and to acknowledging its refined contract **`hsi`**, the **`bps`** refinement can now give a definition to features that were up to now **`deferred`**.

One can see here that top-level requirements of the reference model, based on conceptual system state variables, have been *concretised* into stronger design requirements based on computable, quantified terms (such as **`tolerance.lowerBound`** or **`powerCall.upperBound`**). Stronger here means that satisfaction of the concrete requirements implies satisfaction of the conceptual requirements, but this needs to be verified. However, even as the first design step remains at a fairly high level, its model is significantly bigger (here, 450 lines including comments) than the reference model (here, 100 lines). Manual verification is usually still possible, but tool-aided verification is of a great help and offers stronger assurance.

Like **`BpsReference`**, to benefit from full tool support, **`BpsRedundancy`** needs a scenario model (possibly named **`BpsRedundancyScenario`**) extending it and possibly also **`BpsReference-Scenario`** as shown in Figure 4-9. Figure 4-9 also shows the new simulation data flow.

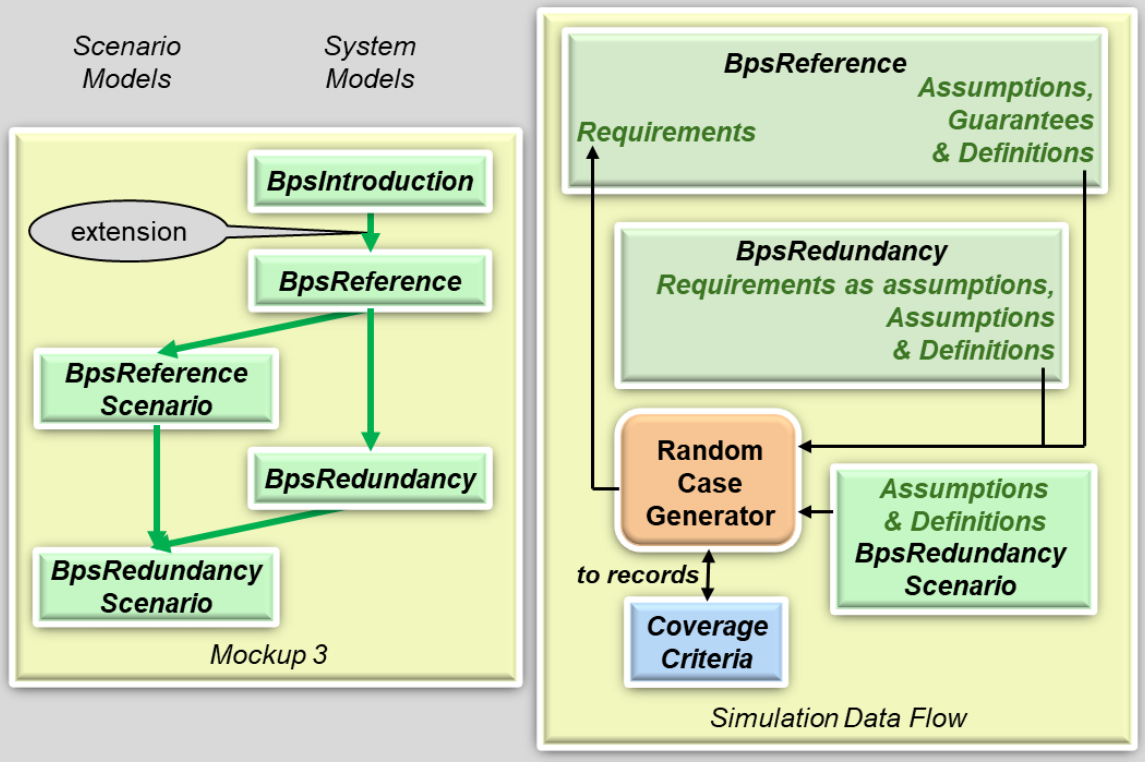


Figure 4-9
As engineering progresses, extension models capture design decisions and disciplinary viewpoints.

Also like for **`BpsReference`**, the dependability team can extend **`BpsPsaReference`** and **`BpsRedundancy`** with a dependability model (**`BpsPsaRedundancy`**) together with its scenario model (**`BpsPsaRedundancy-Scenario`**) also as shown in Figure 4-10. This model considers that the two single failure criterion requirements (**`repower.sfc`** and **`reset.sfc`**) of class **`Division`** are unachievable due to the fact that common-cause failures cannot be excluded, and redeclares them as objectives with probabilistic substitution requirements. It also considers that human errors need to be taken into account and that the guarantees offered by **`operator`** are also unachievable, and redeclares them as objectives with probabilistic substitution requirements. Lastly, it places probabilistic requirements on divisions. Verification that all this is consistent with the probabilistic requirements of **`BpsPsaReference`**, considering assumptions on the frequency and average duration of maintenance sessions (during which only one division is operable) could be done very early in the system life cycle, contrary to many current practices where probabilistic studies are performed only after detailed design is completed, just as a confirmatory measure, which often leads to unnecessarily large and costly design margins, or to last minute design changes, also very costly.

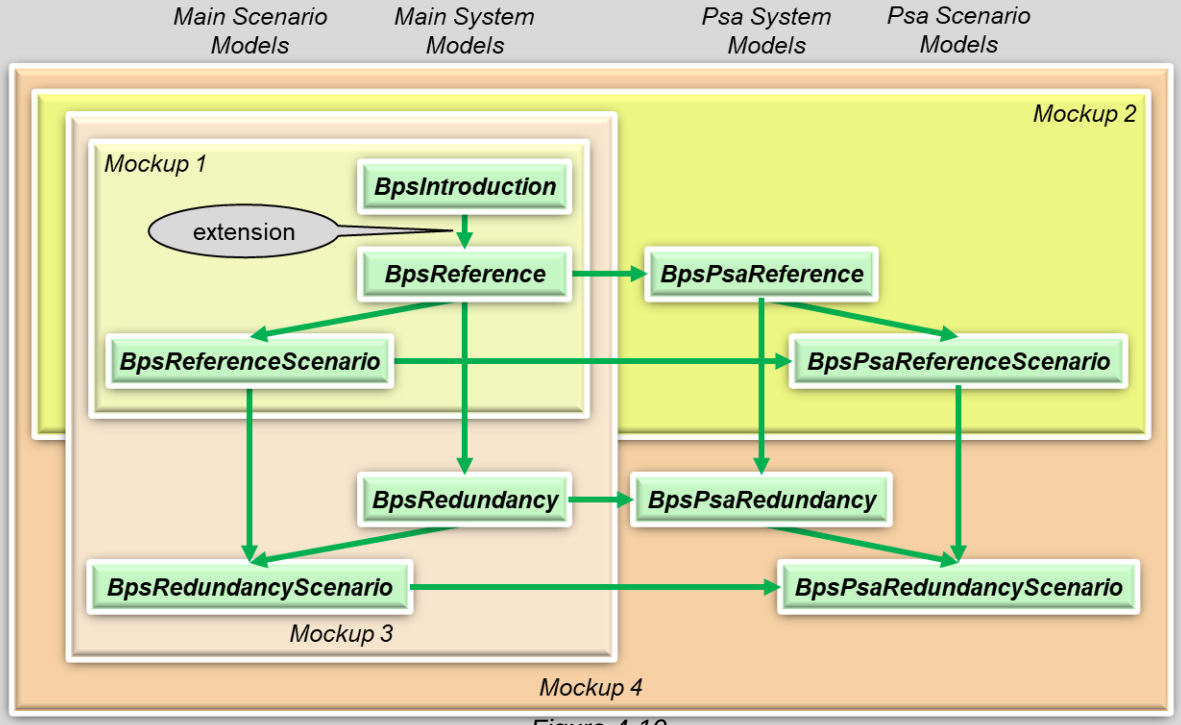


Figure 4-10
Simulation data flow with a first step solution captured in an extension model.

4.10 Complexity Management: Focus and Abstraction

Once the first design step is verified, one can proceed to a second step.

A *BPS* division is now organised into a number of components as shown in Figure 4-11:

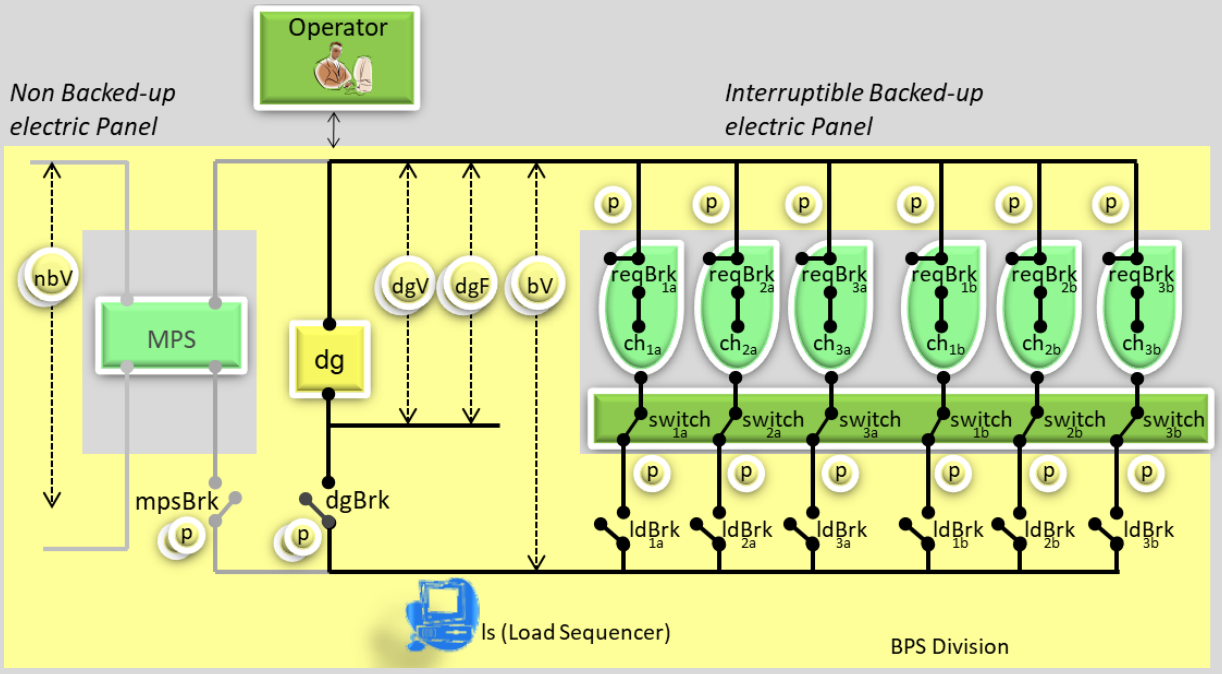


Figure 4-11
Architecture of a BPS *division.*

- A diesel generator (**dg**).
- Several electric circuit breakers (**mpsBrk**, **dgBrk**, and one **ldBrk** per channel). Also, each channel owns and controls a **reqBrk** breaker: it is closed when the channel is required to operate, and open otherwise. Some clients operate on demand, but even for those operating continuously, their channels may sometimes need to be put out of service, e.g., for maintenance. Each client has a requirement such that not all its channels are simultaneously put out of service.
- Several redundant voltmeters (**dgV1** and **dgV2**, **bV1** and **bV2**, and **nbV1** and **nbV2**).
- Redundant frequency meters (**dgF1** and **dgF2**).
- Position sensors for each breaker and switch (redundant sensors for **mpsBrk** and **dgBrk**, one sensor per channel **switch**, and one sensor per channel **reqBrk** breaker).
- A digital load sequencing control system (**ls**).

This is captured in a FORM-L model named **BpsDivision** that extends **BpsRedundancy**. As focus is now on divisions rather than on the total *BPS*, **BpsDivision** declares class **bps.*Division*** as its ***main*** item. Also, the guarantees given by the division in **BpsRedundancy** (which have been verified to collectively ensure the satisfaction of **BpsReference** requirements) are now redeclared as requirements, the satisfaction of which by this extension needs to be verified.

To verify that the division probabilistic requirements and fault tolerance requirements are satisfied, the dependability team will have to develop a **BpsPsaDivision** model capturing the failure modes and the failure rates of the division components (possibly adding components that did not appear in **BpsDivision** but that could fail nonetheless, such as passive components like wires and connectors).

As manual *FMECA* (Failure Modes, Effects and Criticality Analysis, an activity generally required of safety critical systems) is notoriously labour-intensive, time consuming and expensive, it is classically performed after detailed design is completed and frozen, just as a confirmatory measure, often with costly over-designed margins. In the BASAALT approach, tool-aided FMECA can be repeated each time design is refined or altered, with much greater accuracy and much less effort, allowing more optimised designs.

Since the division design includes several complex objects (the diesel generator, the load sequencer, possibly the breakers), in a subsequent engineering step, each will be allocated to a specific team. The diesel generator team will capture its engineering decisions in a model that declares **dg** as its ***main*** object and will declare the other *BPS* division objects as ***external*** and consider them only through their contracts with **dg**. The load sequencer team will do the same but with **ls** as its ***main*** object. As long as the contract between **dg** and **ls** does not need to be modified or refined, the two teams can work separately and focus their attention on their object of interest. If one team needs to modify or complete the interface between the two objects, the two teams must reconvene to jointly re-negotiate or refine the contract.

Work on the *BPS* case study was initiated by the need to make sure that the detailed technical and functional requirements for the load sequencer are adequate for all situations. To this end, the BASAALT approach places it in its full operational context, i.e., in the framework of the environment, requirements and design of its parent system the *BPS*. Even at this fairly high level of design, one realises that for example the response times required of the sequencer depend on many factors such as the grace times allowed by clients in different situations, the response times of the diesel generator and of the various circuit breakers, the response time and accuracy of the many sensors involved, and the possible failure (in various failure modes) of all. Also, even in the absence of failure, many situations need to be taken into account, in particular whether channels are connected and required to operate, and the different possible behaviours of the *MPS* and the actions of the operator. Later in the lifetime of the installation, due to ageing or upgrades, some of these factors may vary, and tool-aided verification will again be of a precious help, particularly if engineers who participated in the initial design are no longer available.

4.11 Integration of Existing Solutions in Bottom-Up Approaches

The previous sections show a mainly top-down approach, where one starts with top-level objects and requirements and then develop solutions. However, real-life systems are rarely if ever engineered in a completely top-down manner: a systems engineering method needs also to support the reuse of

existing, suitable partial solutions in a bottom-up manner. This includes in particular the ability to use models provided by suppliers of commercial-off-the-shelf (COTS) components.

A diesel generator is included in the design of a *BPS* division. It is very unlikely that the engineering organisation in charge of the overall installation (the *designer*) choses to develop its own bespoke generator. Most probably, it will purchase a COTS product or subcontract the generator design to a more specialised organisation (a *supplier*). Following the BASAALT approach, the *designer* has developed a number of FORM-L contracts involving the generator, which together express the behaviour, interfaces and guarantees expected from the generator. To proceed with BASAALT, the *designer* will require that the *supplier* provides behavioural models for the generator to be delivered. In the case of a COTS product, it is very likely that these models are also off-the-shelf and use terms and compute values different from those in the *designer*'s models. Also, the *supplier* will most probably want to protect its know-how and intellectual property by not delivering its models in readable textual form, but only as FMUs (Functional Mock-up Units) complying with the FMI (Functional Mock-up Interface) standard (see [11]).

To allow co-simulation of designer's and supplier's models, BASAALT and FORM-L rely on the notion of *binding*, which provide the pieces of information (e.g., variable values and event occurrences) needed by one model (the *client* model) by collecting raw data from one or more other models (the *provider* models) and transform them as necessary, without having to modify neither the *client* nor the *provider* models. Bindings are not specific to BASAALT and FORM-L and are described in more detail in [3].

4.12 Integration of Disciplinary, non-FORM-L Models

Many engineering teams, disciplines and organisations use their own modelling methods and formal languages to represent detailed, deterministic, discipline-specific aspects of solutions. Thus, model composition must address not only FORM-L models, but also specialised disciplinary models such as physical behavioural models, control models, probabilistic models and economic models. As bindings tie FMUs together, they can also support model composition involving non-FORM-L models, and enable co-simulation of FORM-L models and various disciplinary models all together, so that the deterministic behaviours of the solutions represented by these disciplinary models can be verified against the requirements and in the framework of the assumptions of the FORM-L models.

Now that design is sufficiently detailed, solutions developed in the following engineering steps can be modelled in deterministic, discipline-specific languages. For example, electric phenomena can now be modelled in Modelica [1], whereas sequencing control logic can be modelled in Scade [16], both very different languages from FORM-L.

In particular, the supplier of the diesel generator may not know about BASAALT and FORM-L and provide only deterministic models.

4.13 Simulation Coverage Criteria

When using simulation to verify the adequacy of top-level requirements or to verify solutions against requirements, to reach an adequate level of confidence, one usually needs to examine large numbers of cases. With appropriate modelling in FORM-L, automatic random case generation and automatic verification or requirements satisfaction much facilitate statistical testing and Monte Carlo approaches, where very large numbers of cases (sometimes in the tens of thousand or even more) are necessary.

When such large numbers are impractical (e.g., due to limited computing resources or limited time), one may rely on the notion of *coverage criterion*. This notion has been introduced by software engineering to help ensure that a piece of software is adequately tested by a limited number of test cases. Indeed, formal models have many similarities with software, and so does their verification.

Many different coverage criteria have been proposed for software. That will also be the case for formal behavioural models. Although none are proposed in this document, *Figure 4-3* and *Figure 4-10* show how they can be inserted in simulation flows: by analysing already examined cases, appropriate tools could guide the random case generator (possibly using additional assumptions) towards cases that increase coverage.

4.14 Design Optimisation, Agile Approaches

Step-by-step modelling in FORM-L is relatively low-cost: modelling envelopes can be much simpler than modelling detailed, real physical phenomena (see for example the envelopes shown in *Figure 4-4*, *Figure 4-7* and *Figure 4-8*). Thus, when looking for solutions to a given problem, engineers may model

step-by-step multiple alternative options (each with its own mock-ups) and use tool-aided verification to eliminate those that do not satisfy requirements, as shown in *Figure 4-12*.

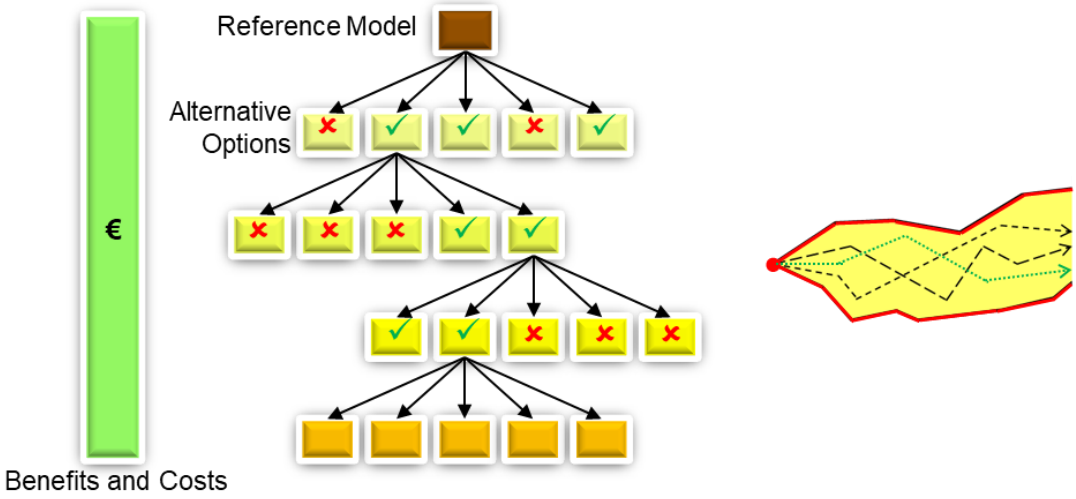


Figure 4-12
Optimisation of solutions by the exploration of multiple options (left-hand side), and optimisation of operation (right-hand side) by the exploration of a constraints model.

The remaining options can then be ranked according to a *merit value* in order to select the best for the next design step (or a few best if one wishes to explore multiple options in depth).

In this *BPS* example, a very simple 1-out-of-2 redundancy architecture has been modelled. Other architectures are possible, e.g., with more redundancies and lower dependability requirements per redundancy. Also, if there are many *BPS* clients, one may consider allocating them among multiple interruptible backed-up electric panels and multiple *BPSs*.

Due to the constraints of hardware implementation, physical systems, including cyber-physical systems, are not as amenable to agile development as pure software systems. However, with modelling and tool-supported simulation and analysis, agile approaches may be applied to help concerned stakeholders evaluate the acceptability of various proposals and to determine what class of system is really needed.

4.15 Support for Operation, Digital Twins

Even though systems engineering is supposed to cover the complete system life cycle, most MBSE (Model-Based Systems Engineering) methods just pay lip service to activities other than design. With BASAALT, the tool-friendly, constraints-based behavioural models developed during system design may be reused to provide effective tool support for operation:

- With redundant or diverse sources of observed operation-time data, behavioural mock-ups and mathematical methods, *data validation and reconciliation* may be used to identify data inconsistencies (generally caused by miscalibration or failure) and to determine which data pieces are the most likely to be incorrect (and thus which components are the most likely to have drifted or failed). It may also be used to correct or reduce measurement errors and margins of uncertainty due to sensors response times and lack of precision, and thus to improve system performance.
- With behavioural mock-ups and mathematical methods, *data assimilation* may be used to interpolate sparse observation data to help determine the most likely current state of the system. *Reverse time simulation* may then help identify the most likely causal factors (e.g., component states or failures) that could have led to the current system state (diagnosis), and possibly, better predict upcoming failures (prognosis). Forward time simulation may also help understand the effects of certain courses of action (*what if*) and determine an appropriate one to reach a desired system state.
- A system may usually be operated in different manners, at different *costs* (not only and not necessarily in monetary terms but also in terms of undesired aspects such as pollution or noise) and with different *benefits* (again, not only and not necessarily in monetary terms but also in terms of other desired aspects such as usefulness to society or well-being of operators and users). In BASAALT and FORM-L, each satisfactory operational case of a mock-up (i.e., each legitimate case satisfying all deterministic requirements) represents a possible manner of operation.

Optimising a system operation activity consists then in finding among these satisfactory cases those maximising a *merit value*, i.e. the value of an ***Integer*** or ***Real*** variable expressing the desired balance between costs and benefits. The determination of the maximal merit value may be done by considering all possible satisfactory cases (with formal analysis) or large numbers of satisfactory cases (with simulation). An optimisation property is satisfied by cases where the merit value is equal (or near) the maximal value.

Maximal merit values are determined *a posteriori* (they have a **?** determiner) and are unknown during an operational case. Thus, they cannot be used in the definition of variables, events, sets and properties that need to be determined *a priori* (before operation starts, with a **!** determiner) or along each operational case (with a **#**, **@** or **^** determiner).

Thus, with BASAALT, a digital twin is not just a model, however accurate and complete it might be: it is a set of up-to-date models (i.e., representative of the real operational system, not just of the initial intentions of designers) from which one can build mock-ups tailored to the specific needs of various operation activities. The fact that each model is useful all along system development is a strong incentive to keep it up-to-date, ready to support operation.

Construction and major maintenance or upgrade actions often require very large numbers of tasks (sometimes in the thousands or even in the tens of thousand) with complex mutual interdependencies dependent on many possible events and conditions. Optimising task schedules is essential but could be extremely difficult. Tasks and their constraints could be captured in specific FORM-L models. Tool-aided analysis or simulation may then help identify contradicting constraints (no schedule could simultaneously satisfy them all). They may also help verify that proposed task schedules satisfy all constraints, or help find optimal schedules, each time constraints are modified or new facts or conditions occur.

4.16 Traceability, Justification of Assumptions and Solutions

Traceability is an essential technique in the engineering of large, complex yet dependable systems. It informs on how a requirement specified at a given engineering step is taken into account in the subsequent steps. Traditionally, it is implemented using links relating the requirement to its solution elements. However, in many cases, simple traceability links do not provide sufficient information. In particular, links just identify the solution elements, which could be numerous: they do not inform on how their possibly complex interactions ensure the satisfaction of the requirement. Also, they do not explain the reasons why a given solution has been chosen. Understanding this is important during initial engineering when the system is designed, constructed and verified, but also during operation or renovation long after the original system engineers are no longer available.

In BASAALT and FORM-L, traceability is supported by the notions of *extension*, *refinement*, *concretisation* and *substitution*, which are much more informative than simple traceability links. Simulation may also help understand how interactions of solution elements contribute to the satisfaction of requirements. However, although they may provide a partial answer, extensions, refinements, concretisations and substitutions do not fully explain WHY a given solution has been chosen. Also, some assumptions cannot be justified by analysis or simulation: they are sometimes based on elements as varied as historical, geological or geographical data; regulation; generally accepted or standardised practices; expert judgment; scientific consensus, etc. Thus, in addition and in complement to the FORM-L based modelling framework presented in the previous sections, BASAALT relies on a *justification framework*.

Over the past years, there has been a trend towards an explicit *claim-based approach* to safety assessment and assurance [5]. The approach described here is a Claim, Argument, Evidence (CAE) approach (see *Figure 4-13*), and is used not just to justify safety, but to justify any kind of engineering claim.

A *claim* is an asserted sentence. For BASAALT and FORM-L, it could be sentences such as "*the system complies with requirement x*" or "*assumption y is legitimate*". It could also be an assertion about a model (e.g., that it had been adequately verified and validated by concerned stakeholders) or about the rigour of the verification of a solution model (e.g., that verification against the specified requirements has been actually performed and was sufficiently rigorous).

A piece of *evidence* is an objective, verifiable fact used in the justification of a claim.

An *argument* links a claim to its supporting pieces of evidence (which could be numerous) in a structured and explicit manner that facilitates the understanding of complex chains of reasoning.

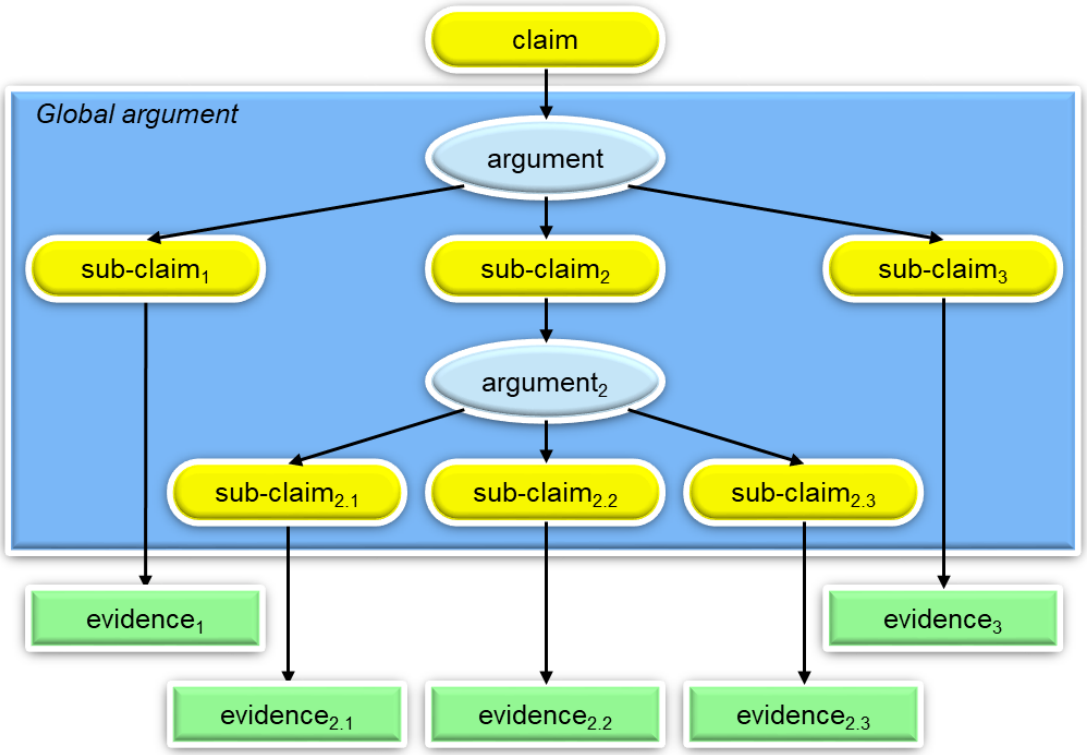


Figure 4-13
Structure of a Claim-Argument-Evidence based justification.

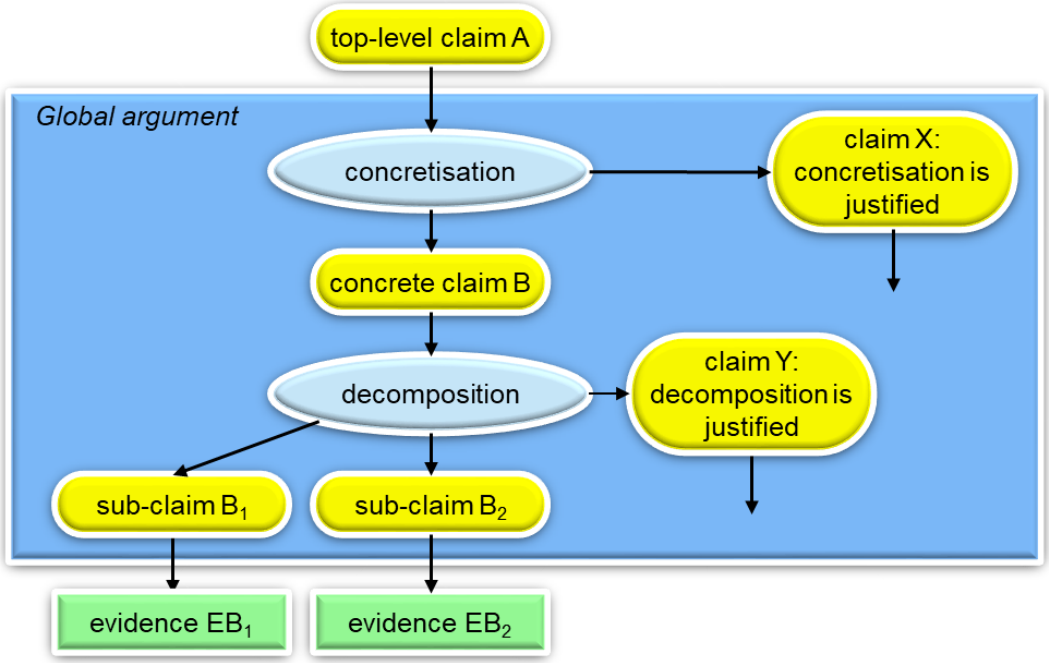


Figure 4-14
Example of Claim-Argument-Evidence based justification with different kinds of argument steps.

Figure 4-14 shows four basic argumentation steps:

- *Concretisation* is used when a claim is given a more precise definition or some interpretation: then, it is sometimes worthwhile to justify that this new definition or this interpretation is appropriate. With BASAALT, concretisation is applied when formalising an object that was initially just described in natural language, or when a conceptual behavioural object (with a **^** determiner) is associated (using the `concretisation` attribute) with more concrete ones (with **!**, **@** or **#** determiners).

BpsReference formalises (i.e., concretises) the natural language requirements of **BpsIntroduction** (**repower**, **reset** and **alarm**), but it does more than a mere translation: it also adds new elements such as margins and grace times. **BpsReference** does not provide itself any justification for the concretisation, but it can provide a link to its justification in a justification framework.

```
repower is
  during not mpsAvailable
  for all x of {Client}
  ensure x.tolerance >= 5*perCent;
repower.justification is "§ link to the justification framework";
```

Once it leaves the standby state (where it just observes the **voltage** provided by the *MPS*) and enters the active state, a *BPS* division first isolates its clients from the *MPS* and then repowers them. It has a low but non zero probability of failure to repower its clients after having isolated them from the *MPS*. As such failure could lead to unacceptable consequences, it is not desirable to activate the *BPS* when not absolutely necessary (e.g., in case of *MPS* micro-cuts). On the other hand, when the *MPS* is really lost, the time delay to repower clients is relatively brief (a few tens of seconds). Thus, the designers opted for a two-step filtering scheme:

- First, based on the voltage at the *MPS* terminals, the division determines the instantaneous **on** / **off** *MPS* state: when **on**, that state MAY be declared **off** when voltage gets below 195 volts, and MUST be declared **off** when voltage gets below 190 volts; when **off**, it MAY be declared **on** when voltage gets above 200 volts, and MUST be declared **on** when voltage gets above 205 volts.
- Then, to determine whether the *MPS* is **available**, the division applies a time-filtering scheme: when **available**, the *MPS* MAY be declared **unavailable** when it has been **off** for more than 1.5 consecutive seconds, and it MUST be declared unavailable when it has been **off** for more than 2 consecutive seconds; when **unavailable**, it MAY be declared **available** again when it has been **on** for more than 8 consecutive seconds, and it MUST be declared available again when it has been **on** for more than 8.5 consecutive seconds.

Is this scheme adequate? And what is the justification for the values of the 8 parameters? The FORM-L models do provide some justification for the 190 volts value (below that, clients are not guaranteed to be operable and to be able to replenish their tolerance level) and for the 195 volts value (which is in part related to the accuracy and response times of voltmeters), but not necessarily for the others, which might be justified by historical and statistical data on *MPS* failures.

- *Substitution* transforms a claim into another one. Like for concretisation, it may be worthwhile to justify that a substitution is legitimate. For example, model-based engineering implies substitution: it substitutes properties of the real system with properties of one or more models. Then, one needs to justify that analysis or simulation of these models provides results representative of what would be obtained with the real system. With BASAALT, substitution is also applied when a requirement cannot be achieved and needs to be replaced with achievable ones.

The *BPS* dependability team realised that the three original requirements were unachievable. It demoted them as mere objectives and substituted each with achievable requirements.

```
redeclared Objective repower begin
  substitution is {°pfd, fro, sar};
  justification is "§ link to the justification framework";
end repower;
```

- *Decomposition* partitions some aspect of a claim into sub-claims, usually according to functions, architecture, properties being considered or with respect to some sequence such as life cycle phases or modes of operation. In addition to the justification of each element of the decomposition, one may also need to justify that the decomposition itself is legitimate.

The justification of the substitution of the real system with a model may be decomposed into sub-claims such as: the model is a correct representation of the system; the scenarios used to challenge the model are representative of the situations the system will face; the tools used do not provide misleading results; tool results are correctly interpreted; etc.

- *Calculation* claims that a given formula computes the value of a system feature from the values of other features, including components features. Then, one may need to justify the correctness of the formula.

For example, the probability of failure on demand of the *BPS* can, with simplifying assumptions, be calculated from the probability of failure on demand of its divisions and on the probability for a division to be in maintenance. Similarly, the response time of a *BPS* division to a loss of the *MPS* can be calculated from the response times of its components (sensors, breakers, diesel generator, load sequencer).

As it can integrate the objective, rigorous logic and elements of formal modelling with subjective human judgement, such a justification framework can offer an insight into the reasoning that led to a chosen solution. This also may help identify possible weaknesses. This would be particularly useful to those who will have to operate, maintain, upgrade and replace the system over multiple decades, when the original engineers are no longer available.

4.17 System Knowledge Representation and Management

Engineering knowledge may be divided into different categories:

- *Disciplinary* and *technological knowledge* are generic (i.e., not system-specific). Education and training provide basic knowledge, which may become expertise with practice and experience.
- *Operational knowledge* (i.e., how to do, how to recognise, how to diagnose) is system-specific but is also obtained through training, practice and experience.
- *System knowledge* is system-specific and is based on a large amount of *engineering information*. However, having that information is not sufficient: effective system knowledge requires *understanding*.

Together, the FORM-L modelling framework and the justification framework constitute not only a rich repository for engineering information: their supporting tools also provide a powerful environment to help understand the information stored. In particular, simulation may help understand the effects and consequences of a given engineering decision in given situations, and the justification framework may help understand the reasons of certain engineering or operational choices. Thus, provided that models and their supporting tools are maintained all along a system life time, MESKAAL (Maintenance of Engineering and Safety Knowledge on a system All Along its Life time) is a by-product of BASAALT.

5 FORM-L, a Constraint-Based Modelling Language

As FORM-L notions are interdependent, this section provides a brief introduction in order to facilitate the understanding of their more detailed presentation in the rest of the document.

5.1 Overview

FORM-L is based on various kinds of *modelling items* (see *Figure 5-1*):

- *Objects*: *variables* (Booleans, finite state automata, integers, reals and strings), *events*, *sets*, *properties* and *non-valued objects*.
- *Classes: templates for objects*.
- *Interface items*: *contracts* and *encroachments*.
- *Temporal items*: *discrete*, *continuous* or *sliding temporal locators*.
- *Organisational items*: FORM-L *models*, *partial models*, *libraries*, *bindings* and *mock-ups*.

They are expressed in terms statements (*declarations*, *definition blocks* and *behavioural instructions*).

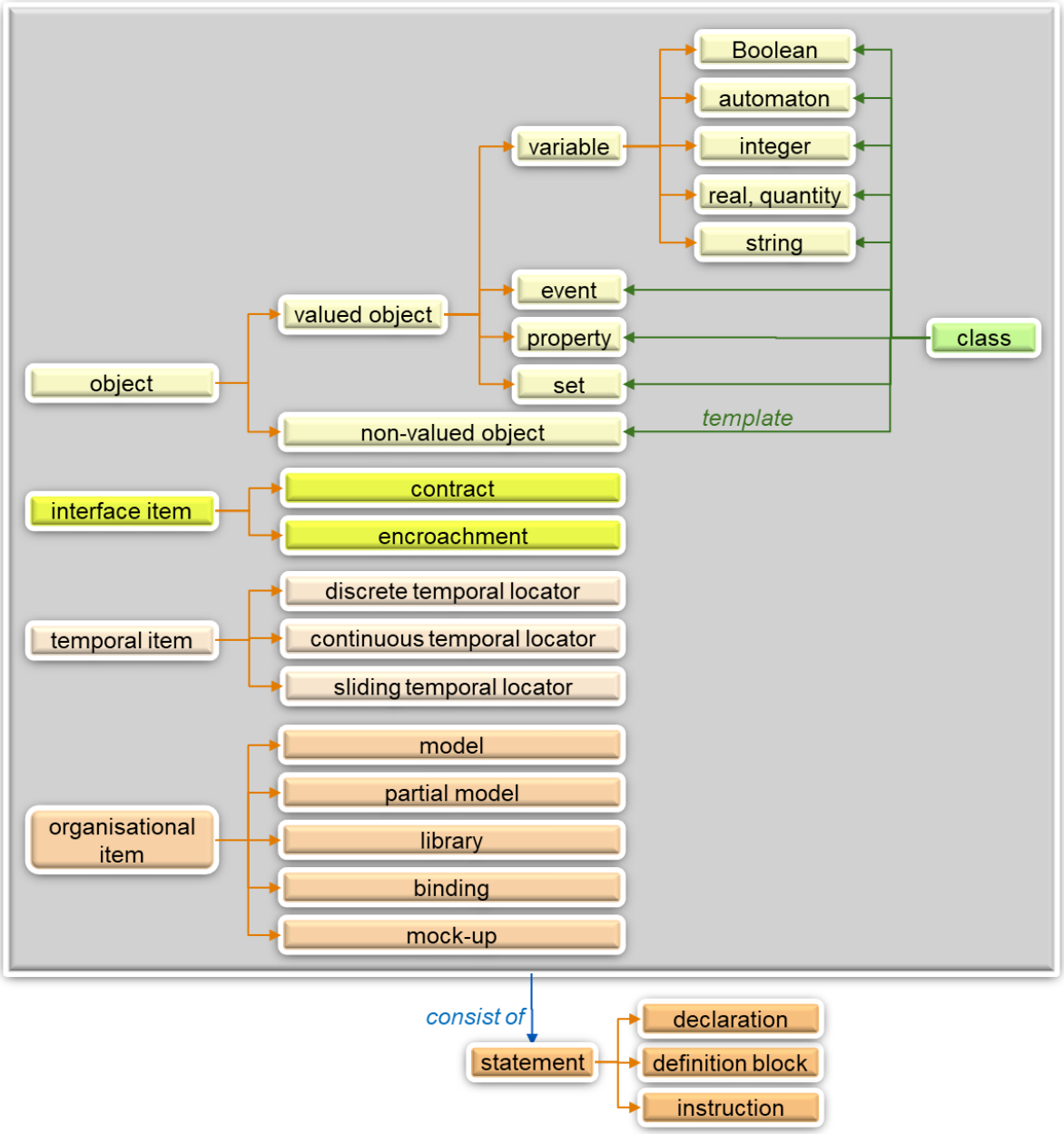


Figure 5-1
Main FORM-L notions.

5.2 Objects

In FORM-L, dynamic phenomena are modelled with *objects*, which are characterised by their *classes* (like in many object-oriented languages, but here, an object may instantiate multiple classes), an optional *determiner*, their *attributes* and their *features*. Determiners and attributes are associated pieces of information built-in the language. Features are model-specific embedded objects: an object has all the features specified by its classes, but may have additional ones of its own.

Some objects (variables, events sets and properties) have an associated *value* and are also called *valued objects*. Such objects have one and only one valued class. Objects that do not have an associated value are *non-valued objects*. A value is an implicit *function of time*: at any instant along an *operational case*, it gives one and only one outcome. The behaviour of an object is entirely defined by its value (if it has one) together with the values of all its direct and indirect valued features. At a higher level, the behaviour of a *mock-up* is entirely defined by the behaviour of all its objects.

A value is `fixed` if it stays the same all along each operational case but may be different in different cases. It is `constant` if it does not change with time and is the same for all cases[4]. In FORM-L, the value of variables, events and sets, is represented by their `value` attribute.

The nature of the value of a variable (Boolean, enumerated, integer, real or quantity) is determined by its valued class.

An event represents a fact of interest that may occur, possibly multiple times, during an operational case but has no duration or the duration of which is neglected. Each time that something happens is an *occurrence* of the event. The value of an event is the set of all its past occurrences.

A *set of objects* is a finite, non-ordered collection of objects of the same class. A *set of values* is a finite, non-ordered collection of values or value intervals of the same nature. Their value is their *membership*.

A property places *constraints* on objects identified by name or by a *selector*, at instants or time periods specified by a *temporal locator*. Contrary to the other valued objects; a property does not have a `value` attribute: instead, its value is given by three Boolean attributes that characterise its state (`untested`, `satisfaction` or `violation`) along an operational case.

Unlike valued objects, *non-valued objects* do not have a value of their own, and their behaviour is determined only by their features.

To reflect the dynamic composition of some systems, in particular of systems of systems, in addition to the named, *static* objects that exist all along operational cases, *dynamic* objects may be created and deleted along operational cases. A dynamically created object is necessarily included in some sets of objects, *a minima* in the implicitly defined set of all instances of each of its classes.

5.3 Classes

A *class* is a template for an object: all what is specified for a class also applies to its *instances*. A *valued class* is a template for valued objects, whereas a *non-valued class* is a template for non-valued objects. There are *predefined classes*, but one may also create *defined classes*. *Extension classes* are created by enriching existing ones. *Quantity classes* are extensions of predefined class `Real`. *Enumerated classes* are created by listing their possible discrete values. *Set classes* are created by associating an existing class with a *set indicator*.

5.4 Attributes

An *attribute* is a variable, event or set-like piece of information built-in the FORM-L language that is used to observe and / or control an object. *Figure 5-2* and *Figure 5-3* give an overview of the available attributes. They are presented in more detail in the following sections.

[4] Yes, a constant variable is an oximoron.

Objects	Attributes		
Non-valued Object			clock *identity* *values* *eNew* *eDelete* justification
Boolean, Set, Integer, String		value next *previous* default	
Real	quantity, unit, scale, offset, *integral, integral* [*n*], *inPIntegral, inPIntegral* [*n*] derivative, derivative [n]		
Automaton	memory		
Event	occurrence, rate, *inPValue*	*value*, default	
Property	substitution concretisation	*pfd, eInPSatisfaction, eInPViolation* *inPUntested, inPSatisfaction,* *inPViolation* *eSatisfaction, eViolation, untested,* *satisfaction, violation, violated, satisfied*	

Figure 5-2
Objects attributes. Those in italics are determined automatically and cannot be assigned or signalled.

Attribute	*Type*	*In-Period?*	*Attribute*	*Type*	*In-Period?*
value (of a variable)	Variable, Set	N	**untested**	Boolean	N
previous			**satisfaction**		
next			**violation**		
default	Variable, Event, Set		**satisfied**		
quantity	String		**violated**		
unit			**inPUntested**		Y
scale	Real		**inPSatisfaction**		
offset			**inPViolation**		
derivative	Quantity / Duration^n		**eSatisfaction**	Event	N
derivative [n]			**eViolation**		
integral	Quantity * Duration^n		**eInPSatisfaction**		Y
integral [n]			**eInPViolation**		
memory	Boolean		**pfd**	Real	AP
value (of an event)	${ Duration }		**rate**	Duration^{-1}	N
value (of a set)	${ identity }		**eNew**	Event	
identity			**eDelete**		
clock	Event		**justification**		
occurrence					
inPIntegral	Quantity * Duration^n	Y	**substitution**	Property or Property { }	
inPIntegral [n]			**concretisation**		

Figure 5-3
Class of attributes. AP *stands for* a posteriori. justification *is implementation-specific.*

Attributes are not objects and have neither features nor attributes, except for event-like attributes, which are objects of class Event and therefore have `value`, `inPvalue`, `clockValue`, `inPClockValue`, `occurrence` and `rate` attributes.

5.5 Interface Items

A *contract* specifies an *engineered* interface between given objects and / or classes (its *parties*), in the form of variables, events, sets and non-valued objects delivered by each party (its *deliverables*), of *guarantees* or *requirements* each party ensures regarding its deliverables, and of *assumptions* it makes and expects regarding the deliverables of other parties.

An *encroachment* models the undesired effects an object or class (the *origin*) may have on another one (the *target*), usually (but not necessarily) due to failures or other abnormal conditions, in the form of instructions specified by the origin forcing the behaviour of the target.

5.6 Operational Cases

An *operational case* (or *case*, for short) is a collective behaviour of all the objects of a *mock-up* (see Section 5.12) and is composed of their *trajectories* as allowed by their definitions. A variable trajectory is a sequence of values. An event trajectory is a sequence of occurrences. A set trajectory is a sequence of memberships. A property trajectory is a sequence of property states.

A *legitimate case* is a case consistent with all the assumptions and guarantees of the mock-up. A legitimate case that violates at least one universal requirement (i.e., a requirement with a universal constraint) is a *delinquent case*. One objective of verification is to make sure that there are no delinquent cases. A legitimate case that violates no requirements is a *satisfactory case*. Another objective of verification is to make sure that there is at least one satisfactory cases, in particular in the presence of capability requirements (i.e., requirements expressing existential constraints)

A case has a *beginning* (a predefined event denoted **`t0`**) before which nothing happens, and an *end* (a pre-declared event denoted **`eoc`**) after which nothing new happens.

5.7 Time

5.7.1 Temporal Locators

Temporal locators are used to precisely place *actions* (i.e., *assignments*, *event signals*, *objects deletions* and *constraints*) in time[5]. There are three kinds of temporal locators (see *Figure 5-4*):

- A *discrete temporal locator* (*DTL*) specifies a finite number of discrete *instants* in time.
- A *continuous temporal locator* (*CTL*) specifies a finite number of possibly overlapping *time periods*.
- A *sliding temporal locator* (*STL*) specifies a possibly infinite number of time periods of a given, fixed duration.

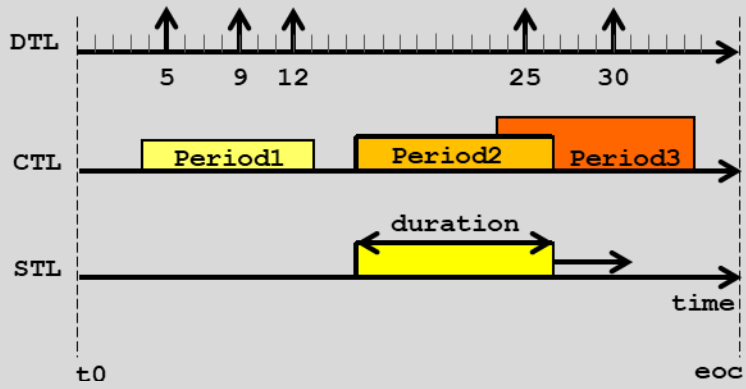


Figure 5-4

Temporal locators. The DTL *is composed of 5 instants. The* CTL *is composed of three periods, the third one overlapping with the second one. The* STL *is composed of all periods of a specified* duration.

5.7.2 Time Domains

Some digital systems are *synchronous*, i.e., they perceive their environment and act on it only at discrete instants. Others (often qualified as GALS: globally asynchronous, locally synchronous) are composed of multiple synchronous subsystems acting asynchronously. Such systems interact with physical elements and human agents operating in continuous time (see *Figure 5-5* and *Figure 5-6*).

[5] Time in FORM-L is local to an operational case and is Newtonian (as opposed to Einsteinian, i.e., relativistic): it is perceived in the same way by all the items involved in the operational case.

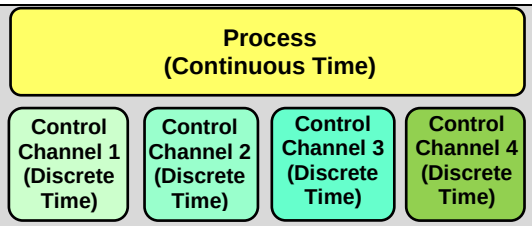


Figure 5-5
Example of multiple time domains. The digital control system is constituted of 4 redundant, independent, and individually synchronous control channels. Each box is a separate time domain

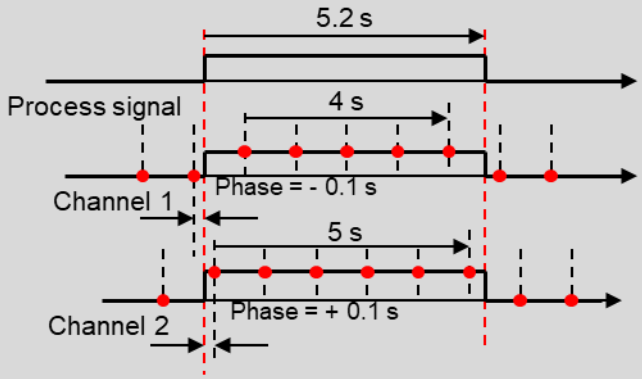


Figure 5-6
Independent time domains. The process signal is in the continuous time domain, whereas the two control channels are in separate discrete time domains, with the same period but different phases. Channel 1 sees the process signal from the 3rd to the 7th tick of its clock (lasting 5 ticks and 4 seconds). Channel 2 sees it from the 2nd to the 7th tick of its clock (lasting 6 ticks and 5 seconds).

In FORM-L, there is one and only one *continuous time domain* where human activities and physical processes occur, and it does not need to be declared. By default, an object is in the continuous time domain, but one can place it in a *discrete time domain* by specifying its `clock` attribute, which is an event the occurrences of which determine the ticks of the clock. A clock is usually periodic with a fixed (generally random) phase with respect to `t0`, but more complex clocks can be defined, for example to add jitter and / or drift to an otherwise periodic clock. A clock may even be completely aperiodic.

5.7.3 Time Domain Interfaces

In general, events not belonging to a discrete time domain cannot be perceived directly by objects in that domain: to be so, they must go through user-defined *time domain interfaces* that perform the necessary adaptations to make these events perceptible at the instants of the clock. Various time domain interface schemes are possible.

Example 1: Holding an event occurrence until the next clock tick. If the objects in the discrete time domain do not react to the occurrence at that clock tick, then the occurrence is lost.

Example 2: Holding an event occurrence until its next occurrence. If the objects of the discrete time domain do not react in time, that occurrence is lost.

Example 3: A FIFO (First In, First Out) queue holding multiple event occurrences.

Note 1: The modelling of time domain interfaces does not require specific FORM-L features and is under the responsibility of model authors.

Note 2: In a discrete time domain, independent events CAN be perceived as simultaneous.

5.7.4 Durations

A *duration* is a stretch of time and can be specified in one of two ways:

- It can be expressed with respect to the continuous physical time as an instance of class `Duration`, which is derived from `Real` as a `quantity` of time.
- In a discrete time domain, it can also be expressed as an `Integer` number of clock ticks. Caution needs to be applied if the clock is not periodic.

Naught durations are ignored, and negative durations raise exceptions.

5.8 Statements

There are three kinds of *statements*: *declarations* (and *redeclarations*), *definition blocks* and *instructions*.

A declaration states the existence of an item of a given name and nature, may provide a natural language description and a definition block. In the case of an object, it may also specify a determiner. An item may be declared multiple times, possibly with additional comments and definition elements, but there must be no contradictions. As engineering progresses, it might be necessary to modify the declaration of an object: a redeclaration may change the determiner, replace one of the classes by one of its extensions, or transform a requirement or guarantee into an objective.

A definition block provides information on the composition and / or behaviour of a declared item, with embedded declarations for composition, and instructions for behaviour. An item may have multiple definition blocks that complement one another: they may be redundancies but there must be no contradictions.

An instruction specifies a specific behaviour (with *assignments*, *signals* and *deletions*) or a constraint on behaviour, for valued objects identified by name or by a *selector*, and at instants or time periods specified by a *temporal locator*. An instruction may be part of an object definition block, or be a *free-floating* instruction coordinating the behaviour of multiple items.

To limit risks of contradiction, in an *extension model*, specification of specific behaviours to a valued object excludes all the instants where the object is already fully determined by the models being extended. Similarly, assignments to a valued object implicitly excludes all the instants at which the object is already fully determined by its class.

An item that has no definition is empty and its behaviour is completely unconstrained.

Note 1. FORM-L is in a large part a declarative language, where the instruction evaluation order may be but is not necessarily specified. At each instant, for instructions that are independent from one another at that instant, any order is appropriate. When at a given instant there are dependencies, these must be acyclic: the order matters, but a natural one (which may vary in time) can be automatically determined.

Note 2. A future extension of FORM-L might introduce *spatial locators* to specify where in 3D, 2D or 1D space a constraint or behaviour applies.

5.9 Extensions, Refinements

An *extension* creates a new item that has all the information of the items it extends, and has new information of its own. As shown in Section 3 *BASAALT, a Systems Engineering* Approach, model extension can serve different purposes and may be used:

- To follow the step-by-step progress of the engineering process.
- To take account of the specific viewpoints of particular teams, disciplines, organisations or stakeholders.
- To introduce generic scenarios.

A *refinement* adds new information to an item while retaining all what was already known on that item.

New information is generally provided in the form of additional statements, but in the case of refinement, it may also be provided in the form of modifications to the item declaration.

5.10 Models

A *FORM-L model* (or *model*, for short when there is no ambiguity) is a named collection of *statements* representing a certain viewpoint on some real-world system and its environment. A statement may be an *elementary statement* or a *block statement* embedding other statements. To bring modularity and greater clarity, and to facilitate the understanding of large and complex models, the statements of a model may be organised into a series. The first one is generally an *overview statement* that declares the model; its embedded statements declare the top-level items of the model, and may provide a partial or complete definition for each. The following statements define or complete the definition of items declared in some previous statement, as shown in Section 3 *BASAALT, a Systems Engineering* Approach.

A model has one ***main*** object or class of interest. It may also indicate that other objects and classes are ***external*** to the item of interest and belong to its environment: then, only what concerns the item of interest needs to be modelled. When an extension model does not specify explicitly its item of interest, then it has the same main and external items as the models it extends, provided there is no contradiction.

```
Model BpsIntroduction "Initial, semi-formal model for the BPS" begin
  main     Object bps      "Backup electrical Power Supply";

  external Object mps      "Main electrical Power Supply";
  external Object operator "Human operator of the installation";
  external Class  Client   "Any system powered by the backed-up panel";
end BpsIntroduction;
```

This is the overview statement of the **BpsIntroduction** model. It is a block statement (keywords ***begin end***) containing four embedded statements. The first one declares the **bps** object and designates it as the object of interest (keyword ***main***) of the model. The three following ones declare objects **mps** and **operator**, and type **Client**. These items are ***external***, i.e., their modelling can and will be limited to what concerns **bps**.

A *partial model* is a named subset of a model that gathers statements addressing a given topic of interest.

A *library* is a model the definitions of which are reused by other models: when a model declares another one as a library, it has access to all its declarations and definitions.

The examples in this document assume the use of a **Standard** library defining the physical quantities and units of the SI system of units, commonly used monetary units, basic physical and mathematical constants, and commonly used mathematical operators.

5.11 Bindings

FORM-L models may be developed by different teams or organisations, independently from one another, each having its own viewpoint on the system or its constituents, its own ways for naming items, and its own modelling rules and conventions. To support co-analysis or co-simulation, information transfer from one model to another may be necessary. However, it is preferable not to modify any of them. *Bindings* bridge the gaps between such models, and also between FORM-L and *non-FORM-L models*, by specifying how information made available by one or more provider models is extracted and transformed as necessary before being transferred to a client model.

Bindings are not specific to FORM-L and are external to it. They are mentioned here just for information. See [3] for details.

5.12 Mock-Ups

A *mock-up* is a collection of FORM-L models, bindings and non-FORM-L models constituting a whole that can be 'compiled' and then analysed or simulated. It is composed of one FORM-L model designated as the *root model*, together with FORM-L and non-FORM-L models directly or indirectly associated with it by extension or binding. Any object or attribute ***deferred*** by a FORM-L model must be given an explicit definition within the mock-up. The object or class of interest for the mock-up is the ***main*** object or class of the root model. The ***external*** objects and classes of the root model may be represented in the mock-up only by their interfaces with the ***main*** object or class, as defined by contract and encroachments.

The root model of a mock-up may be chosen to address the specific needs of a particular engineering activity, and to make abstraction of details not relevant to that activity.

5.13 Justification

FORM-L is intended to be associated with a full-fledged *justification framework* [5] that makes complex chains of reasoning explicit and organises numerous pieces of evidence to justify why a given *claim* is legitimate (or, more precisely, why model authors consider it as legitimate). FORM-L models complement the justification framework:

- By formalizing some of the claims made in the justification framework.
- By providing a basis for the concretisation, substitution, decomposition or calculation formula of certain (sub-)claims made in the justification framework.

- By providing simulation- or analysis-based pieces of evidence.

Conversely, the justification framework may provide a more complete rationale for certain assumptions and refinements, based not only on objective modelling but also on other sources of information (e.g., past experience, historical data or general consensus as expressed in international standards) or subjective reasoning.

Each FORM-L item definition or refinement may have a `justification` attribute in the form of either a property stating the conditions under which the definition or refinement is legitimate or a link to an associated claim in the justification framework.

5.14 Exceptions

A model may lead to invalid conditions during simulation, such as contradictions or divisions by zero. Formal analysis could, in favourable situations, reveal them systematically. When an invalid condition is reached during simulation, pre-defined event `eAbort` occurs and the operational case is stopped (with pre-declared event `eoc`) so that model authors can explore its state and understand what went wrong.

6 Names & Scoping

6.1 Names, Name Spaces & Pathnames

A *name* is a sequence of letters, digits, underscores and ° characters, and begins with a letter or with the ° character[6]. Each name belongs to and is unique within a *name space*. The name spaces of a mock-up are themselves organised into one or more hierarchies, each forming an acyclic oriented graph.

The root model of the mock-up, together with the models in the mock-up that it extends directly or indirectly, together with all their contents, form one hierarchy. Other hierarchies may be formed by models in the mock-up not tied to the root model by extension but by binding.

Except for *dummy names* (i.e., names declared and used by selectors), each name identifies one and only one modelling item in its name space.

```
Assumption consistency is
  for all x of Client
  during x.poweredByBps
  ensure x.powered;
```

A dummy name is a name used by a *selector* to scan through a set, and identifies successively each member of the set. Here, **`x`** is a dummy name for the scanning of set **`Client`** (the set of all named instances of class ***`Client`***).

The name of an item constitutes a single name space for all the item's *definition blocks*: that name space contains the names of the item's sub-items (its *features*) and attributes. Thus, some (generally, most) name spaces are named. However, a *free-floating definition block* (i.e., a definition block not attached to any particular item) constitutes an unnamed name space.

A *pathname* is a sequence of dot-separated names that defines a path within a name hierarchy. A full pathname starts from the top-level name space of a hierarchy, whereas a partial pathname does not: its starting point is then determined by where it appears.

6.2 Scoping

As the FORM-L scoping rules (which define from where and with which name or pathnames an item can be accessed) are specific and somewhat different from those of many other programming and modelling languages, they are described here in detail.

Except for the top-level name space of a hierarchy, a name space is either *embedded* or *injected* into another one.

6.2.1 Name Space Embedment

The embedment of name spaces creates the structure of the hierarchy (see *Figure 6-1*). The scoping rules for embedded name spaces are relatively classic. When name space *B* is embedded into name space *A*:

- A ***`private`*** name in *B* is not accessible from *A*.
- A non-***`private`*** name in *B* is accessible from *A* but must be qualified with the name of *B*.
- Similarly, a pathname not directly accessible from A but accessible from *B* is accessible from A when qualified with the name of *B*.
- A name in *A* is directly accessible from *B*, unless it is also declared in *B*, in which case it denotes a different item. Then, to access the name in *A*, it must be qualified with the name of *A*.
- When a pathname cannot be found in *B*, a match is looked for in the name spaces just above *B* in the hierarchy. If found, it must be only one, otherwise the model must be corrected. If not found, a match is looked for one level up, and so on. If not found in the whole hierarchy, the model must be corrected.

[6] When one needs to use a name that is already a FORM-L keyword, a possible practice is to have a ° as the first character

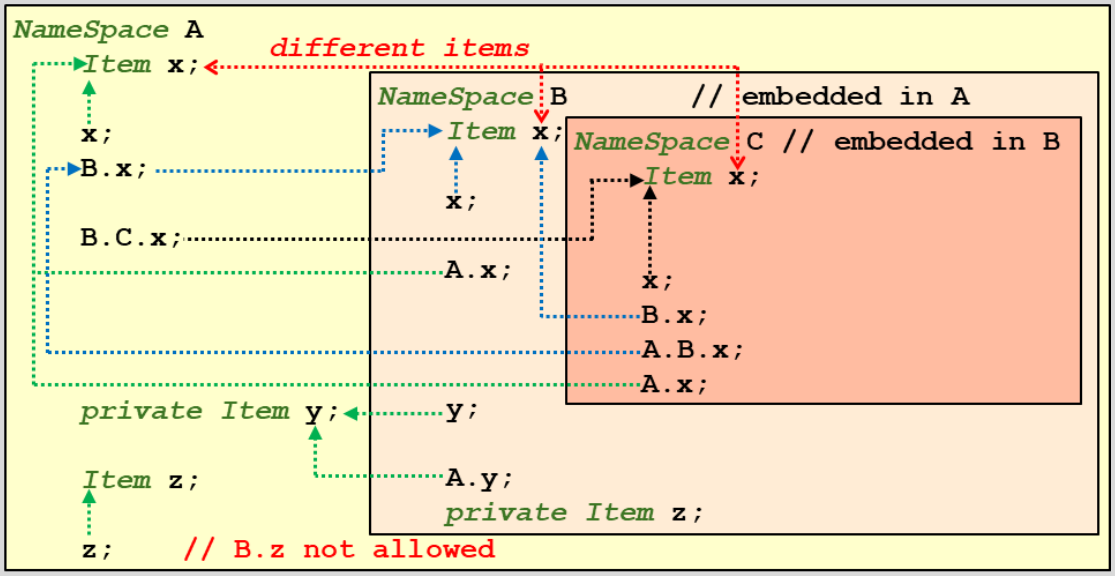


Figure 6-1
Illustration of scoping rules for embedded name spaces.

6.2.2 Name Space Injection

The injection of name spaces creates a mostly flat structure (see *Figure 6-2*): when name space *E* is injected into name space *D*, all *E*'s contents is transferred into *D*, except for names declared by *E* that are also declared in in *D* or in some other name spaces also injected in *D* (e.g., name space *F* in the figure): they remain in *E* and what remains of *E* is embedded in *D*. Names private to *E* or *F* are accessible from *D*, but not from outside of *D*.

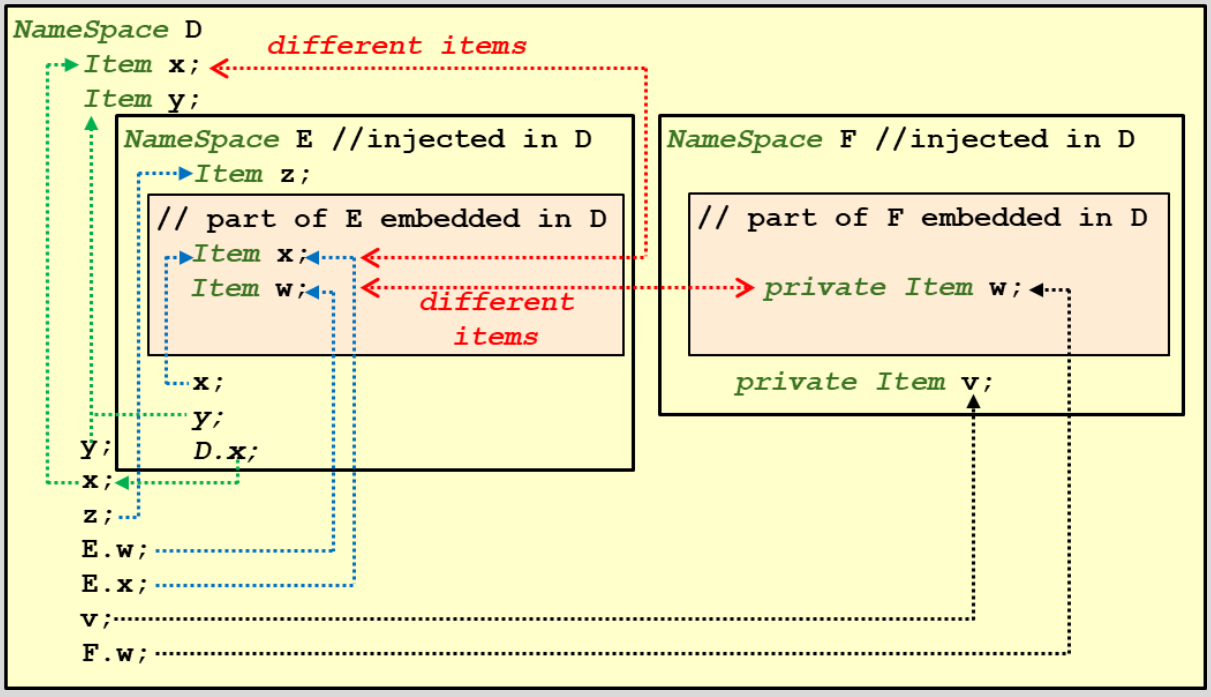


Figure 6-2
Illustration of scoping rules for injected name spaces.

6.2.3 Name Space Rules

The rules regarding name spaces are as follows:

- The root model of a mock-up, together with all the models it extends (directly or indirectly) and all their contents form one hierarchy.

- Other hierarchies are formed by FORM-L models, if any, not tied to the root model by extension but by binding. Each such model that is not extended by any other in the mock-up and that is not a library constitutes the top-level of a specific hierarchy.
- Objects, classes, interfaces, models and partial models have each a separate name space that bears their name.
- The name space of an object, class, interface is unique even through refinement and multiplicity of definition blocks.
- The name space of an interface is injected in the name spaces of its parties.
- The name space of an item is injected in the name spaces of the items that extend it.
- The name space of a model is injected in the name space of models that declare it as a library.
- The name space of a partial model is injected in the name space where it is first declared.
- The name space of an object or template is embedded in the name space of the (partial) model or contract where it is first declared.
- The attributes of an object are in its name space.
- An expression involving one or more automata has a name space embedded in the name space where the expression appears. The names of the automata states are injected in that name space.

```
Class #Division begin // The item of interest is now bps.Division
  Automaton
  [ standby     "Monitoring MPS, ready to produce own power if necessary"
  , active      "Powering channels with own power after an MPS loss"
  , test        "Performing a periodic test"
  , maintenance "Being repaired and not available"
  , failure     "In failure and not available"
  ] #state;
end Division;

Division u;           // The BPS has two redundant divisions u and v
Division v;

Guarantee @noTest
  "No periodic test when one or more divisions are in failure"
  is during (from u.eAlarm until u.state leaves {failure, maintenance})
       or   (from v.eAlarm until v.state leaves {failure, maintenance})
    ensure no (testUCmd or testVCmd) becomes true;
```

`u.state leaves {failure, maintenance}` is an expression where `state` is an automaton. Its states (here, `failure` and `maintenance`) can be used directly and do not need to be qualified by a pathname. A pathname would be necessary if the same expression involves two or more automata of different classes but with states bearing the same name.

- A dummy name (in a selector) is implicitly `private` to the name space of the instruction that declares it.
- Names declared in an unnamed name space are *de facto* `private`, since no pathname from outside can reach them.

6.3 Special Names

Keyword `me` (resp. `Me`) is a proxy for the name of the object (resp. class) within its name space.

Keyword `parent` (resp. `Parent`) is a proxy for the name of the object (resp. class) one level up in the hierarchy with respect to the name space of the item being defined.

A number of keywords are used to name the attributes of objects.

6.4 Grammar

```
name:   (determiner? NAME) | 'me' | 'Me' | 'parent' | 'Parent' ;
```

NAME is the name of an item and is a sequence of letters, digits, underscores and ° characters, and begin with a letter or the ° character.

```
determiner:   ('!' | '#' | '@' | '?' | '^' | '~') ;
```

Determiners are optional and may be specified only for objects or templates. They apply to the name they precede.

```
attribute:
   'value'          | 'previous' | 'next'   | 'default'
|  'quantity'       | 'unit'     | 'scale'  | 'offset'
|  ('derivative'    | 'integral' | 'inPIntegral')
        ('[' INT ']' | '[' NAME ( ',' INT)? ']')?
|  'memory'
|  'untested'       | 'satisfaction' | 'violation'
|  'inPValue'       | 'inPUntested'  | 'inPSatisfaction'  | 'inPViolation'
|  'eSatisfaction'  | 'eViolation'   | 'eInPSatisfaction' | 'eInPViolation'
|  'satisfied'      | 'violated'
|  'pfd'
|  'rate'           | 'clockValue'
|  'substitution'   | 'concretisation'  | 'justification'
|  'clock'          | 'identity' ;
```

```
pathname:   name ('.' name)* ('.' attribute)*
|           attribute        ('.' attribute)* ;
```

7 Objects

In FORM-L, the modelling items that express dynamic behaviours are the *objects*. They may be divided into *valued objects* (i.e., *variables*, *events*, *sets* and *properties*) and *non-valued objects*. This section presents what is common to all. The next section presents the notion of *class*, which is a *template* that factorizes what is common to similar objects. Then the five following sections present what is specific to each category of object.

7.1 Features

All objects (including valued objects) may have *features*, in the form of other, embedded objects. A *direct feature* is a feature embedded directly in the object. An *indirect feature* is a feature embedded in a direct or indirect feature of the object. Term *feature* covers both direct and indirect features.

An object is an instance of one or more classes and has all the features specified by its classe(s), plus possibly some of its own.

7.2 Value, Values & Identity

The behaviour of an object is determined and characterized by its built-in *attributes* and its *features*. The main characteristic of a valued object is its *value*, which is represented by its *value* attribute. It may be an instance of multiple classes, but one and only one of them is a *valued class* and determines the type of the value. (Reminder: the value of an event is the set of its past occurrences, the value of a set is its membership at any given instant, and the value of a property is represented by the three Boolean attributes *untested*, *satisfaction* or *violation*). On the contrary, a non-valued object (i.e., an object that is not an instance of any valued class) does not have a value of its own.

A value generally can evolve in time along an *operational case*, unless it is declared to be ***constant*** or ***fixed***. When the name of a valued object appears in an expression without any qualifying attribute or feature, it stands for its *value*.

Note. In this document, term *value* can have two different meanings. In what follows, *value* stands for a *function of time*, and *value* for an *instantaneous value*.

The complete behaviour of an object is determined by its own *value* (in the case of valued objects) and by the *value* of all its valued features. It is represented by attribute *values*. That attribute can only be compared for equality or difference, and be monitored for changes. An object is ***constant*** if and only if its *values* are ***constant***. It is ***fixed*** if and only if one or more of its *value* are ***fixed***, and all the other are ***constant***.

Each object (be it named or not, be it static or dynamic) has an automatically determined ***fixed*** *identity* attribute that is unique within a mock-up. In particular, the *identity* of deleted objects is not reused. *Identity* can only be compared for equality or difference, and its nature is implementation dependent.

There are thus three different kinds of object comparison:

- Comparison of *identity*.

```
Class ValuedClass extends Integer begin
  Boolean booleanFeature;
  Real    realFeature;
end ValuedClass;

ValuedClass {} set;

Assumption a1 is
  for all x1, x2 of set such that x1.identity <> x2.identity
  ensure ...;
```

- Comparison of *value* (only in the case of valued objects).

```
Assumption a2 is
  for all x1, x2 of set such that x1 <> x2
  ensure ...;
```

```
Assumption a3 "identical to a2" is
  for all x1, x2 of set such that x1.value <> x2.value
  ensure ...;
```

- Comparison of *values*.

```
Assumption a4 is
  for all x1, x2 of set such that x1.values <> x2.values
  ensure ...;
```

7.3 Determiners

A *determiner* is an optional piece of information that may be attached to an object. If specified, it characterises certain aspects of the object and helps understand the intentions of model authors: it does not affect behaviour *per se*. Determiners are denoted with specific marks (`! # @ ~ ? ^`, see *Figure 7-1*). When an object has multiple determiner specifications (be they specified directly for the object or indirectly through some of its classes), they must not contradict one another.

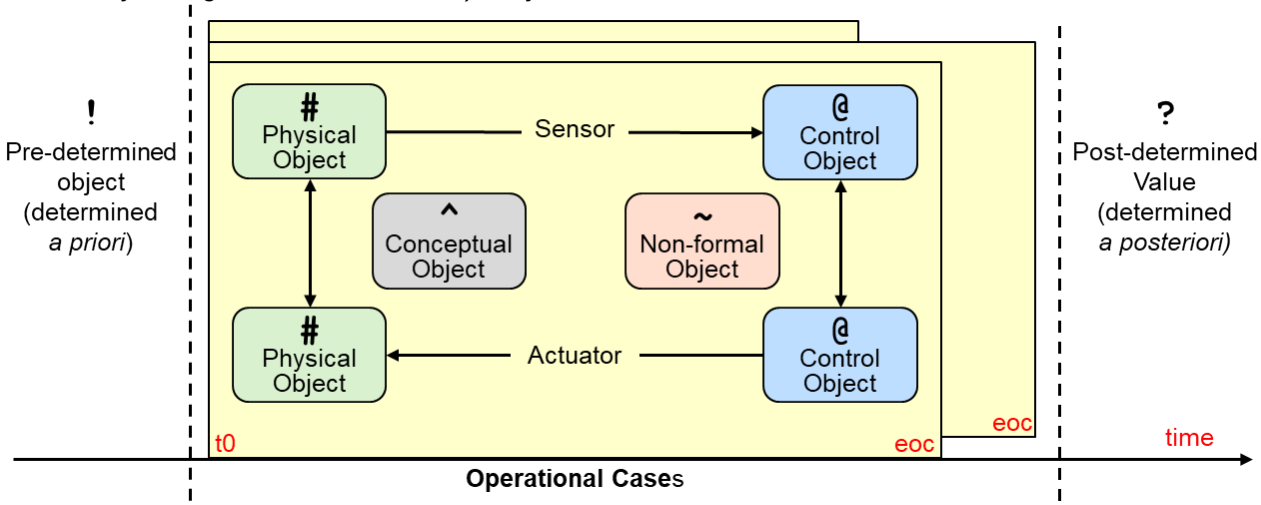


Figure 7-1
The six determiner marks.

The `!` determiner indicates that the object is *pre-determined*, i.e., that its behaviour is or will be fully determined during design and before system operation: a pre-determined object is composed exclusively of pre-determined features, a pre-determined valued object has a fully pre-determined value at each instant of any operational case, that value being the same for all operational cases. Constants are generally but not necessarily pre-determined (some are determined *a posteriori* and have a `?` determiner), and a pre-determined object is not necessarily constant (see the following example). Pre-determined objects must be defined based only on literals and other pre-determined objects, and must not involve random functions. Conversely, an object defined based only on literals and other pre-determined objects, and not involving random functions, is implicitly pre-determined.

```
constant Duration !lifeTime is deferred;   // Defined elsewhere
Real inputSignal is sin (2*pi*time);
Event periodicClock is every 2*s;
```

The `!` determiner attached to **`lifetime`** notifies engineers who will later provide an explicit definition that it must be a pre-determined constant, not a constant determined *a posteriori* (in which case it would have a `?` determiner).

The determiners for **`inputSignal`** and **`periodicClock`** are not specified, but as their definitions are based only on literals (`2`), pre-determined constants (`pi`, `s`) and pre-determined variables (`time`), and do not involve random functions, one can infer that they are also pre-determined.

The `#` determiner indicates that the object is a so-called *physical object*, whereas the `@` determiner indicates that the object is a so-called *control object*. Physical objects represent actual entities and observable and measurable aspects of the real, physical world. Control objects represent the 'beliefs' or the 'desires' that some parts of the system or its environment may have on the physical world, and

are obtained either by measurement of physical aspects or by calculation based on other control or pre-determined objects.

```
Money @bill;                // Desired amount of money
Money #payment;             // Actual amount of money
```

Physical and control objects are *concrete objects*. Their definitions should not be based on *conceptual* or *a posteriori* objects. An object defined based only on literals, pre-determined objects and physical objects is implicitly a physical object. Similarly, an object defined based only on literals, pre-determined objects and control objects is implicitly a control object.

```
// ---- reset ------------------------------------------- reset ----
Client begin
  Boolean #powered      "Whether powered at all" is deferred;
  Boolean #poweredByBps "Whether powered by BPS" is deferred;
  Assumption consistency is during poweredByBps ensure powered;
end Client;

#reset is
  after (operator.eReset while mps.ok) within 10*s
  for all x of Client
  achieve (x.powered and not x.poweredByBps) or not mps.ok;

// ---- alarm ------------------------------------------- alarm ----
Event @eFailure "When failure is detected"              is deferred;
Event @eAlarm   "Signal sent to operator when eFailure" is deferred;
alarm is
  after eFailure within 200*ms
  achieve eAlarm;
```

This example is taken from the *Backup Power Supply* study: it highlights the difference between physical and control objects, and shows why it is important to make the distinction.

powered and **poweredByBps** are physical objects (**#** determiner): that means they represent actual, physical states, not beliefs of some control element in the system.

On the contrary, **eFailure** and **eAlarm** are control objects (**@** determiner). This means that:

- **eFailure** represents the belief of a control element (**"When a failure is detected"**), whereas a **#** determiner would have meant an actual failure (**"When a failure occurs"**), with a very different meaning for requirement **alarm**.
- **eAlarm** represents a control signal that will need to be converted (by a human-system interface element external to the *BPS*) into a physical signal perceptible by the operator. A **#** determiner would have meant that the *BPS* is in charge of that physical signal and that the human-system interface element is part of it.

Requirement **alarm** is defined based only on control objects (**eFailure** and **eAlarm**) and is thus a control requirement. Requirement **reset** is defined based on physical objects (**mps.ok**, **powered** and **poweredByBps**), on a pre-determined object (**Client**) but also on a control object (**operator.eReset**): the **#** determiner indicates that it is considered a physical requirement.

The **^** determiner indicates that an object is a *conceptual object* representing an abstract notion that cannot be observed, measured or calculated. They can later be redeclared as pre-determined, physical or control objects by some subsequent extension model. (The reverse, redeclaring a pre-determined, physical or control object as a conceptual object is not allowed.) They may be used to express high-level properties, but as these would also be conceptual, they will need at some point in the engineering process to be associated (through the *concretisation* attribute) with concrete properties, as shown in Section 4.9 - *Step-By-Step Solutions in Top-Down Approaches.*

```
Client begin
  ...
  Percentage ^tolerance "Instantaneous tolerance to loss of power"
  begin
    value is deferred;
    Assumption range is ensure value in [0, 100]*perCent;
  end tolerance;
end Client;
```

tolerance represents a conceptual notion that can be neither measured nor observed. It could theoretically be calculated when the client is known in sufficient detail, but here, model authors have chosen to consider it as a conceptual object. Later in the engineering process, if a rigorous formula is found to calculate it, it could be redeclared as a concrete object.

The **?** determiner applies to *a posteriori* constants, i.e., values that are unknown during the whole course of individual operational cases and that can be determined only *a posteriori* by the analysis of the outcomes of large, appropriate sets of legitimate cases. This is for example the case of *maximal merit values* (see Section 4.14 *Design Optimisation, Agile Approaches*) and of system-level failure probabilities (see Section 4.6 *Coordination of Different Disciplines*). These values may be used to express objectives and requirements, but these can themselves be evaluated only *a posteriori*.

```
Requirement repower.?°pfd is
  from needed becomes true during 2*mn
  ensure repower.pfd < 1e-4;

Requirement repower.?fro is
  during needed except first 2*mn
  ensure repower.eViolation.rate < 1e-3/h;

Objective repower.^noSpurious is
  for all x of Client
  during not needed
  ensure no x.poweredByBps becomes true;

Requirement repower.?sar is repower.noSpurious.eViolation.rate < 1e-3/year;
```

pfd (probability of failure on demand) and *rate* are *a posteriori* attributes.

The **~** determiner applies to objects (generally, but not necessarily, properties) that are expressed only in natural language and will not be formalised: mock-up verification tools will issue a warning if and only if they are formalised, contrary to all other objects, for which tools issue a warning if they are not. Such objects are provided for information to readers but do not participate in simulation or formal verification, and cannot be used in formal expressions. They must be redeclared with other determiners if model authors eventually decide to formalise them.

```
Contract trackConditions begin
  party track begin
    Guarantee ~smooth        "Flat, no pothole";
    Guarantee ~uniform       "Adherence constant in time, and homogeneous
                              over complete track surface";
    Guarantee ~goodCondition "Ambient conditions have no influence";
    Guarantee !closedLoop;
    Guarantee !continuous;
    Guarantee ~noIntersection "The track does not intersect itself";
  end track;
end trackConditions;
```

This example is taken from the **roboCar** study. This broadcast contract is offered by object **track**, the racing track on which the **roboCar** runs. **smooth**, **uniform**, **goodCondition** and **noIntersection** will not be formalised, whereas **closedLoop** and **continuous** will.

Automatically calculated *attributes* have the same determiner as the object to which they are attached, except property attribute *pfd* which is an *a posteriori* value. Attribute *rate* is also an *a posteriori* value when applied to property event attributes such as *eViolation*..

The determiner of a feature is not necessarily the same as the one of its *parent* object.

The set of all instances of a class is implicitly pre-determined when there is no dynamic object creation.

When a class specifies a determiner, all its instances and extension classes have that determiner.

Though determiners are optional, it is generally a good practice to specify them whenever that makes sense: static analysis can then issue a warning if a definition is not consistent with the specified determiner, thus reducing the potential for modelling errors.

7.4 Clocks & Time Domains

By default, an object is placed in the *continuous time domain*. It may be placed in a *discrete time domain* by defining its `clock` attribute.

Classes may also define a `clock`, which then applies to all their instances. If an object is an instance of multiple classes with their own `clock`, and / or if it defines its own `clock`, then the intersection of all these `clock` is applied.

When an object is placed in a discrete time domain, the `clock` applied to its features is the intersection of the object's and the feature's `clock`.

7.5 Identity

The notion of set of objects implies the notion of object `identity`, which is a fixed attribute assigned to each object (be it named or not), the value of which is unique within a mock-up and all along an operational case (i.e., the `identity` of a deleted object is not reused). Identities can only be compared for equality or difference, and their nature is implementation dependent.

7.6 Object of Interest, Objects of the Environment

An object may be declared as the `main` focus of interest of a model. Others may be declared as `external`, in which case they are just part of the environment of the `main` object and do not need to be described and modelled in full detail: only what concerns the `main` object and its features needs to be. Objects that are neither `main` nor `external` are typically features of the `main` object.

7.7 Declaration

An object declaration specifies:

- Whether it is `private` to the parent object, class or partial model within which it is declared.
- Whether it is the `main` object of the model or an `external` object, or neither. Normally, the `main` object is not `private`.
- Its *variability*, i.e., whether it is `constant` (its `values` attribute does not change in time and is the same for all operational cases), `fixed` (its `values` attribute does not change in time but may be different for different operational cases) or neither.
- Its *classes*. An object is a direct instance of one or more classes, but at most one can be a *valued class* (i.e., a variable, event, property or set class, or one of their extensions). Then, the object is a *valued object*, and its `value` is of the type of that valued class. If none are valued classes, then the object is a non-valued object. A class may be specified by a pathname, but more generally by a class expression. Classes that have parameters must have a corresponding list of *arguments*.
- An optional *determiner*, which must be consistent with those possibly specified by the classes.
- Its *name*.
- An optional natural language *description*.
- An optional *global assignment* or *definition block*.

```
PushButton              @reset;          // Control information on a PushButton
Percentage              ^tolerance;      // Conceptual Percentage
fixed Duration          #phase;          // Fixed physical Duration
Automaton [on, off]     @command;        // Control automaton without the
                                         // mediation of an enumeration
Length/Duration         speed;           // Speed without a determiner

Event                   eEndOfLifetime;
private Event           @eFailure;

Client {3..10}          clients          "Set of 3 to 10 instances of Client";
AcVoltage  {# 3}        electricPhases   "Set of three actual voltages";
Duration {fixed # 3}    phases           "Set of three actual, fixed phases";
```

```
Requirement #repower
  "During the lifetime of the installation, when the MPS is not
   available, the BPS shall ensure that none of its clients exhaust
   their tolerance level to loss of power";

Object motor;                          // Non-valued object
private Pump pump;                     // Private instance of Pump
(Motor, Pump) motoPump;                // Instance of both Pump and Motor
```

Examples of object declarations without a definition part.

7.8 Definition

The *definition* of an object specifies its composition and its behaviour. Composition is specified by the declaration of the object's *features*. Behaviour (including *concretisation*, *substitution* and *justification*) is specified with *instructions* (see Section 18).

The composition and behaviour of an object are specified in part by its classes. In addition, its own definition may:

- Define or refine the definition of features inherited from its classes. In particular, it MUST provide a definition for those defined by its classes as **specific**.
- Declare, redeclare, define and refine its own specific features.
- Declare, redeclare, define and refine embedded classes, which can be instantiated only within the definition of the object.
- Specify its own instructions.

A definition may be provided within or separately from a declaration. For valued objects that have no features, a single *global assignment* (keyword ***is*** followed by an expression) can be sufficient. Otherwise, the definition may be provided in one or more *definition blocks*, to reflect the step-by-step nature of *refinement* (see Section 7.10). A definition block for an object or a class may be specified within a definition block of another one to signify that it is that other one that is responsible for these features.

```
Real maxFailureProbability is 1 - exp(-k*time);
```

Example of combined declaration and global definition.

```
Real maxFailureProbability;                    // Declaration

...

maxFailureProbability is 1 - exp(-k*time);     // Global definition
```

This example of separate declaration and global definition is equivalent to the preceding one.

```
Power ^powerDemand is deferred;
```

Example of combined declaration and global definition. ***deferred*** means that a more precise definition of **powerDemand** will be provided later in the engineering process, in some future extension model or through a binding.

```
AcVoltage #voltage "Electric tension at the MPS terminals" begin
  value is deferred;
  Assumption range is ensure value in [0, 260]*V;
end voltage;
```

This example of combined declaration and definition block (bracketed with ***begin*** and ***end***) is taken from the *Backup Power Supply* study.

```
AcVoltage #voltage "Electric tension at the MPS terminals"; // Declaration
...
voltage begin                                               // Definition
  value is deferred;
  Assumption range is ensure value in [0, 260]*V;
end voltage;
```

This example of separated declaration and itemised definition is equivalent to the preceding one.

```
AcVoltage #voltage "Electric tension at the MPS terminals"; // Declaration
...
voltage: value is deferred;                                 // Definition
...
voltage: Assumption range is ensure value in [0, 260]*V;    // Definition
```

Here, the definition is not only separated from the declaration, it is given in two separate definition blocks. As each of these blocks specifies only one statement, they do not need a ***begin end*** brackets and are content with a **:** mark. This example is equivalent to the two preceding ones.

```
Boolean @resetCmd "A control variable" begin
  DcVoltage #physicalSignal "Physical aspect of the control variable" begin
    value is deferred;
    Assumption  init  is when t0 ensure value = 0*V;
    Requirement range is ensure value in [0, 5]*V;
    Requirement shortTransients is
      during value in [1, 4]*V
      ensure inPTime < 1*ms;
  end physicalSignal;

  during           physicalSignal in [0, 1]*V define value is false
  otherwise during physicalSignal in [4, 5]*V define value is true
  otherwise                                  define value is previous;
end resetCmd;
```

In this example, control variable **resetCmd** has an associated physical counterpart modelled as a feature named **physicalSignal**.

Within the definition blocks of an object, special name **me** denotes the object itself, and special name **parent** denotes the object of which it is a direct feature, if any.

```
Object bps begin
  Contract mainPower (mps);           // Acknowledgment of contract
  Contract backupPower (me, Client);  // Acknowledgment of contract
  Contract hsi (me, operator);        // Acknowledgment of contract
  ...
end bps;
```

This example is taken from the *Backup Power Supply* study. Here, object **bps** is declared and given a definition.

```
bps begin
  ...
  Client begin
    Boolean #powered      "Whether powered at all" is deferred;
    Boolean #poweredByBps "Whether powered by BPS" is deferred;
    Assumption consistency is during poweredByBps ensure powered;
  end Client;
  ...
end bps;
```

This example is also taken from the *Backup Power Supply* study: it shows a definition block for class ***Client*** inserted in a definition block for object **bps**, highlighting the fact that it is **bps** that is responsible (and will give a definition) for ***Client*** features **powered** and **poweredByBps**..

At certain times along an operational case, e.g., due to *assignments*, a `value` is *fully constrained*, i.e., at each instant it can have one and only one value. At other times, it may be *under-constrained* (with *assumptions* or when there is no constraints at all): its value could be any one among different possible values (see *Figure 7-2*). What must be avoided are *over-constraints* when, at certain times, no value can possibly comply with all specified assignments and assumptions. When analysis or simulation reaches such a situation, an *exception* is raised.

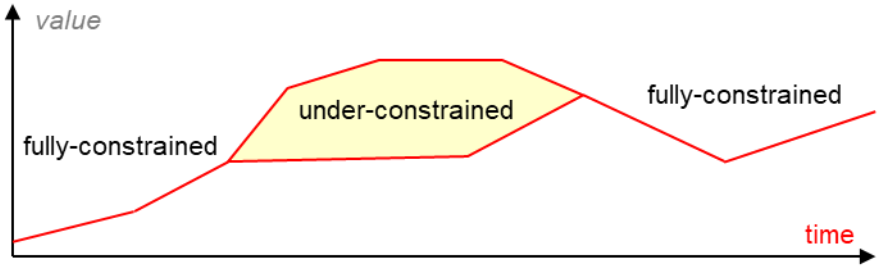


Figure 7-2
Alternance of periods where a `value` is fully constrained and under-constrained.

7.9 Deferment

When developing a model at an early engineering stage, one does not necessarily know everything about a given object: there may be instants or periods of time where some of its `values` cannot be defined yet but must be so at a later engineering stage, in some extension model. That is indicated by keyword `deferred`.

```
Client: Boolean #poweredByBps is deferred;
```

The value of **poweredByBps** is currently unknown but will be determined at some later engineering stage.

```
Boolean @resetCmd begin
  when t0 define value is false
  otherwise      value is deferred;
end resetCmd;
```

The value of **resetCmd** at `t0` is known (it is `false`), but not after `t0`.

```
Class @PushButton extends Boolean begin
  when t0 define value is false;
  ...
end PushButton;

PushButton testUCmd "Start periodic test of bps.u" is deferred;
```

The value of **testUCmd** at `t0` is known (it is `false`, as specified by its class, `PushButton`), but not after `t0`. Keyword `deferred` here applies to instants where `PushButton` does not specify a value.

7.10 Refinement

The *refinement* of an object already declared may provide new definition blocks for that object.

This example is taken from the *Backup Power Supply* study. Here, the **mps** object is just declared in the **BpsIntroduction** model, without any definition block.

```
external Object mps "Main electrical Power Supply";
```

mps is then refined in the **BpsReference** model (an extension of **BpsIntroduction**) with two definition blocks, one of which is within a definition block of another object, **bps**.

```
refined mps:
  AcVoltage #voltage "Electric tension at the MPS terminals" begin
    value is deferred;
    Assumption range is ensure value in [0, 260]*V;
  end voltage;
end mps;
```

```
bps begin
  ...
  mps: Boolean #ok is voltage >= Client.minimumVoltage;
  ...
end bps;
```

This example is taken from a *Car Wipers* study. The definition of **active** specifies the requirements regarding when the wipers must be active and when they must not.

```
Automaton [on, off] @command "Commands to wipers";

Boolean #retracted "Whether wipers are fully retracted":
  Assumption is during value = true achieve inPTime > 0.2*s;

Boolean #active "Whether wipers are wiping" begin
  value is deferred;
    when t0
    Assumption is ensure value = false
  otherwise
    during command = on except first 0.1*s    // Wipers need time to react
    Requirement wipe is ensure value = true
  otherwise
    during not retracted                      // Active when not retracted
    Requirement dontStop is ensure value = true
  otherwise
    during retracted and command = off except first 0.1*s
    Requirement rest is ensure value = false // Rest only when retracted
  otherwise
    during command = off
    Requirement noSpurious is ensure no value becomes true;
end active;
```

The following refinement of **active** specifies a solution to these requirements:

```
refined active begin
    during command = on except first 0.05*s
    define value is true
  otherwise
    when retracted becomes true while command = off
    define next is false
  otherwise
    define next is value;
end active;
```

A refinement may also add precision to the declaration of the object, and may:

- Provide additional natural language descriptions.

This example is taken from the *Backup Power Supply* study. In model **BpsIntroduction**, **alarm** is given a first natural language description.

```
Requirement #alarm
 "When the BPS detects failures that could prevent it from satisfying
  'repower' or 'reset', it shall send an alarm to the operator";
```

Then, in model **BpsReference**, **alarm** is refined and given an additional description that complements and adds precision to the first one.

```
refined alarm "with a 200 ms time margin";
```

- Specify a determiner when there was none, of confirm an existing determiner.

This example is inspired from the *Backup Power Supply* study. In model **BpsIntroduction**, **alarm** is declared without a determiner.

```
Requirement alarm;
```

Then, in model **BpsReference**, **alarm** is given a **#** determiner.

```
refined #alarm;
```

- In the case of a finite state automaton, transform one or more atomic states into sub-automata (see Section 9.5.2).
- In the case of a set, place additional constraints on the set indicator (see Section 11).
- Define the ***clock*** attribute, but if the object being refined has already defined it, then the refinement can only restrict the existing ***clock***, i.e., the refined ***clock*** must be a subset.

7.11 Redeclaration

As engineering progresses, some previous decisions may need to be called into question in ways that are not consistent with those previous decisions. This is the case when one realises that a requirement is not achievable and can only be an objective, or when one realises that an object initially thought to be physical (with a **#** determiner) is in fact and abstract one (with a **^** determiner). Such changes of mind and breaches of consistency must be done with caution, and must be announced with keyword ***redeclared*** instead of ***refined***.

This example is taken from the *Backup Power Supply* study. In model **BpsIntroduction**, **repower** is first declared as a physical requirement.

```
Requirement #repower;
```

Then, in model **BpsReference**, one realises that it depends on **^tolerance**, which is an abstract object. Therefore **repower** is also abstract.

```
redeclared ^repower;
```

Lastly, in model **BpsPsaReference**, one realises that it is unachievable and cannot be a requirement.

```
redeclared Objective repower;
```

One can also substitute one or more classes of the object with one of their extensions, or specify additional classes for the object.

This example is inspired from the *Backup Power Supply* study. **resetCmd** is first defined as a ***Boolean***.

```
Boolean @resetCmd "Reset command" is deferred;
```

Then, in an extension model, **resetCmd** is redeclared as a ***PushButton***, an extension class of ***Boolean***.

```
Class @PushButton extends Boolean;
redeclared PushButton resetCmd;
```

```
Class Pump;
Pump  pump;
...
Class Motor;
redeclared (Pump, Motor) pump;
```

In this example, object **pump**, initially of class ***Pump***, is redeclared to be both a ***Pump*** and a ***Motor*** (i.e., a pump actuated by a motor).

7.12 Dynamic Creation and Deletion

Whereas objects declared without a temporal locator are *static* (i.e., they exist for the complete duration of every operational cases), to reflect the dynamic composition of certain systems (e.g., certain systems of systems), objects declared under the scope of a *DTL* or *CTL* with keyword ***new*** are created dynamically along an operational case. Under the scope of a *DTL*, a new object is created at each instant of the *DTL* and exists until it is explicitly deleted, using action ***delete***. Under the scope of a *CTL*, a new object is created at the beginning of each period of the *CTL* and exists until the end of the period, unless it is explicitly deleted during the period, also using action ***delete***. Upon creation, dynamically created objects are automatically inserted in the sets of all the instances of their classes. Conversely, they are removed from all sets they belong to when they are deleted.

Named objects are static and cannot be deleted: an exception is raised when that occurs.

```
Class Vehicle extends Object begin
   when eNew + 15*year delete me;
end Vehicle;

Vehicle {} fleet begin
     when t0 define value is {}
   otherwise
     when eNewPurchase define value is previous union new Vehicle;
end fleet;
```

In this very simplistic example, one considers a **fleet** of ***Vehicle***. Vehicles may be purchased (and created) in the course of a case (event **eNewPurchase**) and added to set **fleet**. They are deleted when aged 15 years (*eNew* is the attribute event marking the dynamic creation of the object) and automatically removed from all sets to which they belong.

The features of a dynamically created object do not exist before its creation and after its destruction: their life time covers only the existence of the object. Thus, for the object and its features, attribute *eNew* is the event that marks their creation, and attribute *eDelete* is the event that marks their destruction.

7.13 Functions

A *function* is a template that has parameters other than **time** and that generates a specific object wherever it appears. Even though it is declared in a manner similar to objects, it is not one and it can be used only as a term in an expression. Like for objects, its definition can be given in one or more definition blocks within or separate from its declaration(s),

A function declaration specifies:

- Whether it is ***private*** to the parent object, class or partial model within which it is declared.
- Whether the object resulting from each invocation is ***fixed*** and does not change in time.
- The *class* of the generated objects.
- An optional *determiner*, which must be consistent with the one possibly specified by the class.
- Its *name*.
- A non-empty list of *parameters*. When a function is declared multiple times, the name of each parameter must be the same in all declarations. This is to allow the notation of partial derivatives and integrals (see Section 9.2 *Attributes derivative, integral & inPIntegral*).
- An optional natural language *description*.
- An optional *global assignment* or *definition block*.

Definition blocks for functions are similar to definition blocks for objects, but can declare only properties.

```
Real average (Real p, fixed Duration d) begin
  Guard dPositive is ensure d > 0*s;
  when t0            define value is 0
  otherwise until d define value is p.integral/time
  otherwise          define value is (p.integral -(p.integral delayed d))/d;
end average;
```

Function **average** returns the average value of ***Real*** variable **p** over fixed ***Duration*** **d**.

```
Event @eStartDg;
Event @eStopDg;
State @state;

Boolean @output (Event @evt1, Event @evt2)
  "Binary output of the BPS control system"
  is from evt1 until evt2;

Boolean startDg is
  output (eStartDg, state becomes operational.(standby, *));

Boolean stopDg  is
  output (eStopDg,  state becomes operational.(standby, *));
```

This example is taken from the *Backup Power Supply* study. **output** is a ***Boolean*** function with two control ***Event*** parameters (@ determiner) **evt1** and **evt2**. It is a control function (@ determiner) that is **true** just after each occurrence of **evt1** until (and including) the following occurrence of **evt2**, and **false** otherwise. **startDg** and **stopDg** also have the @ determiner.

```
Event ePeriodicClock (fixed Duration period) begin
  Guard positive is ensure period > 0*s;
  private fixed Duration phase is random (period); // Local variable
  (every period) + phase signal one occurrence;
end ePeriodicClock;

Event ePeriodicClock1 is ePeriodicClock (50*ms);
Event ePeriodicClock2 is ePeriodicClock (50*ms);
Event ePeriodicClock3 is ePeriodicClock (70*ms);
```

ePeriodicClock signals periodic occurrences with a random phase, and has a parameter that is the period of the clock. **ePeriodicClock1** and **ePeriodicClock2** have the same period, whereas **ePeriodicClock3** has a different period. All three have different phases.

A *parameter declaration* is similar to an object declaration, with an optional variability, one or more classes, an optional determiner, a name and an optional natural language description. However, it may also be just the name of an event, in which case each invocation will specify an occurrence of that event, as shown in the following example.

```
external Event eRequest
  "Randomly occurring service requests"
  is deferred;
Event eAcknowledge (eRequest)
  "The parameter is the eRequest occurrence that is acknowledged"
  is deferred;
Requirement acknowledgment
  "Each request occurrence shall be acknowledged within 10 seconds"
  is after eRequest within 10*s
     achieve eAcknowledge (bop);
```

7.14 Grammar

```
Object:
   ( 'main'
   | 'external' variability?
   | ('private'? & 'common'?) variability // common only in class definition
   | ('private'? & 'common')
   | 'private')
   classes
   (determiner? NAME)? // The name of an Assumption may be omitted
   STRING*
   (Behaviour | ';')
;

variability: 'constant' | 'fixed' ;

classes: class | '(' class (',' class)* ')' ;

determiner: '#' | '@' | '^' | '!'| '?'| '~' ;

Behaviour:
   'is' (expr | propertyDef | 'deferred' | 'specific') ';'
                                  // 'specific' only in class definition
| ':' ObjectStatement
| 'begin' ObjectStatement* 'end' (('for' ('all'| 'some'))? NAME)? ';'
;
```

```
ObjectStatement:
   Object
|  ObjectDefinition
|  ObjectRefinement
|  ObjectRedeclaration
|  Class            // See Section 9.6
|  Instruction      // See Section 18.4
|  STRING           // for comment
;

ObjectDefinition: determiner? pathToObject (STRING+ | STRING* Behaviour) ;

ObjectRefinement:
   'refined' classes? determiner?
   pathToObject STRING* (Behaviour | ';')
;

ObjectRedeclaration:
   'redeclared' (classes determiner? | determiner)
   pathToObject STRING* (Behaviour | ';')
;

Function:
   'private'? valuedClass
   determiner? NAME parameters STRING* (Behaviour | ';')
;

valuedClass: class ;

parameters: '(' parameter (',' parameter)* ')' ;

parameter: (variability? classes determiner?)? NAME STRING* ;

pathToObject: pathname ;

pathToClass:  pathname ;

Creation: 'new' determiner? classes (Behaviour | ';') ;
```

8 Classes

Classes are *templates* that can be instantiated into *objects*. The *statements* (declarations, definition blocks and instructions) specified by a class apply to all its instances.

8.1 Predefined Classes

Predefined classes are built-in in the FORM-L language and have no associated statements. These are:

- Predefined *variable classes* ***Boolean***, ***Integer***, ***Real*** and ***String***.
- Predefined *event class* ***Event***.
- Predefined *property classes* ***Property***, ***Assumption***, ***Objective***, ***Requirement***, ***Guarantee*** and ***Guard***.
- Predefined *non-valued class* ***Object***.

8.2 Defined Classes

New classes may be declared and defined by *enumeration*, *extension* or *set class definition*:

- Enumeration creates variable classes the instances of which can take value only within the discrete values listed in the enumeration (see Section 9.5).
- Extension creates classes that have all the statements of the class(es) they extend (their *super classes*), plus statements of their own. It corresponds to the notion of *multiple inheritance* in many object-oriented languages. The instances of an extension class are also instances of its super classes. Features inherited from different super classes and bearing the same name are distinct and must be identified by unambiguous pathnames, unless the super classes inherited them from a single super super class. A class defined by extension can extend at most one valued class and is then a valued class of the same type. A class defined by extension that does not extend any valued class is a *non-valued class*.
- *Set classes* are created by associating an existing class with a *set indicator* (see Section 11).

8.3 Declaration

A class declaration specifies:

- Whether it is ***private***: this applies only if the class is declared as a feature of a *parent* object or class, and means that its scope (where it can be referred to and used) is limited to the definitions blocks of its parent.
- Whether it is ***abstract***: this means that it can be extended but cannot be directly instantiated; only its non-abstract extensions can.
- Whether it is the ***main*** class of a model, which means that its instances are the ***main*** objects (i.e., the systems of interest) of the model.
- Whether it is ***external***: this means that its instances are neither the ***main*** systems of interest nor part of the ***main*** systems. ***main*** and ***external*** are exclusive.
- Whether it is an extension, an enumeration or a set class. By default, it is an extension of ***Object***.
- An optional *determiner*. if present, it applies to all the *instances* of the class.
- Its *name*.
- An optional list of *parameters*.
- If an extension class, its super classes, with a list of *arguments* for each super class that has parameters.
- An optional natural language *description*.
- An optional *global definition* or *definition block*.

```
Class Probability extends Real begin
  Guard range is ensure value in [0, 1];              // Property
end Probability;
```

Class **Probability** extends predefined valued class *Real* by specifying that the *value* of its instances cannot be out of range **[0, 1]**, otherwise the model is inconsistent.

```
Class @PushButton extends Boolean "Boolean HSI input device" begin
  when t0 define value is false;                          // Assignment

  Requirement notTooBrief is                              // Property
    after value becomes true during 0.5*s
    ensure value = true;

  Requirement noTooLong is                                // Property
    after value becomes true within 2*s
    achieve value = false;
end PushButton;
```

This example is taken from the *Backup Power Supply* study. **PushButton** extends predefined valued class *Boolean* by specifying that all its instances are initially **false** and by guaranteeing that they will be **true** neither too briefly nor for too long.

```
Class Division begin
  ...
end Division;

Class RedundantSystem is Division {2..4};
```

This example is inspired from the *Backup Power Supply* study. **Division** is implicitly an extension of *Object*. A **RedundantSystem** is a set of 2 to 4 **Division**.

8.4 Parametered Classes

When a class has parameters, each of its instantiations and extensions must specify a corresponding list of arguments as shown in the following examples.

```
Class Output1 (Event @evt1, Event @evt2) extends Boolean
  "Binary outputs of the BPS control system"
begin
  value is from evt1 until evt2;
end Output1;

Output1 (eStartDg, state becomes operational.(standby, *)) startDg1;
Output1 (eStopDg,  state becomes operational.(standby, *)) stopDg1;
```

This is an equivalent variation of one of the examples of Section 7.13. Since its definition is based on **evt1** and **evt2** which both have the **@** determiner, **Output1** also has the **@** determiner, and so do its instances **startDg1** and **stopDg1**.

Choosing between this style and the one based on functions is a matter of taste.

```
Class Output2 (Event @evt1, Event @evt2) extends Boolean begin
  from evt until evt2 define value is true;
end Output2;
```

The difference with *Output1* is that the *value* of *Output3* is unconstrained after **evt2** until the next occurrence of **evt1**.

```
Class Output3 (Event evt1, Event evt2, fixed Duration phase)
  extends Boolean
begin
  clock is (every 50*ms) + phase;                         // Assignment
  value is from evt1 until evt2;                          // Assignment
end Output3;
```

Here, each instance of **Output3** is placed in a discrete time domain, the ticks of which are periodic, with a 50 ms period and a specified **phase**.

```
Class PositionSensor (Breaker breaker) extends Component begin
  Boolean open is breaker.functionalState.open delayed 10*ms;
end PositionSensor;

Breaker mpsBrk;
Breaker dgBrk;

PositionSensor (mpsBrk) mpsBrkPos1;
PositionSensor (mpsBrk) mpsBrkPos2;
PositionSensor (dgBrk)  dgBrkPos1;
PositionSensor (dgBrk)  dgBrkPos2;
```

This example is also taken from the *Backup Power Supply* study. **PositionSensor** is a parametered class: each of its instances is attached to a specific ***Breaker***. The definition of **PositionSensor.open** is a simplistic one, just for the sake of the example. A more realistic model would include a less deterministic response time and the effects of failures (where the sensor would be stuck open or stuck closed for example).

The extension of a parametered class must specify a corresponding list of arguments for it.

```
Class Output4 (Event @evt, State @state)
  extends Output3 (evt, state becomes operational.(standby, *))
begin
  default is false;
end Output4;
```

Though functions and parametered classes have similarities, there is an important difference: whereas class instantiation results in new objects (which can be included in sets of objects), function invocation results in new value trajectories (which can be included in sets of values but not in sets of objects – see Section 11).

8.5 Definition

A class definition may be specified using multiple definition blocks and comes in addition to what is inherited from its super classes, if any. It may contain all what an object definition may. In addition:

- It may declare some features to be ***common***, i.e., they are shared by and are the same for all the instances of the class. ***common*** features are defined only by the class (not by instances).
- It may declare some attributes and features to be **specific**, in which case they must be defined by each instance of the class, possibly through the mediation of an extension class. A feature cannot be both ***common*** and **specific**.

```
Class Client begin
  common constant AcVoltage !minimum is deferred;
  Percentage ^tolerance is specific;
end Client;
```

This example is taken from the *Backup Power Supply* study. Class **Client** implicitly extends class ***Object*** and states that each of its instances has a state variable named **tolerance** of class ***Percentage***. Constant **minimum**, of class ***AcVoltage***, is the same for all instances and is thus declared ***common***.

```
abstract Class #Component begin
  Automaton [ok, [primary, secondary] failure] #status is deferred;
end Component;
```

This example is also taken from the *Backup Power Supply* study. **Component** is an ***abstract*** extension of ***Object***. Its instances have a feature named **status**, which is just declared and needs to be defined somewhere else in the model. The **#** determiner specifies that all its instances (and also all the instances of its extension classes) represent actual, physical objects.

```
Class          Breaker   extends Component;  // Circuit breakers
abstract Class MSensor   extends Component;  // Measurement sensor
Class          Voltmeter extends MSensor;    // Voltmeters
```

Here, abstract and non-abstract extensions of class ***Component*** are just declared (and not defined yet). They all inherit the **#** determiner of ***Component***.

```
Breaker begin
  Automaton [open, closing, closed, opening] #functionalState;
end Breaker;

Breaker mpsBrk; Breaker dgBrk;
```

Here, since it has already been declared, class **Breaker** is just defined. Its instances have feature **functionalState** in addition to feature **status** (provided by super class ***Component***).

```
Class Supplier extends Object begin
   Consumer {} #clients is specific;
   Money {}    @clients.bill;
end Supplier;
```

Class **Supplier** declares **clients** as a set of ***Consumer***, each instance of **Supplier** having its specific set of **clients**. **clients.bill** is a set of variables with a one-to-one relationship with **clients**. From a ***Consumer*** instance standpoint, **bill** is a set with one member for each of its **suppliers**.

8.6 Refinement

The *refinement* of a class may complete its natural language description, specify or modify (within authorised limits) the determiner, refine existing features, and add new parameters, features and statements.

```
Model BpsIntroduction begin
  external Class #Client
    "Any system powered by the backed-up panel"
    extends Object;
end BpsIntroduction;

Model BpsReference extends BpsIntroduction begin
  refined Client begin
    common constant AcVoltage !minimumVoltage is deferred;
    Percentage ^tolerance begin
      value is deferred;
      Assumption range is ensure value in [0, 100]*perCent;
    end tolerance;
  end Client;
end BpsIntroduction;
```

This example is taken from the *Backup Power Supply* study. Here, class **Client** is first introduced in the **BpsIntroduction** model. It is then refined in the **BpsReference** model, which is itself an extension of the **BpsIntroduction** model.

Instances that existed prior to the refinement of some of their classes have all the additional information and features specified, but to specify instance-specific properties or attributes, they must themselves be refined individually.

When a class refinement adds parameters, it specifies only the new ones as shown in the following example.

```
Model BpsReference extends BpsIntroduction begin
  ...
  external Class #Client;
  ...
end BpsReference;
```

Here, class ***Client*** has no parameters.

```
Model BpsRedundancy1 extends BpsReference begin
  ...
  refined Client
  ( constant Rate     !kcMax
  , constant Rate     !krMin
  , constant Duration !latency
  );
  Client (2.5*perCent/s, 10*perCent/mn, 5*s) client1;
  Client (4.0*perCent/s,  8*perCent/mn, 8*s) client2;
  ...
end BpsRedundancy1;
```

This first refinement of *Client* adds three parameters (**kcMax**, **krMin** and **latency**, see *Figure 4-7*).

```
Model BpsRedundancy2 extends BpsRedundancy1 begin
  ...
  refined Client
  ( constant Power    !initial
  , constant Power    !stabilised
  , constant Duration !settleTime
  );
```

This second refinement of *Client* adds three new parameters (**initial**, **stabilised** and **settleTime**, see *Figure 4-8*) that come in addition to and after the three already specified by **BpsRedundancy1**. They must be explicitly defined for the two pre-existing instances (**client1** and **client2**).

```
  refined Client (12*kW, 5*kW, 4*s) client1;
  refined Client ( 8*kW, 3*kW, 3*s) client2;
```

New instances are declared with the 6 parameters.

```
  Client (5.0*perCent/s,  6*perCent/mn, 2*s, 15*kW, 6*kW, 3*s) client3;
  ...
end BpsRedundancy2;
```

When a model extends other models which independently refine a class with their own parameters, the consolidated list of parameters merges the initial lists, in the order given by the model extension list.

```
Model BpsTolerance extends BpsReference begin
  ...
  refined Client
  ( constant Rate     !kcMax
  , constant Rate     !krMin
  , constant Duration !latency
  );
  ...
end BpsTolerance;

Model BpsPower extends BpsReference begin
  ...
  refined Client
  ( constant Power    !initial
  , constant Power    !stabilised
  , constant Duration !settleTime
  );
  ...
end BpsPower;

Model BpsRedundancy extends BpsTolerance, BpsPower begin
  //       kcMax      krMin      latency   initial stabilised settleTime
  Client (2.5*pC/s, 10*pC/mn, 5*s,      12*kW,  5*kW,       4*s) client1;
  Client (4.0*pC/s,  8*pC/mn, 8*s,       8*kW,  3*kW,       3*s) client2;
  Client (5.0*pC/s,  6*pC/mn, 2*s,      15*kW,  6*kW,       3*s) client3;
end BpsRedundancy;
```

BpsRedundancy extends **BpsTolerance** and **BpsPower**, which independently refine class ***Client*** with their own parameters.

8.7 Grammar

```
class:
   'Boolean'   | 'Integer'     | 'Real'       | 'String'       |   'Event'
|  'Property'  | 'Assumption'  | 'Objective'  | 'Requirement'  | 'Guarantee'
|  'Guard'     | 'Object'
|  pathToClass arguments?
|  quantityClass                  // Quantity class expression, see 9.6
|  'Automaton' states             // Enumeration    expression, see 9.6
|  setClass                       // Set class      expression, see 11.6
;

arguments:  '(' expr (',' expr)* ')' ;

Class: ('main'? | ('abstract'? & 'external'?)
(  ExtensionClass        // See 9.6
|  QuantityClass         // See 9.6
|  EnumeratedClass       // See 9.6
|  SetClass              // See 11.6
) ;
```

9 Variables

With *events*, *variables* are the most basic of FORM-L items. A variable has a `value` that may change in time and that represents a continuous or discrete state. Each variable has a *class* that specifies the nature of its `value` and possibly some features and aspects of its behaviour.

9.1 Attributes value, previous & next

All variables have the three following attributes:

- `value` denotes the value of the variable at the current instant.

```
Duration lifetime is 60*year;          // Installation's lifetime
```

This example of global assignment (keyword `is`) to **`lifetime`** is taken from the *Backup Power Supply* study. It specifies that the `value` is always (no temporal locator) 60 years (a constant expression, implicit **`!`** determiner).

```
Percentage ^tolerance begin
  when t0 define value is 100*perCent;
  Assumption range is ensure value in [0, 100]*perCent;
end tolerance;
```

This example of declaration with a definition is also taken from the *Backup Power Supply* study. The first statement is an assignment instruction specifying that at initialisation (`when t0`), the `value` of **`tolerance`** is 100% (`define value is 100*perCent`). The second statement (`Assumption` **`range`**) is an assumption (i.e., a particular kind of property) specifying that at all times (no temporal locator), the `value` stays in range 0-100% (`ensure value in [0, 100]*perCent`).

At initialisation, **`tolerance`** is fully constrained. After initialisation, it is under-constrained. The two statements overlap in time, but that is acceptable since they do not contradict one another.

It is `value` that is used when a variable is defined or used without any explicit mention of attribute.

- `previous` denotes the value of the variable immediately before the current instant. If the variable is in the *continuous time domain*: `previous` (t) = limit `value` (t-dt) dt → 0+

  If the variable is in a *discrete time domain*, `previous` denotes the value of the variable at the clock tick immediately and strictly before the current instant, or the value at `t0` if there is no such tick. Since it belongs to the past, `previous` cannot be modified, i.e., it cannot be assigned.

```
Automaton [normal, failure] status is
    when t0 define value is normal     // Initial value
  otherwise                            // Transition normal -> failure
    when eFailure while previous = normal  define value is failure
  otherwise                            // Transition failure -> normal
    when eRepair  while previous = failure define value is normal
  otherwise
    define value is previous;
```

This example shows how `value` and `previous` may be used to specify the transitions of a finite state automaton. There is a transition from **`normal`** (the origin state) to **`failure`** (the target state) when event **`eFailure`** occurs while the automaton is in **`normal`** state. As **`status`** is a variable and can have only one value at any given time, at the instant of the transition it cannot be in both the origin and the target states, i.e., it cannot be in **`normal`** and **`failure`** states at the same time. Here, one uses `previous` to denote the origin state and `value` to denote the target state.

Keyword `otherwise` is used to limit risks of contradiction: a statement in an `otherwise` chain (also called a temporal exclusion chain, see Section 18.2.1) has precedence over the statements that follow it in the chain. The last assignment (`define value is previous`) specifies that apart from the initial instant and the instants where a transition is explicitly specified, the value does not change.

- `next` denotes the value to be taken by the variable immediately after the current instant. In the continuous time domain: `next` (t) = limit `value` (t+dt) dt → 0+

In a discrete time domain, *next* denotes the value to be taken by the variable at the clock tick immediately and strictly after the current instant. Since it belongs to the future, *next* can be assigned but cannot be used in an expression.

```
Automaton [normal, failure] status is
    when t0 define value is normal     // Initial value
  otherwise                            // Transition normal -> failure
    when eFailure while value = normal  define next is failure
  otherwise                            // Transition failure -> normal
    when eRepair  while value = failure define next is normal
  otherwise
    define next is value;
```

This example is very similar to the previous one, except that instead of pair (*previous*, *value*), it uses pair (*value*, *next*). In continuous time, where the time difference between *previous* and *value* or between *value* and *next* is infinitesimal, the two examples lead to practically the same behaviour. That is not the case in discrete time, where the time difference is perceptible.

9.2 Attributes derivative, integral & inPIntegral

A *Real* or *quantity* variable (see Section 9.4) has in addition the following attributes:

- *derivative* denotes the first derivative of the variable with respect to time and is meaningful only for variables in the continuous time domain. If there is a discontinuity, it is equal to **+%** or **-%** (the upper and lower bounds of representable *Real*).

  When used in an expression, it is determined by the *value* and *previous* attributes:

  derivative (t) = limit (*value* (t) - *value* (t-dt)) / dt
  dt → 0+

  When assigned a value, it determines the *next* attribute based on the *value* attribute[7]:

  next (t) = limit *value* (t) + *derivative* (t)*dt
  dt → 0+

  Thus, one cannot make assignments to *next* and *derivative* at the same time. The quantity of *derivative* is the quantity of the original variable divided by *Duration*.

- *derivative*[n] (where n is a literal for a positive whole number) denotes the n^{th} derivative of the variable with respect to time. Thus, *derivative*[1] is equivalent to *derivative*, *derivative*[2] is the derivative of *derivative*[1] and so on.

- In the case of a *Real* or quantity function (that has parameters other than time), *derivative*[x] (where x is the name of a *Real* or quantity parameter) denotes the first derivative with respect to x. The quantity of the *derivative* is the quantity of the function divided by the quantity of x. *derivative*[x, n] denotes the n^{th} derivative with respect to x. Its quantity is the quantity of the function divided by the quantity of x to the power n.

- *integral* denotes the first integral of the variable with respect to time. *inPIntegral* denotes the first integral of the variable with respect to time within a time period. The quantity of *integral* and *inPIntegral* is the quantity of the original variable multiplied by *Duration*. However, as they are based on the past, they cannot be assigned.

- *integral*[n] denotes the n^{th} integral of the variable with respect to time. *inPIntegral*[n] denotes the n^{th} integral of the variable with respect to time within a time period. Similarly to derivatives, *integral*[1] is equivalent to *integral*, *integral*[2] is the integral of *integral*[1] and so on, and *inPIntegral*[1] is equivalent to *inPIntegral*, *inPIntegral*[2] is the integral of *inPIntegral*[1] and so on. As all are based on the past, they cannot be assigned.

- In the case of a *Real* or quantity function, *integral*[x] (where x is the name of a *Real* or quantity parameter) denotes the first integral with respect to x. *inPIntegral*[x] denotes the first integral with respect to x within a time period. The quantity of the *integral* is the quantity of the function multiplied by the quantity of x. *integral*[x, n] denotes the n^{th} integral with respect to x.

[7] This fomula is useful when, in simulations, continuous time is discretised.

inPIntegral`[x, n]` denotes the n^{th} integral with respect to `x` within a time period. Its quantity is the quantity of the function multiplied by the quantity of `x` to the power `n`.

9.3 Attribute *default*

A defined variable class may use attribute *default* to specify a value to be taken by its instances when they are under-constrained. When used, it must be consistent with the applicable constraints, otherwise there is a contradiction and an exception is raised.

```
Class Switch1 extends Boolean "Default only at initialisation" begin
  when t0 define default is false;
end Switch1;
```

Class **Switch1** extends predefined class **Boolean**. Its definition specifies a default value (**false**) at initialisation time, and only at initialisation time (*when* **t0**). Instances that do not specify an initialisation value themselves will start with value **false**.

```
Class Switch2 extends Boolean "Default all along operational cases" begin
  default is false;
end Switch2;
```

Class **Switch2** also extends predefined class **Boolean** by providing a default value (**false**), but for all times. At times when an instance does not specify a value, it will take value **false**.

9.4 Quantity Classes

As FORM-L aims at addressing physical systems and their dynamic phenomena, a frequent use of *Real* extensions is to represent measurable *quantities* according to a *system of units*, such as the SI International System of Units [12].

The *SI* International *System of Units* starts with seven *base quantities* and their *units*:

- The *second* (symbol *s*) is the unit for quantity *time*.
- The *metre* (*m*) is the unit of *length*.
- The *kilogram* (*kg*) is the unit of *mass*.
- The *ampere* (*A*) is the unit of *electric current*.
- The *kelvin* (*K*) is the unit of *thermodynamic temperature*.
- The *mole* (*mol*) is the unit of *amount of substance*.
- The *candela* (*cd*) is the unit of *luminous intensity*.

An unlimited number of *derived quantities* (and their corresponding *units*) can be obtained as products of powers of base quantities. For example, the *SI* unit for *velocity* is the *metre per second* ($m*s^{-1}$) and the unit for *acceleration* is the *metre per second squared* ($m*s^{-2}$).

Twenty-two derived *SI* units are provided with special names and symbols. For example, the *newton* (*N*, $kg*m*s^{-2}$) is the unit of force and the *joule* (*J*, $kg*m^2*s^{-2}$) is the unit of energy.

In this document, one assumes the use of a **Standard** library defining a class for each commonly used quantity (not only *SI* quantities but also other useful ones such as a monetary quantity) and a constant for each commonly used unit (not only *SI* units but also ones such as the hour (*h*) or the euro (*euro*)).

9.4.1 Attributes *quantity*, *unit*, *scale* & *offset*

To that end, each instance of *Real* and its extensions has the following attributes:

- *quantity* is a constant specifying the quantity it represents. It is defined only by classes and cannot be modified by instances. Classes representing an actual quantity are called *quantity classes*, and those that do not are called *scalar classes*.

  Its actual contents is implementation-specific. In the examples, it is assumed to be as follows:

  - It is an empty character string for scalar classes.

- It is a non-empty string for classes representing a base quantity. That string must be different for and specific to each base quantity. For example: `"time"`, `"length"` or `"money"`.
- It is an automatically calculated normalised dimension formula[8] for classes and instances representing derived quantities. For example: `"length^1 * time^-2"` for acceleration.

- *unit* specifies the unit in which *value* is expressed. It is mainly useful for displaying analysis or simulation results. The *unit* of a class is constant and is the default unit. An instance may specify a specific *unit* that may vary in time, but this should be done with care, in consistency with *value*, *scale* and *offset*.

Its actual contents is implementation-specific. In the examples, it is assumed to be a character string.

```
Class Duration extends Real begin  // Base quantity time
  quantity is "time";
  unit     is "s";                 // String for the default unit
end Duration;

Class Length extends Real begin
   quantity is "length";
   unit is "m";
end Length;

Class Velocity is Length / Duration;

Class Mass extends Real begin
   quantity is "mass";
   unit is "kg";
end Mass;

Class Force extends Real begin
   quantity is Mass*Length / Duration^2;
   unit is "N";
end Force;

Class Money extends Real begin  // Monetary, non-physical quantity
  quantity is "money";
  unit is "€";
end Money;

Money euro is 1;
```

- *scale*: In an instance, it specifies the ratio between the *unit* of the instance and the *unit* of its class; it may vary in time but this should be done with care, in consistency with *value*, *unit* and *offset*. In a class, it is constant and specifies the ratio between the *unit* of the class and the *unit* of its super class. It is equal to 1 by default.

```
Class Percentage extends Real begin
  scale is 1e-2;
  unit is "%";
end Percentage;

Percentage perCent is 1;
```

```
Duration s is 1;                  // Allows notations such as 60*s

Duration ms begin                 // Allows notations such as 50*ms
  unit  is "ms";
  value is 1;
  scale is 0.001;                 // ms stands for millisecond
end ms;
```

- *offset* is defined only by instances: it is expressed in the *unit* of the instance and added after *scale* has been applied. It is nought by default. In practice, offsets are used only for temperatures. It may vary in time but this should be done with care, in consistency with *value*, *unit* and *scale*.

[8] A normalised dimension formula is expressed as a product of base quantities raised to integer powers (positive or negative), in a fixed order (e.g., in alphabetical order of the base quantity names).

```
Class Temperature extends Real begin
  quantity is "temperature";
  unit     is "°K";
end Temperature;

Temperature °K is 1;                  // Allows notations such as 300*°K

Temperature °C  begin                 // Allows notations such as 20*°C
  unit   is "°C";
  value  is 1;
  offset is 273.15;                   // °K = °C + offset
end °C;

Temperature °F  begin                 // Allows notations such as 20*°F
  unit   is "°F";
  value  is 1;
  scale  is 5/9;
  offset is 273.16-32*5/9;            // °K = (°F * scale) + offset
end °F;
```

Expressions with quantities are subject to the customary rules. In particular:

- One cannot add or subtract variables representing different quantities.
- When adding or subtracting variables representing the same quantity but in different units, a conversion is made based on `scale` and `offset` attributes so that the result is expressed in the `unit` of the left-hand side of the operator.
- The dimension formula for products, divisions and integer powers is respectively the addition, subtraction and multiplication of powers in the normalised dimension formula.

As time is a fundamental notion when addressing dynamic phenomena, class `Duration` needs to be defined. Also, the definitions of quantity classes must be consistent across a mock-up.

Conversion from `Real` to `Integer` and the reverse can be made on request with predefined operators. However, the operators for all other class conversions (including from a quantity to a scalar) must be defined explicitly, preferably in a library ensuring consistency with the quantity and unit system.

9.5 Enumerations

An *enumerated class* (or *enumeration*, for short) specifies a finite list of named, discrete values (also called *states*). An instance of an enumeration is a *finite state automaton* (or *automaton*, for short) the value of which is confined to the specified list of states. The *initial state* and *state transitions* (or *transitions*, for short) are jointly determined by the assignments and properties of the enumeration and of the automaton: the former specify what is common to all instances, whereas the latter specify what is specific. The two must not be contradictory.

An *atomic state* does not contain any sub-automata. A *composite state* contains either a single sub-automaton or the Cartesian product of two or more sub-automata. The states of sub-automata are themselves either atomic or composite. Thus, the states of an enumeration or automaton may be represented by a tree (see example of *Figure 9-1*):

- Its root node (or *root*) is the top-level enumeration or automaton.
- Its intermediate nodes are the sub-automata of the composite states.
- Its terminal nodes (or *leaves*) are the atomic states.

```
Enumeration [f, g, h]   B;    // Three states: f, g and h
Enumeration [i, j]      D;
Enumeration [m, n, o]   K;
Enumeration [p, q]      L;
Enumeration [K k & L l] E;    // Cartesian product of k and l
Enumeration [B b, c, [D d1 & D d2 & D d3] d, E e] A;
A a;   // declaration of automaton a as an instance of enumeration A
```

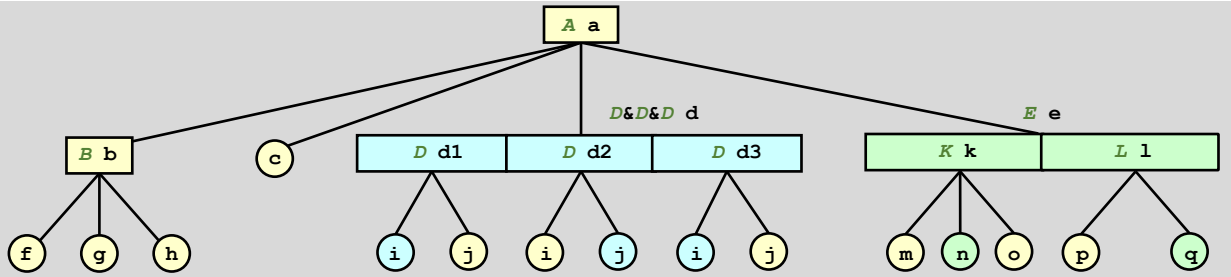


Figure 9-1
Artificial example of complex enumeration.

Here are a few examples of states that automaton *a* can take and of how they are denoted:

- `b.f`
- `c`
- `d.[d1.i & d2.j & d3.i]` `// In blue in Figure 9-1`
- `d.[i & j & i]` `// Same as d.[d1.i & d2.j & d3.i]`
- `e.[k.n & l.q]` `// In green in Figure 9-1`
- `e.[n & q]` `// Same as e.[k.n & l.q]`

In this artificial example, automaton ***a*** is an instance of enumeration ***A***. *Figure 9-1* shows the corresponding tree. Intermediate nodes are shown as rectangles, leaves are shown as circles.

In the list of state, an individual state is defined:

- Only by its name if it is atomic.

```
Enumeration [b, [f, g] c, H h, [I i1 & I i2] e] A;
```

- By its class and name if it is composite (i.e., containing one or more sub-automata). The class of a composite state may be:
  - A list of sub-states.

```
Enumeration [b, [f, g] c, H h, [I i1 & I i2] e] A;
```

  - The name of an existing enumeration.

```
Enumeration [b, [f, g] c, H h, [I i1 & I i2] e] A;
```

  - A Cartesian product of atomic states and automata. Atomic states in the Cartesian product are usually intended to be refined into sub-automata. Each element of the Cartesian product is named individually.

```
Enumeration [b, [f, g] c, H h, [I i1 & I i2] e] A;
```

9.5.1 Attribute *memory*

Each sub-automaton has a `default` attribute and a constant Boolean `memory` attribute of its own:

- When an automaton enters a composite state for the first time, if for some sub-automata the transition does not specify a specific sub-state, then their `default` attribute is used if defined; if not, an exception is raised.
- When the automaton enters a composite state not for the first time, if for some sub-automata the transition does not specify a specific sub-state, then for those with attribute `memory` set to **`true`**, the sub-state taken is the one at the last exit. For those for which attribute *memory* has not been defined or is set to **`false`**, attribute `default` is used if it is defined; if not, an exception is raised.

The `default` and `memory` attributes specified by an automaton must not contradict those specified by its enumeration.

```
Enumeration [retracted, moving, extended] GearState;
Enumeration [closed,    moving, open]     DoorState;
Enumeration [
  unknown,
  [ GearState fg & GearState rg & GearState lg
  & DoorState fd & DoorState rd & DoorState ld
  ] known
] LgsState;
```

This example is taken from the *Landing Gear System* study. **LgsState** is an enumeration that has:

- One atomic state named **unknown**.
- One composite state named **known**, which is the Cartesian product of 3 *GearState* automata named **fg**, **rg** and **lg** (for front, right and left gears) and 3 *DoorState* automata named **fd**, **rd** and **ld** (for front, right and left doors).

```
Enumeration [slow, rebound] Degraded;
Enumeration [stuckOpen, stuckClose, stuckAsIs] Failed;
Enumeration [nominal, Degraded degraded, Failed failed] OpState begin
  when t0 define value is nominal;     // Initial state
  degraded.memory is true;
  failed.default is stuckAsIs;
end OpState;
```

OpState is an enumeration where:

- **nominal** is atomic and the initial state.
- **degraded** is an instance of separately declared enumeration *Degraded*. As degradation could be intermittent, **degraded** is specified to have *memory*.
- **failed** is an instance of another separately declared enumeration named *Failed*. It does not have *memory* and its *default* value is **stuckAsIs**.

```
Enumeration [standby, active, finished] Sequencing;
Enumeration [normal, test] Mode;
Enumeration
[ [Sequencing seq & Mode mode] operational,
  maintenance,
  failure
] State begin
  when t0 define value is operational.(standby & normal);
end State;
```

This example is taken from the *Backup Power Supply* study. Enumeration **State** identifies the main states of a system:

- **operational** is the nominal state where the system is ready to perform or is performing its mission.
- **maintenance** is the atomic state taken when the system is under maintenance.
- **failure** is the atomic state taken when a failure prevents the system from accomplishing its mission.

The **operational** state is the Cartesian product of two enumerations, *Mode* and *Sequencing*. **mode** indicates whether the system is under periodic test (**test**) or not (**normal**): what is required of the system depends on it. **seq** indicates the current phase of system operation:

- In the **standby** sub-state, the system stands ready to perform its mission should a demand occur.
- In the **active** sub-state, a demand has occurred and the system is performing the required actions.
- In the **finished** sub-state, a demand has occurred and the system has completed all the required actions and is just waiting for a reset signal to return to the standby mode.

9.5.2 Refinement

For enumerations or automata, refinement may transform some atomic states into composite ones.

```
Enumeration [ok, failure] Mode;

refined [stuckOpen, stuckClose, stuckAsIs, spuriousOpen] Mode.failure;
```

Enumeration **Mode** has two states, **ok** and **failure**. Then, atomic state **failure** is refined to be no longer atomic and to represent four possible failure modes.

9.6 Grammar

```
predefVariableClaas: 'Boolean' | 'Integer' | 'Real' | 'String' ;
```

```
ExtensionClass:
  'private'? 'Class' determiner? NAME parameters? STRING* 'extends' classes
  (Behaviour | ';')
;
```

```
QuantityClass:
  'private'? 'Class' determiner? NAME parameters? STRING*
  'is' quantityClass ';'
;
```

A quantity class may be defined by a quantity expression based on already defined quantity classes.

```
quantityClass: quantityTerm (('*' | '/') quantityTerm)* ; // Quantity expr
```

A quantity class expression is either a single quantity term (***quantityTerm***) or the multiplication and division of quantity terms.

```
quantityTerm:
  (pathname /* quantity class */ | '(' quantityClass ')') ('^' '-'? INT)? ;
```

A quantity term is either a defined quantity class (identified by a ***pathname***) or a parenthesed quantity class expression (***quantitClass***), possibly elevated to a power (positive or negative integer literal).

```
EnumeratedClass:
  'private'? 'Enumeration' states determiner? NAME parameters? STRING*
  (Behaviour | ';') ;

states: '[' state ((',' state)+ | ('&' state)+) ']' ;
```

The ***states*** of an enumeration or automaton are specified either by a list of comma-separated ***state***, or by a Cartesian product of ampersand-separated ***state***.

```
state: (pathname | states)? NAME STRING* ;
```

A ***state*** is specified by its **NAME** and an optional **STRING** description. A composite ***state*** is itself an instance of an enumeration specified by a ***pathname*** or by its own ***states***.

```
StateRefinement: 'refined' states pathname STRING* ';' ;
```

An atomic state of an enumeration or an automaton may be refined into a composite state by specifying a ***pathname*** to the atomic state being refined, and the ***states*** defining its structure.

10 Events

With *variables*, *events* are the most basic FORM-L items. An event represents a fact of interest that may occur (possibly multiple times) in the course of an operational case and the duration of which is neglected. Each event has a *class* that may specify or place constraints on its behaviour (i.e., its occurrences or non-occurrences) and may specify attached features.

10.1 Attributes *rate*, *value* & *occurrence*

The definition of an event specifies its behaviour (i.e., occurrence or non-occurrence) by placing *CTL*-controlled *assignments* or *assumptions* on *attributes* `occurrence`, `default` and `rate`, but also by *DTL*-controlled *signals* (keyword `signal`). The behaviour of an event may, at certain times, be *fully constrained*: it has then fully specified occurrences or absence of occurrences. At other times, it may be *under-constrained*, e.g., with *rates* of occurrence (see *Figure 7-2*). At times when no assignment or signal applies, the event does not occur.

- `occurrence` denotes when the event occurs or does not occur.

```
Event eEndOfLifetime1 begin
  when t0 + 60*year signal occurrence;
end eEndOfLifetime1;

Event eEndOfLifetime2 is when t0 + 60*year;
```

These two events are identical and occur only once, 60 years after `t0`.

```
Event @eFailure:
  Assumption consistency is during not operation signal no occurrence;
```

This specifies that when not in **operation**, it is assumed that there is no occurrence of **eFailure**.

```
Event eReset begin
  Assumption notTooOften is
    after occurrence during 20*s
    ensure no occurrence;

  Assumption notTooRare is
    after t0 or occurrence within 2*h
    achieve occurrence;
end eReset;
```

- When assigned, `rate` specifies the background probability of occurrence of the event per unit of time: background means that it comes in addition to explicitly specified `signal occurrence` and is superseded during instants or time periods where `signal no occurrence` is specified. At times where neither `rate` is specified, nor `occurrence` or `no occurrence` are signalled, the event is completely unconstrained and occurs completely randomly. When used in an expression, that expression must be under the scope of a CTL (or of implicit CTL `always`): `rate` is then the *a posteriori* value of the effective rate of occurrence of the event during each time period of the CTL.

```
eFailure:
  Assumption reliability is during operation ensure rate < 1e-4/h;
```

This additional definition completes the one given in previous example with assumption **reliability**, which states that during **operation**, the `rate` of occurrence is below 10^{-4} per hour.

- `value` is a *value set* of fixed `Duration` providing the time position of each past occurrence. It is automatically calculated and can be used but not modified by a model.

10.2 Attribute *clockValue*

Like all objects, events have a `clock` attribute which, when defined, places it in a *discrete time domain*: then, there can be occurrences only at `clock` ticks.

`clockValue` is a *value set* of fixed `Integer` that is non empty only if the event is in a discrete time domain (i.e., only if its `clock` attribute is defined): it gives the position of each past occurrence in terms of tick number of the `clock`. It is automatically calculated and can be used but not modified by a model.

10.3 Predefined Classes

There is only one predefined event class: ***Event***.

10.4 Defined Classes

Defined event classes may be created only by the extension of an existing event class.

```
Class EClock extends Event begin
  fixed Duration phase is random (50)*ms;          // Local variable
  (every 50*ms) + phase signal occurrence;         // Signal
end EClock;

EClock eclock1;
EClock eclock2;
```

EClock extends predefined class ***Event*** by signalling periodic occurrences with a random phase. **eclock1** and **eclock2** are two instances: they have the same period (**50*ms**), but different phases.

10.5 Grammar

```
predefEventType: 'Event' ;
```

11 Sets

11.1 Sets of Objects

The objects constituting a system of interest and its environment, and the variables and events necessary to model their behaviour are often not fully known at early stages of the engineering process.

This is the case in the *Backup Power Supply* study: in the **`BpsIntroduction`**, the **`BpsReference`** and the **`BpsPsaReference`** models, the clients and also the components of the *BPS* are not known yet.

Sets of objects, which are finite, non-ordered collections of objects (their *members*) of the same class, are thus an essential ingredient of the FORM-L language: when they are defined *in intention* (by specifying what characterises their members), they enable reasoning on, and the specification of, behaviours and properties of objects that have particular roles or characteristics but are not individually identified yet.

In the framework of an expression (including *selectors*), the name of a class denotes the set of all its named and dynamically created instances. It is the largest possible set for that class: all the other sets for that class, be they defined in intention or extension, are subsets of it.

A subset may be defined in intention using a *subset selector* (as shown in the following example) or set operators (union, intersection, subtraction and complement).

```
Class Component "Components of the BPS" extends Object begin
  Automaton [ok, [primary, secondary] failure] #state;
end Component;

Component {} inPrimaryFailure is
   all x of Component such that x.state = failure.primary;
```

This example is taken from the *Backup Power Supply* study. **`inPrimaryFailure`** is a set of ***`Component`*** defined in intention using a subset selector: it contains all the *BPS* components that are in a primary failure state (***`all`*** **`x`** ***`of`*** **`Component`** ***`such that`*** **`x.state = failure.primary`**), even though the full list of ***`Component`*** is not known yet.

Alternatively, a set of objects may be defined *in extension* with an explicit curly-bracketed list of all its members, which are either identified by name or created on the spot by an instantiation, as shown in the following examples. The curly-bracketed list is itself a set literal.

```
Component c1;  Component c2;  Component c3;
Component {} subset "Subset of Component" is {c1, c2};
```

Here, **`subset`** is defined in extension by explicitly naming its members in a literal (**`{c1, c2}`**).

```
Enumeration [nil , a, b, ab, ris, risa, risb, risab] Role;

Class Variant (Role r1, Role r2, Role r3, Role r4) extends Object;

Class Configuration (Role r1, Role r2, Role r3, Role r4) begin
  Variant {} variants is
    {Variant (r1, r2, r3, r4),
     Variant (r2, r1, r3, r4),
     Variant (r1, r2, r4, r3),
     Variant (r2, r1, r4, r3)};
end Configuration;
```

This example is taken (in a simplified form) from the *Intermediate Cooling System* study. The cooling system and its clients are subsystems of an overall plant. The mission of the system is to cool its clients, but depending on plant state, some of them are not active and do not need to be cooled. It is composed of four trains of equipment that must remain independent, i.e., a client must never be cooled by more than one train at a time.

A ***`Configuration`*** specifies the ***`Role`*** of each train, i.e., which clients it must cool. ***`Role`*** is declared as an enumeration, each value of which designating a particular subset of clients.

To deal with possible train failures, the first and second trains constitute a redundant pair and may exchange their ***Role***, and so do the third and fourth trains. Thus, each configuration has four functionally equivalent ***Variant***.

Here, feature **variants** of class ***Configuration*** is a set of ***Variant*** defined in extension by four unnamed ***Variant*** instances.

The same set may, at some instants, be defined in intention, and at others in extension.

11.2 Sets of Values

A *set of values* in FORM-L is a finite, non-ordered collection of values or value intervals (its *members*) of the same valued class. Intervals are possible only for valued classes that have a total order relation (***Integer***, ***Real*** and their extension classes). Sets of values and sets of objects are different, as shown in the following example.

```
Boolean b1;  Boolean b2;  Boolean b3;

Boolean  {} objects   "Set of Boolean objects" is  {b1, b2, b3};
Boolean ${} values    "Set of Boolean values"  is ${b1, b2 or b3, true};
Real    ${} intervals "Set of Real intervals"  is ${1.5, 3, [4, +%]};
```

This artificial example shows the difference between sets of objects and sets of values:

- **objects** is a set of objects defined in extension: its cardinal is 3 since it contains 3 members (**b1**, **b2** and **b3**).
- **values** is a set of values: its cardinal is 1 or 2 since a Boolean value is either **true** or **false**.
- **intervals** is also a set of values: its cardinal is 3. Intervals **1.5** and **3** are reduced to a single value.

Sets of values are declared in a manner very similar to sets of objects, but to make a distinction, set indications and literals for sets of values are prefixed by a **$** sign. Also, a set of values indication can only specify the domain of value of the cardinal of the set.

A set of values may also be defined either in intention or in extension. In the framework of an expression, the name of a variable class prefixed by the **$** sign denotes the set of all its possible values. Subset selectors and set operators may also be used.

When a set of values is defined in extension, values and interval boundaries are specified by expressions of the appropriate class. An ***Integer*** interval is denoted **n..m**, or **n** when it is reduced to a single value. A ***Real*** or quantity class interval is denoted **[m, n]**, **]m, n]**, **[m, n[** or **]m, n[** depending on whether it includes its boundaries, or **n** when is reduced to a single value. An interval is empty if expression **n** is strictly lower than expression **m**, and may be downwardly and / or upwardly unbounded using **-%** and **+%** to denote the infinite.

11.3 Attributes value, previous, next & clock

Overall, a set, be it of objects or of values, is treated as a variable the ***value*** of which is its membership: thus a ***constant*** (or ***fixed***) set has a ***constant*** (or ***fixed***) membership, and is not to be confounded with a set of ***constant*** (or ***fixed***) members.

In the same spirit, ***previous*** and ***next*** apply to its membership, and ***clock***, when defined, specifies the instants when membership can change.

11.4 Defined Classes

There are no predefined set classes. A set class may be defined either by associating an existing class or list of classes (which will be the class or classes of its members) with a *set indication*, or by extending one set class.

A set indication is composed a minima of a pair of curly brackets **{}** in the case of a class of set of objects, or of **$** marked pair of curly brackets **${}** in the case of a class of set of values. The contents of the brackets may specify possible values for the cardinal. For a class of set of objects, it may also specify the variability and / or the determiner of the objects.

```
AcVoltage {3} threePhaseCurrent;
```

threePhaseCurrent is a set of 3 instances of ***AcVoltage***.

```
Class ThreePhaseCurrent "Three AC voltages" is AcVoltage {3};
```

ThreePhaseCurrent is a class of set of objects. The cardinal of its instances must be 3, and their members must be instances of ***AcVoltage***.

```
Class ActualThreePhaseCurrent "Three actual AC voltages"
  extends ThreePhaseCurrent {#};
```

ActualThreePhaseCurrent is a class of set of objects extending **ThreePhaseCurrent**. Like for **ThreePhaseCurrent**, the cardinal of its instances must be 3 and their members must be instances of ***AcVoltage***, but in addition, they must all have a **#** determiner.

```
Class ActualThreePhaseCurrent "Three actual AC voltages"
  is AcVoltage {3, #};
```

This example is similar to the previous one, but without the mediation of **ThreePhaseCurrent**.

11.5 Selectors

The role of a *selector* is to scan through one or more sets, with an optional Boolean selection criterion. Without a selection criterion, all the members of the set(s) are taken into account. Selectors may be used for different purpose:

- Random extraction of a single element.

```
Class Component "Components of the BPS" begin
  Automaton [ok, [primary, secondary] failure] #state;
end Component;

Component selected is any Component;
```

- Extraction of subsets.

```
Component {} inPrimaryFailure is
   all x of Component such that x.state = failure.primary;
```

- Boolean expressions based on existential quantifiers.

```
Boolean partialFailure is
  whether some x of Component satisfy x.state=failure;
```

This example shows an existential quantifier applied to a single set, without a selection criterion.

```
Integer {} set1;  Integer {} set2;

Boolean intersect1 is
  whether some x of set1, y of set2 satisfy x = y;

Boolean intersect2 is
  whether some x of set1, y of set2 satisfy x.identity = y.identity;
```

This example shows existential quantifiers applied to multiple sets, without a selection criterion. The first one determines whether the two sets of objects have members with the same value, even though these may be distinct items. The second one determines whether the two sets have items in common.

- Boolean expressions based on universal quantifiers.

```
Boolean totalFailure is
  whether all x of Component satisfy x.state=failure;
```

Universal quantifier applied to a single set.

```
Boolean difference1 is
  whether all x of set1, y of set2 satisfy x <> y;
```

```
Boolean difference2 is
  whether all x of set1, y of set2 satisfy x.identity <> y.identity;
```

Universal quantifiers applied to multiple sets. The first one determines whether the two sets of objects have members with different values. The second one determines whether the two sets have items not in common.

- In a property definition, an *existential selector* specifies that the constraint is to be satisfied by at least one selected member.

```
Property notAllCavitate is
  during normalOperation
  for some p of pumps such that p.location = "room A"
  ensure not p.cavitates;
```

This states that during each period of **normalOperation**, some members of set **pumps** in room A should not be cavitating.

- In a property definition, a *universal selector* specifies that the constraint is to be satisfied by all selected members.

```
Property noCavitation is
  during normalOperation
  for all p of pumps such that p.location = "room A"
  ensure not p.cavitates;
```

This states that during **normalOperation**, no member of set **pumps** in room A should cavitate.

```
Guarantee sensorsOk is
  during normalOperation
  for all s1 of Sensor, s2 of Sensor
    such that s1.input.previous = s2.input.previous
  ensure s1.output = s2.output;
```

This guarantee ensures that instances of class **Sensor** receiving the same input do provide the same **output**.

- When applied to a statement or block of statements, a selector applies the statement or block to each selected member. Objects declared by the statement or block are created and deleted dynamically when necessary.

```
for all x of Client: x.poweredByBps is
     u.x.a.poweredByDivision
  or u.x.b.poweredByDivision
  or v.x.a.poweredByDivision
  or v.x.b.poweredByDivision;
```

This example is taken from the *Backup Power Supply*. It states that a client (i.e., a member of set **Client**) receives power produced by the BPS when any of its channels receives power produced by any of the divisions.

- When applied to an expression, a selector produces a set of values the members of which are the application of the expression to each selected member.

```
Integer ${} multiplication is
  for all x of set1, y of set2 such that x < 0 and y > 5 : x * y;
```

11.6 Grammar

```
SetClass: 'private'? 'Class' determiner? NAME parameters? STRING*
  'is' setClass (Behaviour | ';') ;

setClass:  class setIndication;

setIndication: '$'? '{' (variability? & determiner? & cardinal?) '}' ;
```

A *setIndication* is either for a set of objects or for a set of values. In the latter case, there is no *determiner*. The *variability* here applies to set members.

```
cardinal: INT ('..' (INT | '+%') (',' INT ('..' (INT | '+%')?)* }' ;
```

A *cardinal* places a constraint on the cardinal of a set. It has the same syntax as a set of Integer values, except that values and interval boundaries must be positive literals.

```
Set:
   ('private'? & variability?) setClass  determiner? NAME STRING*
   (Behaviour |  ';') ;
```

```
selector:   NAME (',' NAME)* 'of' set ('such' 'that' bool)? ;
```

The same *set* may be scanned multiple times: all **NAME** must then be different. *bool* is an optional selection criterion.

```
constraintApplicator: 'for' ('some' | 'all') selector ;

statementApplicator:  'for' 'all' selector;
```

12 Properties

A *property* specifies one or more *constraints*, at times specified by *temporal locators*, on the expected, desired or required behaviour of other objects identified either by name or by *selectors*. It either specifies these constraints directly (together with their temporal locators and selectors) or is based on an expression involving other properties. The first form is discussed in Section 12.2 whereas the latter one is discussed in Section 12.7 after a detailed presentation of property evaluation in Section 12.6.

12.1 Declaration

The name of an ***Assumption***. may be omitted.

12.2 Constraint-Based Definition

In a constraint-based property definition, *temporal locators* (see Sections 15 *Discrete Temporal Locators (DTL)*, 16 Continuous Temporal Locators (*CTL)* and 17 *Sliding Temporal Locators (STL)*) may be used to specify at which *instants* or during which *time periods* each of these constraints applies. If no temporal locator is specified for a constraint, then it applies at all times, all along each *operational case*.

Selectors (see Section 11.5) may also be used to specify which members of one or more sets are concerned with the constraints. If no selector is specified for a constraint, then the objects concerned must be named explicitly. There are two kinds of selectors for properties: universal selectors (beginning with ***for all***) and existential selectors (beginning with ***for some***).

```
Requirement #repower is
  for all x of Client
  during mps.voltage < Client.minimumVoltage
  ensure x.tolerance >= 5*perCent;
```

This example is taken (in a slightly modified form) from the *Backup Power Supply* study. It shows the three basic parts of a formal, constraints-based property definition:

- The first part (***for all*** **x** ***of*** **Client**) is a universal selector, specifying that all the instances of class **Client** are concerned with the constraint.
- The second part (***during*** **mps.voltage < Client.minimumVoltage**) is a temporal locator specifying when the constraint applies. Here, it is a *CTL*. Since it does not depend on **x**, it could have been specified before the selector.
- The third part (***ensure*** **x.tolerance >= 5*perCent**) is the constraint to be satisfied.

Like variables, events and sets, a property may be defined globally (as shown in the preceding example) or in an itemised manner with a definition block.

```
Guarantee #noSwitch
  "Switches must not be manipulated when the BPS is operating, unless
   a division signals an alarm during a real demand (i.e., not a test)"
begin
  private Boolean failure is
    from (u.eAlarm or v.eAlarm) while bps.operating
    until bps.operating becomes false;

  satisfaction is
    during bps.operating and not failure
    ensure no OR (Client.Channel.switch CHANGES);
end noSwitch;
```

This example is also taken from the **BpsRedundancy** model and specifies a guarantee ensured by the **operator**. Here, the expression of the temporal locator is based on ***private Boolean*** feature **failure**.

One can note that since a property does not have a ***value*** attribute, it is represented in the definition block by attribute ***satisfaction***.

When multiple constraints apply but are mutually exclusive in time, they may be specified in a unique property with a *temporal exclusion chain* (keyword ***otherwise***, see Section 18.2.1).

```
Boolean #active "Whether wipers are wiping" begin
  when t0 define value is false otherwise value is deferred;
  Requirement activity is
      during command = on except first 0.1*s  // Wipers need time to react
      ensure active = true
    otherwise
      during not retracted                    // Rest only when retracted
      ensure active = true
    otherwise
      during retracted and command = off except first 0.1*s
      ensure active = false
    otherwise
      during command = off
      ensure no active becomes true;
end active;
```

This is a variation of the *Car Wipers* study seen in Section 7.10. The definition of **`activity`** specifies multiple constraints, each with its own temporal locator. That the constraints do not overlap in time (which might lead to contradiction) is guaranteed by keyword ***`otherwise`***.

However, when the property is violated, it might be more difficult to determine which of the constraints is at fault. It is often preferable to define single-constraint properties using the more general time coordination mechanisms of *composite instructions* (see Section 18.2 and the original *Car Wipers* example).

12.3 Constraints

There are different kinds of constraints (see *Figure 12-1*).

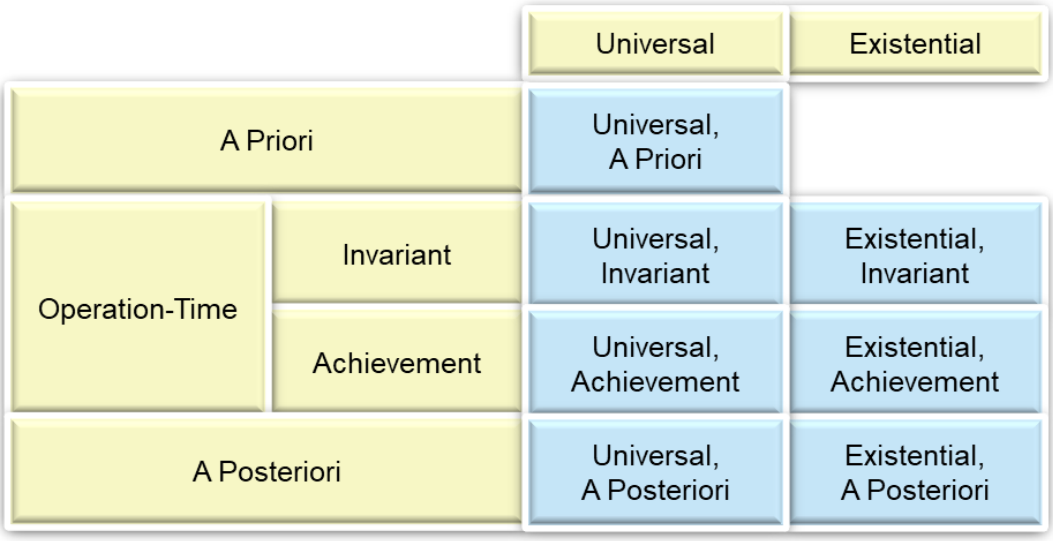


Figure 12-1
The different kinds of constraints.

First, one needs to distinguish *operation-time constraints* from *a priori constraints* and *a posteriori constraints*. An *a priori* constraint specifies a Boolean expression that involves only pre-determined values and that can be verified independently of operational cases. Such constraints are generally used in ***`Guards`***. An operation-time constraint specifies a Boolean expression that can be evaluated in the framework of each operational case (i.e., it does not involve any values that can only be calculated *a posteriori*). An *a posteriori* constraint specifies a Boolean expression that involves *a posteriori* values and that can be verified only on the basis of many operational cases.

Second, one needs to distinguish *universal constraints* from *existential constraints*. A universal constraint needs to be satisfied in *all* legitimate operational cases (i.e., operational cases where all assumptions are satisfied). An existential constraint needs to be satisfied in *at least one* legitimate case. They are used to express *capabilities* such as "*There shall be a way for X to operate the system such that goal Y is achieved*". By definition, existential constraints cannot be used in an ***`Assumption`***. Also, this distinction does not apply to *a priori* constraints.

Operation-time universal and existential constraints are each divided into two sub-kinds: *invariant constraints* and *achievement constraints*:

- Under the scope of a *DTL*, the constraint is necessarily an invariant constraint.
- Under the scope of a *CTL* or an *STL*, if the constraint needs to be satisfied all along each time period then the constraint is an invariant constraint.
- Under the scope of a *CTL* or an *STL*, if the constraint only needs to become satisfied in the course of each time period and to remain so until the end of the period, then the constraint is an achievement constraint.

A *universal invariant constraint* begins with keyword ***ensure***, whereas a *universal achievement constraint* begins with keyword ***achieve***. Their existential counterparts begin with ***assert*** **x** ***can ensure*** or ***assert*** **x** ***can achieve***, where **x** represents one or more objects. The meaning is that for each possible behaviour allowed by the assumptions that are not features of **x**, at least one behaviour allowed by the assumptions features of **x** satisfies the constraint

In the previous *Car Wipers* example, the constraint is a universal invariant constraint.

```
Requirement alarm is
  after eFailure within 200*ms
  achieve eAlarm;
```

This example, also taken from the *Backup Power Supply* study, shows a universal achievement constraint. Event **eAlarm** must occur at least once within 200 ms after each occurrence of event **eFailure**.

```
Objective resilience is
  during exceptionalCondition
  for all x of Client
  assert operator can ensure x.temperature < x.maxTemperature;
```

This example shows an existential invariant constraint: whenever **exceptionalCondition** is **true**, there should be a legitimate behaviour of **operator** that maintains the **temperature** of each **Client** below its **maxTemperature**.

```
Objective resilience2 is
  during exceptionalCondition
  for all x of Client
  assert controlRoomOperator, fieldOperator
    can ensure x.temperature < x.maxTemperature;
```

This variation of the previous example shows an existential invariant constraint to be ensured by the joint actions of two objects, **controlRoomOperator** and **fieldOperator**.

```
Objective   reset;
Requirement resetPfd is ensure reset.pfd < 1e-4;
```

This example, also taken from the *Backup Power Supply* study, shows an *a posteriori* universal constraint. Attribute *pfd* (probability of failure on demand) is an *a posteriori* property attribute, thus **resetPfd** is also *a posteriori*.

12.4 Predefined Classes

There are six predefined property classes, each having a particular meaning:

- A ***Property*** has a neutral connotation and is generally used as an intermediate term to express other, more complex properties.
- An ***Objective*** is a property that is desired but not necessarily achievable and that might be violated (in case of component failure or exceptional conditions for example).
- A ***Requirement*** is a property that is required and the violation of which is not acceptable.

- An ***Assumption*** is a property that by definition is never violated by legitimate cases. As its non-violation is guaranteed, it is the only property (together with instances of ***Assumption*** extensions) that may be unnamed.
- A ***Guarantee*** is a property ensured by one party to a contract: the other parties can then take it as an ***Assumption***. The party responsible for the guarantee can later refine it either as a ***Requirement*** or as an ***Assumption***.
- A ***Guard*** is a property that specifies when a model is valid: when it is violated, an exception is signalled since the model no longer gives meaningful results.

12.5 Defined Classes

A defined property class may be created by extending an existing one, and has the same general intention and meaning as its super-class, but specifies additional statements that apply to all its instances. However, as a property value is completely determined by its definition, defined property classes are property functions (see Section 7.4) or declare specific features.

```
Enumeration [classA, classB, classC, nc] SafetyClass;

Class SafetyRequirement extends Requirement begin
  constant SafetyClass safetyClass is specific;
end SafetyRequirement;
```

In this example, a **SafetyRequirement** is a ***Requirement*** that has a specific ***SafetyClass*** feature.

```
Class FaultTolerance (Objective obj, Boolean tolerance) extends
Requirement:
  satisfaction is during tolerance ensure no obj.eViolation;
```

In this example, a **FaultTolerance** is a ***Requirement*** function that requires that ***Objective*** **obj** is not violated when ***Boolean*** condition **tolerance** is **true**.

Note. As explained in Section 12.6 *Property Evaluation*, the evaluation of the value of a property is *non-univocal*, i.e., there are different possible legitimate definitions. FORM-L has a chosen one that is built in the predefined property classes. For applications that need an alternative definition, future versions of the language could propose a ***basic*** flavour for each of the predefined property class: they would not be intended to be instantiated directly but only through extensions. They would provide a full definition only for *univocal* attributes, *non-univocal* attributes being just declared so that extensions can specify definitions different from those of Section 12.6.

12.6 Property Evaluation

One of the goals of formal properties analysis or simulation is to determine whether the specified properties have been *tested* or not, and if so, whether they have been satisfied or violated. *A posteriori* properties such as probabilistic and optimisation properties are evaluated with specific approaches not discussed here. To support the *operation-time* evaluation process, a property has the following attributes:

- If the temporal locator is a *CTL*[9], in-period events ***eInPSatisfaction*** and ***eInPViolation*** mark the satisfaction or violation of the property constraint within each period, and in-period Booleans ***inPUntested***, ***inPSatisfaction*** and ***inPViolation*** represent the property value within each period, one and only one of them being **true** at any time along a period. ***eInPSatisfaction*** occurrences mark transitions to ***inPSatisfaction***, and ***eInPViolation*** occurrences mark transitions to ***inPViolation***. There is no ***eInPUntested*** event: ***inPUntested*** is the initial in-period state but there is no transition back to it.
- Events ***eSatisfaction*** and ***eViolation*** mark the satisfaction or violation of the property constraint along time, and Booleans ***untested***, ***satisfaction*** and ***violation*** represent the property value along each operational case, one and only one of them being **true** at any time along a case. ***eSatisfaction*** occurrences mark transitions to ***satisfaction***, and ***eViolation***

[9] *STL* are defined in terms of *CTL*. See Section 17 *Sliding Temporal Locators (STL)*.

occurrences mark transitions to `violation`. There is no `eUntested` event: `untested` is the initial state but there is no transition back to it.

- Booleans `satisfied` and `violated` represent the case verdict.

Evaluation for operation-time properties is done in a four-step process illustrated in *Figure 12-2*:

- Step 1 : a local satisfaction or violation *decision* is made for each instant of a *DTL* (events `eViolation` and `eSatisfaction`) or for each time period of a *CTL* (in-period events `eInPViolation` and `eInPSatisfaction`).
- Step 2 : a time-dependent property value (Booleans `untested`, `satisfaction` or `violation`) is determined along each operational case based on the sequence of local decisions.
- Step 3 : a *verdict* (Booleans `satisfied` and `violated`,) for each operational case is determined based on that value.
- Step 4 : a campaign *conclusion* is determined based on the verdicts of a set of cases.

Campaign conclusions are determined *a posteriori* with approaches not discussed here.

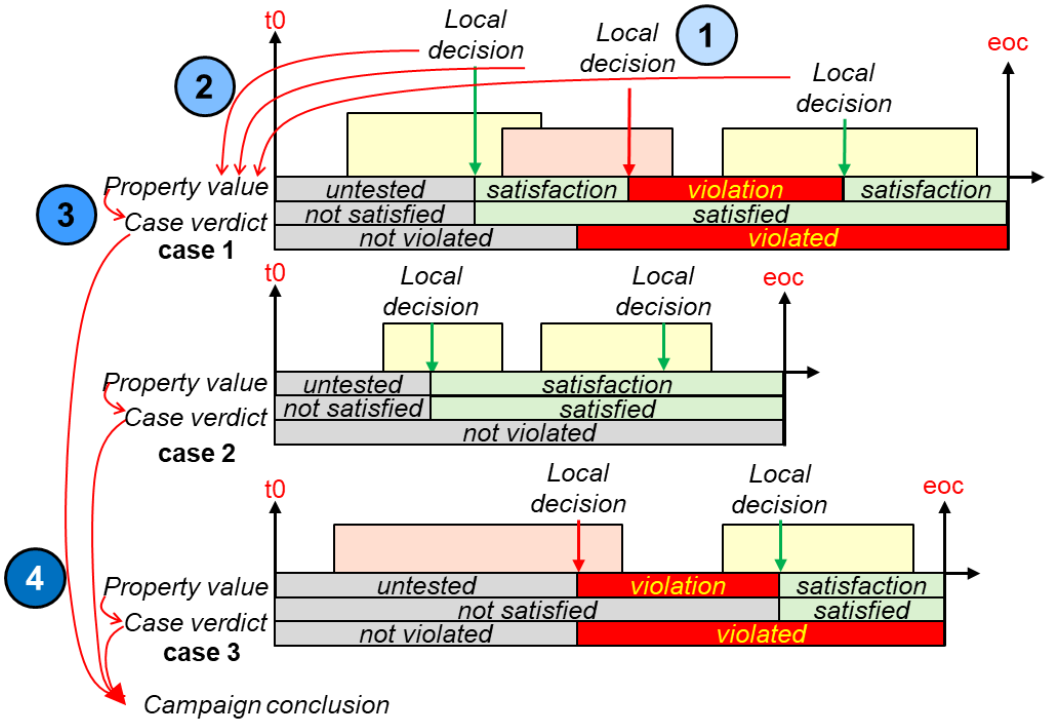


Figure 12-2
The four-step property determination process in the case of a CTL-based property.

Note 1. `satisfaction` and `satisfied` do not mean that the property is satisfied in all cases: `satisfaction` just means that it is satisfied in the current operational case at the current instant or up to the current instant, and `satisfied` just means that it has been satisfied at least once in the current operational case. The same applies, *mutatis mutandis*, to `violation` and `violated`.

Note 2. By definition, in legitimate operational cases, an `Assumption` or a `Guarantee` is never in `violation` and never `violated`, and its `eViolation` and `eInPViolation` never occur.

Local decisions under a DTL

A local decision is made at each instant of the *DTL* and is represented by an occurrence of either `eSatisfaction` or `eViolation`, depending on whether the constraint is satisfied at that instant.

Local decisions under a CTL

A local decision is made for each period that reaches its normal term, and is represented by the occurrence of either `eInPSatisfaction` or `eInPViolation`. (There is no decision at all if the period

is aborted before a local decision can be made.) Decision making is more subtle than in the case of a *DTL*:

- For invariant constraints, an `eInPSatisfaction` decision can be made only at the end of the period, but an `eInPViolation` decision can be made as soon as the constraint is violated during the period.
- For achievement constraints, the local decision can also be made only at the end of the period, except in the case of a fixed constraint on a *monotonous* in-period expression where the decision can sometimes be made earlier.

Only one of these two in-period events can occur for the same period, and only once.

Property Value

Although local decision making is univocal (i.e., for each kind of constraint, there is only one logical way to make a local decision), multiple logical ways are possible to define the property value. For example, one could choose to say that at any instant not covered by its temporal locator, or when no decision can be made yet during a time period, a property is `untested`. However, *DTL*-based or invariant properties would then never be in `satisfaction`, except for the fleeting *DTL* instants or period ends. It is thus preferable to say that:

- A *DTL*-based property is initially `untested`, and then either in `satisfaction` or `violation`, depending on the successive local decisions (`eSatisfaction` or `eViolation`).
- A *CTL*-based property is, for each period of the *CTL*, initially `inPuntested`, and then either in `inPSatisfaction` or `inPViolation`, depending on the successive local decisions (`eInPSatisfaction` or `eInPViolation`).

Also, as the time periods of a *CTL* may overlap, a *CTL*-based property may, at the same instant, be in `inPSatisfaction` in one period and in `inPViolation` in another. In the following, precedence has been given to violation, since verification usually aims at ensuring that requirements are never violated.

In the case of *CTL*-based properties, the determination of the property value begins by the determination of `eSatisfaction` and `eViolation`, based on the local decisions (i.e., the occurrences of `eInPSatisfaction` and `eInPViolation`):

- `eViolation` occurs when `eInPViolation` occurs.
- `eSatisfaction` occurs when `eInPSatisfaction` occurs while no period is in `inPViolation`.

The value can now be determined in the same manner for *DTL* and *CTL* based properties:

- It is initially `untested` before the first occurrence of `eSatisfaction` or `eViolation`.
- It transits to `satisfaction` when `eSatistaction` occurs.
- It transits to `violation` when `eViolation` occurs.

Case Verdict

`satisfied` is initially `false` and transits to `true` when `eSatisfaction` occurs, and remains so to the end of the case.

`violated` is initially `false` and transits to `true` when `eViolation` occurs, and remains so to the end of the case.

pfd

The `pfd` attribute of a property denotes its probability of failure on demand. It is calculated *a posteriori* based on the results of a number of operational cases with the following formula:

```
pfd = count (eViolation) / (count (eSatisfaction) + count (eViolation))
```

***Early local decisions under a* CTL**

Operation-time constraints to be evaluated during a period fall into one of three kinds:

- An *invariant constraint* (or *i-constraint*) is introduced by keywords ***ensure*** or ***assert*** **x** ***can ensure*** and needs to be satisfied all along the period: *eInPSatisfaction* can be decided only at the end of the period, but *eInPViolation* could be decided before.
- An *achievement constraint* is introduced by keywords ***achieve*** or ***assert*** **x** ***can achieve*** and needs to be satisfied only at the end of the period. Thus, *eInPSatisfaction* or *eInPViolation* can normally occur only at the end of the period.
- However, if the constraint is based only on constants, fixed values and on monotonic terms, it is a *monotonic constraint* (or *m-constraint*) for which a decision could be made before. In the following, an *a-constraint* is an achievement constraint that is not monotonic.

Examples of m-constraints:

```
achieve inPMax (expr) < constant
achieve inPDuration (bool) > fixed
achieve inPClockCount (event) = 1
```

An a-constraint generally involves in-period terms and non-fixed expressions. It is sometimes more convenient and more expressive to use such a constraint under a *CTL* where a *DTL* would be sufficient.

Examples of a-constraints, where **bool** is a Boolean expression, **expr1** an expression, **expr2** a non monotonic expression, and **evt** an event:

```
during bool achieve inPMax (expr1) < expr2;
```

```
after evt within 10*s achieve bool;
```

This last example is nearly equivalent to:

```
when evt + 10*s ensure bool;
```

Decisions for a-constraints are made as with *DTL* and are straightforward: the following focuses on i-constraints and m-constraints.

If **nd** denotes a never-decreasing expression, **nd0** its value at the beginning of a period, and **nd1** and **nd2** fixed or constant values such that **nd2** > **nd1** > **nd0**, and if **ni** denotes a never-increasing expression, **ni0** its value at the beginning of a period, and **ni1** and **ni2** fixed or constant values such that **ni2** < **ni1** < **ni0** then:

```
nd =  nd0                    ni =  ni0
nd <  nd1                    ni >  ni1
nd <= nd1                    ni >= ni1
```

are i-constraints, and

```
nd in [nd1, nd2]             ni in [ni2, ni1]
not nd in [nd1, nd2]         not ni in [ni2, ni1]
nd >  nd0                    ni <  ni0
nd >  nd1                    ni <  ni1
nd >= nd1                    ni <= ni1
nd <> nd1                    ni <> ni1
nd =  nd1                    ni =  ni1
```

are m-constraints.

If **i**, **i1** and **i2** are i-constraints, and **m**, **m1** and **m2** are m-constraints

```
i1 or  i2    is an i-constraint
i1 and i2    is an i-constraint
i  or  m     is an m-constraint
i  and m     is an i-constraint
m1 or  m2    is an m-constraint
m1 and m2    is an m-constraint
not i        could be a p or an m-constraint
not m        could be a p or an m-constraint
```

In addition to Boolean time function **b** that defines it, each constraint has a decision-making Boolean time function **d** function (see Figure 12-3). The cases in a yellow background are those for which an early violation decision can be made, whereas those with a green background are those for which an early satisfaction can be made. These early decisions are made when **d** becomes `true` (event **ed**). Also, in the Figure:

- **s** stands for *inPSatisfaction*, and **es** for *eInPSatisfaction*.
- **v** stands for *inPViolation*, and **ev** for *eInPViolation*.
- They all depend on **b** and **d**.
- **s** and **v** are both initially `false`.
- They cannot be `true` at the same time.
- Once they become `true`, they remain so.
- `eop` denotes the normal end of the period (it does not occur and there might be no decision if the period is aborted before its normal end, by `eoc`, by an escape or by truncation in the framework of a composite action).
- **p** stands for a Boolean that is `true` at all instants of the period and `false` otherwise.

b	d	es	ev
bool	¬ b	eop ∧ *if* eop ∈ p *then* ¬ v *else* ¬ v.*previous*	d *while* p
nd = nd_0	¬ b		
nd < nd_2	¬ b		
nd <= nd_2	¬ b		
i_1 *or* i_2	i_1.v ∧ i_2.v		
i_1 *and* i_2	i_1.v ∨ i_2.v		
i *and* m	i.v ∨ m.v		
nd = nd_2	nd > nd_2		
nd *in* [nd_1, nd_2]	nd > nd_2		
not nd *in* [nd_1, nd_2]	nd > nd_2	d *while* p	eop ∧ *if* eop ∈ p *then* ¬ s *else* ¬ s.*previous*
nd <> nd_2	nd > nd_2		
nd > nd_0	b		
nd > nd_2	b		
nd >= nd_2	b		
m_1 *or* m_2	m_1.s ∨ m_2.s		
m_1 *and* m_2	m_1.s ∧ m_2.s		
i *or* m	i.s ∨ m.s		

Figure 12-3
Local decision making for a CTL-based constraint. The cases with never increasing expressions can be deduced from those with never decreasing expressions.

Non-Formalised Properties

Properties that are not formalised yet have the same attributes, but:

- Assumptions and guarantees are in *satisfaction*, *satisfied* is `true` and *violated* is `false`.
- The other properties are in *untested*, and *satisfied* and *violated* are both `false`.

12.7 Property Expressions

Three operators are available to combine the value and events of two or more properties (see Figure 12-4):

- *weakOr3* is in *satisfaction* if at least one input is in *satisfaction*, and in *violation* if all are in *violation*.
- *strongOr3* is in *satisfaction* if at least one input is in *satisfaction* and none are in *violation*, and in *violation* if any is in *violation*.

- ***and3*** is in *satisfaction* if all inputs are in *satisfaction*, and in *violation* if any is in *violation*.

weakOr3 , ***strongOr3*** and ***and3*** apply to sets of properties.

	U	eS	S	eV	V
U	U	eS	S	U	U
eS	eS	eS	S	eS	eS
S	S	S	S	S	S
eV	U	eS	S	eV	V
V	U	eS	S	V	V

weakOr3

	U	eS	S	eV	V
U	U	eS	S	eV	V
eS	eS	eS	S	eV	V
S	S	S	S	eV	V
eV	eV	eV	eV	eV	V
V	V	V	V	V	V

strongOr3

	U	eS	S	eV	V
U	U	U	U	eV	V
eS	U	S	eS	eV	V
S	U	eS	S	eV	V
eV	eV	eV	eV	eV	V
V	V	V	V	V	V

and3

Figure 12-4

Property operators. U *stands for* *untested*, eS *for* *eSatisfaction*, S *for* *satisfaction*, eV *for* *eViolation* *and* V *for* *violation*.

Other combinations must be obtained by explicit expressions based on property attributes.

Note. Property expressions must not mix *a posteriori* properties with operation-time properties.

12.8 Redeclaration

Through redeclaration:

- A ***Guarantee*** may be transformed either into a ***Requirement*** or into an ***Assumption***.
- An ***Objective*** may be transformed into a ***Requirement***.
- A ***Requirement*** may be transformed (and downgraded) into an ***Objective***, in which case a *substitution* specifying a set of substitute ***Requirement*** is expected. The substitute requirements for a formally defined ***Requirement*** must themselves be formally defined within the mock-up.
- A non-formally defined property of a certain class may be given a *concretisation* with one or more other, better defined properties of that same class.

```
Requirement #repower
  "During the lifetime of the installation, when the MPS is not
   available, the BPS shall ensure that its clients do not exhaust
   their tolerance level to loss of power";
...
repower is                                    // Implicit refinement
  during not mps.ok
  for all x of {Client}
  ensure x.tolerance >= 5*perCent;
...
redeclared Objective repower begin            // Explicit refinement
  substitution is {repowerPfd, repowerFro, repowerSar};
end repower;
```

12.9 Grammar

```
predefPropertyClass:
  'Property'    | 'Assumption' | 'Objective'
| 'Requirement' | 'Guarantee'  | 'Guard' ;
```

```
propertyDef:
  (constraintApplicator* & temporalLocator?) constraint
  In case of a single constraint, the optional constraint applicators and the optional temporal locator
  can be specified in any order.
```

```
| (constraintApplicator* ':')?
    temporalLocator constraintApplicator* constraint
    ('otherwise' temporalLocator constraintApplicator* constraint)*
    ('otherwise' constraintApplicator* constraint)?
```

In case of a temporal exclusion chain, each rung that is not the last one must specify a temporal locator, before possible constraint applicators. Some constraint applicators can be factored in.

```
;
```

```
constraint:
  ('assert' pathname (',' pathname)* 'can')? ('ensure' | 'achieve') bool ;
```

The ***pathname*** identify objects that must have at least one legitimate collective behaviour that ensures or achieves the ***bool*** condition.

13 Non-Valued Objects

A *non-valued object* generally represents a real world entity that does not have a natural `value` and that is instead characterised only by its *features* (valued or not). It may be a *one-of-a-kind object*, in which case it is an instance of predefined class `Object`. However, when several objects are of the same nature and have similar features, a defined class may be used to factor in and highlight the similarities: they are then *instances* of the class. An object may be an instance of multiple classes.

FORM-L non-valued objects and their classes are similar but not exactly the same as objects and classes in most object-oriented languages. In addition to the notion of *extension* (which, for classes, is similar to the notion of multiple inheritance), FORM-L has the notion of *refinement* (see Sections 5.9 and 7.10).

```
bps begin
  ...
  refined Client begin
    Boolean #poweredByBps
      "Whether the client receives power produced by the BPS"
      is deferred;

    Boolean #powered
      "Whether the client receives power (always through the BPS).
       Power may be produced either by the MPS or the BPS"
    begin
        during poweredByBps define value is true
      otherwise
        during not mps.ok and not poweredByBps define value is false
      otherwise value is deferred;
    end powered;
  end Client;
  ...
end bps;
```

This example is taken from the *Backup Power Supply* study. Non-valued object **`bps`** specifies a refinement of class **`Client`** with a definition block declaring and defining two additional features, **`poweredByBps`** and **`powered`**.

14 Expressions

An *expression* is a syntactic term that can be continuously evaluated to determine a value. The result of an expression based only on constants and not involving random or noise operators is constant. The result of an expression based only on constant and fixed items is fixed.

An expression that includes one or more *in-period terms* is an *in-period expression* and is evaluated separately in the framework of each time period of a *CTL* (see Section 16) or *STL* (see Section 17). An expression that includes one or more *a posteriori terms* is an *a posteriori expression* and is evaluated off-line based on the outcome of large sets of operational cases. An expression cannot be both in-period and *a posteriori*.

Expression **specific** is used only in template definitions: each instance MUST provide an explicit definition each item defined as **specific**. Expression **deferred** means that a definition will be given within the mock-up. That definition may be provided elsewhere in the model, in an extension model or through a binding.

14.1 In-Period Terms

An *in-period term* is a pre-defined term to be used under the scope of a temporal locator (generally a *CTL* or *STL* and evaluated separately in the framework of each time period of the temporal locator. When an in-period term is not used under the scope of a *CTL* or *STL*, then it is evaluated as its *in-time* counterpart (see *Figure 14-1*).

In-Period Terms		Corresponding In-Time Terms
inPTime	Time elapsed since the beginning of the period	**time**
inPClockTime	Number of clock ticks since the beginning of the period	**clockTime**
bop	Event occurrence at the beginning of the period, or the instant of a *DTL*. Its features are those of the event itself.	**t0**
eop	Event occurrence at the end of the period. Its features are those of the event itself.	**eoc**
inPCount **(event)**	Number of occurrences of **event** in the period, up to and including the current instant	***count*** **(event)**
inPMax **(int)** ***inPMax*** **(real)** ***inPMin*** **(int)** ***inPMin*** **(real)**	Min or max value in the period, up to and including the current instant	***max*** **(int)** ***max*** **(real)** ***min*** **(int)** ***min*** **(real)**
inPDuration **(bool)**	Duration in the period where **Bool** has been **true** (not necessarily continuously)	***duration*** **(bool)**
inPClockDuration **(bool)**	Number of clock ticks in the period where **bool(which** must be in discrete time**)** has been **true**	***clockDuration*** **(bool)**
inPIntegral *inPIntegral* **[n]** *inPIntegral* **[x]** *inPIntegral* **[x,n]**	Attributes, integral of a **real** in the period, up to the current instant	*integral* *integral* **[n]** *integral* **[x]** *integral* **[x,n]**

Figure 14-1
In-period *terms and their* in-time *counterparts.*

expr:
bool | ***int*** | ***real*** | ***state*** | ***string*** | ***event*** | ***set*** | ***property***

| '(' ***expr*** ')'

| ***pathname*** /* to a valued object or attribute */
value of a valued object or attribute identified by a ***pathname***.

| 'any' **NAME?** 'of' ***set*** ('such' 'that' ***bool***)?
Random selection of a single element of a ***set***, possibly after application of a filtering Boolean condition ***bool***.

| 'deferred' | 'specific'
To be used only in definitions, and only in class or class definitions in the case of **specific**.
;

14.2 Boolean Expressions

The value of a FORM-L *Boolean* variable is either `true` or `false`[10]. It could be viewed as a two-state automaton. Using one class or the other is just a matter of convenience: Booleans support the classical logical operations (*not*, *and*, *or*, ...) whereas automata states can be given meaningful names and may be refined into sub-automata when necessary.

bool:
  'true' | 'false'

| ***expr*₁** ('=' | '<>') ***expr*₂**
  expr*₁** ***and ***expr*₂** must be value expressions of the same nature, and the comparison applies to their value.

| (***int*₁** | ***real*₁**) (' <' | '<=' | '>=' | '>') (***int*₂** | ***real*₂**)
  real*₁** ***and ***real*₂** must be of the same *quantity* and the comparison takes the *scale* and *offset* into account. As ***int*** have no quantity, they cannot be compared to ***real*** with *quantity*.

| 'not' ***bool***
| ***bool*₁** ('and' | 'or' | 'xor' | 'excluding') ***bool*₂**
  Classical Boolean operators. *xor* is also obtained with the <> operator.

| 'AND' '(' ***set*** /* of Booleans */ ')'
  `true` if and only if all the members of ***setBool*** (a set of *Boolean* items or values) are `true` or if ***setBool*** is empty. (`true` is the neutral element of the Boolean *and* operator.)

| 'OR' '(' ***set*** /* of Booleans */ ')'
  `false` if and only if all the members of ***setBool*** are `false` or if ***setBoolExpr*** is empty. (`false` is the neutral element of the Boolean *or* operator.)

| ***expr*** 'in' ***set*** /* of Booleans */
| ***pathname*** 'in' ***set*** /* of items */
  Test of membership.

| ***set*₁** 'strictly'? 'in' ***set*₂**
  Test of set inclusion.

| ('no' | 'one') ***event***
  Equivalent respectively to 'inPCount' '(' ***event*** ')' '=' '0'
  and 'inPCount' '(' ***event*** ')' '=' '1'

| ***ctl***
  `true` at instants belonging to the ***ctl***, `false` otherwise.

| 'whether' 'some' ***selector*** 'satisfy' ***bool***
| 'whether' 'all' ***selector*** 'satisfy' ***bool***
  Existential and universal quantifiers.
;

14.3 Integer Expressions

The domain of value of a FORM-L *Integer* is the classical set of mathematical integers. However, as computers cannot represent arbitrarily large numbers[11], computer-based implementations of the language inevitably provide only an approximation. One just needs to make sure that they do not hinder the study of real-world systems.

int:
  INT
  Integer literal.

| '+%' | '-%'
  Largest representable positive or negative *Integer*

[10] In FORM-L, a Boolean is NOT an Integer.

[11] There are multiple ways to represent Integers in a computer. The most common are fixed-bit-number representations. 64 bit representations should be sufficient for most cases. Variable-bit-number representations are also possible but there is nonetheless a limit set by the amount of available computer memory.

```
| '-' int
| int1 ('+' | '-' | '*' | '/' | 'modulo' | '**') int2
```
Classical Integer operators.

```
| 'integer' '(' real ')'                    // Standard operator
```
Conversion by truncation of the *value* of **real** into an **Integer**. Rounding must be explicit.

```
| 'abs' '(' int ')'                         // Standard operator
```
Absolute value of **int**.

```
| 'min' '(' int1 (',' intn)+ ')'            // Standard operator
| 'max' '(' int1 (',' intn)+ ')'            // Standard operator
```
Mini / max of two or more **Int**.

```
| 'min' '(' int ')'                         // Standard operator
| 'max' '(' int ')'                         // Standard operator
```
Min / max of **int** up to current instant.

```
| 'inPMin' '(' int ')'                      // Standard operator
| 'inPMax' '(' int ')'                      // Standard operator
```
Min / max of **Int** in a period of a *CTL* up to current instant.

```
| 'sum' '(' setInt ')'                      // Standard operator
```
Sum of all the members of **setInt** (a set of *Integers*) or **0** if **setInt** is empty.

```
| 'product' '(' setInt ')'                  // Standard operator
```
Product of all the members of **setInt** or **1** if **setInt** is empty.

```
| 'MIN' '(' setInt ')'                      // Standard operator
```
Smallest member of **setInt** or **+%** if **setInt** is empty.

```
| 'MAX' '(' setInt ')'                      // Standard operator
```
Largest member of **setInt** or **-%** if **setInt** is empty.

```
| 'count' '(' set ')'                       // Standard operator
```
Cardinal of **set**.

```
| 'count' '(' event ')'                     // Standard operator
```
Number of occurrences of **event** up to and including the current instant.

```
| 'inPCount' '(' event ')'                  // Standard operator
```
Number of occurrences of **event** in a period, up to and including the current instant.

```
| 'clockTime'                               // Standard operator
```
In a discrete time domain, number of clock ticks since **t0**, up to and including the current instant.

```
| 'inPClockTime'                            // Standard operator
```
In a discrete time domain, number of clock ticks in a period, up to and including the current instant.

```
| 'clockDuration' '(' bool ')'              // Standard operator
```
In a discrete time domain, number of clock ticks since **t0**, up to and including the current instant, where **bool** has been **true**.

```
| 'inPClockDuration' '(' bool ')'           // Standard operator
```
In a discrete time domain, number of clock ticks in a period, up to and including the current instant, where **bool** has been **true**.

```
| ('optimin' | 'optimax') int dtl           // Standard operator
```
A posteriori minimal or maximal value of **int** at the first instant of **dtl**, over a set of operational cases. Used in particular to specify optimisation properties.

```
;
```

14.4 Real Expressions

FORM-L **Real** variables aim at representing the classical real numbers of mathematics. However, even more so than for **Integer**, computer-based tools inevitably provide only an approximation[12]: not only

[12] There are multiple ways to represent **Reals** in a computer. The most common are IEEE-style representations. However, they may have counter-intuitive effects (e.g., 10 * 0.1 is not equal to 1) that could be solved by other styles of representation, e.g., fraction-based representations (where a Real is approximated by two integer numbers and a power of 10).

computers cannot represent arbitrarily large numbers, they also have difficulties representing irrational numbers (such as $\sqrt{2}$) and cannot represent transcendent numbers (such as π) at all.

***real*:**
REAL
Real literal.

| **'+%'** | **'-%'**
Largest representable positive or negative ***Real***

| **'-'** ***real***
| ***real*$_1$ **('+'** | **'-'** | **'*'** | **'/'** | **'**'**)** ***real*$_2$**
| ***real*** **'**'** ***int***
Classical ***Real*** operators. For binary **+** and **-**, the arguments must be of the same ***quantity***. For ***Real*** powers (******), they must not be quantities.

| **'real'** **'('** ***int*** **')'** **// Standard operator**
Conversion of ***int*** into a ***Real***.

| **'conversion'** **'('** ***real*$_1$** **','** ***real*$_2$** **')'** **// Standard operator**
Conversion of ***real*$_1$** to the unit of ***real*$_2$**. Both must be of the same ***quantity***. The conversion is based on the ***scale*** and ***offset*** attributes: the ***value*** of ***real*$_2$** is not taken into consideration.

| **'abs'** **'('** ***real*** **')'** **// Standard operator**
Absolute value of ***real***.

| **'min'** **'('** ***real*$_1$** **(','** ***real*$_n$**)+ **')'** **// Standard operator**
| **'max'** **'('** ***real*$_1$** **(','** ***real*$_n$**)+ **')'** **// Standard operator**
Mini / max of two or more ***real*** of the same quantity.

| **'min'** **'('** ***real*** **')'** **// Standard operator**
| **'max'** **'('** ***real*** **')'** **// Standard operator**
Min / max of ***Real*** up to and including the current instant.

| **'inPMin'** **'('** ***real*** **')'** **// Standard operator**
| **'inPMax'** **'('** ***real*** **')'** **// Standard operator**
Min / max of ***real*** in a period of a *CTL*, up to and including the current instant.

| **'sum'** **'('** ***setReal*** **')'** **// Standard operator**
Sum of all the members of ***setReal*** (a set of *Real*s of the same quantity) or 0 if ***setReal*** is empty.

| **'product'** **'('** ***setReal*** **')'** **// Standard operator**
Product of all the members of ***setReal*** or 1 if ***setReal*** is empty.

| **'MIN'** **'('** ***setReal*** **')'** **// Standard operator**
Smallest member of ***setReal*** or **+%** if ***setReal*** is empty.

| **'MAX'** **'('** ***setReal*** **')'** **// Standard operator**
Largest member of ***setReal*** or **-%** if ***setReal*** is empty.

| **'log'** **'('** ***real*** **')'** **// Standard operator**
| **'log10'** **'('** ***real*** **')'** **// Standard operator**
| **'exp'** **'('** ***real*** **')'** **// Standard operator**
| **'exp10'** **'('** ***real*** **')'** **// Standard operator**
| **'sin'** **'('** ***real*** **')'** **// Standard operator**
| **'cos'** **'('** ***real*** **')'** **// Standard operator**
| **'tan'** **'('** ***real*** **')'** **// Standard operator**
| **'arcSin'** **'('** ***real*** **')'** **// Standard operator**
| **'arcCos'** **'('** ***real*** **')'** **// Standard operator**
| **'arcTan'** **'('** ***real*** **')'** **// Standard operator**
Classical mathematical operators. ***real*** must be scalar. Hyperbolic functions could also be added.

| **'time'** **// Standard operator**
Duration since the beginning of the operational case.

| **'inPTime'** **// Standard operator**
Duration since the beginning of a period of a *CTL* or *STL*.

| **'duration'** **'('** ***bool*** **')'** **// Standard operator**

Duration where ***bool*** has been *true* since *t0*.

```
| 'inPDuration' '(' bool ')'              // Standard operator
```
Duration where ***bool*** has been *true* since the beginning of a period of a *CTL*.

```
| ('optimin' | 'optimax') real dtl        // Standard operator
```
A posteriori minimal or maximal value of ***real*** at the first instant of ***dtl***, over a set of operational cases. Used in particular to specify optimisation properties.

```
;
```

14.5 Finite State Expressions

A state value is determined either by a state literal (in which case it is constant) or by the current state of a *permanent automaton* (in which case it may vary in time). An automaton is permanent when it is in a well-defined state at all times. This is the case of top-level automata, i.e., automata not embedded as a composite state in another one. Automata embedded as a composite state of another automaton is active only when the parent automaton is in that state. If a permanent automaton is a Cartesian product, then each element of the product is also permanent.

No operators on state values are provided beyond those applicable to general expressions.

A state expression may vary in time, but its value must always be relative to the same enumeration or automaton.

When an intermediate node is a Cartesian product, then a bracketed, &-separated notation is used, with one item per product element.

```
Enumeration [g, h, i] C;
Enumeration [k, h] D;
Enumeration [l, m, n] E;
Enumeration [m, n] F;
Enumeration [C c, p, q, [D d1 & D d2] d] A;
Enumeration [q, [E e & F f] e] B;

A a; B b;
```

Examples of state literals:

```
• A.p
• a.p
• p            // In A's or a's name space

• A.q
• b.q
• q            // In A's or b's name space

• A.c.g
• c.g          // In A's or a's name space
• g            // In C's or c's name space

• a.d          // Sub-automaton and its set of states
• a.d.[k & h]
• a.d.[k & k]
• d.[k & h]    // In A's or a's name space
• [k & k]      // In D's or d1's or d2's name space
```

```
Enumeration [on, off, inBetween] SwitchState;

[SwitchState a & SwitchState b & SwitchState c] systState;
```

In this example, the state of a system (**systState**) is defined by the **on** / **off** / **inBetween** states of its three switches **a**, **b** and **c**. Expression

```
systState.b
```

denotes an automaton (a variable representing the state of the **b** switch of the system) and its current state, whereas the following expression denotes a particular state literal (a constant) of the system, where each switch is in a specific state:

```
systState.[on & inBetween & off]
```

The following one denotes a system state where switches **a** and **c** are respectively **on** and **off**, and switch **b** is in any state:

```
systState.[on & b & off]
```

Lastly, the following one denotes a system state where switches **a** and **c** are again respectively **on** and **off**, and switch **b** is either **on** or **inBetween**:

```
systState.[on & {on, inBetween} & off]
```

```
state:
   pathname
   Current state of the automaton designated by pathname.
| localState
   localState without a preceding path is allowed only in the name space of a specific
   enumeration or automaton, and designates a single but complete state of that enumeration or
   automaton.
| pathname '.' localState
   Single but complete state of the enumeration or automaton identified by pathname.
;
localState:
   NAME ('.' NAME)* ('.' cartProductState)?
| cProductState
;
cartProductState: '[' localState ('&' localState)+ ']' ;
   State of an embedded Cartesian product.
```

14.6 String Expressions

FORM-L *Strings* aim at representing any text in any natural language. Character representation is implementation specific but should not be a hindrance in the use of natural languages. Operations on *Strings* are very limited in FORM-L: they are either constant or fixed.

```
string: STRING ;
```

14.7 Set Expressions

```
set:
   setObjects | setValues
| set1 ('union' | 'inter' | 'excl') set2
   Union, intersection and subtraction of two compatible set.
| ('UNION' | 'INTER') set
   Union and intersection of all the members of a set of compatible sets.
| set binOp expr
| expr binOp set
   Shortcut for   'for' 'all' NAME 'of' set ':' NAME binOp expr.
   or             'for' 'all' NAME 'of' set ':' expr binOp. NAME
   binOp stands for a binary, infixed operator, such as +, * or /.
| unOp set
   Shortcut for      'for' 'all' NAME 'of' set ':' unOp. NAME
   unOp stands for a unary, prefixed operator, such as -.
| 'for' 'all' selector ':' expr
   Application of expr to all selected set members.
| 'all' selector
   Selection of a subset.
;
```

14.7.1 Set of Objects Expressions

setObjects:
  '{' (***pathname***$_1$ (**','** ***pathname***$_n$)*)? **'}'**
  Literal in extension definition, with a curly-bracketed list of ***pathname*** to member items. A ***pathname*** leading to a class stands for all its instances.

| ***pathname***
  In the framework of an expression, a ***pathname*** to a class denotes the implicitly defined set of all named and dynamically created instances of that class.

| ***setObjects*** (**'with'** | **'without'**) ***pathname***
  Insertion in or removal from a ***setObjects*** of an object identified by a ***pathname***.
;

14.7.2 Set of Values Expressions

setValues:
  '$'** **'{'** (***expr***$_1$ (','** ***expr***$_n$)*)? **'}'**
  Definition in extension with a curly-bracketed list of expressions.

| ***setValues*** (**'with'** | **'without'**) ***expr***
  Insertion in or removal from a ***set*** of a value.
;

14.8 Event Expressions

event:
  't0'
  This pre-defined event marks the beginning of time for operational cases.

| **'eoc'**
  This pre-declared event marks the end of time for operational cases. It may be signalled explicitly by a mock-up, or occurs when simulation or analysis is terminated.

| **'every'** ***duration***
  First occurrence at **t0**. Then, iteratively, a new occurrence after each ***duration*** interval.

| ***expr*** **'becomes'** ***setValues***
  Signalled each time ***expr*** becomes equal to a value in ***setValues***.

| ***expr*** **'leaves'** ***setValues***
  Signalled each time ***expr*** becomes different from all values in ***setValues***.

| ***expr*** **'leaves'** ***setValues***$_1$ **'becomes'** ***setValues***$_2$
  Signalled each time ***expr*** leaves ***setValues***$_1$ AND enters ***setValues***$_2$.

| ***expr*** **'changes'**
  Signalled each time the value of ***expr*** is modified. Used mainly for discrete values (Booleans, integers, states and sets).

| ***expr*** **'mutates'**
  Signalled each time the value or any feature of ***expr*** is modified. May be applied to any object.

| ***set*** **'CHANGES'**
  Produces a set of events, one per ***set*** member, signalled each time the value of the member is modified.

| ***set*** **'MUTATES'**
  Produces a set of events, one per ***set*** member, signalled each time the value or any feature of the member is modified.

| ***event***$_1$ **'or'** ***event***$_2$
  Union of the occurrences of the two ***event***.

| **'OR'** **'('** ***set*** **/* of events */** **')'**
  Union of the occurrences of the events of the ***set***.

| ***event***$_1$ **'xor'** ***event***$_2$
  Union of the occurrences of the two ***event***, except when both occur at the same time.

| ***event₁*** **'and'** ***event₂***
Occurrences of the two ***event*** that occur at the same time.

| **'AND'** **'('** ***set*** **/* of events */** **')'**
Occurrences of the events of the ***set*** that occur at the same time.

| ***event₁*** **'excluding'** ***event₂***
Occurrences of ***event₁*** that do not occur at the same time as an occurrence of ***event₂***.

| ***event₁*** **'following'** ***event₂***
First occurrence of ***event₁*** after an occurrence of ***event₂***.

| ***event*** **'+'** ***duration***
| ***event*** **'delayed'** ***duration***
Occurrences of ***event*** delayed by the value of ***duration*** at the occurrence.

| **'first'** **('('** **INT** **')')?** ***event***
First occurrence or first **INT** occurrences of ***event***.

| ***event*** **'except'** **'first'** **('('** **INT** **')')?**
All occurrences of ***event*** except the first or first **INT** ones.

| ***event*** **'while'** ***bool***
All occurrences of ***event*** at times where ***bool*** is **true**.

| ***event*** **'such'** **'that'** ***bool***
All occurrences of ***event*** for which ***bool*** (a Boolean expression involving features of ***event***) is **true**.

14.9 Property Expressions

```
property:
| property₁ ('weakOr3' | 'strongOr3' | 'and3') property₂
| ('weakOR3' | 'strongOR3' | 'AND3') set /* of properties */
;
```

15 Discrete Temporal Locators (*DTL*)

A *DTL* defines a finite number of positions in time that have no duration. This notion is thus the same as the notion of event, and there is no need for named *DTL*. One usually use term *event* in expressions or when expressing *CTL*, and term *DTL* when expressing the temporal locator of an instruction. The difference is purely syntactic: the expression of a *DTL* may be preceded by keyword ***when***.

```
dtl:
   '(' dtl ')'
|  'when'? event
;
```

16 Continuous Temporal Locators (*CTL*)

A *CTL* defines a finite number of possibly overlapping time periods that have a strictly positive duration. In some cases, it is important to specify whether the beginning and end boundaries belong or not to a period. As a general rule:

- With keyword ***always***, both boundaries (**t0** and **eoc**) belong to the period ▶■■◀.
- With keyword ***from***, the beginning boundary belongs to the period ▶■. With ***after***, it does not ■◀.
- With keywords ***until*** or ***during***, the end boundary belongs to the period ■◀. With ***before*** or ***within***, it does not ▶■◀.

Ctl: **'private'? 'Ctl' NAME STRING* ('is' *ctl*)? ';' ;**
Declaration and possibly definition of a named *CTL*.

ctl:
'(' *ctl* ')'
| **'during' *pathname***
Named *CTL* identified by a ***pathname***.

16.1 Basic *CTL*

| **'always'**
If no temporal locator is specified where one is expected, then the default temporal locator is ***always***, a single time period from and including **t0**, until and including **eoc** (see *Figure 16-1*).

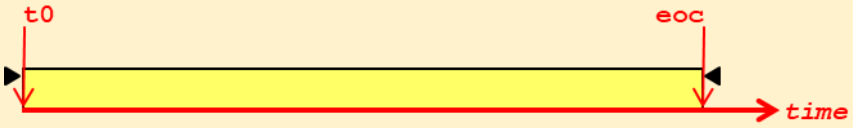


Figure 16-1
always

| **'during' *bool***
The time periods are those where the ***bool*** condition is continuously **true** (see *Figure 16-2*). In this case, they cannot overlap. It is nearly equivalent to.
***from* bool *becomes* true *until* bool *becomes* false**
The only difference is that **eoc** may end normally the last period, but does not truncate it.

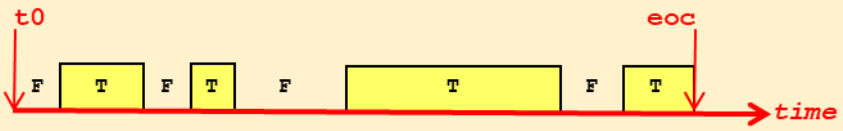


Figure 16-2
during* *bool

| **('from' | 'after') (*event* | *bool*)**
This opens a new time period at each occurrence of ***event*** or of ***bool* *becomes* true**, ending at **eoc**. With ***from***, the beginning boundary belongs to the period, whereas with ***after***, it does not. In both cases, the end boundary belongs to the period (see *Figure 16-3* and *Figure 16-4*). There is no period at all if **eoc** occurs before or at the same time as the first occurrence of ***event*** or of ***bool* *becomes* true**. If there are more than one period, they will overlap.

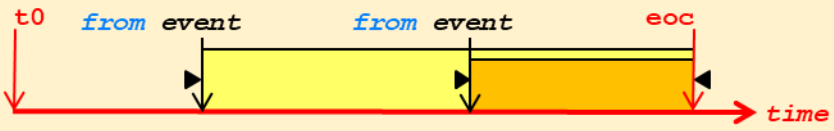


Figure 16-3
from* *event

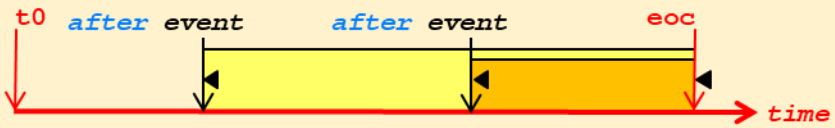


Figure 16-4
after event

| **('from' | 'after') *duration***

This defines a single time period beginning at **t0** + a fixed ***duration*** and ending at **eoc**. With ***from***, the beginning boundary belongs to the period, whereas with ***after***, it does not. In both cases, the end boundary belongs to the period (see *Figure 16-5* and *Figure 16-6*). There may be no period at all if **eoc** occurs before or at the same time as **t0** + ***duration***.

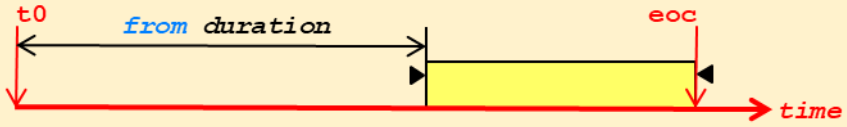


Figure 16-5
from duration

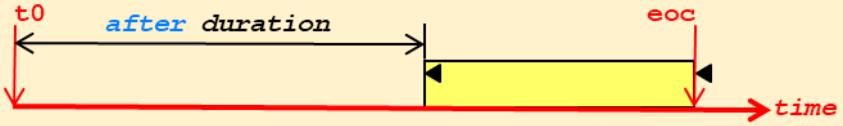


Figure 16-6
after duration

| **('until' | 'before') (*event* | *bool*)**

This defines a single time period from and including **t0** and normally ending at the first occurrence of ***event*** or of ***bool*** ***becomes*** **true**. With ***until***, the normal end boundary belongs to the period, whereas with ***before***, it does not (see *Figure 16-7* and *Figure 16-8*). The period may be truncated if **eoc** occurs before the first occurrence of ***event*** or of ***bool*** ***becomes*** **true**.

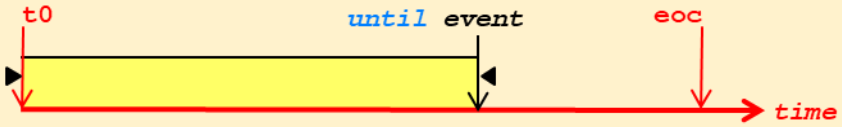


Figure 16-7
***until* event**

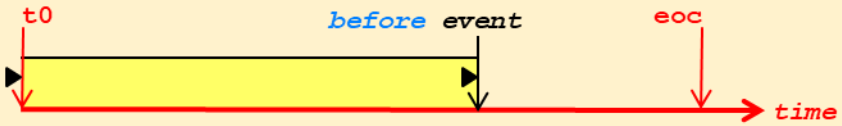


Figure 16-8
***before* event**

| **('until' | 'before') *duration***

This defines a single time period from and including **t0** and normally ending at **t0** + a fixed ***duration***. With ***until***, the normal end boundary belongs to the period, whereas with ***before***, it does no (see *Figure 16-9* and *Figure 16-10*). The period is truncated if **eoc** occurs before **t0** + ***duration***.

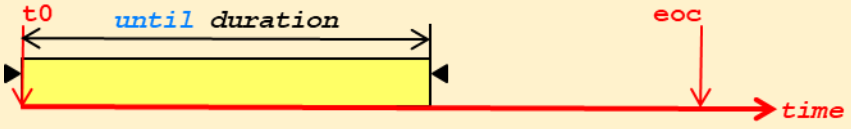


Figure 16-9
until duration

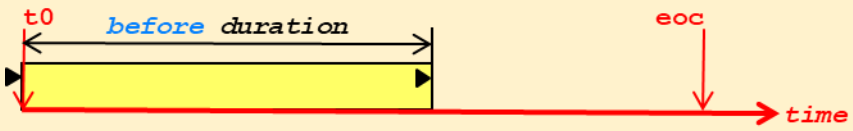


Figure 16-10
before duration

| **('from' | 'after') (event₁ | bool₁) ('until' | 'before') (event₂ | bool₂)**
This opens a new time period at each occurrence of ***event₁*** or of ***bool₁*** ***becomes*** **true**, normally ending at the following occurrence of ***event₂*** or of ***bool₂*** ***becomes*** **true**. Whether the period boundaries belong to it depends on the keywords used. A time period is truncated if **eoc** occurs before its normal end. If there are more than one period, they may overlap.

| **('from' | 'after') (event | bool) 'during' duration // ensure**
This is equivalent to
from **(event | bool)** ***until*** **bop + duration**
or ***after*** **(event | bool)** ***until*** **bop + duration**
and is intended to facilitate understanding in the case of invariant constraints.

| **('from' | 'after') (event | bool) 'within' duration // achieve**
This is equivalent to
from **(event | bool)** ***before*** **bop + duration**
or ***after*** **(event | bool)** ***before*** **bop + duration**
and is intended to facilitate understanding in the case of achievement constraints.

| **'from' 'every' duration 'until' (event | bool) ...**
| **'after' 'every' duration₁ 'during' duration₂ ... // ensure**
| **'from' 'every' duration₁ 'within' duration₂ ... // achieve**

These use periodic event ***every*** **duration** as periods beginning (see respectively *Figure 16-11*, *Figure 16-12* and *Figure 16-13*).

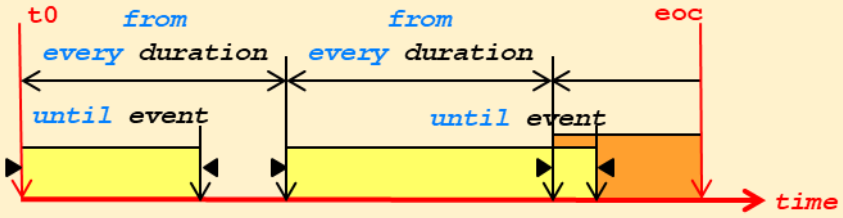


Figure 16-11
from every **duration** ***until*** **event**.
The second and third periods overlap. The third period is truncated by **eoc**.

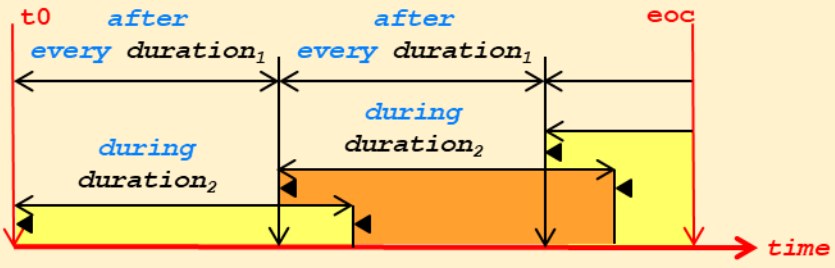


Figure 16-12
after every **duration₁** ***during*** **duration₂**.
As **duration₂** *is longer than* **duration₁**, *the periods overlap.*

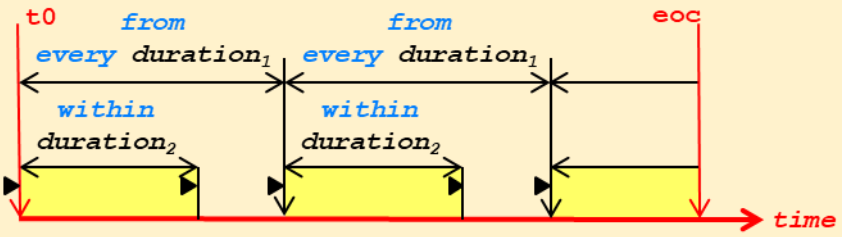


Figure 16-13
`from every duration1 within duration2`

16.2 Combining & Transforming *CTL*

`'not' ctl`
The time periods of the resulting *CTL* are those obtained by excluding from **`always`** any instant belonging to a time period of the original **`ctl`** (see *Figure 16-14*). It is to be noted that if the original **`ctl`** has overlapping time periods, then **`not not ctl`** is not equivalent to **`ctl`** but to **`flat ctl`**.

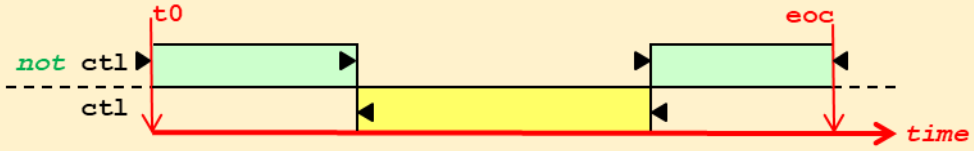


Figure 16-14
`not ctl`, *with a few boundary examples.*

`'flat' ctl`
In the resulting *CTL*, the overlapping time periods of the original **`ctl`** are merged. Instants not covered by the original **`ctl`** are also not covered by the resulting *CTL* (see *Figure 16-15*).

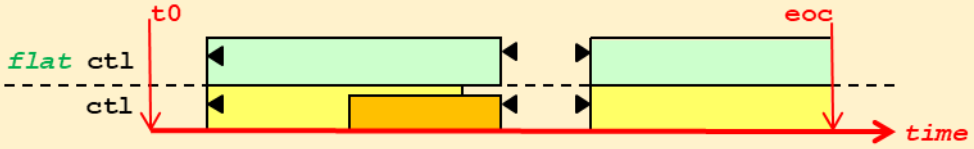


Figure 16-15
`flat ctl`, *with a few boundary examples.*

`ctl 'delayed' duration`
The resulting *CTL* is the time-shift of each time periods of the original **`ctl`** (see *Figure 16-16*). The **`duration`** of the time-shift is evaluated at the beginning of each of the time periods to be shifted and must be positive. Some of the shifted periods may be truncated by **`eoc`**.

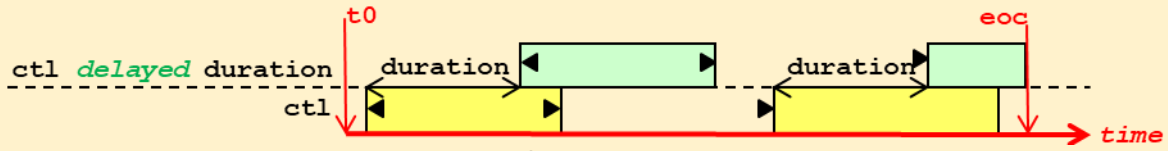


Figure 16-16
`ctl delayed duration`, *with a few boundary examples.*

`ctl 'except' 'first' duration`
The resulting *CTL* is composed of the time-truncation of the time periods of the original **`ctl`**. If the truncation **`duration`** is longer or equal to the duration of an individual time period, then that period is eliminated (see *Figure 16-17*). The **`duration`** is evaluated at the beginning of each of the time periods to be truncated and must not be negative.

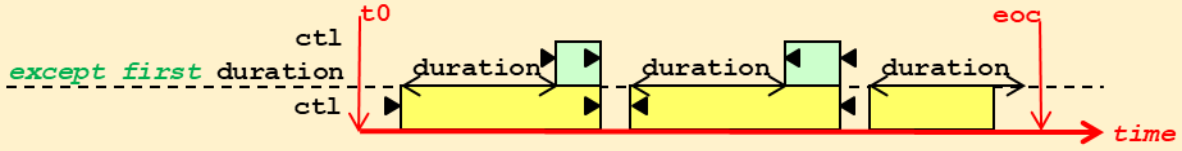


Figure 16-17
`ctl except first duration`, *with a few boundary examples.*

| ***ctl*** **'prolonged'** ***duration***

The resulting *CTL* is composed of the time-extension of the time periods of the original ***ctl*** (see *Figure 16-18*). The ***duration*** is evaluated at the beginning of each of the time periods to be prolonged and must not be negative.

Note that the converse operator (time-truncation) is not feasible in simulation, since the end of a truncated period would occur before the end of the original one: that would require to have full knowledge of the future..

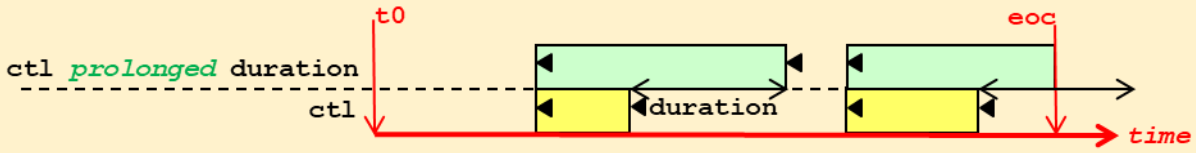


Figure 16-18
ctl ***prolonged*** **duration**.

| ***ctl*** **'while'** ***bool***

The time periods of the resulting *CTL* are obtained by excluding from each time period of the original ***ctl*** any instant where ***bool*** is *false* (see Figure 16-19). Empty periods are eliminated.

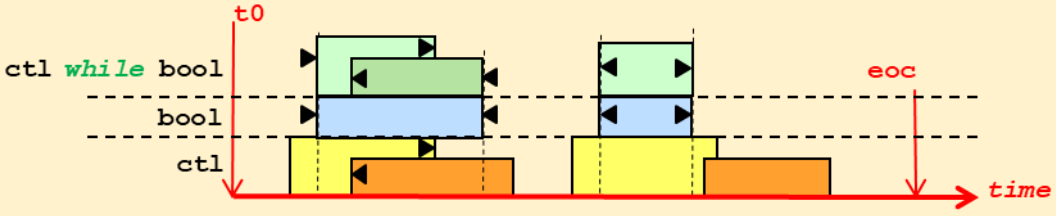


Figure 16-19
ctl ***while*** **bool**, *with a few boundary examples.*

| ***ctl*** **'excluding'** ***dtl***

The time periods of the resulting *CTL* are obtained by excluding the instants of ***dtl*** from each time period of ***ctl*** (which means splitting it into multiple time periods, and / or excluding one or both boundaries).

| ***ctl*** **'escape'** **('from'** | **'after')** ***event***

The ***event*** occurrences forcefully terminate the periods during which they occur.

;

17 Sliding Temporal Locators (*STL*)

An *STL* is introduced by keywords ***during all*** or ***during some***. It denotes all the time periods of a given constant or fixed ***Duration***, possibly excluding those during which a Boolean condition introduced by keywords ***such that*** is not continuously **true**. Both boundaries belong to the periods. With keywords ***during all***, the constraint needs to be satisfied by all periods, whereas with keywords ***during some***, it needs to be satisfied by at least one.

STL in discrete time are not necessary, as they always represent a finite number of periods, which is exactly the role of CTL.

Stl: **'private'? 'Stl' NAME STRING* ('is' *stl*)? ';'** ;
Declaration and possibly definition of a named *STL*.

StlDefinition: ***pathname*** **STRING* 'is'** ***stl*** **';'** ;
Definition of a previously declared *STL*.

stl: **'during' ('all' | 'some')** ***duration*** **('such' 'that'** ***bool*****)?** ;

Property **p** ***is during all*** **5*mn** ***such that*** **normalOperation** ***achieve*** **eSignal;**

Property **p** is in *satisfaction* in each 5-minute time period during which Boolean **normalOperation** has been continuously **true** when event **eSignal** occurs. Conversely, it is in *violation* at the end of each 5-minute time period during which Boolean **normalOperation** has been continuously **true** and event **eSignal** has not occurred.

STL are more difficult to handle than *DTL* or even *CTL*. Thus, they do not have operators, can only be used as temporal locators of properties, and the use of in-period terms is limited. Their semantics is defined in the following by specifying when *eInPViolation* and *eInPSatisfaction* occur.

17.1 Unconditional *STL*, Universal Constraint, No In-Period Term

Here:

- **d** stands for a strictly positive constant or fixed ***Duration***.
- **b** stands for a ***Boolean*** expression that contains no in-period term.

during all **d** ***ensure*** **b;**

- *eInPSatisfaction* occurs at the end of a period during which **b** has been continuously **true**.
- *eInPViolation* occurs when **b** is **false** at the beginning of a period or when **b** ***becomes*** **false** during a period.
- *eInPSatisfaction* for one period and *eInPViolation* for another cannot occur simultaneously since that would require **b** to be **true** and **false** at the same time.

eSatisfaction ***is when*** **(*during*** **b** ***except first*** **d)** ***becomes*** **true;**
eViolation ***is when*** **(t0** ***while not*** **b)** ***or*** **(b** ***becomes*** **false);**

17.2 Unconditional *STL*, Universal Constraint, Single Monotonic In-Period Term

STL constraints are severely restricted in their use of in-period terms: they can only compare a single monotonic in-period term with a constant or fixed value. This sections applies to in-period terms the in-time correspondent of which (see 14.1 *In-Period Terms*) is also monotonic: ***inPCount***, ***inPDuration*** or ***inPClockDuration*** and **inPClockTime**. (In this context, **inPTime** is not very useful.) *inPIntegral* is allowed only for variables that are guaranteed to be always positive or always negative.

As it is impossible to deal with an infinite number of periods one by one, the constraint must be substituted with an equivalent one without any in-period term. In the following:

- **d** is a strictly positive constant or fixed ***Duration***.
- **inPNd** is a single monotonic (non-decreasing) in-period term.

- **k** is a strictly positive constant or fixed expression of the same class as **inPNd**.
- **inTNd** is the in-time correspondent of **inPNd**.

If **bop** the beginning of a period and **t** is an instant in that period:

$$\mathbf{inTNd}_{\mathbf{bop+t}} = \mathbf{inTNd}.\mathit{previous}_{\mathbf{bop}} + \mathbf{inPNd}_{\mathbf{t}}$$

$$\mathbf{inPNd}_{\mathbf{t}} = \mathbf{inTNd}_{\mathbf{bop+t}} - \mathbf{inTNd}.\mathit{previous}_{\mathbf{bop}}$$

In particular:

$$\mathbf{inPNd}_{\mathbf{d}} = \mathbf{inTNd}_{\mathbf{bop+d}} - \mathbf{inTNd}.\mathit{previous}_{\mathbf{bop}}$$

represents the value of **inPNd** at the end of the period and is expressed in FORM-L as

```
nd is inTNd – (inTNd delayed d).previous
```

The advantage of this expression is that it does not contain any in-period terms.

Before **t0+d**, **inTNd** *delayed* **d** is naught, and thus **inTNd**, **inPNd** and **nd** are all equal.

Simultaneous *eInPViolation* for one period and *eInPSatisfaction* for another cannot occur since that would require a contradiction on **nd**.

```
during all d achieve inPNd >  k;
during all d achieve inPNd >= k;
during all d achieve inPNd <> k;
```

- *eInPViolation* occurs when and only when, at the end of a period, **inPNd** *op* **k** is **false**, i.e., when **nd** *op* **k** is **false**. It cannot occur before **t0+d**.
- All periods during which **inPNd** *op* **k** *becomes* **true** signal *eInPSatisfaction*. (In the case of **<>**, that happens only when **inPNd** becomes greater than **k**.) From **t0+d**, that necessarily happens at the end of some other period, the **inPNd** of which is necessarily superior or equal to the one that signal *eInPSatisfaction*. Thus at that instant, **nd** *op* **k** is **true**.
- *eInPSatisfaction* also occurs when, at the end of a period, **inPNd** *op* **k** is **true**, i.e., when **nd** *op* **k** is **true**.

```
eSatisfaction is when nd op k becomes true;
eViolation    is when nd op k becomes false;
```

```
during all d ensure  inPNd <  k;
during all d ensure  inPNd <= k;
during all d achieve inPNd =  k;
```

- *eInPSatisfaction* occurs when and only when, at the end of a period, **inPNd** *op* **k** is **true**, i.e., when **nd** *op* **k** is **true**. It cannot occur before **t0+d**.
- All periods during which **inPNd** *op* **k** *becomes* **false** signal *eInPViolation*. (In the case of **=**, that happens only when **inPNd** becomes greater than **k**.) From **t0+d**, that necessarily happens at the end of some other period, the **inPNd** of which is necessarily superior or equal to the one that signal *eInPViolation*. Thus at that instant, **nd** *op* **k** is **true**.
- *eInPViolation* also occurs when, at the end of a period, **inPNd** *op* **k** is **false**, i.e., when **nd** *op* **k** is **false**.

```
eSatisfaction is when nd op k becomes true;
eViolation    is when nd op k becomes false;
```

17.3 Unconditional *STL*, Universal Constraint With *inPMax*, *inPMin*

The equivalences are given for *inPMax*: the same apply to *inPMin*, *mutatis mutandis*. Here:

- **d** is a strictly positive constant or fixed *Duration*.
- **e** is a numerical expression (i.e., of class *Integer*, *Real* or one of their extensions).
- **k** is a strictly positive constant or fixed expression of the same class as **e**.

```
during all d achieve inPMax (e) > k;
during all d achieve          e  > k;
```

```
during all d achieve inPMax (e) >= k;
during all d achieve         e  >= k;
```

```
during all d ensure  inPMax (e) < k;
during all d ensure          e  < k;
```

```
during all d ensure  inPMax (e) <= k;
during all d ensure          e  <= k;
```

```
during all d achieve inPMax (e) = k;
(during all d achieve e = k) and3 (during all d ensure e <= k);
```

```
during all d achieve inPMax (e) <> k;
(during all d ensure e < k) weakOr3 (during all d achieve e > k);
```

17.4 Conditional *STL*

With a filtering Boolean condition (currently without in-period terms), the evaluation of the property changes radically, since a decision (*eInPViolation* or *eInPSatisfaction*) for a period can be made only at its end, and only if the filtering condition has been continuously true during the whole period. Here:

- **d** is a strictly positive constant or fixed *Duration*.
- **f** is a *Boolean* filtering expression that contains no in-period term.
- **b** is a *Boolean* expression that also contains no in-period term.
- **k** is strictly positive constant or fixed expression of the same type as **inPNd**.
- **inPNd** is a single monotonic (non-decreasing) term.
- **nd** *is* **inTNd – (inTNd** *delayed* **d)**, where **inTNd** is the in-time correspondent of **inPNd**.

```
during all d such that f ensure b;
```

- *eInPSatisfaction* occurs at the end of periods during which both **f** and **b** have been continuously **true**.
- *eInPViolation* occurs at the end of periods during which **f** has been continuously **true** and **b** has not.

```
eSatisfaction is when (during (f and b) except first d) becomes true;
eViolation is when ((during not b prolonged d)
              and   (during f except first d)) becomes true;
```

```
during all d such that f achieve inPNd >  k;
during all d such that f achieve inPNd >= k;
during all d such that f achieve inPNd <> k;
during all d such that f ensure  inPNd <  k;
during all d such that f ensure  inPNd <= k;
during all d such that f achieve inPNd =  k;
```

- *eInPSatisfaction* occurs when, at the end of periods during which **f** has been continuously **true**, **inPNd** *op* **k** is **true** (i.e., **nd** *op* **k** is **true**).
- *eInPViolation* occurs when, at the end of periods during which **f** has been continuously **true**, **inPNd** *op* **k** is **false** (i.e., **nd** *op* **k** is **false**).

```
eSatisfaction is when nd op k becomes true while (during f except first d);
eViolation is when nd op k becomes false while (during f except first d);
```

```
during all d such that f achieve inPMax (e) > k;
during all d such that f achieve         e  > k;
```

```
during all d such that f achieve inPMax (e) >= k;
during all d such that f achieve         e  >= k;
```

```
during all d such that f ensure  inPMax (e) < k;
during all d such that f ensure           e  < k;
```

```
during all d such that f ensure  inPMax (e) <= k;
during all d such that f ensure           e  <= k;
```

```
during all d such that f achieve inPMax (e) = k;
(during all d such that f achieve e = k) and3
(during all d such that f ensure e <= k);
```

```
during all d such that f achieve inPMax (e) <> k;
(during all d such that f ensure e < k) weakOr3
(during all d such that f achieve e > k);
```

17.5 Existential Constraints

Here:

- `d` is a strictly positive constant or fixed *Duration*.
- `f` is a *Boolean* filtering expression that contains no in-period term.
- `c` is a constraint.

```
during some d c;
```

- *eSatisfaction* occurs like for `during all d c`.
- *eViolation* occurs at `eoc` if at least one `d` period has elapsed and *eSatisfaction* has never occurred.

```
eSatisfaction is first (during all b c).eSatisfaction;
eViolation is when eoc while time >= d and count (eSatisfaction) = 0;
```

```
during some d such that f c;
```

- *eSatisfaction* occurs like for `during all d such that f c`.
- *eViolation* occurs at `eoc` if at least one `d` period where `f` has been continuously `true` has elapsed and *eSatisfaction* has never occurred.

```
eSatisfaction is first (during all b such that f c).eSatisfaction;
eViolation is when eoc while
      count ((during f achieve inPTime >= d).eSatisfaction) > 0
  and count (eSatisfaction) = 0;
```

18 Instructions

In FORM-L, behaviour is specified with *instructions*. There are three main kinds of instructions: *elementary instructions*, *exclusion chains* and *coordinated instructions*.

18.1 Elementary Instructions

An elementary instruction is composed of three parts: an *action* (WHAT), an optional *selector* (WHICH) and an optional *temporal locator* (WHEN).

18.1.1 Actions

There are different kinds of actions:

- An *assignment* specifies a definite value for an attribute and begins with keyword ***define***.
- A *signal* specifies the occurrence (or non occurrence) of an event and begins with keyword ***signal***.
- A *constraint* specifies a Boolean condition not to be violated (see Section 12.3 *Constraints*).
- A *deletion* begins with keyword ***delete*** and eliminates a dynamically created object. (Object creation is performed in the framework of expression evaluations, see the example of Section 0 *One can also substitute* one or more classes of the object with one of their extensions, or specify additional classes for the object.

This example is inspired from the *Backup Power Supply* study. **resetCmd** is first defined as a ***Boolean***.

```
Boolean @resetCmd "Reset command" is deferred;
```

Then, in an extension model, **resetCmd** is redeclared as a ***PushButton***, an extension class of ***Boolean***.

```
Class @PushButton extends Boolean;
redeclared PushButton resetCmd;
```

```
Class Pump;
Pump  pump;
...
Class Motor;
redeclared (Pump, Motor) pump;
```

In this example, object **pump**, initially of class ***Pump***, is redeclared to be both a ***Pump*** and a ***Motor*** (i.e., a pump actuated by a motor).

- Dynamic Creation and Deletion.)

18.1.2 Selectors

A selector specifies which elements of one or more sets are concerned (see Section 11.5). If absent, the items concerned must be named explicitly.

18.1.3 Temporal Locators

A temporal locator specifies at which instants or during which time periods the action applies (see Sections 15 *Discrete Temporal Locators (DTL)*, 16 *Continuous Temporal Locators (CTL)* and 17 *Sliding Temporal Locators* (*STL*)). If absent, the action applies at all times. *STL* can be used only with constraints.

18.2 Exclusion Chains

18.2.1 Temporal Exclusion

To facilitate the avoidance of contradictions (e.g., different values being assigned at the same time to the same attribute), instructions may be organised into *temporal exclusion chains*, each rung of a chain being separated from the following one by keyword ***otherwise***. The effective temporal locator for an individual instruction in the chain is the one it specifies, but excluding all the temporal locators of the instructions preceding it in the chain.

```
Power ^powerCall "Call for BPS power" begin
```

```
  ...
  Power @upperBound begin
    during not (poweredByBps and required)
      define value is 0*W
    otherwise from poweredByBps and required during settleTime
      define value is initial
    otherwise
      define value is stabilised;
  end upperBound;
end powerCall;
```

This example is taken from the *BPS* study. It specifies an upper bound for the call for electric power of a *BPS* client (see *Figure 4-8*). The exclusion chain ensures that the definition of **upperBound** is adequate even when the duration of **poweredByBps** *and* **required** is shorter than **settleTime**.

The definition of an attribute implicitly excludes all the instants where it is already fully determined by applicable classes (i.e., classes of the object to which the attribute is attached, or classes where that object is a feature). Similarly, in an extension model, the refined definition of an object implicitly excludes all the instants where it is already fully determined by some of the models being extended.

18.2.2 Set Exclusion

Much like with temporal exclusion, instructions may also be organised into *set exclusion chains*, each rung being separated from the following one by keywords ***for all other*** (or possibly ***for some other*** when the action is a constraint). The effective selector of an individual instruction of the chain is the one it specifies, but excluding the outcomes of all the selectors of the preceding rungs in the chain.

```
Objective goodPractice is
  for all x of potentialClients such that x.isClient
    during openingHours
    ensure goodService
  for all other
    every year within 1*year
    achieve x.eContact;
```

This simplistic example shows a set exclusion chain where **potentialClients** that are already clients (**x.isClient**) get **goodService** and the others are contacted once a year.

18.3 Temporal Coordinations

Sometimes, multiple instructions need to be precisely coordinated in time. This could be done by specifying adequate temporal locators for each. However, to highlight the nature of the coordination, to simplify the expression of the coordinated instructions, and thus to facilitate understanding and avoid modelling errors, FORM-L offers four temporal coordination mechanisms:

- *Sequence* places two or more instructions in time, one after the other.
- *Concurrence* places the beginning of two or more instructions at the same instant.
- *Iteration* repeats an instruction in time.
- *Selection* selects one instruction among several, based on a Boolean or probabilistic criterion.

The temporal locator of a coordination (not to be confounded with the temporal locators of the coordinated instructions) controls it as follows:

- If it is a *DTL*, the coordination is initiated at each instant of the *DTL* and performed to completion or is truncated by **eoc**.
- If it is a *CTL*, the coordination is initiated at the beginning of each period and truncated at the end of the period if it is not completed.
- If no temporal locator is specified, *CTL* ***always*** is assumed.

The action of a coordinated instruction is initiated at instants determined by the coordination and its own temporal locator:

- If it does not specify any temporal locator, then the action is instantaneous (i.e., it has no duration) and is performed once at the first instant allowed by the coordination.

- if specifies a *DTL*, then the action is instantaneous and is performed once at the first *DTL* occurrence at the same time or after the first instant allowed by the coordination.
- If it specifies a *CTL*, then the action is performed for a single period, the first one that begins at the same time or after the first instant allowed by the coordination. For *CTL* that normally begin at `t0`, the period begins at the first instant allowed by the coordination. If the action is a constraint, then it terminates when a satisfaction or violation decision is made.

18.3.1 Sequence

A sequence is introduced by keyword ***sequence*** and places a number of instructions in time, one after the other, in the order specified..

18.3.2 Concurrence

A concurrence is introduced by keyword ***concurrence*** and places the beginning of a number of instructions at the same time.

```
when motor.eNeeded sequence
  Requirement r1 is within 0.5*s achieve motor.eStart;
  Requirement r2 is after motor.eReady within 0.5*s achieve mpsBrk.eOpen;
  concurrence
    Assumption  a3 is within 1*s achieve mpsBrk.open;
    Requirement r4 is after 5*s within 0.5*s achieve eProceed;
  end concurrence;
end sequence;
```

This instruction is taken from the *BPS* example. It specifies that (see *Figure 18-1*):

- When the **motor** becomes needed (event **motor.eNeeded**), then within 0.5 seconds an order to start it (event **motor.eStart**) must be issued (requirement **r1**).
- Then, after the **motor** becomes ready (event **motor.eReady**), within 0.5 seconds an order to open the *MPS* circuit breaker (event **mpsBrk.eOpen**) must be issued (requirement **r2**).
- 1 second after event **mpsBrk.eOpen**, the state of **mpsBrk** should be **open** (requirement **r3**).
- After a delay of 5 seconds after event **mpsBrk.eOpen**, an order to proceed (event **eProceed**) must be issued within 0.5 seconds (requirement **r4**).

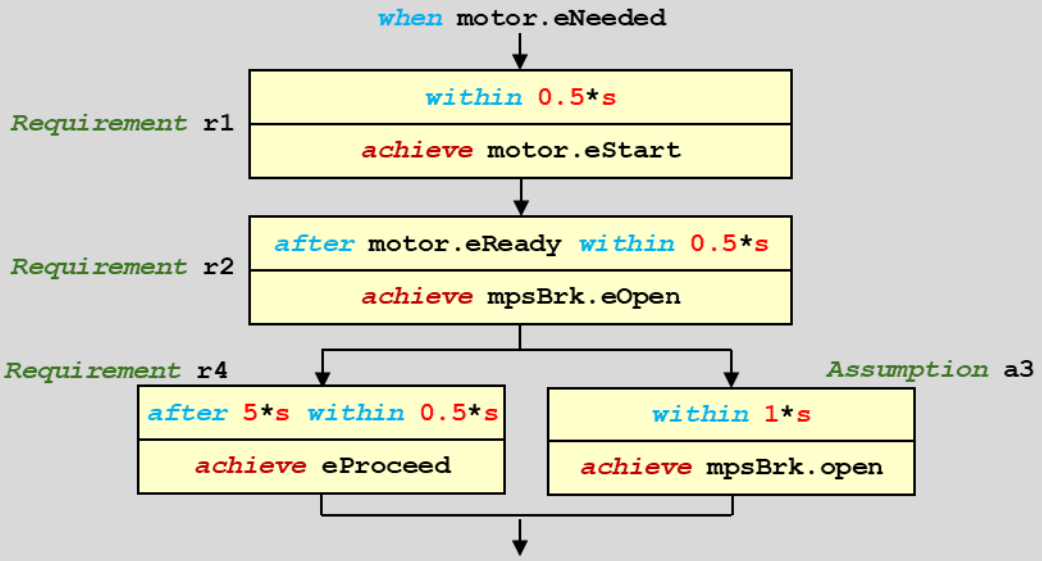


Figure 18-1
Example of sequential and concurrent instructions

18.3.3 Iteration

An iteration repeats a single instruction. There are two kinds of iterations:

- With keyword ***repeat***, the instruction is repeated until abortion.

- With keywords ***iterate n times***, the instruction is repeated the number of times specified, as evaluated when the iteration is initiated.

18.3.4 Selection

A *selection* is introduced by keyword ***selection***, and at each instant of its temporal locator (which must be a *DTL*), it chooses and applies one instruction among several. The action of that instruction must be an assignment, an event signal or an object deletion. There are two kinds of selections: *deterministic selections* and *probabilistic selections*.

In a *deterministic selection*, the instructions are in an ordered list, each being guarded by a Boolean condition (introduced by keyword ***case***): the instruction selected is the first in the list the Boolean condition of which is **true**. If none of the Boolean conditions are **true**, then no instruction is applied. The Boolean condition of the last instruction of the list may be **true**, in which case it selected when none of the others are eligible.

In a *probabilistic selection*, instruction is guarded by a probability expression (introduced by keywords ***with probability***). If the sum of the probability guards is less than **1**, then there is an implicit do-nothing instruction the probabilistic guard of which is the complement to **1**. The probabilistic guard of the last instruction may be **1**, in which case it complements the sum of the others to **1**. If the sum of the probability guards is greater than **1**, then the last instructions are never taken into consideration.

```
Event eFailure: rate is 1e-2/year;
Event eRepair is deferred;

Automaton [ok, [stuckOpen, stuckClose, stuckAsIs, slow] failure] state
begin
  when t0 define value is ok
  otherwise when eFailure while value = ok selection
    with probability 1/4: next is failure.stuckOpen;
    with probability 1/4: next is failure.stuckClose;
    with probability 1/4: next is failure.stuckAsIs;
    with probability 1:   next is failure.slow;
  end selection
  otherwise when eRepair define next is ok
  otherwise next is value;
end state;
```

This example is taken from the *BPS* case study: it models equiprobable component failure modes.

18.4 Grammar

```
Instruction: instruction ';' ;

instruction:
( actionApplicator *     (dctl actionApplicator*)?       action
| constraintApplicator* (tl   constraintApplicator *)? qConstraint
| timeExclusionChain
| setActionExclusionChain
| setConstraintExclusionChain
) ;
```

actionApplicator allows only universal selectors, whereas ***constraintApplicator*** also allows existential selectors. Also, *STL* can be used only with constraints.

```
action:
   'define' pathname 'is' expr                        // assignment
| 'signal' 'no'? (pathname | set)                     // event signalling
| 'delete' expr                                       // object deletion
| 'sequence' Instruction+ 'end' 'sequence'?           // sequence
| 'concurrence' Instruction+ 'end' 'concurrence'?     // concurrence
| ('repeat' | 'iterate' int 'times') instruction     // iteration
| 'selection' choices 'end' 'selection'?             // selection
;

choices:
   ('case'                bool ':' Instruction)+      // deterministic
| ('with' 'probability' real ':' Instruction)+       // probabilistic
```

```
;

qConstraint:                                          // qualified constraint
   ('private'? propertyClass (determiner? NAME)? STRING* 'is')?
   constraint
;
```

The qualification part declares and defines a property. If absent, the constraint is an assumption. If present, **NAME** may be omitted only if ***propertyClass*** is an ***Assumption*** or an ***Assumption*** extension.

```
dctl: dtl | ctl;

tl:   dtl | ctl | stl;

timeExclusionChain:
               dctl setAction
 ('otherwise' dctl setAction)*
 ('otherwise'      setAction)?
;
```

Each rung that is not the last one in the chain must explicitly specify a temporal locator.

```
setAction:
   actionApplicator*     action
|  constraintApplicator* qConstraint
;

setActionExclusionChain:
   actionApplicator*                        timeAction
   ('for' 'all' 'other' 'such' 'that' bool timeAction)*
   ('for' 'all' 'other'                     timeAction)?
;
```

Each rung that is not the last one in the chain must explicitly specify filtering condition.

```
timeAction:          action
|               dctl action
   ('otherwise' dctl action)*
   ('otherwise'      action)?
;

setConstraintExclusionChain:
   constraintApplicator* timeConstraint
   ('for' ('all' | 'some') 'other' ('such' 'that' bool)? timeConstraint)*
;
```

The **constraintApplicator*** determine sets that are concerned. Each rung in the chain excludes the set members selected by preceding rungs. Only the last rung may specify an unconditional **'for' 'all' 'other'**.

```
timeConstraint:      qConstraint
|               dctl qConstraint
   ('otherwise' dctl qConstraint)*
   ('otherwise'      qConstraint)?
;
```

19 Interfaces

19.1 Contracts

A *contract* specifies an engineered interface between *parties*, which may be objects or classes. Besides identifying the concerned parties, a contract specifies the *obligations* of each party, i.e.,:

- The *deliverables* (variables, events, sets and non-valued objects) that party provides to the other ones.
- The *guarantees* and *requirements* it ensures regarding its deliverables. A guarantee or requirement from a party is an *assumption* for the other parties.
- The *assumptions* it makes and that must be collectively guaranteed by the other parties. This is mostly useful in contracts with three or more parties.

To make sure that a contract is not established without the consent of its parties, each must acknowledge it by declaring it as a feature.

```
Object bps begin
  Contract backupPower (me, Client);
  ...
end bps;
```

Object **bps** acknowledges contract **backupPower**. Note the use of keyword **me**.

19.1.1 Broadcast Contracts

A contract that has only one party is a *broadcast contract*: that party offers the interface to all objects and classes of the mock-up that acknowledge the contract.

```
Contract mainPower (mps) "Normal electric power service";
```

This example is also taken from the *Backup Power Supply* study. The **mainPower** contract is simply declared and is between object **mps** and all the other objects of the model that acknowledge it.

19.1.2 Contracts Between Static Objects

```
Contract hsi (bps, operator) "Human-System Interface" begin
  party operator begin
    Boolean @resetCmd;
  end operator;

  party bps begin
    Event @eAlarm;                  // May not be able to fulfil its mission
    Requirement eAlarm.sendIt;      // When necessary
    Requirement eAlarm.noSpurious; // When not necessary
  end bps;
end hsi;
```

This example is taken from the *Backup Power Supply* study. Object **bps** represents the *Backup Power Supply* system, and object **operator** represent the human operator. As per this contract, **operator** provides **bps** with a Boolean named **resetCmd**, which is a reset order. Conversely, **bps** provides **operator** with an event named **eAlarm** signalling that it might be unable to perform its mission. **bps** also mentions two requirements regarding **eAlarm**: **sendIt** when necessary, and **noSpurious** when not. The **@** determiners indicate that the deliverables are control information, not physical quantities.

```
Contract backupPower (bps, Client) "Backup electric power service" begin
  party Client begin
    Boolean @required;                  // Whether required to operate
    Percentage ^tolerance;              // Tolerance to loss of power
  end Client;

  party bps: AcVoltage Client.#voltage; // At each client's terminals
end backupPower;
```

This example is also taken from the *Backup Power Supply* study. The contract is between the **bps** and class **Client**. Here, the obligations of each party are just declared. They may be more precisely defined within the definition of their party or in a refinement of the contract, at model authors' choice.

19.1.3 Contracts Involving Classes

A party to a contract may be a class, in which case the contract applies to each instance of the class. When a contract involves one or more classes, different cases need to be considered regarding deliveries, as shown in *Figure 19-1* and *Figure 19-2*.

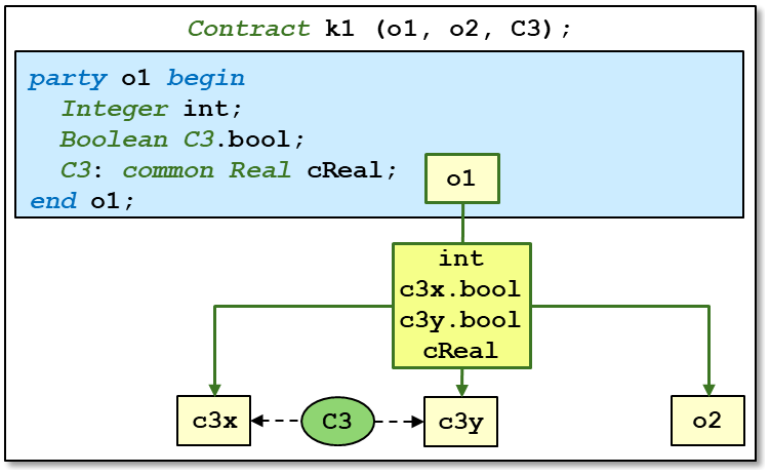


Figure 19-1
Deliverables of an object party. Here, object **o1** *delivers an* **int**, *a* **bool** *specific to each instance of class* **C3**, *but a single* **cReal**, *as it is* **common** *to all instances of* **C3**.

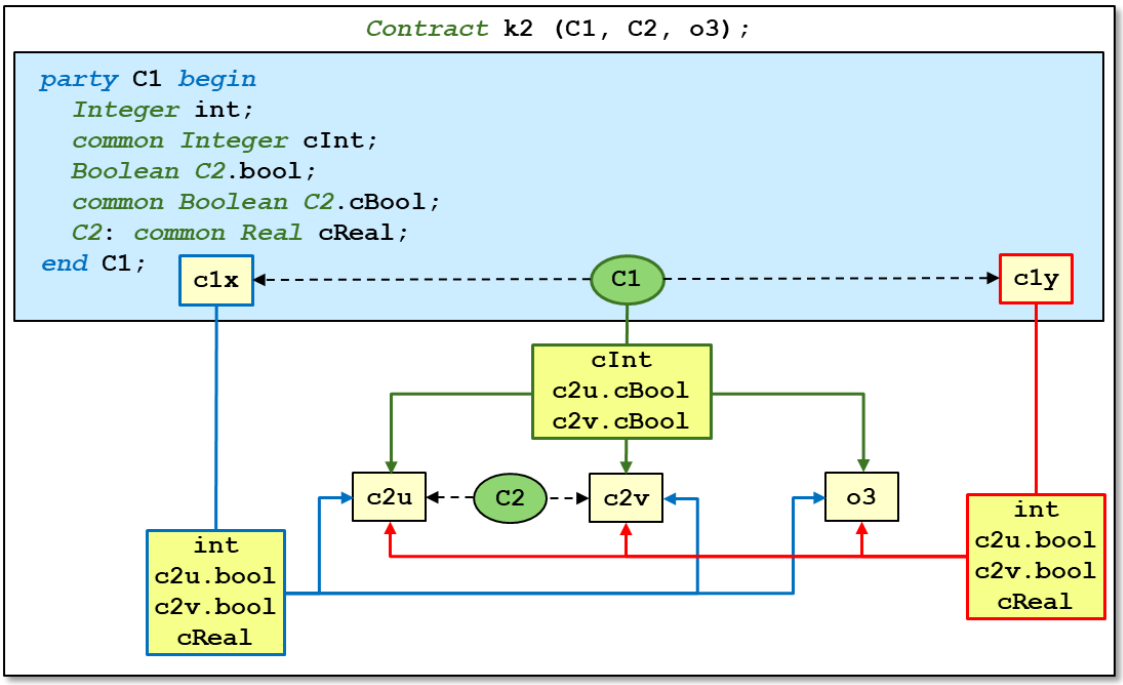


Figure 19-2
Deliverables of a class party. Here, each instance of class **C1** *delivers its own* **int** *and* **cReal** *(which is* **common** *to all instances of class* **C2**), *and its own* **bool** *specific to each instance of* **C2**. *As both* **cInt** *and* **cBool** *are* **common** *to all instances of* **C1**, *it is* **C1** *as a whole that delivers* **cInt** *and then a* **cBool** *specific to each instance of* **C2**.

```
Contract backupPowerDiv (bps.Division, Client.Channel) begin
  party bps.Division begin
    Client.Channel begin
      AcVoltage #voltage;          // From division to each channel
      Guarantee voltage.#range;    // 0-260 volts
    end Client.Channel;
    Guarantee #repower;            // tolerance > 5%
    Guarantee #reset;              // Restoration of powering by MPS
  end division;

  party Client.Channel begin
    Boolean @required;             // Whether required to operate
    Boolean #powered;              // Whether powered, by MPS or BPS
    Percentage ^tolerance;         // Tolerance to loss of power
    Guarantee tolerate
     "When required to operate, a channel shall tolerate not being powered
      for a certain amount of time";
    Guarantee replenish
     "When powered, a channel shall replenish its tolerance capability";
  end Client.Channel;
end backupPowerDiv;
```

This example is also taken from the *Backup Power Supply* study. Contract **backupPowerDiv** is between class **bps.Division** and class **Client.Channel**, i.e., between each division and each channel. Each channel provides the same deliverables and offers the same guarantees to each division. The reverse is not true: though a division ensures **repower** and **reset** globally to the set of channels, it provides **voltage** individually to each channel.

In some cases, the same class plays multiple roles in an interface (contract or encroachment) and therefore appears several times in the list of parties. Each appearance (and role) can be followed by a substitute identifier that will distinguish it from the others (see following examples).

19.2 Encroachments

An *encroachment* has two parties and formally models the undesired, non-engineered side effects one party (the *origin*, the first party of the encroachment) may have on the other one (the *target*, the second party of the encroachment), possibly due to failures and other exceptional conditions. Parties may be individual objects or classes. The origin may force the behaviour of some of the target's variables, events and sets, with precedence (with time and set exclusion) with respect to their definitions by the target. Normally, the target has no active role.

Like for contracts, encroachments must be acknowledged by their origin and target. The order of acknowledgments defines the order of precedence: an encroachment that is acknowledged (or re-acknowledged) after another one has lower precedence.

```
abstract Class #Component begin
  Automaton [ok, failure] #state;
end Component;

Encroachment propagation (Component Origin, Component Target)
begin
  for all origin of Origin, target of Target
  such that origin.identity <> target.identity
  after origin.state becomes failure
    while target.state <> failure
  during 1*mn
  ensure probability (target.state becomes failure) = 0.25;
end propagation;
```

propagation models failure propagation between components: when any one fails (**state** *becomes* **failure**), another one has a 0.25 probability of failing within 1 minute (*during* 1**mn**), if it was not already in a failed state.

```
abstract Class #RedudantPart;
Encroachment independence (RedundantPart Origin, RedundantPart Target)
begin
  for all origin of Origin, target of Target
  such that origin.identity <> target.identity
  ensure no effect;
end independence;
```

Keyword `no effect` may be used to specify absence of side effect.

19.3 Refinement

A contract, standard contract, encroachment or standard encroachment may be refined. A refined contract or standard contract keeps all the original definition, but some or all parties may specify additional deliverables and obligations. In a refined encroachment or standard encroachment, the origin may force additional effects on the target.

```
refined powerSupply;
```

This simply declares a refinement of **powerSupply**. The definition of the refinement will be provided elsewhere in the model.

```
refined powerSupply begin
  party mps begin
    Event @eWarning;
  end mps;
end powerSupply;
```

This re-declares and defines the refinement of **powerSupply**. It specifies that **mps** has an additional deliverable: event **eWarning**.

19.4 Grammar

```
Interface: Contract | Encroachment ;

InterfaceDefinition: ContractDefinition | EncroachmentDefinition ;

InterfaceRefinement: ContractRefinement | EncroachmentRefinement ;

Contract:
   'Contract' NAME '(' party (',' party)* ')' contractBlock? ';' ;
   The contract being declared is identified by its NAME.

party:         determiner? pathname STRING* ;
   The pathname identifies a party to a contract.

contractBlock: statementApplicator? 'begin' PartyBlock+ 'end' NAME? ;

PartyBlock:    'party' pathname /* to party */ (Block | ';') ;

ContractDefinition:  pathname STRING* contractBlock ';' ;
   The contract being defined is identified by a pathname.

ContractRefinement:  'refined' pathname STRING* contractBlock? ';' ;
   The contract being refined is identified by a pathname.

Encroachment:
   'Encroachment' NAME '(' party ',' party) ')' (EncroachmentBlock? | ';') ;
   The encroachment being declared is identified by its NAME and has only two parties.

EncroachmentBlock: statementApplicator? Block ;

EncroachmentDefinition: pathname STRING* Block ;
   The encroachment being defined is identified by a pathname.

EncroachmentRefinement: 'refined' pathname STRING* (Block | ';') ;
   The encroachment being refined is identified by a pathname.
```

20 Models

20.1 Models, Partial Models, Libraries & Statements

A FORM-L *model* (or *model*, for short when there is no ambiguity) is a named collection of statements that generally represents a certain viewpoint on a system of interest (represented by the ***main*** behavioural item or class) and its environment (represented by ***external*** objects and classes). A *library* is a FORM-L model the statements of which are reused in other FORM-L models.

A *partial model* is a named subset of a FORM-L model that is used to organise and provide structure to a large and complex model.

20.2 Extension

A model may *extend* one or more other models:

- When model *B* extends model *A*, *B* includes all what is in *A*; it may then refine items declared by *A*, and add new statements of its own.
- In particular, *B* may provide explicit definitions for items that *A* has *deferred*. *B* may also take responsibility for some *guarantees* offered by some objects or classes of *A* by refining them as *requirements*, or on the contrary refine them as *assumptions*.
- *B* may also refine a requirement by downgrading it as an objective, as shown in model **BpsPsaReference**, in which case the refinement must specify a *substitution* requirement, preferably with a *justification*.
- An item ***external*** to *A* is also ***external*** to *B*, unless B declares it as its ***main*** item.
- A model that does not extend any other model MUST specify its ***main*** behavioural item or class.
- When a model does not specify itself a ***main*** item and extends one or more models, its ***main*** item is the one of the first extended model in the list.
- When *B* extends multiple models, there may be name conflicts (models being extended using the same name for different entities, e.g., because they were developed by different teams), unambiguous pathnames must be used.

Extension models may be used to represent successive engineering stages (see Section 3 *BASAALT, a Systems Engineering* Approach):

- A first model, **BpsIntroduction**, expresses in natural language the top-level *BPS* requirements and identifies the elements constituting its environment.
- **BpsReference**, an extension of **BpsIntroduction**, formalises these top-level requirements and the assumptions made regarding the environment.
- **BpsRedundancy**, an extension of **BpsReference**, specifies a first design step (a redundant architecture) to be assessed in the framework of the requirements and assumptions of **BpsReference**.
- **BpsDivision**, an extension of **BpsRedundancy**, specifies a second design step (the architecture of each *BPS* division) to be assessed in the framework of the requirements and assumptions of **BpsRedundancy** and **BpsReference**.

Another possible use of extension models is the introduction of generic scenarios:

- **BpsReference** models the system and its environment "as they are" and “as they are known” at the initial stage of the systems engineering process.
- **BpsReferenceScenario**, an extension of **BpsReference**, introduces a generic scenario in the form of additional assumptions and definitions specifying an envelope of *legitimate cases* of interest.
- **BpsReferenceScenario2**, another extension of **BpsReference**, could be used to introduce a different generic scenario specifying a different envelope of legitimate cases of interest.

Extension models may also be used to represent the viewpoints of particular stakeholders (such as sponsors, clients, operators, regulators and owners) or engineering disciplines (such as project management, physics, instrumentation and control, safety, security, dependability, probabilistic analysis, human factors engineering, economics, operation and maintenance). In

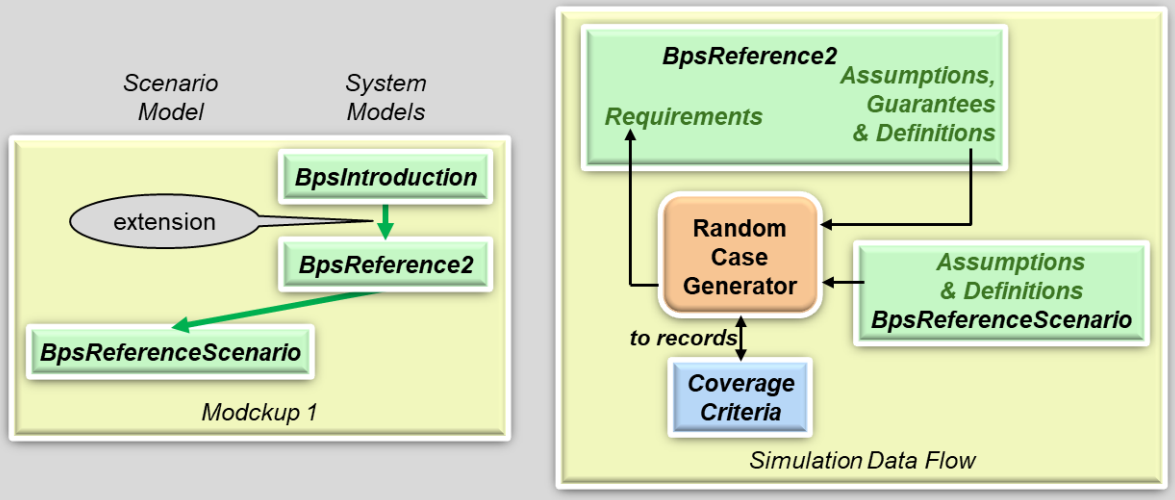


Figure 4-3, *Figure 4-5* and *Figure 4-9*:

- **`BpsIntroduction`**, **`BpsReference`**, **`BpsRedundancy`** and **`BpsDivision`** are developed by the main engineering team but do not address dependability issues.
- **`BpsDivision`** extends **`BpsRedundancy`** which itself extends **`BpsReference`** which itself extends **`BpsIntroduction`**: this reflects the step-by-step progress of the main engineering team.
- The dependability team adds the information it needs in **`BpsPsaReference`** (which extends **`BpsReference`**), then in **`BpsPsaRedundancy`** (which extends both **`BpsPsaReference`** and **`BpsRedundancy`**), and lastly in **`BpsPsaDivision`** (which extends both **`BpsPsaRedundancy`** and **`BpsDivision`**).

20.3 Non-FORM-L Models & Bindings

A *non-FORM-L model* is a model expressed in another language than FORM-L. A *binding* is a non-FORM-L model that specifies how two or more models (FORM-L or non-FORM-L models) can exchange information along operational cases. They are not addressed in this document.

20.4 Grammar

```
Model:
   'Model' NAME1 ('extends' NAMEn (',' NAMEn)*)? STRING* (Block | ';')
   Segment* ;
```

`NAME`$_1$ is the name of the model being declared, whereas **`NAME`**$_n$ are the names of models it refines.

```
Segment: Declaration | Definition | Refinement | Instruction ;
```

The ***`Declaration`***, ***`Definition`***, ***`Refinement`*** or ***`Instruction`*** belongs to the ***`Model`*** declared at the beginning.

```
PartialModel: 'partial' 'Model' NAME STRING* (Block | ';') ;
```

`NAME` is the name of the partial model.

```
ModelDefinition: NAME STRING* block ';' ;
```

`NAME` is the name of a model or partial model that has already been declared.

```
Library:       'Library' NAME /* of a model */ STRING* ';' ;
```

`NAME` is the name of a model that has already been declared and defined.

```
Declaration:
   Library
|  PartialModel
|  Object
|  Class
|  Ctl
|  Stl
|  Interface
;
Definition:
   ObjectDefinition
|  ClassDefinition
|  InterfaceDefinition
|  ModelDefinition
;
Refinement:
   ObjectRefinement
|  StateRefinement
|  PropertyRefinement
|  ClassRefinement
|  InterfaceRefinement
;
Statement:
  Declaration
| ObjectRedeclaration
| ClassRedeclaration
| Definition
| Refinement
| Instruction
| STRING ;
```

STRING may be provided for comment.

21 Conclusion

BASAALT and FORM-L have been developed progressively, based on a number of case studies. Further steps will be following:

- Other case studies could be developed to confirm that the notions presented here are appropriate and sufficient to specify behaviours and properties in a legible, understandable manner.
- Formal specification of FORM-L grammar (in *Xtext*) and semantics (based on mathematical notations) is ongoing.
- Development of support tools is ongoing, in particular translators towards existing modelling languages that already have support tools and environments.
- Development of a graphical version of the language that could also be used to present different views of a mock-up (e.g., like in KAOS [4], goal views, responsibility views, object views, operation views, etc.) and of simulation or analysis results.
- Development of variants other than the English variant presented here (e.g., a French variant), so that users can express their models in a variant close to a natural language with which they are comfortable. As natural languages often have very different grammars, this could be a significant endeavour.
- Spatial locators in 3D, 2D and 1D spaces could be introduced.
- Einsteinian, relativistic space-time could also be introduced, but that could add significant complexity to the language.

Acronyms

AST	Abstract Syntax Tree
BASAALT	Behaviour Analysis and Simulation All Along systems Life Time
BNF	Backus-Naur Form
BPS	Backup Power Supply
COTS	Commercial Off-The Shelf
CPS	Cyber-Physical System
CTL	Continuous Temporal locator
DTL	Discrete Temporal locator
DSL	Domain-Specific Language
EuroSysLib	EUROpean leadership in SYStem modelling and simulation through advanced modelica LIBraries
FMI	Functional Mock-Up Interface
FMU	Functional Mock-Up Unit
FORM-L	FOrmal Requirements Modelling Language
GALS	Globally Asynchronous, Locally Synchronous
HARMONICS	Harmonised Assessment of Reliability of MOdern Nuclear Instrumentation and Control Software
MBSE	Model-Based Systems Engineering
MESKAAL	Maintenance of Engineering and Safety Knowledge on a system All Along its Life time
MODRIO	Model Driven Physical Systems Operation
MPS	Main Power Supply
SoS	System Of Systems
STL	Sliding Temporal locator

Glossary

A Posteriori Value	Value that cannot be evaluated in the framework of individual *Operational Cases*, but based on the outcomes of large numbers of *Legitimate Operational Cases*.
Abstract Class	*Class* that cannot be instantiated directly but only through *Extension Classes*.
Action	Expression specifying the WHAT part of an *Elementary Instruction*: an assignment, an event occurrence signal, a constraint or the deletion of an object.
Assumption	*Property* that is supposed never to be violated.
Attribute	Built-in characteristic of a *Behavioural Item*. See also *Feature*.
Automaton	*Variable* that is an instance of an *Enumeration*.
Behavioural Item	An *Object* or a *Class.*
Binding	Specification of how information is transferred between *Models*, without having to modify any of them.
Case	See *Operational Case*.
Class	Template specifying what is common to multiple *Objects*. These *Objects* are instances of the *Class*.
Clock	*Attribute* of a *Behavioural Item* that specifies the discrete instants where the item can perceive and act on its environment. Defined by a *Discrete Temporal locator*. See also *Discrete Time Domain*
Composite Instruction	*Instruction* composed of multiple, coordinated *Elementary Instructions*.
Conceptual Item	*Behavioural Item* representing an abstract notion that cannot be measured, perceived and computed directly.
Conditional Probability	Value in the [0., 1.] range that states the probability of an *Event* occurring at least once during a specified *Temporal locator*.
Constant Object	*Object* the *Value* and *Features* of which do not change with time and that are the same for all cases. See also *Fixed Object*.
Constraint	*Action* specifying a condition that must be satisfied at specific times (see *Temporal Locator*) by specific *Objects* (see *Selector*).
Continuous Time Domain	*Time Domain* where time is perceived continuously. There is only one such domain.
Continuous Temporal Locator (CTL)	*Temporal locator* that specifies a finite number of possibly overlapping *Time Periods*. See also *Discrete Temporal locator* and *Sliding Temporal locator*.
Contract	FORM-L item that formally specifies the engineered, desired interface and interactions between two or more *Parties*.
Cyber-Physical System	System integrating physics, computing and networking
Cyber System	Systems based exclusively on digital electronics, computing and networking
Declaration	Expression of the existence, name, nature and possibly behaviour and / or contents of a FORM-L item.
Defined Class	*Class* that is explicitly defined by a model (as opposed to *Predefined Class*)
Definition	Possibly partial specification of the behaviour and / or contents of a FORM-L item.
Determiner	Indication attached to an *Object* or a *Class*.

Discrete Time Domain	*Time Domain* where time is perceived only at the instants specified by a *Clock*. There may be several such domains.
Discrete Temporal Locator (DTL)	*Temporal locator* that specifies a finite number of *Instants*. See also *Continuous Temporal locator* and *Sliding Temporal locator*.
Elementary Instruction	Formula composed of an optional *Selector* (WHICH), an optional *Temporal locator* (WHEN) and an *Action* (WHAT).
Embedded Automaton	*Automaton* that is considered as a single state in a higher-level automaton.
Encroachment	FORM-L item that formally specifies the undesired effects one *Party* may have on another *Party*.
Enumeration	Defined *Class* specifying a finite number of named discrete values.
Event	*Object* characterising the occurrences in time of a fact that has no duration.
External Model	Model in a language different from FORM-L (e.g., in Modelica) with which one or more *Models* may exchange information (through *Bindings*).
Extension Class	*Defined Class* that adds precision and *Features* to some existing *Classes*.
Feature	User-defined characteristic of a *Behavioural Item*, represented by an embedded *Object*. See also *Attribute*.
Fixed Object	*Object* the *Value* and *Features* of which are determined anew for each operational case but do not change during the case. See also *Constant Object*.
Guarantee	*Property* that is interpreted as an *Assumption* but that could be later refined as a *Requirement* or an actual *Assumption*.
Guard	*Property* that must not be violated for a *Model* to be valid.
In-Period Term	Term of an expression evaluated in the framework of each *Time Period* of a *CTL*.
Instant	Position in time that has no duration (as opposed to a *Time Period*, which has a strictly positive duration).
Instruction	Either an *Elementary Instruction* or a *Composite Instruction*.
Legitimate Operational Case	*Operational Case* that complies with all the specified *Assumptions* and *Definitions*.
Library	*Model* that provides *Definitions* that can be used by other *Models*.
Mock-Up	Self-sufficient set of *Models* that can be simulated or analysed.
Model	Set of *Declarations*, *Definitions* and *Instructions*.
Object	*Item* that represents the dynamic aspect and behaviour of some entity or aspect of the actual world..
Operational Case	Specific value and occurrence trajectories for all the *Objects* of a *Model*.
Party	*Object* or *Class* that participates in a *Contract* or in an *Encroachment*.
Period	See *Time Period*.
Predefined Class	*Boolean*, *Integer*, *Real*, *String*, *Event*, *Property*, *Assumption*, *Requirement*, *Guarantee*, *Objective*, *Guard*, *Object*
Pre-Determined Object	*Object* the behaviour of which is fully known at design time and is the same for all *Operational Cases*.
Probability	Value in the [0., 1.] range, the value of which at a given instant represents the probability that a specified *Event* occurs at least once.

Property	*Object* expressing something that is assumed, desirable, expected or required, at specific times, for specific *Objects*.
Requirement	*Property* that must be satisfied: it is the objective of analysis or simulation to verify that it is not *violated*.
Selector	Formula that specifies which members of a *Set* are concerned by a *Statement*.
Set of Objects	*Object* the *Value* of which is a finite collection of *Objects* of the same *Class*
Set of Values	*Object* the *Value* of which is a finite collection of values of the same nature
Sliding Temporal Locator	*Temporal locator* that specifies all *Time Periods* lasting a certain fixed duration during which a specified condition is satisfied. See also *Continuous Temporal locator* and *Discrete Temporal locator*.
Socio-Technical System	System integrating human actions and technological aspects
System of systems	Dynamic sets of multiple, more or less independent but interacting systems
Time Domain	FORM-L modelling of perception of time. See *Continuous Time Domain* and *Discrete Time Domain*.
Time Period	Continuous stretch of time that has a beginning, and a strictly positive duration (as opposed to an *Instant*, which has no duration).
Temporal Locator	Formula that specifies WHEN an *Action* is to be taken into account.
Variable	*Object*, the *value* of which is an implicit function of time: for a given *Operational Case*, it has one and only one Boolean, enumerated, integer, real or string *value* at any instant

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] Bouskela D., “MODRIO Project - Modeling Architecture for the Verification of Requirements - MODRIO deliverable D2.1.1”, EDF technical report H-P1C-2014-15188-EN, 2015.

[4] Respect-IT, “A KAOS Tutorial”

[5] ISO/IEC 15026-2:2011, “Systems and Software Assurance: Assurance Case”

[6] Bettini L., “Implementing Domain-Specific Languages with Xtext and Xtend”, PACKT Publishing, 2016

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

[8] Azzouzi E., Jardin A., Mhenni F., “A Survey on Systems Engineering Methodologies for Large Multi-Energy Cyber-Physical Systems” 13th Annu. Int. Syst. Conf. SysCon 2019 - Proc., April 2019.

[9] MODRIO project. https://modelica.org/external-projects/modrio

[10] Thuy N., “The BASAALT (Behaviour Analysis & Simulation All Along Life Time) systems engineering method, and the FORM-L Language (FOrmal Requirements Modelling Language)", EDF report 6125-3113-2019-03969-EN, 2021.

[11] Modelica Association, "Functional Mock-Up Interface for Model Exchange and Co-Simulation", 2021

[12] BIPM, "The International System of Units (SI)", 2019

[13] https://notepad-plus-plus.org

[14] OMG, "SysML, OMG Systems Modeling Language", 2017

[15] Dassault Systèmes, StimuLus, https://www.3ds.com/products-services/catia/products/stimulus/

[16] Ansys, Scade suite, https://www.ansys.com/fr-fr/products/embedded-software/ansys-scade-suite

[17] OECD-NEA, “Computer-based systems important to safety (COMPSIS) project: final report” NEA/CSNI/R(2012)12, July 2012

[18] Electric Power Research Institute (EPRI), "Operating experience insights on common-cause failure in digital instrumentation & control systems", TR 1016731, December 2008

[19] Electric Power Research Institute (EPRI), "Severe nuclear accidents: lessons learned for instrumentation & control and human factors", TR 3002005385, December 2015

[20] The Cranbrook Manoeuvre - https://aviation-safety.net/database/record.php?id=19780211-0

[21] Bouquerel M., Kremers E., Van Der Kamp J., Thuy N., "Requirements Modeling to Help Decision Makers to Efficiently Renovate Urban Energy Districts", Summer Simulation Conference, Berlin, July 2019

[22] Bouissou M., Thuy N., " Early Integration of Dependability Studies in the Design of Cyber-Physical Systems", Proceedings of the 29th ESREL (European Safety and Reliability) conference, Hanover, September 2019

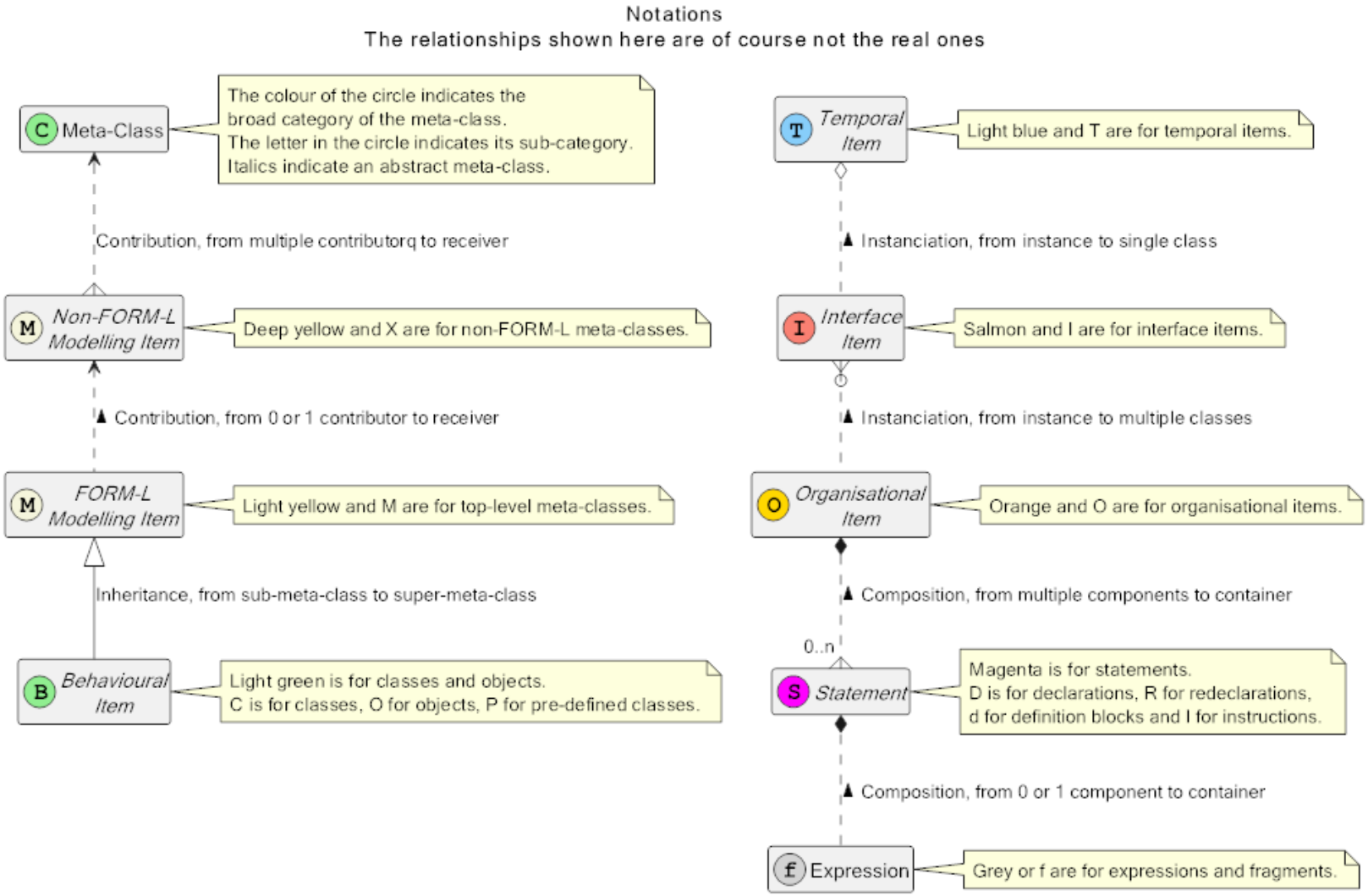
[23] Thuy N., "Formal Requirements and Constraints Modelling in FORM-L for the Engineering of Complex Socio-Technical Systems", Proceedings of RE'19 (7th IEEE International Requirements Engineering Conference), Jeju Island, (South Korea), September 2019

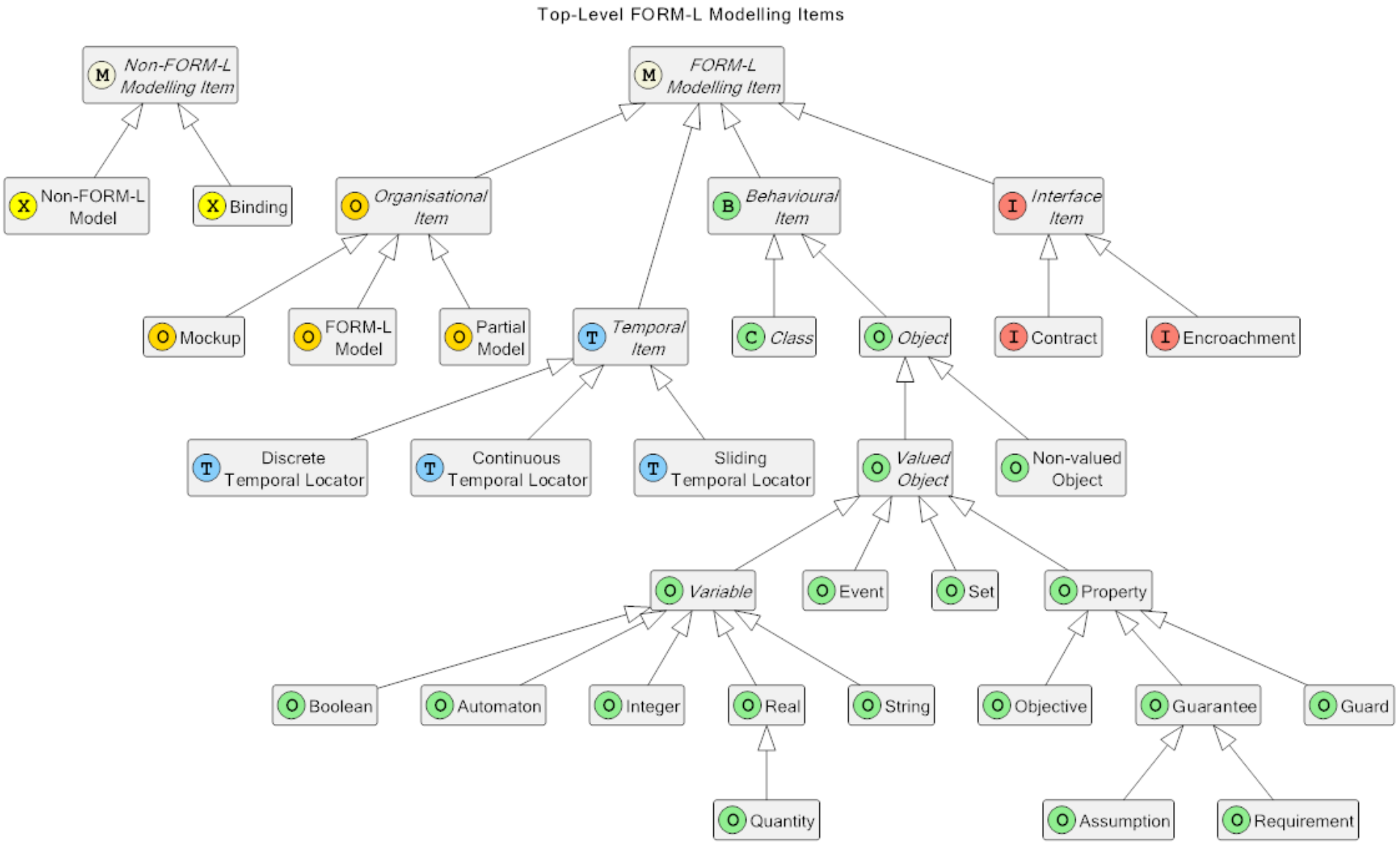
[24] Thuy N., "An Improved Approach to Traceability in the Engineering of Complex Systems ", Proceedings of the 2018 IEEE International Systems Engineering Symposium (ISSE), Rome, October 2018

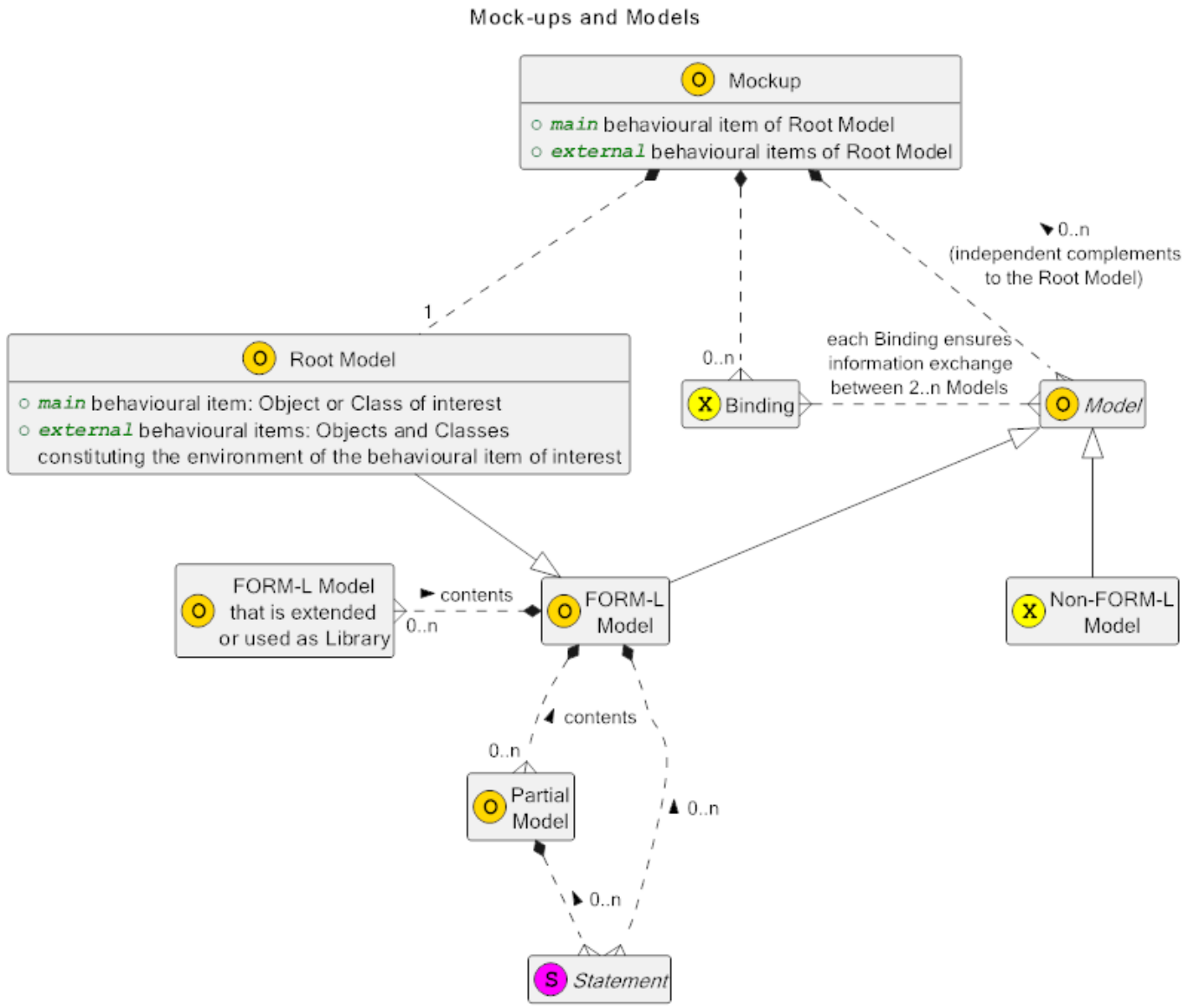
[25] Thuy Nguyen, "A Modelling & Simulation Based Engineering Approach for Socio-Cyber-Physical Systems", 14th ICNSC (IEEE International Conference on Networking, Sensing and Control), Calabria (Italy), May 2017

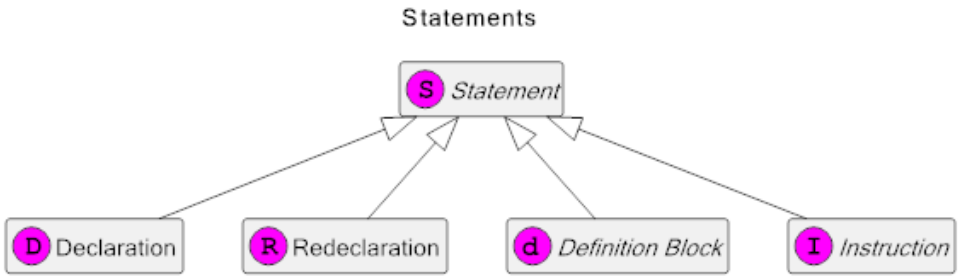
[26] Thuy Nguyen, "The MODRIO M&S-Based Engineering Approach Applied to a Backup Power Supply (BPS)", 14th ICNSC (IEEE International Conference on Networking, Sensing and Control), Calabria (Italy), May 2017

Annex 1 - FORM-L Meta-Model in PlantUML Format

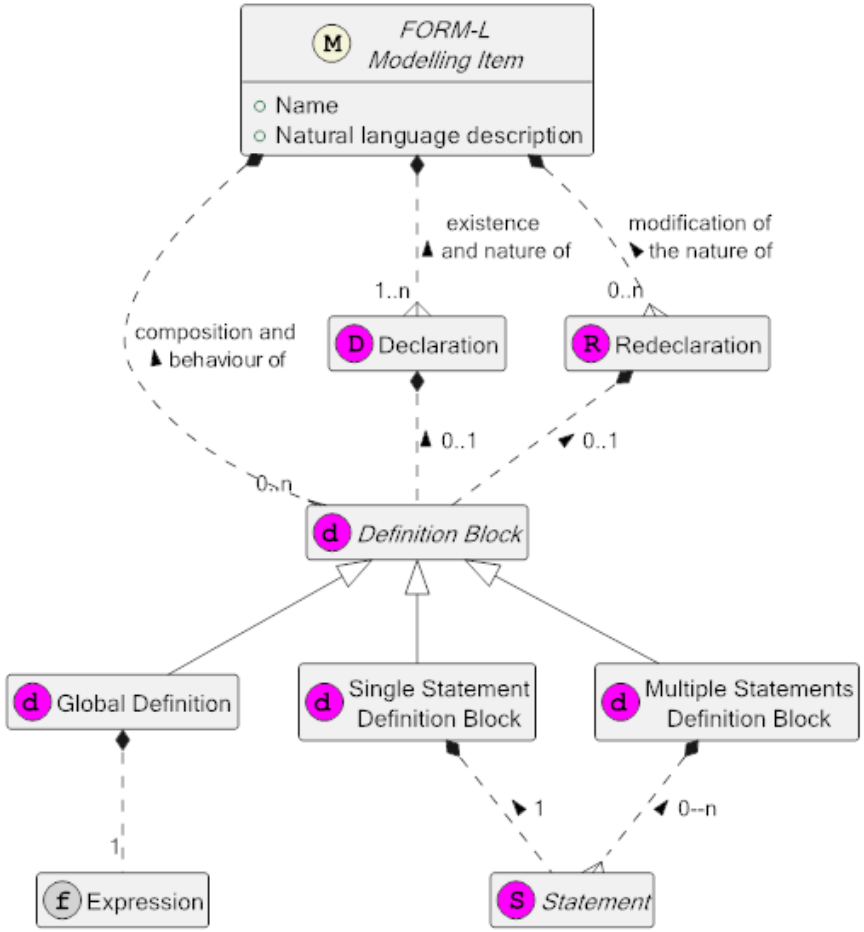


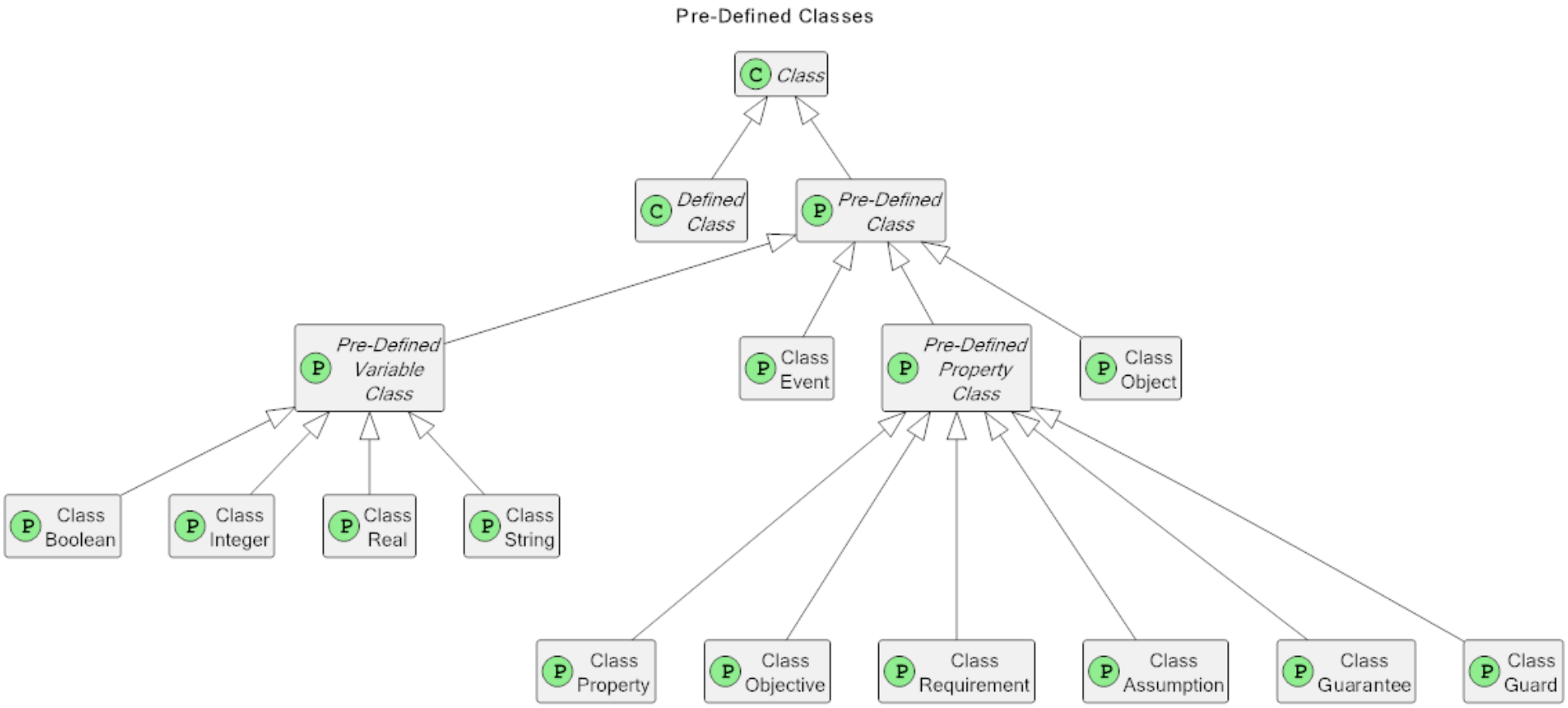

Top-Level FORM-L Modelling Items
M Non-FORM-L Modelling Item
M FORM-L Modelling Item
X Non-FORM-L Model
X Binding
O Organisational Item
B Behavioural Item
I Interface Item
O Mockup
O FORM-L Model
O Partial Model
T Temporal Item
C Class
O Object
I Contract
I Encroachment
T Discrete Temporal Locator
T Continuous Temporal Locator
T Sliding Temporal Locator
O Valued Object
O Non-valued Object
O Variable
O Event
O Set
O Property
O Boolean
O Automaton
O Integer
O Real
O String
O Objective
O Guarantee
O Guard
O Quantity
O Assumption
O Requirement

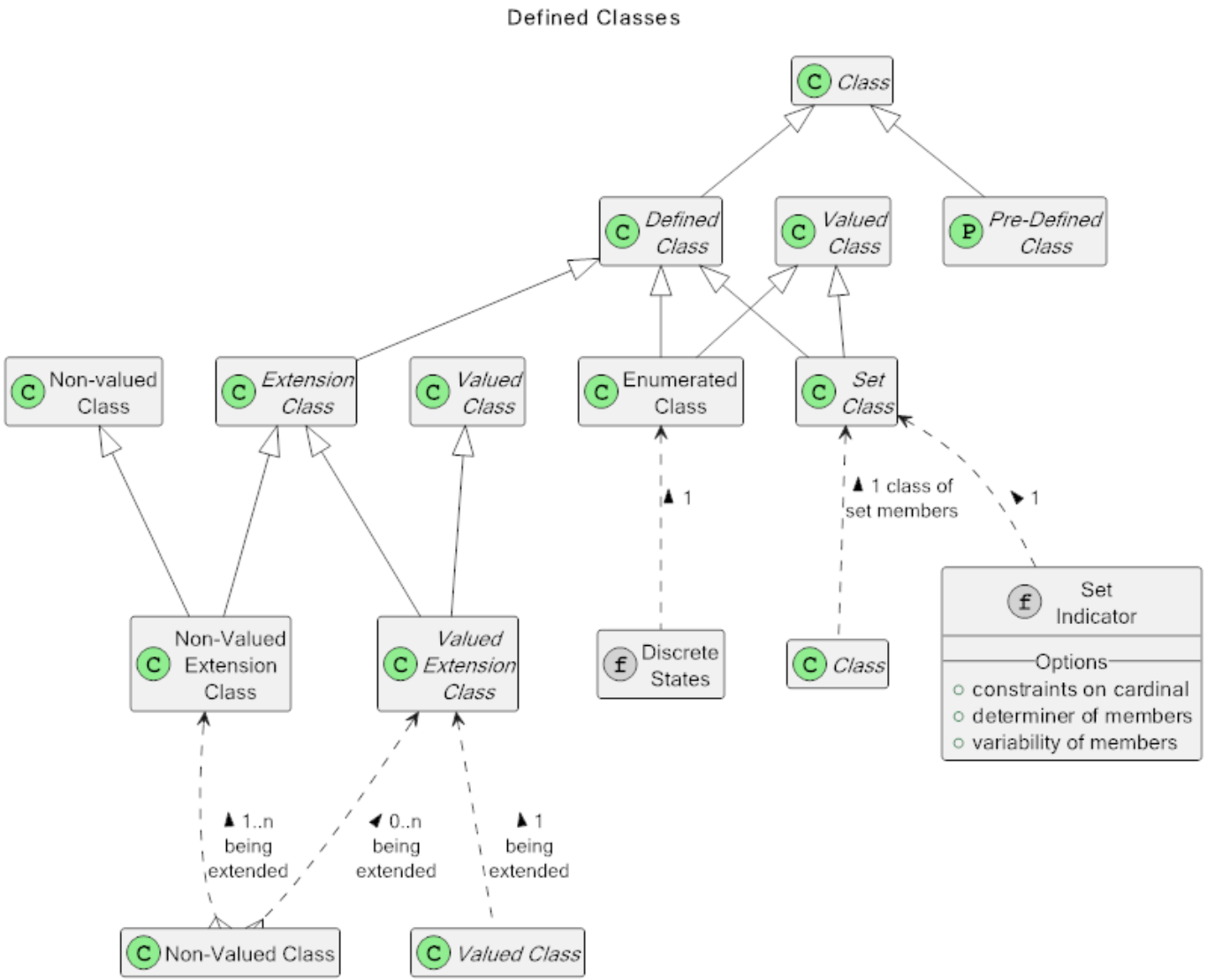

Mock-ups and Models
Mockup
main behavioural item of Root Model
external behavioural items of Root Model
1
0..n
(independent complements
to the Root Model)
Root Model
main behavioural item: Object or Class of interest
external behavioural items: Objects and Classes
constituting the environment of the behavioural item of interest
0..n
Binding
each Binding ensures
information exchange
between 2..n Models
Model
FORM-L Model
that is extended
or used as Library
contents
0..n
FORM-L
Model
Non-FORM-L
Model
contents
0..n
0..n
Partial
Model
0..n
Statement

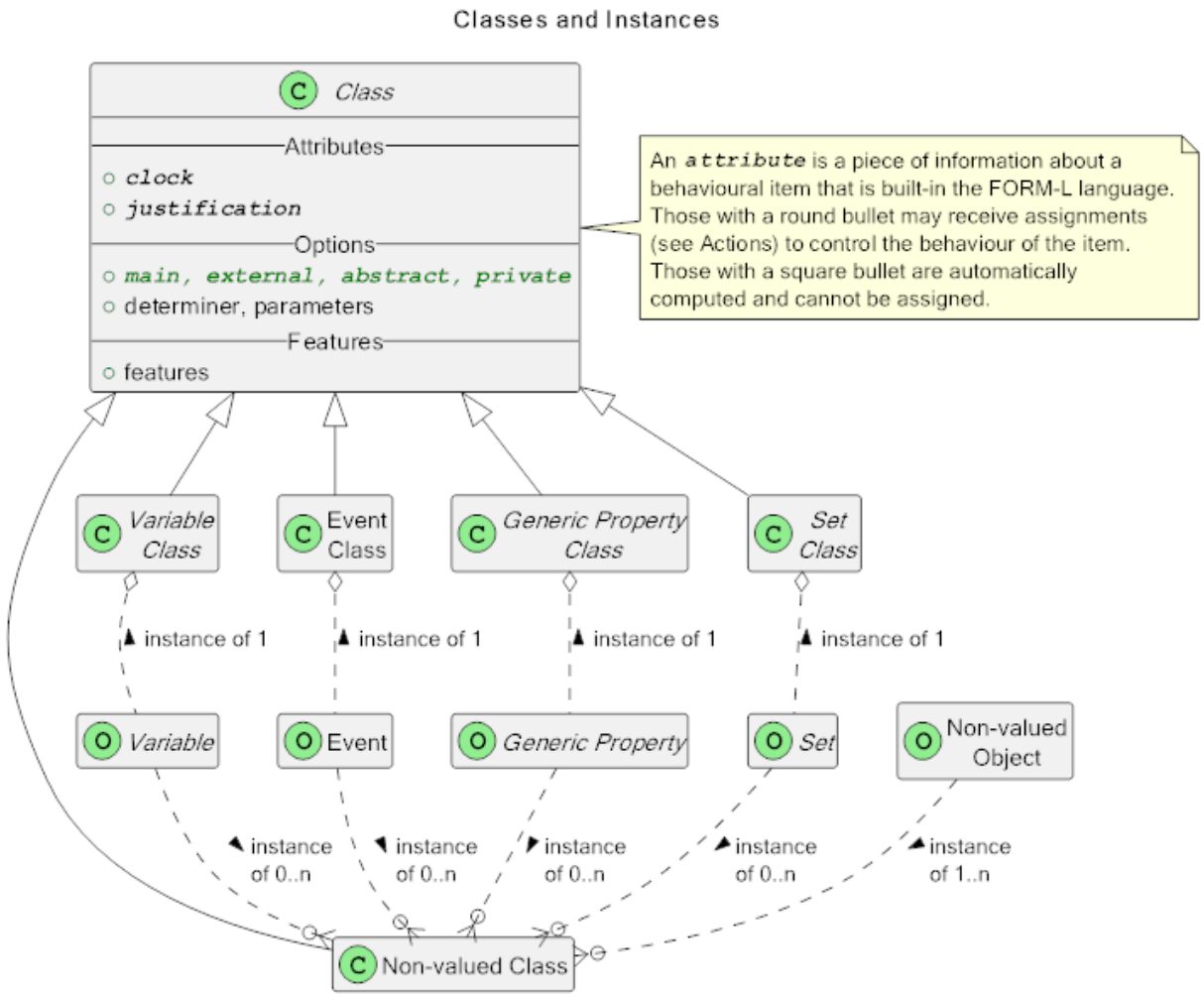

Statements
S Statement
D Declaration
R Redeclaration
d Definition Block
I Instruction

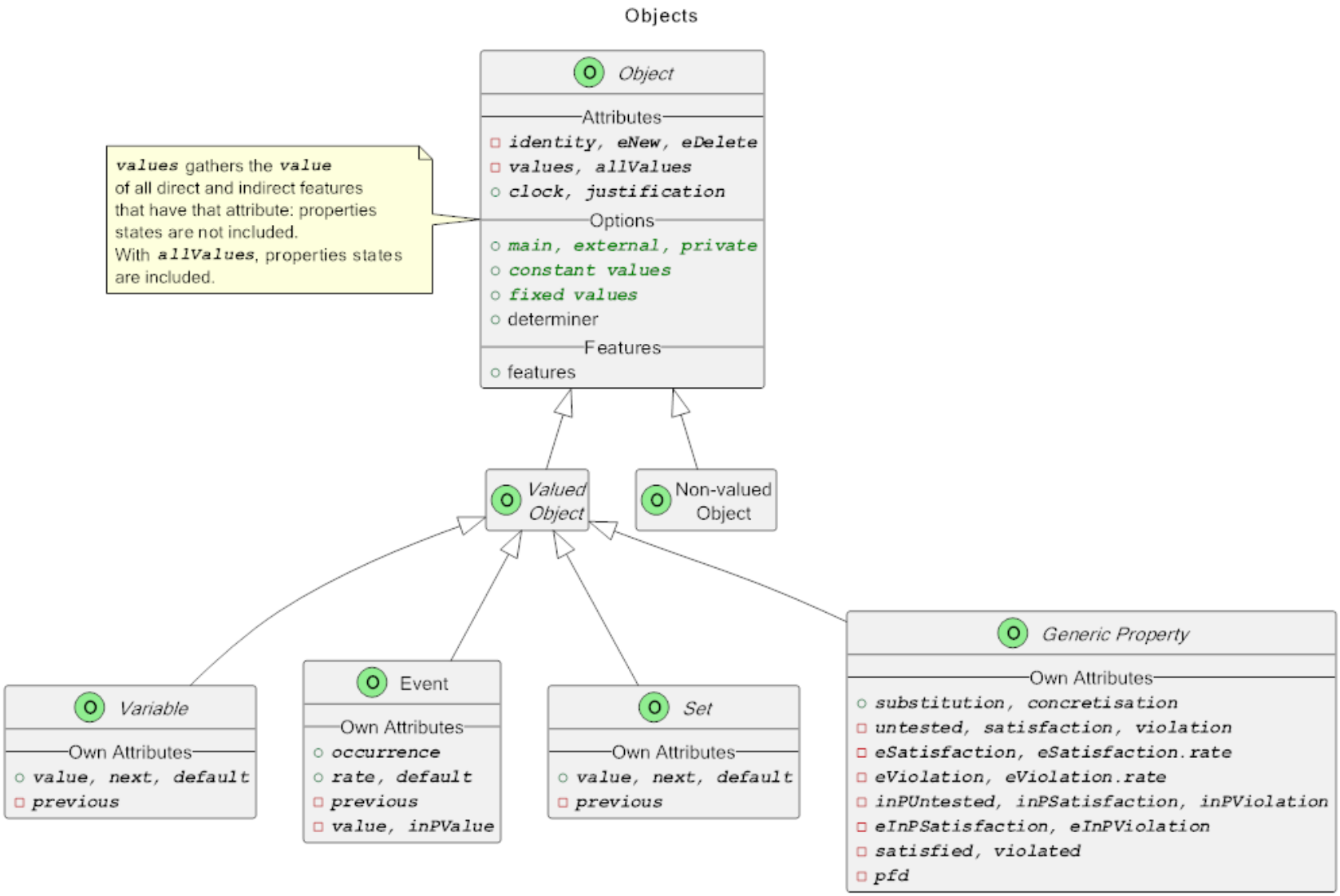
Declaration, Redeclaration and Definition

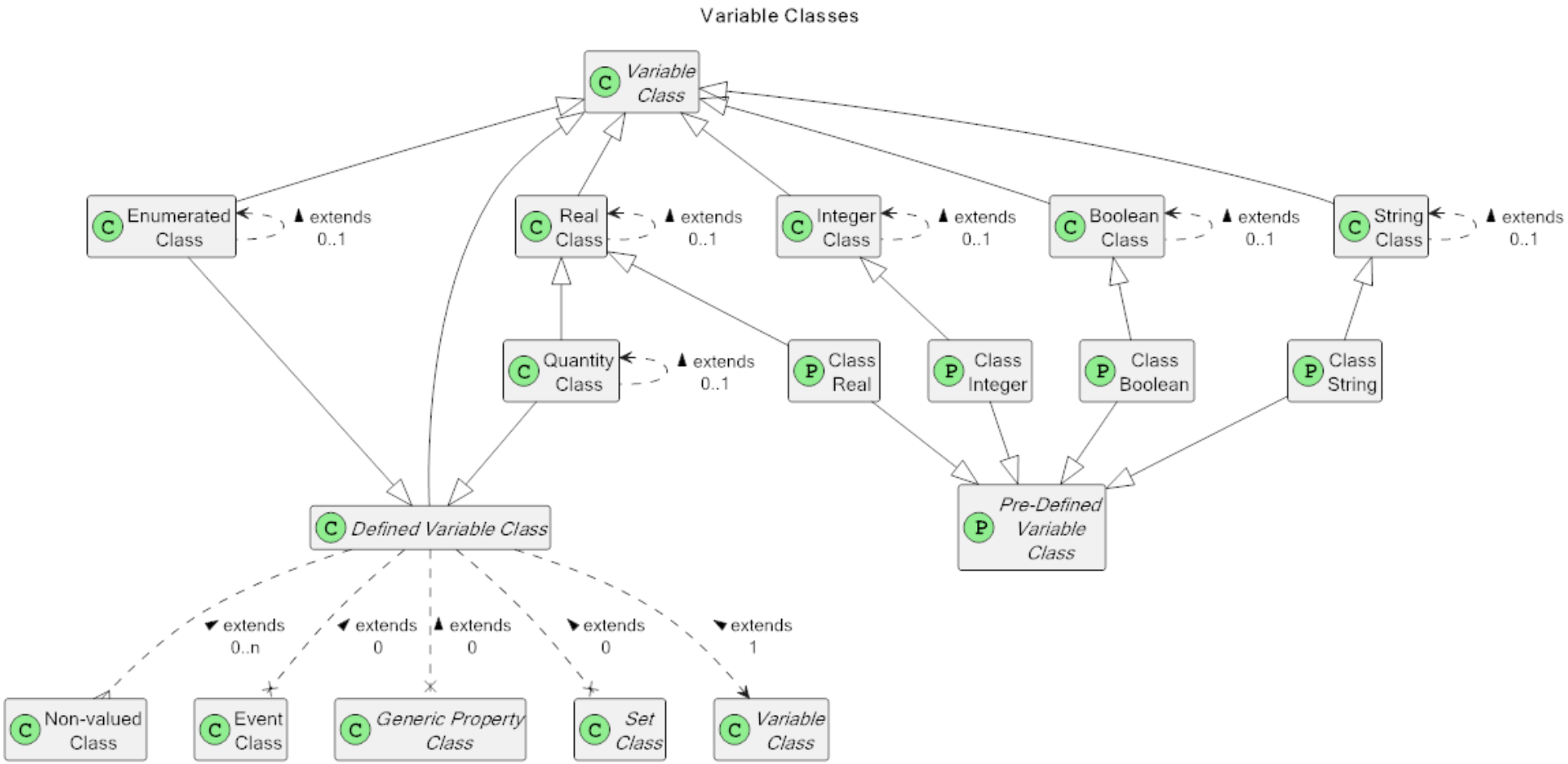


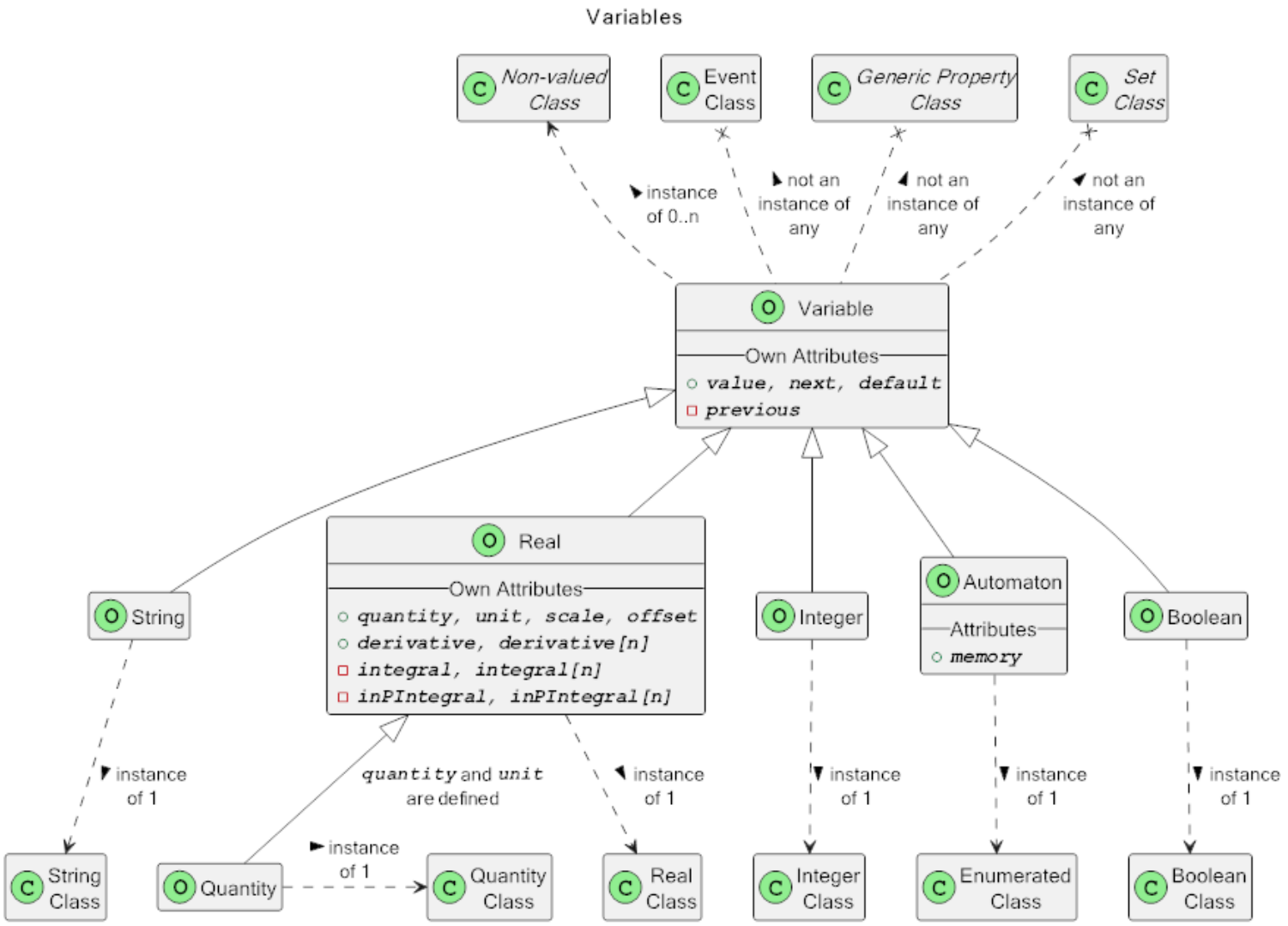

Pre-Defined Classes
Class
Defined Class
Pre-Defined Class
Pre-Defined Variable Class
Class Event
Pre-Defined Property Class
Class Object
Class Boolean
Class Integer
Class Real
Class String
Class Property
Class Objective
Class Requirement
Class Assumption
Class Guarantee
Class Guard

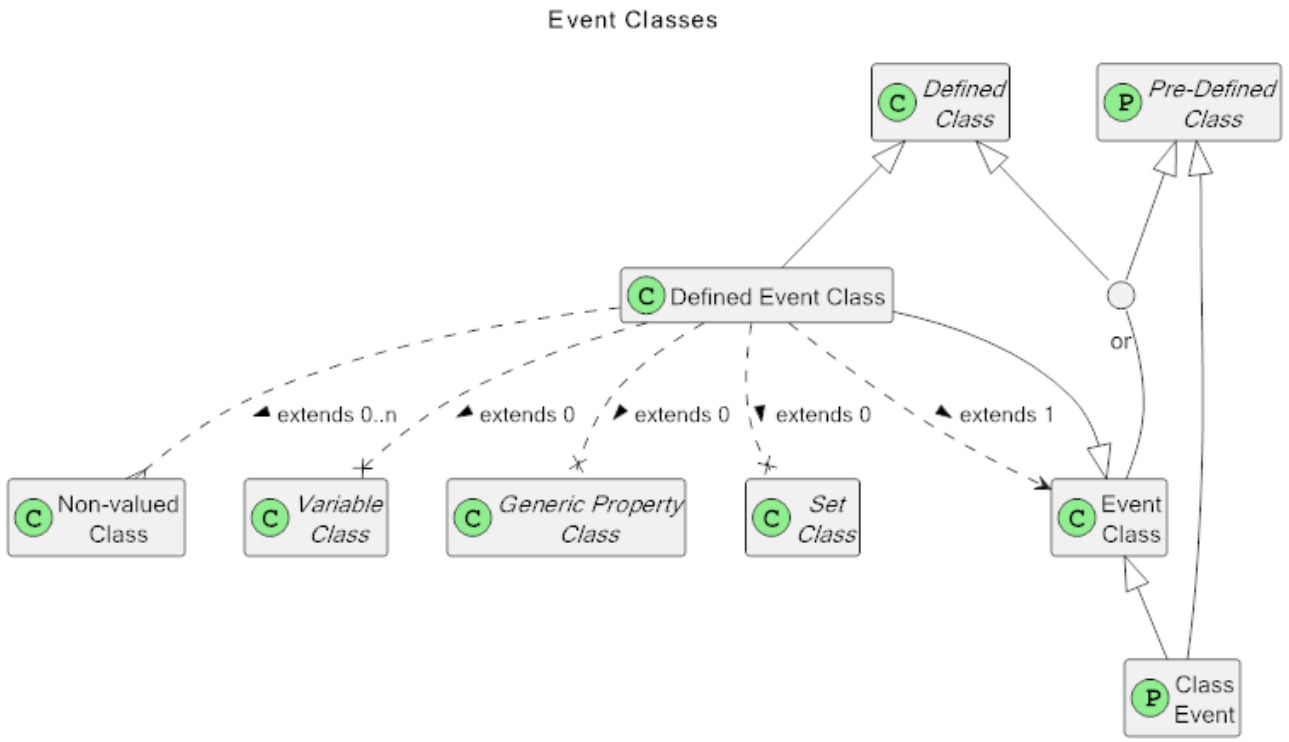

Defined Classes
Class
Defined Class
Valued Class
Pre-Defined Class
Non-valued Class
Extension Class
Valued Class
Enumerated Class
Set Class
1
1 class of set members
1
Non-Valued Extension Class
Valued Extension Class
Discrete States
Class
Set Indicator
Options
constraints on cardinal
determiner of members
variability of members
1..n being extended
0..n being extended
1 being extended
Non-Valued Class
Valued Class

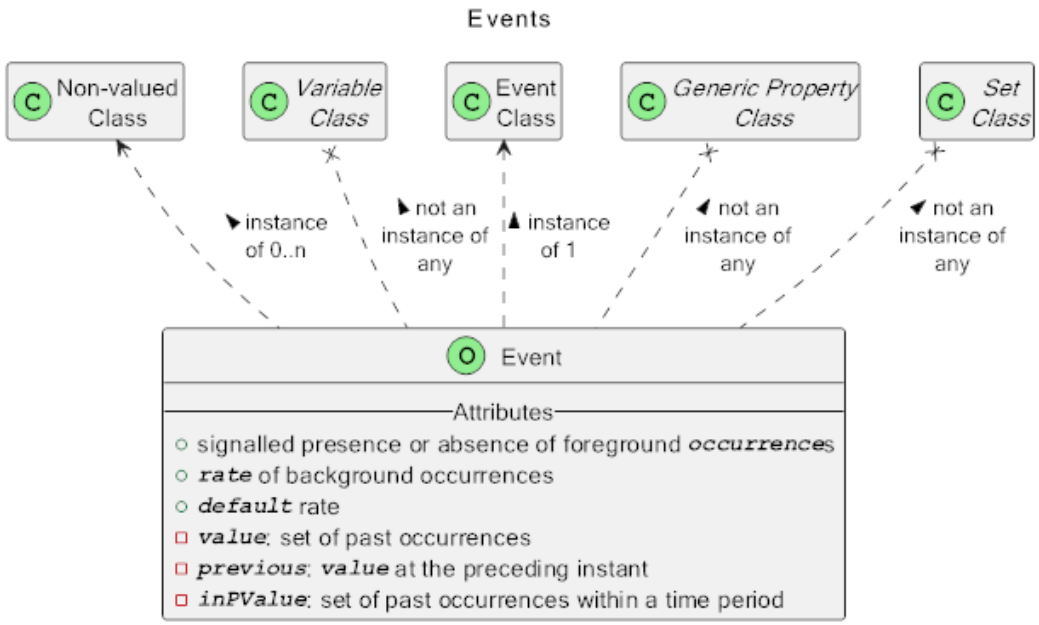

Classes and Instances
Class
Attributes
clock
justification
Options
main, external, abstract, private
determiner, parameters
Features
features
An attribute is a piece of information about a behavioural item that is built-in the FORM-L language. Those with a round bullet may receive assignments (see Actions) to control the behaviour of the item. Those with a square bullet are automatically computed and cannot be assigned.
Variable Class
Event Class
Generic Property Class
Set Class
instance of 1
instance of 1
instance of 1
instance of 1
Variable
Event
Generic Property
Set
Non-valued Object
instance of 0..n
instance of 0..n
instance of 0..n
instance of 0..n
instance of 1..n
Non-valued Class

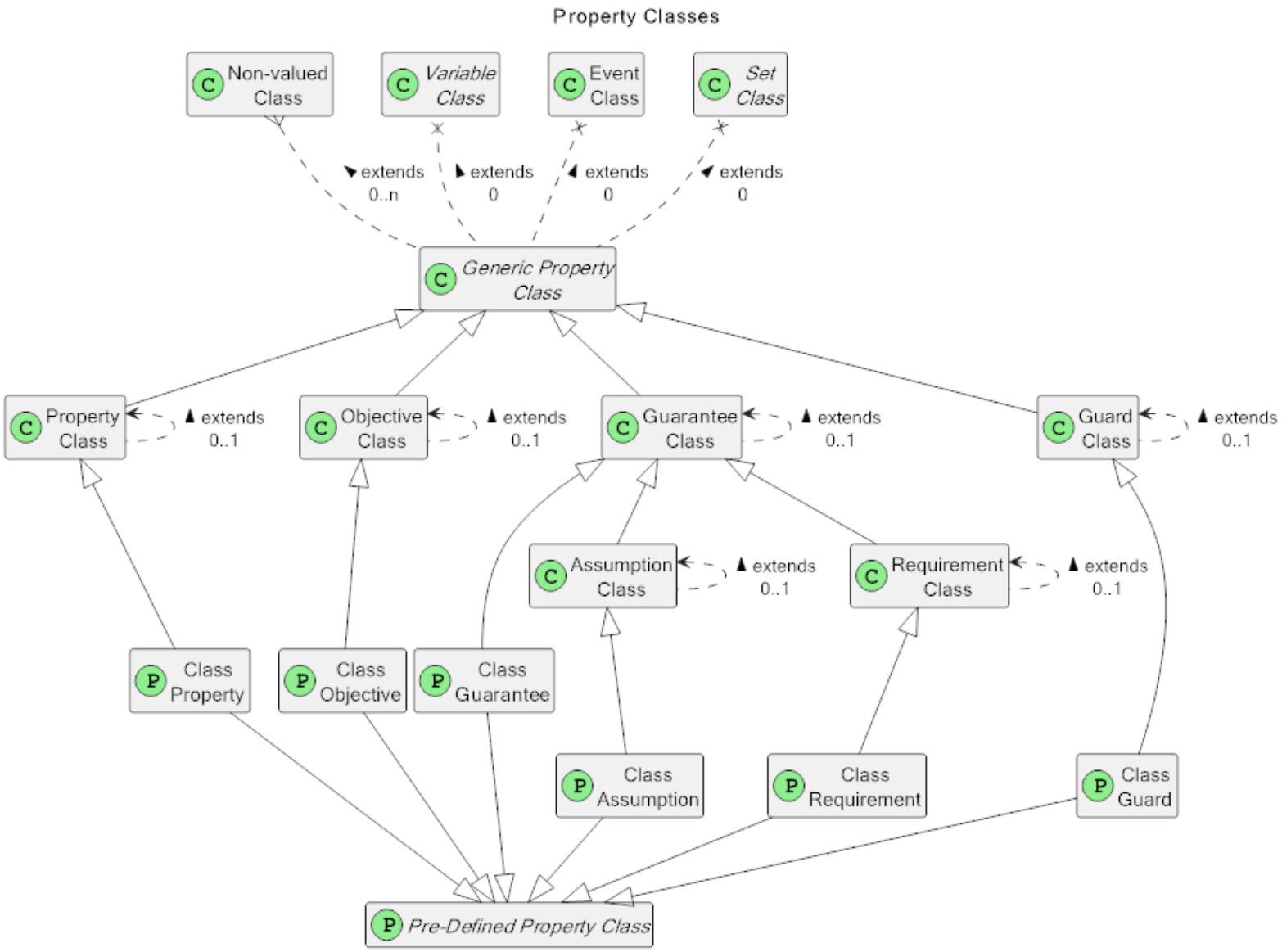

Objects
Object
Attributes
identity, eNew, eDelete
values, allValues
clock, justification
Options
main, external, private
constant values
fixed values
determiner
Features
features
values gathers the value of all direct and indirect features that have that attribute: properties states are not included.
With allValues, properties states are included.
Valued Object
Non-valued Object
Variable
Own Attributes
value, next, default
previous
Event
Own Attributes
occurrence
rate, default
previous
value, inPValue
Set
Own Attributes
value, next, default
previous
Generic Property
Own Attributes
substitution, concretisation
untested, satisfaction, violation
eSatisfaction, eSatisfaction.rate
eViolation, eViolation.rate
inPUntested, inPSatisfaction, inPViolation
eInPSatisfaction, eInPViolation
satisfied, violated
pfd

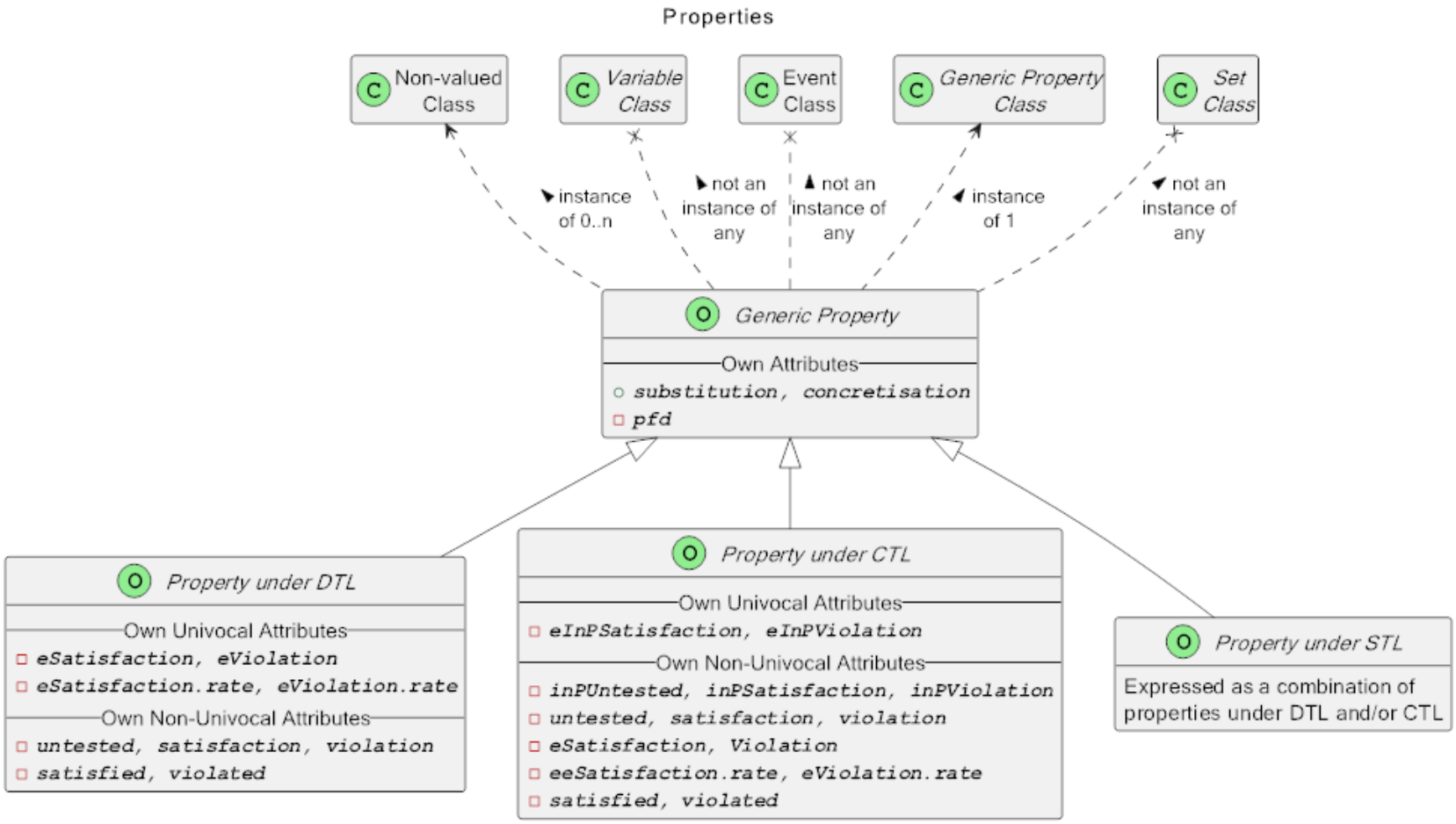

Variable Classes
Variable Class
Enumerated Class
extends 0..1
Real Class
extends 0..1
Integer Class
extends 0..1
Boolean Class
extends 0..1
String Class
extends 0..1
Quantity Class
extends 0..1
Class Real
Class Integer
Class Boolean
Class String
Defined Variable Class
Pre-Defined Variable Class
extends 0..n
extends 0
extends 0
extends 0
extends 1
Non-valued Class
Event Class
Generic Property Class
Set Class
Variable Class

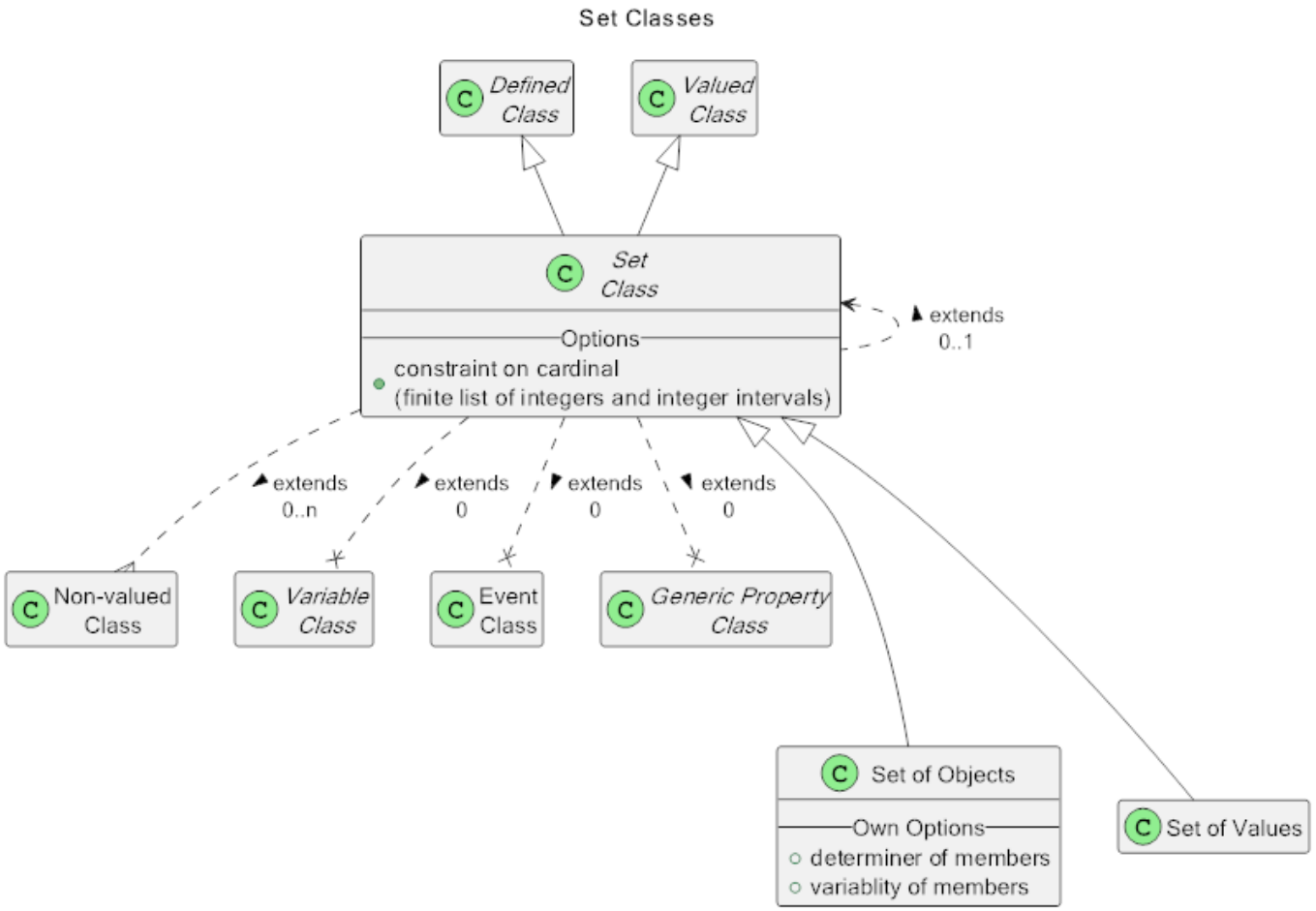

Variables
Non-valued Class
Event Class
Generic Property Class
Set Class
instance of 0..n
not an instance of any
not an instance of any
not an instance of any
Variable
Own Attributes
value, next, default
previous
Real
Own Attributes
quantity, unit, scale, offset
derivative, derivative[n]
integral, integral[n]
inPIntegral, inPIntegral[n]
String
Integer
Automaton
Attributes
memory
Boolean
instance of 1
quantity and unit are defined
instance of 1
instance of 1
instance of 1
instance of 1
instance of 1
String Class
Quantity
Quantity Class
Real Class
Integer Class
Enumerated Class
Boolean Class

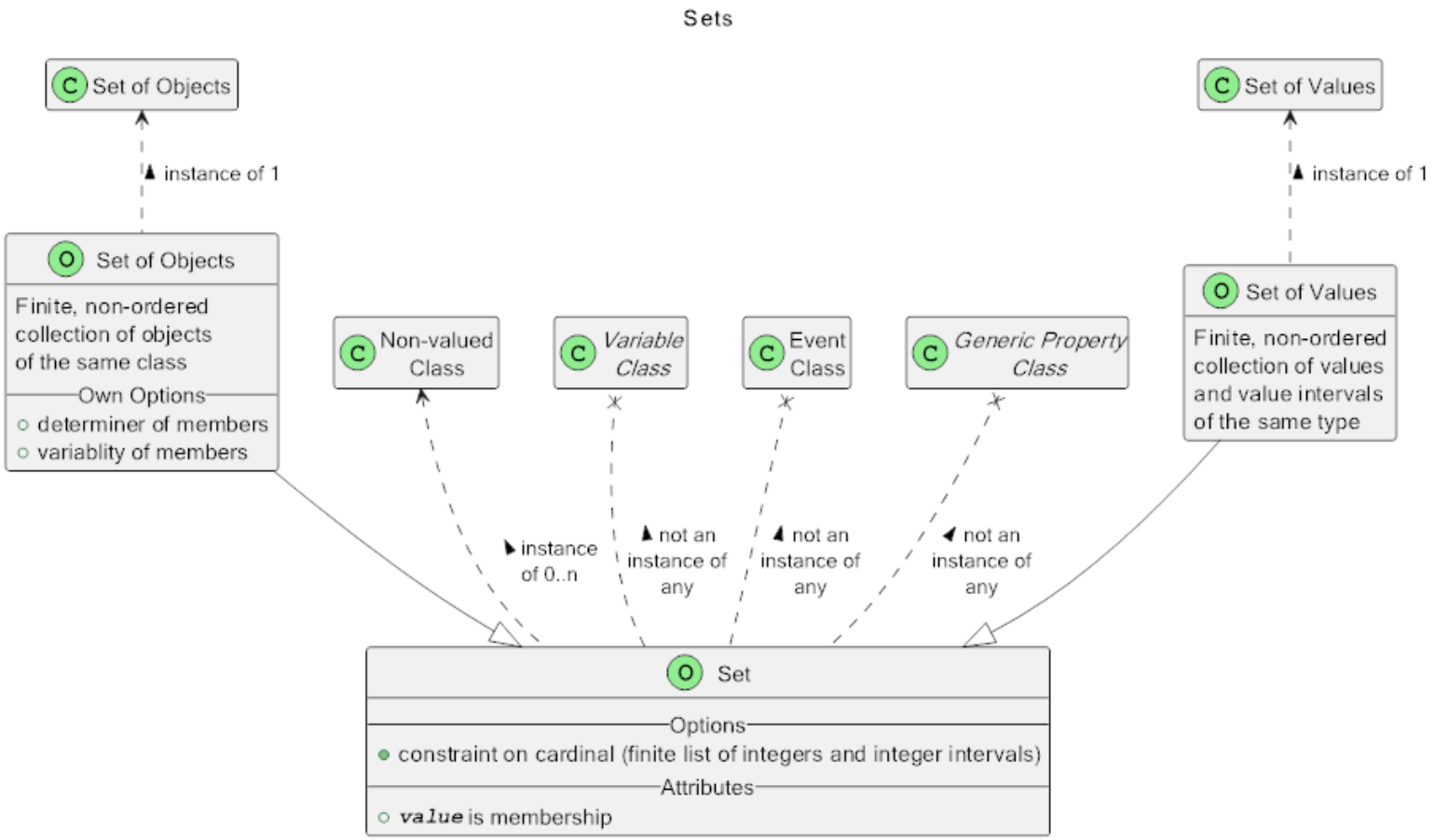

Event Classes
Defined Class
Pre-Defined Class
Defined Event Class
or
extends 0..n
extends 0
extends 0
extends 0
extends 1
Non-valued Class
Variable Class
Generic Property Class
Set Class
Event Class
Class Event

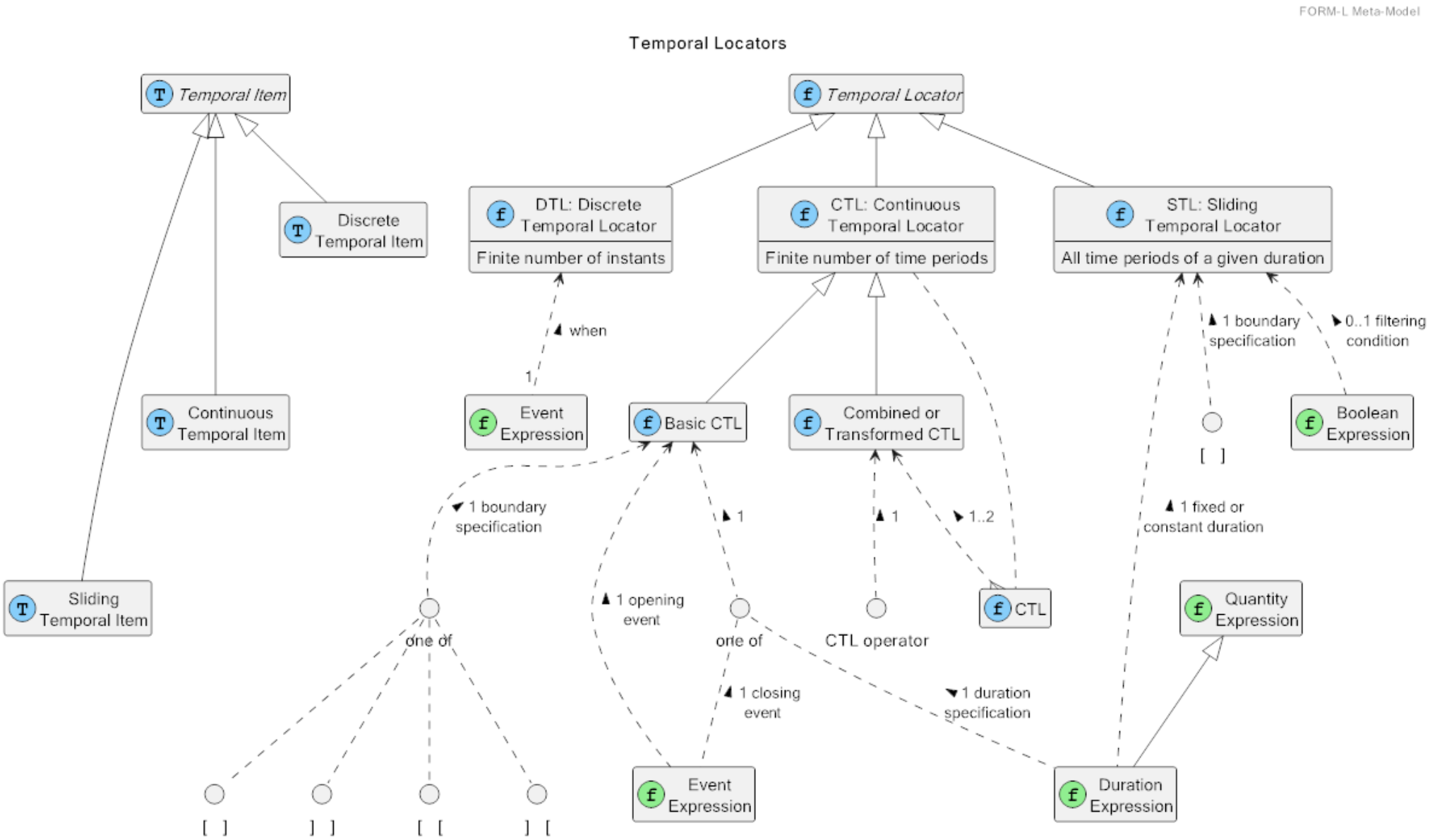

Events
Non-valued Class
Variable Class
Event Class
Generic Property Class
Set Class
instance of 0..n
not an instance of any
instance of 1
not an instance of any
not an instance of any
Event
Attributes
signalled presence or absence of foreground occurrences
rate of background occurrences
default rate
value: set of past occurrences
previous: value at the preceding instant
inPValue: set of past occurrences within a time period


Property Classes
Non-valued Class
Variable Class
Event Class
Set Class
extends 0..n
extends 0
extends 0
extends 0
Generic Property Class
Property Class
extends 0..1
Objective Class
extends 0..1
Guarantee Class
extends 0..1
Guard Class
extends 0..1
Assumption Class
extends 0..1
Requirement Class
extends 0..1
Class Property
Class Objective
Class Guarantee
Class Assumption
Class Requirement
Class Guard
Pre-Defined Property Class

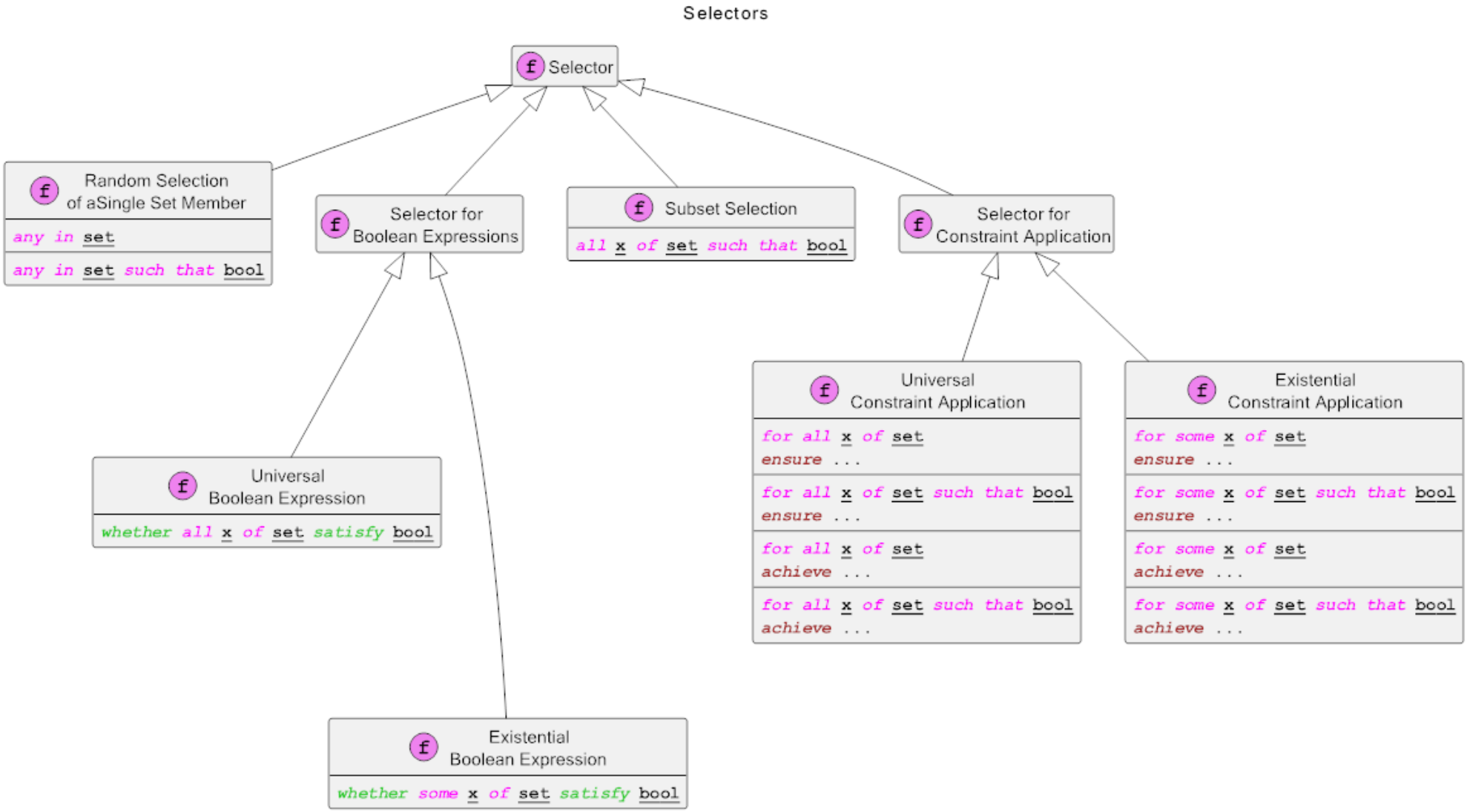

Properties
Non-valued Class
Variable Class
Event Class
Generic Property Class
Set Class
instance of 0..n
not an instance of any
not an instance of any
instance of 1
not an instance of any
Generic Property
Own Attributes
substitution, concretisation
pfd
Property under DTL
Own Univocal Attributes
eSatisfaction, eViolation
eSatisfaction.rate, eViolation.rate
Own Non-Univocal Attributes
untested, satisfaction, violation
satisfied, violated
Property under CTL
Own Univocal Attributes
eInPSatisfaction, eInPViolation
Own Non-Univocal Attributes
inPUntested, inPSatisfaction, inPViolation
untested, satisfaction, violation
eSatisfaction, Violation
eeSatisfaction.rate, eViolation.rate
satisfied, violated
Property under STL
Expressed as a combination of properties under DTL and/or CTL

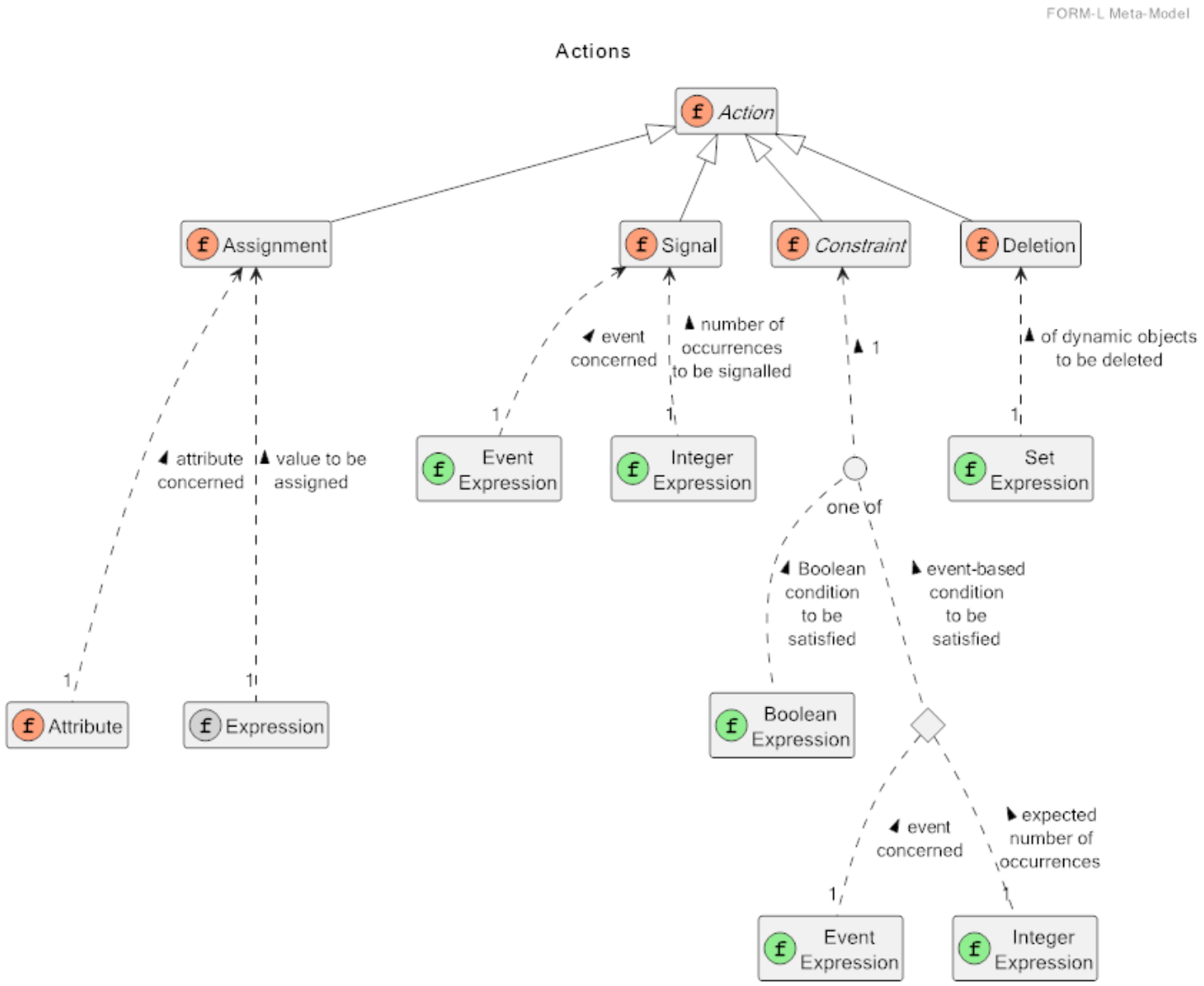

Set Classes
Defined Class
Valued Class
Set Class
Options
constraint on cardinal
(finite list of integers and integer intervals)
extends
0..1
extends
0..n
extends
0
extends
0
extends
0
Non-valued Class
Variable Class
Event Class
Generic Property Class
Set of Objects
Own Options
determiner of members
variablity of members
Set of Values

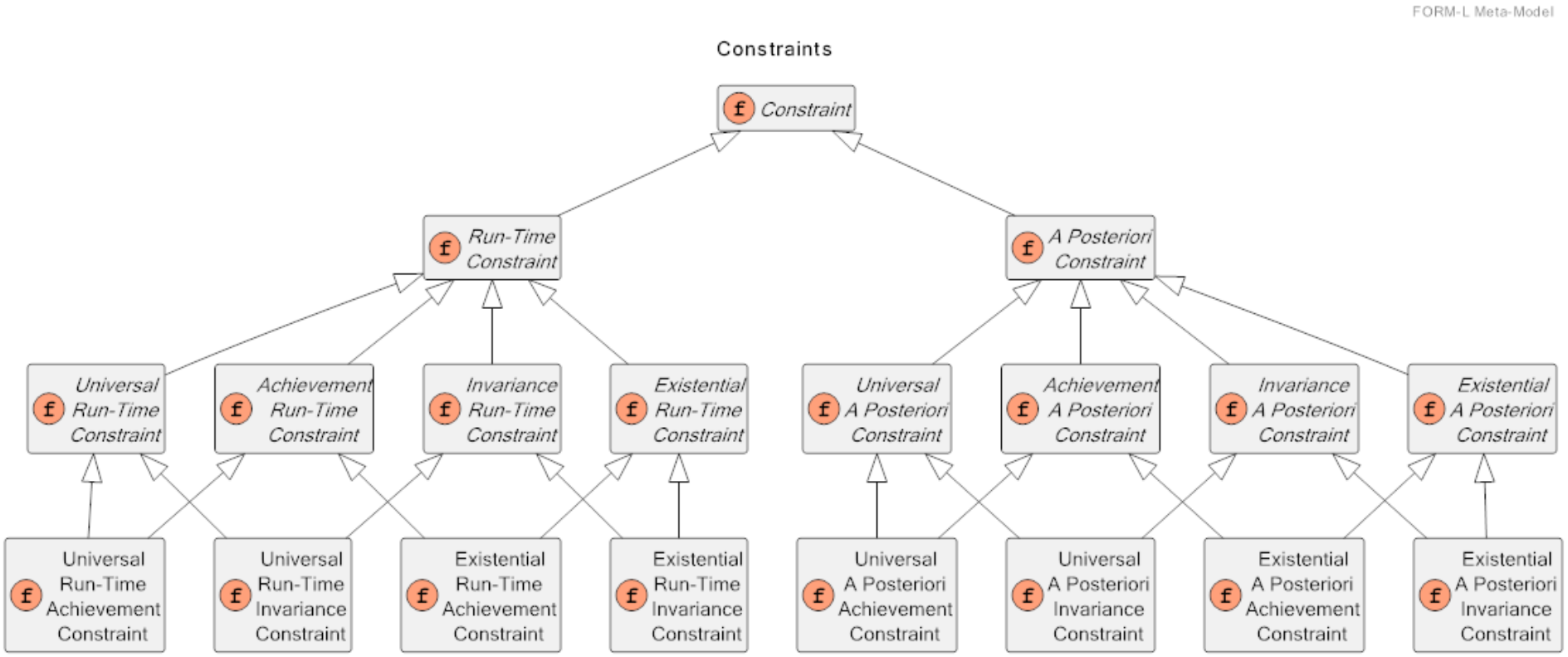

Sets
Set of Objects
instance of 1
Set of Objects
Finite, non-ordered collection of objects of the same class
Own Options
determiner of members
variablity of members
Non-valued Class
Variable Class
Event Class
Generic Property Class
Set of Values
instance of 1
Set of Values
Finite, non-ordered collection of values and value intervals of the same type
instance of 0..n
not an instance of any
not an instance of any
not an instance of any
Set
Options
constraint on cardinal (finite list of integers and integer intervals)
Attributes
value is membership


FORM-L Meta-Model
Temporal Locators
Temporal Item
Temporal Locator
Discrete Temporal Item
DTL: Discrete Temporal Locator
Finite number of instants
CTL: Continuous Temporal Locator
Finite number of time periods
STL: Sliding Temporal Locator
All time periods of a given duration
when
1
1 boundary specification
0..1 filtering condition
Continuous Temporal Item
Event Expression
Basic CTL
Combined or Transformed CTL
Boolean Expression
[ ]
1 boundary specification
1
1
1..2
1 fixed or constant duration
Sliding Temporal Item
one of
1 opening event
one of
CTL operator
CTL
Quantity Expression
1 closing event
1 duration specification
[ ]
]]
[[
] [
Event Expression
Duration Expression

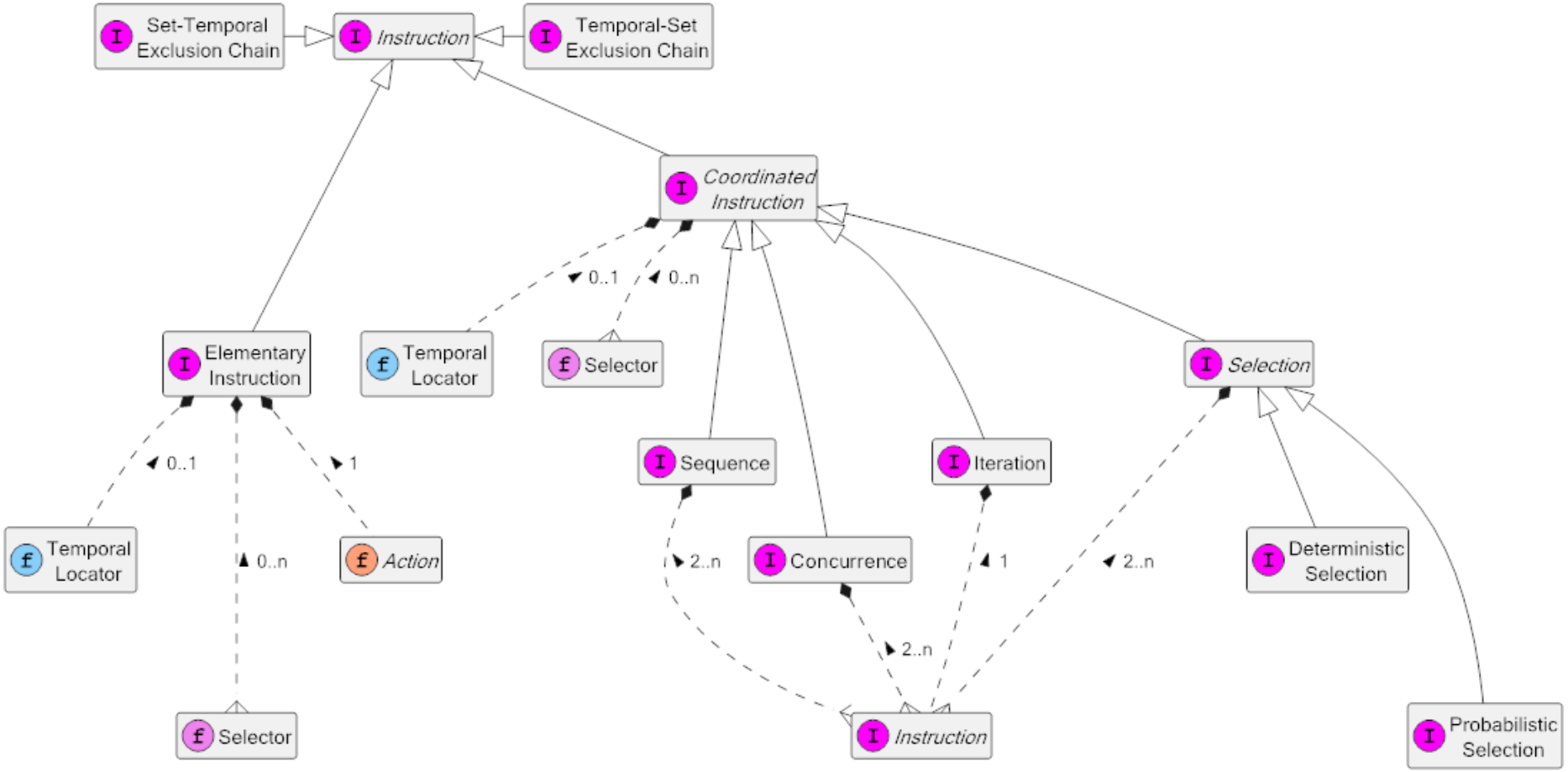
FORM-L Meta-Model

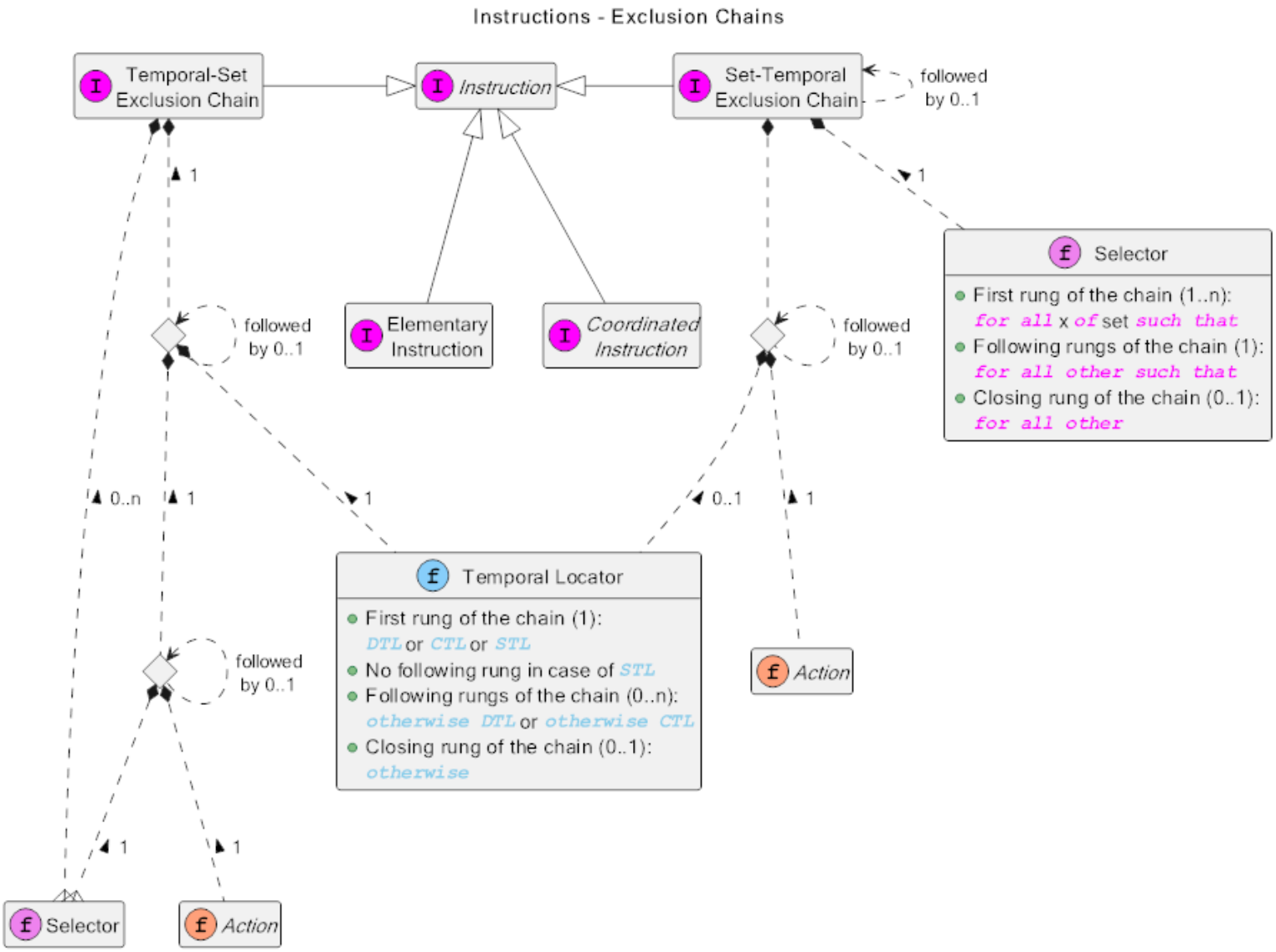


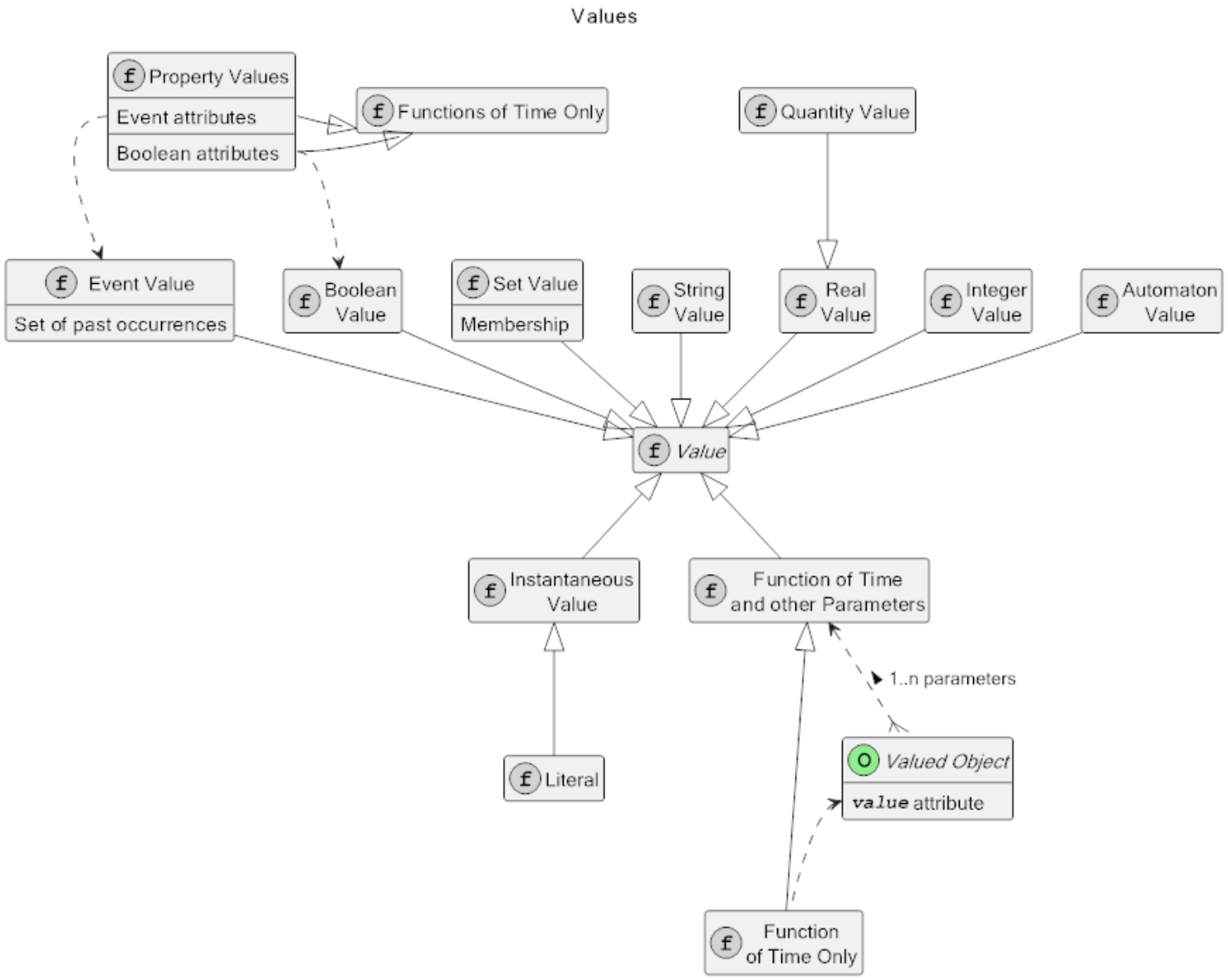

FORM-L Meta-Model
Actions
Action
Assignment
Signal
Constraint
Deletion
attribute concerned
value to be assigned
event concerned
number of occurrences to be signalled
1
of dynamic objects to be deleted
Event Expression
Integer Expression
Set Expression
one of
Boolean condition to be satisfied
event-based condition to be satisfied
Attribute
Expression
Boolean Expression
event concerned
expected number of occurrences
Event Expression
Integer Expression

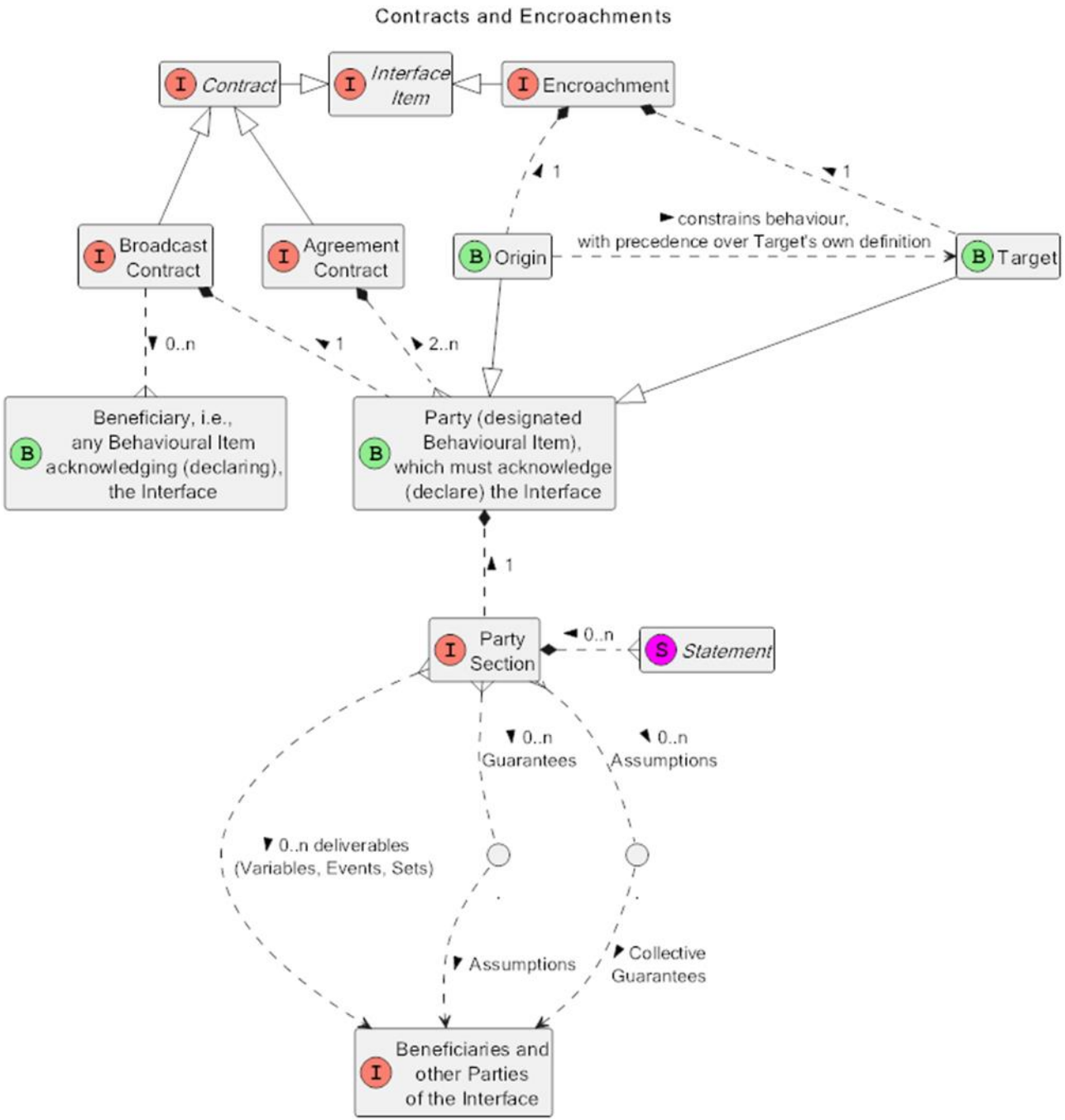

FORM-L Meta-Model
Constraints
Constraint
Run-Time Constraint
A Posteriori Constraint
Universal Run-Time Constraint
Achievement Run-Time Constraint
Invariance Run-Time Constraint
Existential Run-Time Constraint
Universal A Posteriori Constraint
Achievement A Posteriori Constraint
Invariance A Posteriori Constraint
Existential A Posteriori Constraint
Universal Run-Time Achievement Constraint
Universal Run-Time Invariance Constraint
Existential Run-Time Achievement Constraint
Existential Run-Time Invariance Constraint
Universal A Posteriori Achievement Constraint
Universal A Posteriori Invariance Constraint
Existential A Posteriori Achievement Constraint
Existential A Posteriori Invariance Constraint

Instructions - Elementary and Coordinated Instructions




Instructions - Exclusion Chains
Temporal-Set Exclusion Chain
Instruction
Set-Temporal Exclusion Chain
followed by 0..1
Elementary Instruction
Coordinated Instruction
Selector
First rung of the chain (1..n): for all x of set such that
Following rungs of the chain (1): for all other such that
Closing rung of the chain (0..1): for all other
followed by 0..1
followed by 0..1
followed by 0..1
Temporal Locator
First rung of the chain (1): DTL or CTL or STL
No following rung in case of STL
Following rungs of the chain (0..n): otherwise DTL or otherwise CTL
Closing rung of the chain (0..1): otherwise
Action
Selector
Action
1
0..n
0..1


Values
Property Values
Event attributes
Boolean attributes
Functions of Time Only
Quantity Value
Event Value
Set of past occurrences
Boolean Value
Set Value
Membership
String Value
Real Value
Integer Value
Automaton Value
Value
Instantaneous Value
Function of Time and other Parameters
1..n parameters
Literal
Valued Object
value attribute
Function of Time Only


Contracts and Encroachments
Contract
Interface Item
Encroachment
Broadcast Contract
Agreement Contract
Origin
Target
constrains behaviour, with precedence over Target's own definition
1
1
0..n
1
2..n
Beneficiary, i.e., any Behavioural Item acknowledging (declaring), the Interface
Party (designated Behavioural Item), which must acknowledge (declare) the Interface
1
Party Section
0..n
Statement
0..n
Guarantees
0..n
Assumptions
0..n deliverables (Variables, Events, Sets)
Assumptions
Collective Guarantees
Beneficiaries and other Parties of the Interface