# D2.1.1 – Modelica Extensions for Properties modelling

# Part III: FOrmal Requirements Modelling Language (FORM-L)

## WP2.1 – Properties modelling language
## WP2 – Properties modelling and Safety

Version 2.0

**Date** 29/04/2016


Authors

Thuy NGUYEN EDF

## Executive summary

This document is part III of the MODRIO D2.1.1 deliverable. It presents the main notions and the outline of the "canonical" syntax for the FOrmal Requirements Modelling Language (FORM-L) that is proposed in the framework of MODRIO WP2.1 to support systems engineering (development and operation) methodologies, as presented in "MODRIO D2.1.2 *Properties Modelling Method*". It also fits in the modelling architecture proposed in "MODRIO D2.1.1 Part II:*Modelling Architecture for the Verification of Requirements*". To illustrate the notions and the syntax proposed, extensive examples are given. Further, more complete examples are given in "MODRIO D8.1.3 *The BPS case Study*" and "MODRIO D8.1.3 *The SRI case Study*".

The creation of FORM-L has been decided after an unsuccessful search for an existing property specification language answering to the needs of the engineering of so-called "cyber-physical and human systems", i.e., complex systems combining multi-physics, computational and networking, and human aspects. Many property specification languages were found, but they only address the "cyber" aspects.

FORM-L has been specifically designed to formally express requirements on functionality (what the system must do and must not do, depending on situation, 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 formal expression of assumptions made, in particular regarding the environment of the system and human instructions. With suitable tools, such formal modelling of assumptions and requirements enables the use of massive simulation to verify the adequacy of system requirements, system overall design, detailed design, implementation and operational procedures, and to support activities such as failure analyses.

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 an "imperative" model which, given initial and boundary conditions, determines only one behaviour, a FORM-L model is a constraints model that allows many possible solutions and behaviours. Thus, 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 developed to support the first steps of design, where subsystems are identified and system requirements are allocated to subsystems. Particularly when each subsystem is to be subcontracted on the basis of a specific subsystem requirements specification, it is essential to make sure beforehand that the combined subsystems requirements specifications collectively satisfy the system requirements.

The FORM-L canonical 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. To further facilitate its use (particularly by non-English speaking persons), a graphical variant of FORM-L named FORM-GL (FOrmal Requirements Modelling Graphical Language), is under study. It will allow text and graphic "boiler-plates", i.e., natural language (e.g., French, German or Swedish) sentences or free-style diagrams containing placeholders. Each boiler-plate is associated with a FORM-L parametric model, the parameters being the contents of the placeholders. This should facilitate the formalisation of pre-existing system requirements: experiences at Dassault-Aviation and other systems engineering organisations have shown that their pre-existing natural language requirements for a given system can be represented with a limited number of boiler-plates.

The FORM-L canonical syntax can be used as a basis to define other variants, to facilitate tool support. For example, a Modelica variant is currently being defined, based on a specific Modelica library (ReqSysPro). In the same spirit, a FIGARO variant is being studied for probabilistic analyses. Also, a StimuLus[1] variant is being defined to allow automatic test case generation and requirements verification.

[1] StimuLus is a random generator of test cases that are complying with specified constraints, assumptions and scenarios. It is developed and marketed by ArgoSim.

# Summary

# Acronyms

| | |
|---|---|
| **COTS** | Commercial-Off-The-Shelf |
| **CTL** | Continuous Time Locator |
| **DTL** | Discrete Time Locator |
| **FMECA** | Failure Modes Effects and Criticality Analysis |
| **FORM-L** | FOrmal Requirements Modelling Language |
| **LTL** | Linear Temporal Logic |
| **OUS** | Object Under Study |
| **SCPS** | Socio-Cyber-Physical System |
| **SL** | Spatial Locator |
| **SoS** | System of Systems |
| **STW** | Sliding Time Window |
| **TL** | Time Locator |
| **WP** | Work Package |

# Glossary

| | |
|---|---|
| **Arity** | Specification of whether an item is a scalar, a vector of undetermined size, a vector of known size, or a set. |
| **Assumption** | *Property* that is supposed to be satisfied by the environment of a system, or that is specified to guide the supporting tools (e.g., test case generators or optimisation tools) towards cases of interest. |
| **Attribute** | Predefined characteristic of a FORM-L item. As opposed to *Feature*, which is a user-defined characteristic. |
| **Behavioural Model** | *Model* that specifies a behaviour in an imperative manner, e.g., in "classical" Modelica or in functional diagrams. |
| **Binding** | *Model* that specifies how information from one *Property* or *Behavioural Model* is transferred to another. |
| **Boolean Condition** | Expression that can be evaluated over time to true or false. |
| **Category** | Expression that precisely specifies the nature (*Type*, *Class* or *Event Class*) and the *Arity* of an item. |
| **Class** | Template describing *Objects* of the same nature and having the same set of *Features*. These *Objects* are instances of the *Class*. See also *Extension Class* and *Partial Class*. |
| **Clock** | *Discrete Time Locator* specifying when in continuous time an item placed in a *Discrete Time Domain* can perceive and act on its environment. |
| **Closed Definition** | A complete *Definition*, in a single place, of a *Variable* or an *Event*. |
| **Composite Instruction** | *Instruction* composed of multiple coordinated member *Instructions*. |
| **Condition** | See *Boolean Condition*. |
| **Conditional Probability** | Value in the [[0., 1.]] range that states the probability of an *Event* occurring at least once during a specified *Continuous Time Locator*. |
| **Configuration** | *Model* identifying a set of *Models* that constitutes a whole that can be simulated or analysed for optimisation. |
| **Constant** | Variable the value of which does not change with time and that is the same for all cases. |
| **Continuous Time Domain** | *Time Domain* where time is perceived continuously. There is only one such domain in a *Property Model*. |
| **Continuous Time Locator** | *Time Locator* that specifies possibly overlapping *Time Periods*. |
| **Contract** | *Model* that specifies the *Deliverables* and mutual *Obligations* and *Rights* between two or more *Parties*. |
| **Declaration** | Statement of the existence, name and *Category* of a FORM-L item. |
| **Definition** | Statement of the existence, name, *Category*, and specification of a FORM-L item. See also *Closed Definition* and *Open Definition*. |
| **Definition Snippet** | See *Snippet*. |
| **Derived Real Type** | Type defined by modifying some of the *Attributes* of *Primitive Type* Real. |
| **Discrete Time Domain** | *Time Domain* where time is not perceived continuously but only at specific instants, defined by a *Clock*. There may be several such |

| | |
|---|---|
| | domains in a *Property Model*. Different *Discrete Time Domains* usually specify different *Clocks*. |
| **Discrete Time Locator** | *Time Locator* that specifies time *Instants*. |
| **Elementary Instruction** | A verification (check), a *Value Definition*, an *Event Signalling* or the signalling of the end of a test case. |
| **Embedded Finite State Automaton** | *Finite State Automaton* that is considered as a single state in a higher-level automaton. |
| **Event** | Expression characterising the occurrences in time of a fact that has no duration. See also *Naked Event* and *Rich Event*. |
| **Event Class** | Template describing *Rich Events* of the same nature and having the same set of *Features*. These *Events* are instances of the *Event Class*. |
| **Extension Class** | *Class* that adds precision and *Features* to another *Class*. Its instances are also instances of the extended *Class*. |
| **Event Signalling** | Part of a *Snippet* that specifies instants of occurrence of the *Event* being defined. |
| **Feature** | User-defined characteristic of a FORM-L item. As opposed to *Attribute*, which is a predefined characteristic. |
| **Finite State Automaton** | *Variable* that is an instance of an enumeration. |
| **Fixed Variable** | *Variable,* the value of which is determined anew for each operational case but does not change during the case. |
| **Function** | Formula that calculates a scalar Boolean, Integer or Real (primitive or derived) based on one or more arguments. |
| **Guard** | *Property* that must be satisfied for a model to be valid: when a *Guard* is violated, the analysis, simulation or optimisation must be continued with another model the *Guard* of which is satisfied. |
| **In-Period Value** | Value the meaning of which is relative to a specific *Time Period*. |
| **Instruction** | See *Elementary Instruction* and *Composite Instruction*. |
| **Instant** | See *Time Instant*. |
| **Library** | *Model* that provides definitions that can be used by other *Models*. |
| **Model** | Either a *Configuration*, or a basic unit for the composition of *Configurations*: a *Property Model* , an *Object Model*, a *Contract*, a *Binding*, a *Library* or a *Behavioural Model*. Note that this definition of Model is NOT the definition of a Modelica model. |
| **Naked Event** | *Event* with no other information than the *Instants* of its occurrences. |
| **Object** | FORM-L item that describes something one wishes to represent in a *Model*, and that has user-defined *Features*. It can be defined as an instance of a *Class* (when several *Objects* share the same characteristics) or can be defined directly. |
| **Object Model** | *Property Model* that defines a single *Object*. |
| **Obligation** | *Requirement* that a *Party* in a *Contract* must satisfy. |
| **Open Definition** | *Variable* or *Event Definition* that allows *Definition Snippets* scattered throughout the *Property Model*. |
| **Partial Class** | Abstract *Class* that cannot be instantiated directly. |
| **Party** | *Property Model* that participates in a *Contract*. |

| | |
|---|---|
| **Period** | See *Time Period*. |
| **Primitive Type** | *Boolean*, *Integer*, *Real*, *String*, enumerated type |
| **Probability** | Value in the [[0., 1.]] range. Its value at a given instant represents the probability that a specified *Event* has already occurred at least once. |
| **Procedure** | *Statement* template possibly based on a number of arguments. |
| **Product Finite State Automaton** | Cartesian product of two or more *Finite State Automata*. |
| **Property** | Expression of something that is desirable, expected or required. |
| **Property Model** | *Model* that specifies constraints on behaviours, rather than the behaviours themselves. |
| **Relative Time Locator** | *Time Locator* expressed relative to a *Time Instant*. |
| **Requirement** | *Property* that must be satisfied: it is the objective of analysis or simulation to verify that it is not *violated*. |
| **Rich Event** | *Event* the occurrences of which carry specific information in addition to the instants of occurrence. See also *Event Class*. |
| **Right** | *Assumption* that a *Party* in a *Contract* may make, and that is satisfied by another *Party* as an *Obligation*. |
| **Snippet** | Formula that partially defines a *Variable* or an *Event*. It is composed of an optional *Spatial Locator*, a *Time Locator*, and an *Equation*. Special *Time Locator else* may be used once for a given *Variable* or *Event* to specify any *Time Instant* not covered by the other *Snippets*. |
| **Spatial Locator** | Formula that specifies which members of a set or a vector are concerned by a *Statement*. |
| **Statement** | Combination of an optional *Time Locator*, an optional *Spatial Locator* and an *Instruction*. |
| **Thread** | Sequence of *Instructions* performed one after the other. |
| **Time Domain** | Part of a *Property Model* where *Objects* have the same perception of time. See *Continuous Time Domain* and *Discrete Time Domain*. |
| **Time Domain Interface** | Part of a *Property Model* where *Events* and *Variables* in one *Time Domain* are made perceptible in another *Time Domain*. |
| **Time Instant** | Position in time that has no duration (as opposed to a *Time Period*, which has a strictly positive duration). That position is expressed relative to the beginning of time. |
| **Time Period** | Continuous stretch of time that has a beginning, and a strictly positive duration (as opposed to a *Time Instant*, which has no duration). |
| **Time Locator** | Expression that specifies WHEN an instruction is to be performed. |
| **Type** | *Primitive Type* or *Derived Real Type*. |
| **Value Definition** | Part of a *Snippet* that specifies the value of an attribute (e.g., attribute *value*) of the FORM-L item (e.g., a *Variable*) being defined. |

# 1. Context and objectives

## 1.1. Context and objective of the document

This document is part of the MODRIO D2.1.1 deliverable. Its main objective is to present the notions supported by the FOrmal Requirements Modelling Language (FORM-L). It also proposes a syntax for, and the semantics of, the language. To illustrate the notions, semantics and syntax, extensive examples are provided.

The language aims at supporting the systems engineering methodologies presented in [13]. It also fits in the modelling architecture proposed in [12]. To illustrate the notions and the syntax proposed, extensive examples are given. Further, more complete examples are given in [14] and [15].

Work done in other MODRIO work packages is of importance for, or could be affected by, FORM-L, in particular WP4 and WP2.2.

## 1.2. Rationale for FORM-L

At the beginning of the MODRIO project, the plan was first to list the desirable features for a formal modelling language suitable for complex **socio-cyber-physical systems** (SCPSs) and large, distributed **systems of systems** (SoSs), then to perform a survey of the existing languages, and finally to select the most appropriate one. Unfortunately, although the survey identified many languages suitable for "cyber" systems (i.e., systems based on computing and networking), none was found that also covered the "socio" and "physical" aspects of SCPSs and SoSs. Thus, the list of desirable language features gradually morphed into the FORM-L language.

One important objective of FORM-L is to support the engineering of a system **all along its lifecycle**, from prospective and scoping studies (aiming at defining the overall scope and missions of the system), to requirements specification (defining what the system must do and not do, and how well), system specification, architectural design, detailed design, verification and validation of implementation, preparation of operation and maintenance, training, effective operation and maintenance, upgrades and modifications.

In the initial phases, FORM-L allows the expression of the **assumptions** made regarding the environment of the system and the **requirements** placed on the system itself. This should be done while avoiding over specification, i.e., the specification of particular solutions rather than of the problem itself. This is particularly important when one is looking for optimal solutions: over specification often means that better solutions are likely to be overlooked. *Figure 1-1* illustrates the form of modelling supported by FORM-L: the modelling of **envelopes** of possible external conditions that the system may face (assumptions on the environment), and of legitimate system behaviours (requirements), as opposed to the modelling of specific behaviours or precise solutions. An envelope can be **absolute**, i.e., it MUST be satisfied. It can also be **probabilistic**, i.e., it CAN be violated, but there is a limit to the probability that it is so. The notion of envelope is also particularly important when dealing with the uncertainties inherent to prospective studies, human behaviour and physical systems (due to their analogue nature, noise and random failures).

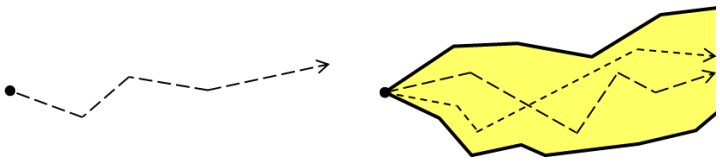

*Figure 1-1*
*Two very different forms of modelling. On the left hand side, modelling of a specific behaviour or individual solution. On the right hand side, modelling of envelopes of possible behaviours or solutions.*

The requirements for a system cover a very large spectrum that would be difficult and inconvenient to address with a single formal language. The scope of FORM-L is centred on **phenomena occurring in physical time**., even though it can take into account other aspects such as geometry and topology. The timing aspects are expressed with **time locators** specifying time **instants** or time **periods**. To address

the "cyber" aspects of SCPSs and in particular the so-called synchronous subsystems (which can perceive their environment and act on it only at particular discrete instants), FORM-L supports **discrete time domains**, in addition to the **continuous time domain** where human and physical activities occur.

The phenomena covered by FORM-L are those that can be represented by **events**, and **time functions** (called **variables**) yielding Boolean values, integer or real numbers (including physical quantities according to a **system of units**) or enumerated discrete values. Time functions yielding enumerated values can support **statecharts**, which are a generalisation of finite state **automata**. The phenomena covered include but are not restricted to physical behaviour: one could for example model operating costs, operators workloads or tasks ordering.

Some requirements or assumptions need to be expressed with respect to information, components or subsystems that are not known yet. For example, one might specify that "in normal operation, no pump in the system shall cavitate", or the "the system shall perform its mission even in the presence of any single component failure". FORM-L has the notion of **external** information, i.e., information that will be provided by other models. It also has the notion of **set** to represent parts of the system that are not known yet in enough detail to be defined in *extension* (i.e., by individually naming each of their members) and that are just defined in *intension*. **Spatial locators**, such as universal or existential quantifiers, specify which parts of a set (and of the system) are concerned with a property or a definition.

For a complex system, the specification of requirements and the design and verification of solutions require the intervention and cooperation of many different disciplinary teams working more or less independently and that often have difficulties understanding one another. Some disciplines may even be represented by multiple teams, each working on different parts of the system. Indeed, one of the great challenges of systems engineering is to ensure the necessary and sufficient coordination of multiple teams, in a rigorous and timely manner. Another issue is the need to adequately support the reuse of existing, suitable 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.

To these ends, FORM-L supports **modelling modularity** and **models composition**, where teams and organisations can develop separate models representing their view points on the system and its components at various levels of detail, and where models can be assembled together to offer a more complete and hopefully consistent view. As some teams and organisations are likely to use specific methods and tools, including for modelling, models composition should address not only FORM-L models, but also disciplinary models such as physical behavioural models, controls models, economics model, 3D geometric and topological models. Models composition with "consenting" models and top-down approaches is supported by the notion of **contract** which specifies the mutual obligations between models. Composition involving models developed separately (e.g., in bottom-up approaches) or with non-FORM-L models is supported by the notion of **binding**, which can provide the information needed by one model (the client model) by taking raw data in one or more other models (the provider models) and transform it as needed, without having to modify any of the client and provider models.

Later in the lifecycle, one needs to describe potential solutions, and verify whether they comply with the requirements. For a complex system, designing a solution at a level of detail allowing conventional behavioural simulation takes a very long time: the development of a behavioural model that can be simulated is a huge undertaking in itself. Any errors in the solution are likely to be revealed late in the engineering process, with significant impacts on cost and schedule. FORM-L allows the expression and the **verification** of preliminary designs, where subsystems and components are not fully detailed and represented by assumptions or requirements. Also, as FORM-L specifies acceptable envelopes, tool-supported **optimisation** (for design or operation) is another important potential use.

Simulation and testing is an important use of models. Large and complex systems, or activities like FMECA (Failure Modes, Effects and Criticality Analysis) or certain types of probabilistic analyses, need extremely large number of cases, which must be all consistent with the assumptions. The formal expression of these assumptions enables automatic cases generation. Specific case scenarios could be expressed as additional assumptions. The formal expression of requirements enables automatic verification of results.

Even though it is a formal language, FORM-L should be close to natural language, so that models can be inspected, reviewed and understood by application experts with limited expertise in FORM-L. Also, FORM-L models could be a very practical means for representing engineering information by explicitly

representing the WHY of, and the dependencies between, design decisions. The syntax presented in this report is English-based: for those not fluent in this language, a graphical variant of FORM-L based on boilerplates and natural language is already under study. This is important

## 1.3. Structure of the document

As the FORM-L notions are interdependent, Section 3 provides an overview of the main notions. The following Sections present the notions in more detail:

- Section 4: Variables & Types
- Section 5: Functions & Function Calls
- Section 6: Events & Event Occurrences
- Section 7: Classes & Objects
- Section 8: Durations
- Section 9: Discrete Time Locators (DTL)
- Section 10: Sliding Time Windows (STW)
- Section 11: Continuous Time Locators (CTL)
- Section 12: Spatial Locators
- Section 13: Properties
- Section 14: Coordination
- Section 15: Time Domains
- Section 16: Models Organisation

# 2. Preliminary Remarks

## 2.1. Scope

The properties targeted by FORM-L are dynamic properties, i.e., properties that vary with time and that can be evaluated by analysis or simulation, or that can be used for optimisation These properties could be associated with a wide range of systems engineering aspects, such as physics, controls, human behaviour, operating costs and task scheduling.

## 2.2. Methodology Support

The language proposed here has been designed to support the methodologies described in MODRIO 2.1.2 *Methodology*. This includes in particular a clear separation of models serving different purposes in the systems engineering lifecycle or in the support to systems operation. This is supported by FORM-L with notions like property models, requirements, assumptions, external information (to be supplied either by other models or by engineering databases), probabilistic properties, sets, etc.

The methodology also proposes various usages to support the different phases and activities of the systems engineering lifecycle and of system operation: this may include functional validation of system requirements, verification of preliminary designs, verification of detailed design, dysfunctional analyses (probabilistic failure analyses, FMECA, ...), subsystems and system testing (Hardware-in-the-Loop testing, Monte-Carlo testing, statistical testing, ...), verification of operational procedures, what-if prediction during operation, optimisation within constraints, etc.

## 2.3. Legibility

The clarity of the FORM-L syntax is important, as the language is mainly intended to be used or read by application specialists rather than modelling experts. A graphical variant of the language (FORM-GL) is

described in a separate document D2.1.1 *FORM-GL*. The objective of the graphical version is to further facilitate legibility. It allows models authors to find a right balance between text and graphics. It also allows 'boiler-plate' expressions in natural languages other than English.

## 2.4. Notations

The FORM-L syntax[1] is defined in terms of **terminal tokens** (that appear directly in a FORM-L expression, i.e., keywords, and symbols such as **`+ - * / =`** etc.), and **syntactic terms** (which are artefacts used to describe the syntax). In this document:

- The FORM-L syntax is "formally" described in a **`bold Courier New`** font.
- Terminal tokens are in **`black`**. FORM-L keywords are in ***`black italics`***.
- Symbols describing the syntactic terms, and the names of the syntactic terms, are in **`blue`**.
- The names of **basic syntactic terms** (i.e., terms that are not defined relative to other terms and / or terminal tokens) are in ***`blue italics`***.
- The names of syntactic terms that are defined relative to other syntactic terms and / or terminal tokens are in a **`straight`** font.
- A syntactic term may have complementary information in **`red`**. E.g., **`set(bool)`** is composed of a syntactic term for set expression, and of an additional information informing the reader that the set must be a set of Booleans.
- The definition of a syntactic term is in the form **`sTerm = sExpression ;`** , where **`sTerm`** is the name of the syntactic term being defined, and **`sExpression`** is a syntactic expression that defines the syntax of the term.
- A syntactic sub-expression enclosed in **`[ ]`** is optional.
- A syntactic sub-expression enclosed in **`( )*`** can be repeated 0 to n times.
- A syntactic sub-expression enclosed in **`( )+`** is repeated 1 to n times.
- A syntactic sub-expression enclosed in **`( )i+`** is repeated **`i`** to n times.
- A syntactic sub-expression enclosed in **`{ ;}*`** can be repeated 0 to n times, using the character before the **`}`** as a separator.
- A syntactic sub-expression enclosed in **`{ ;}+`** is repeated 1 to n times, using the character before the **`}`** as a separator.
- A syntactic sub-expression enclosed in **`{ ;}i+`** is repeated **`i`** to n times, using the character before the **`}`** as a separator.
- Syntactic sub-expressions enclosed in **`( )`** and separated by **`|`** represent a 1 out-of n choice.
- Syntactic sub-expressions enclosed in **`[ ]`** and separated by **`|`** represent an optional 1 out-of n choice.
- When the same syntactic term appears multiple times in a syntactic expression, a specific subscript is associated with each role.

[1] Note for compiler developpers: this description of the FORM-L syntax aims mainly at clarity. It is therefore not expressed in an LL(1) manner, though intrinsically, the syntax is (or is expected to be) LL(1).

Illustrative examples are given in shaded boxes for easy identification. By convention, in the examples, the event names begin with an e , the type and class names begin with an upper-case letter, and the instance names begin with a lower-case letter. Also, the use of a predefined FORM-L library defining physical **constants** of various types is assumed. For example:

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

# 3. Main FORM-L Notions

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

## 3.1. Overall Modelling Organisation

Modelling in FORM-L is organised into **models** of different sorts: property models, object models, libraries, behavioural models, bindings, contracts and configurations.

### 3.1.1. Property Models

A **property model** is a set of consistent **declarations** and **definitions** of FORM-L items (e.g., variables, objects, events, types, classes, event classes, functions, properties, coordinations, procedures, time domains and contracts) that collectively represent a complete or partial view of a real-world system and its environment. Contrary to behavioural models, a property model usually does not specify a behaviour in an imperative and deterministic manner: it specifies constraints (a.k.a. **properties**) that should be satisfied by candidate designs and / or implementations, and also by operational cases.

An **object model** is a property model that directly defines a single main object, possibly with embedded objects. In a similar manner, a **class model** is a property model that directly defines a single class, possibly with its embedded classesn and a **contract model** is a property model that directly defines a single contract.

Multiple property models may be used to represent the same real-world system, each describing different aspects, parts or views. For example, these might include a property model specifying the requirements on the system and the assumptions on its environment, and other property models representing various design stages.

A property model specifying the requirements on the system and the assumptions on its environment should clearly and formally state:

- The boundaries of the system under study, and its possible states (normal and abnormal).
- The entities, including human actors, interacting with the system (which collectively constitute the environment of the system), and their possible states.
- The possible operational goals that operators can have with respect to the system at a given instant, including possible changes of goals.
- Any assumptions made regarding the environment, and any requirements placed on the system, possibly depending on situation (defined as a combination of system states, environment states and operational goals).
- The system operational requirements, including quality of service requirements and operational constraints aiming for example at reducing wear and tear of system components.

Property models may also be used to define **generic scenarios** in the form of additional assumptions on system inputs that could be used to automatically generate **operational cases** of interest.

### 3.1.2. Libraries

A **library** provides general-purpose definitions that can be used and re-used in other models.

The examples provided in this document assume the use of libraries that define the physical quantities and units of the SI system of units, the basic physical and mathematical constants, and general purpose functions.

### 3.1.3. Non-FORM-L Models

At some point in the engineering process, the system, or parts of it or of its environment, can be represented by one or more non-FORM-L models such as **behavioural models** (e.g., in Modelica or in functional diagrams) specifying not constraints but precise phenomena or values determined by specific and specialised disciplines. FORM-L does not define these models: it just references them.

### 3.1.4. Bindings

Different property and behavioural models may be developed by different teams independently from one another, each having their own viewpoint on the system, modelling issues, rules and conventions. When they are grouped together into a configuration, some information transfer from one model to another might be necessary. However, it is preferable not to modify any of these models. **Bindings** are used to bridge gaps between models, and specify how information from one model is taken from, translated and transferred to another.

### 3.1.5. Contracts

Whereas a binding associates property and behavioural models that have been developed independently from one another, a **contract** associates "consenting" property models that have agreed on its terms. Besides identifying the concerned parties, a contract identifies the items (variables, objects and events) that are to be delivered by a given party and made accessible to the other parties. It also specifies the rights (in terms of assumptions that can be made regarding these items) and obligations (in terms of requirements that must be satisfied, also regarding these items) of each party.

Contracts may be used for example to model and specify, and then verify, the mutual obligations and expectations between the different parts of a system.

### 3.1.6. Configurations

A property model may have inputs (identified by **external declarations**) that must be provided by other models (property or behavioural models) identified either directly through contracts or indirectly through bindings. These other models may in turn have their own inputs necessitating additional models, contracts and bindings.

A **configuration** specifies a set of property and behavioural models, libraries, bindings and contracts that constitute a whole that can be ‘compiled’ and then analysed for optimisation or simulated. As with contracts some properties may be considered either as requirements or as assumptions depending on context and point of view, a configuration also identifies one or more property models are the **reference models**: whether a given property is viewed as a requirement or an assumption is made from the point of view of the reference models. When there are multiple reference models, they must not contradict one another.

Many configurations may be defined for the same real-world system, depending on the engineering activities to be supported, on the design alternatives to be assessed, on the generic scenarios to be tested, etc.

### 3.1.7. Operational Cases

For a given configuration, an **operational case** (or **case**, for short) has a specific **trajectory** for each variable and event. The variables and events are those declared directly in the configuration, or indirectly as objects or rich events features. A trajectory for a variable is a value sequence in physical time. A trajectory for an event is the occurrences sequence, also in physical time. A case has a **beginning of**

**time**, and may or mayt not explicitly specify an **end of time**, when all what is relevant for the case or for the configuration has occurred.

**Simulation** and **optimisation** are made from the perspective of one or more objects designated to be the **object(s) under study** (OUSs). A case must be consistent with the assumptions made by the OUSs and with all the definitions in the configuration. A case that violates none of the requirements specified by the OUSs is a **legitimate case**. Otherwise, it is a **delinquent case**. The set of all possible legitimate cases defines the **envelope** of authorised behaviours.

The objective of verification is to check that there exists no delinquent case. The objective of optimisation consists to look for a legitimate case that minimises a given cost function.

## 3.2. Declarations & Definitions

### 3.2.1. Declarations

A **declaration** states the existence, name and nature of a FORM-L item. It allows the item to be used before it is defined. An **external** declaration introduces an item that is used but not defined by a property model. That definition must be provided by another model (through a contract or a binding).

An item declared to be **external** cannot be defined in the model (that would be a contradiction). The declaration of a non-external variable or event can serve as an open definition. It is a good practice then to specify an initial value through the *start* attribute.

### 3.2.2. Definitions

A **definition** not only states the existence, name and nature of an item: it also specifies it. Most definitions are closed, i.e., they completely specify the item in a single place in the model, and do not allow alterations anywhere else. A variable or an event can have an **open definition** that is scattered throughout the property model. The choice between these two styles is at the convenience of models authors.

For a variable or event that has no, or limited and simple dependencies with others, a closed definition is generally preferable. Such a definition can be expressed either globally by a single expression covering all times, or in an itemised manner with **definition snippets** (see Section 3.2.3 *Definition Snippets*). When using snippets, the definition is closed by an **else clause** covering all times not covered by any snippets.

An open definition is itemised, i.e., based on snippets, but does not have an else clause. Some snippets can be stated within the definition, the others are scattered throughout the model. Such definitions are convenient for example when several variables or events change value or occur in a coordinated manner. However, readers need to collect the complete definition throughout the property model.

Note. FORM-L is a declarative language, where, at a given instant, the evaluation order for variables and events can, but is not necessarily, specified. When all variables and events are independent from one another, any order is appropriate. When there are acyclic dependencies, the order matters, but a natural one can be automatically determined. When there are circular dependencies, the model must impose one. Analysis tools can verify that such is the case.

### 3.2.3. Definition Snippets

A definition snippet addresses three basic questions: **WHAT**, **WHEN** and **WHERE**.

Note. **HOW WELL** is mostly expressed with probabilistic properties (see Sections 3.9 *Properties* and 13 *Properties*). **WHY** and **HOW** are also important questions for systems engineering. They are not addressed directly by FORM-L, but by a suitable property modelling approach (see [13]).

### 3.2.4. WHAT: Instructions

When defining a variable (see Section 3.3 *Variables & Types*) or an event (see Section 3.5 *Events & Events Occurrences*), the WHAT is specified by a **value expression**. When defining a property (see Section 3.9 *Properties*), the WHAT is expressed by a **check**, i.e., a constraint to be satisfied.

### 3.2.5. WHEN: Time Locators

The WHEN is expressed with **time locators**, which specify at which **instants** or during which **time periods** the WHAT is assumed or required. There are three types of time locators (see Figure 3-1):

- A **Discrete Time Locator** (DTL) defines a number of **time instants** (see Section 9 *Discrete Time Locators (DTL)*). An instant has a position in time and a rank in the DTL, but no duration.
- A **Continuous Time Locator** (CTL) defines a number of possibly overlapping **time periods** (see Section 11 *Continuous Time Locators (CTL)*). A time period has a beginning and an end, and has a strictly positive duration.
- A **Sliding Time Window** (STW) consists of all possible time periods of a given duration (see Section 10 *Sliding Time Windows (STW)*).

To express WHEN, in addition to time locators, finite state automata and statecharts (see Section 3.3.7 *Enumerations & Automata*), FORM-L also has the notion of **time domain**, (see Sections 3.11 *Time Domains* and 15 *Time Domains*), which is useful, or even necessary, when considering certain types of hybrid systems.

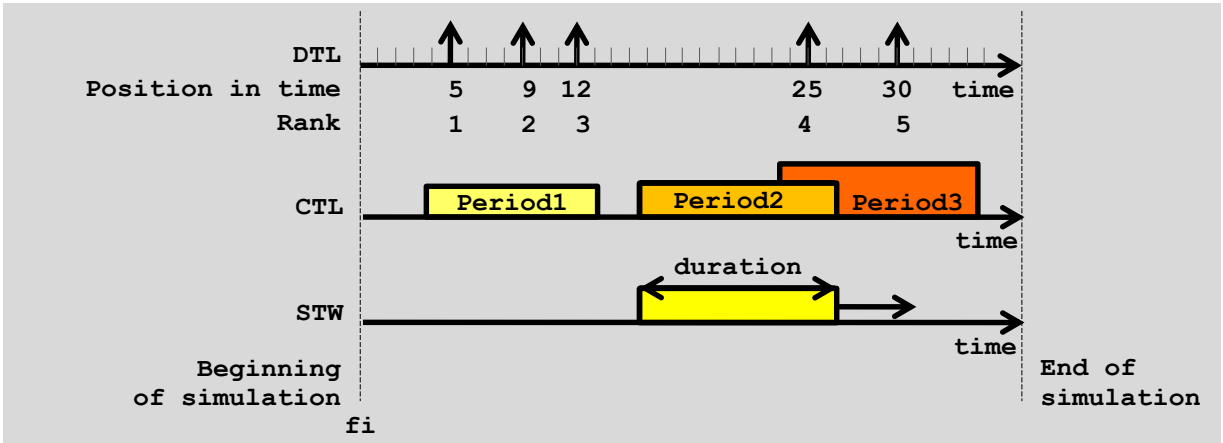


*Figure 3-1*
*Time Locators. The DTL is composed of 5 instants. The CTL is composed of three periods, the third one overlapping with the second one.*

### 3.2.6. WHERE: Sets

The WHERE part of a snippet specifies which items are concerned with the WHAT. In FORM-L, this is expressed either by the naming explicitly the items concerned, or by using **spatial locators** (see Section 12 *Spatial Locators*) dealing with vectors or sets possibly having yet unknown size or membership.

## 3.3. Variables & Types

### 3.3.1. Variables, Fixed Variables & Constants

Variables and events are the most basic FORM-L items. A **variable** is a pure time function: it has one and only one value all along physical time, and it has no other explicit argument. A variable is **fixed** if its value remains the same all along an operational case, but may be different for each case. A **constant**

is a variable the value of which does not change with time, and is the same for all operational cases[1].

### 3.3.2. Types

The **type** of a variable specifies the nature of the value it provides. FORM-L has four **primitive types** that do not need to be defined: **Boolean**, **Integer**, **Real** and **String**[2]. FORM-L also has **enumerated types** (or **enumerations**) that specify a list of possible values and must be explicitly defined. **Derived Real** types may be created to support quantities expressed in terms of particular units. These usually are physical quantities, but there could be non-physical quantities, such as monetary quantities.

Each type has a predefined and fixed set of **attributes** (see Section 3.12 *Attributes*) providing supplementary information (in addition to the *value* of the variable).

### 3.3.3. Booleans

The FORM-L Booleans are the "classical" booleans, taking only one of two values: *true* or *false*[3]. A Boolean can be viewed as a two state automaton, and very often, using one or the other is mainly a matter of modelling preference. The main difference is that Booleans can be subjected to the classical logical operations (not, and, or, ...).

When used to specify time locators, they are sometimes called **conditions**, or **Boolean conditions**.

### 3.3.4. Integers

The difference between the FORM-L Integers and the mathematical integers is that due to implementation constraints, FORM-L Integers might be bounded in value and might not represent arbitrarily large numbers[4]. The largest positive and negative numbers that can be represented are implementation dependent, but they should be sufficiently large that in practice, this is not a hindrance in the modelling of real-world systems.

### 3.3.5. Reals

FORM-L Reals are also an approximation of the mathematical real numbers. The details of the approximation are implementation specific, but should not be a hindrance in the modelling of real-world systems[5].

As FORM-L addresses the physical aspects of systems, a frequent use of Reals is to represent quantities. This is supported by attributes *dimension*, *unit*, *scale* and *offset* that are specific to Real types (see Section 3.12 *Attributes*). Reals the *dimension* of which is the empty string are called **Real numbers**. Reals the *dimension* of which is not the empty string are called **quantities**. Real expressions are subject to the customary dimensional rules. In particular:

- One cannot add or substract Reals with different normalised *dimension* equations[6].
- The product of a *length* by a *length* is a $length^2$, and the division of a *length* by a *duration* is a $length \times duration^{-1}$.

---

1 Yes, a constant variable is an oximoron.

2 In this document, capitalised Boolean, Integer, Real and String denote the FORM-L types. Non-capitalised boolean, integer, real and string denote the mathematical or general notions.

3 In FORM-L, a Boolean is NOT an Integer.

4 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. However, variable-bit-number representations could also be possible.

5 There are also multiple ways to represent Reals in a computer. The most common are IEEE-style representations. However, such representations 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 represented by two integer numbers).

6 A normalised dimension equation is expressed a product of base dimensions raised to integer powers (positive or negative), in a fixed order (e.g., in alphabetical order of the dimensions names).

- When adding a Real in *metres* to a Real in *centimetres*, a unit conversion is made so that the result is expressed in the units of the left hand side of the addition. Unit conversion is based on a formula automatically deduced from the *scale* and *offset* attributes.
- When a Real variable is specified a legitimate value (i.e., of the right dimension), a unit conversion is performed if necessary to match the unit specified for the variable.

As physical time is essential for the expression of time locators, a fundamental ingredient of FORM-L, derived type **Duration** needs to be defined. Also, the definitions of derived types must be consistent across all models in a configuration.

Type conversion from Real to Integer and the reverse can be made on request with predefined functions. However, the functions for all other type conversions (including from a quantity to a Real number) must be defined explicitly, preferably with library functions ensuring consistency with the dimension and unit system.

### 3.3.6. Strings

Operations on character strings are very limited in FORM-L: Strings are either constants or fixed variables. The character set is implementation dependent.

### 3.3.7. Enumerations & Automata

In FORM-L a **finite state automaton**, or **automaton** for short, is a variable of an enumerated type. The possible values specified by the enumeration are the discrete **states** that can be taken by the automaton. Like any other types, enumerations can be instantiated into scalars and vectors[1]. A vector of automata can be considered as the **Cartesian product** of its elements, in the order of the index.

The **state transitions** (or **transitions**, for short) for an automaton are specified by its definition. A transition from one or more states to a target state is made at the instants specified by a DTL. The definition can also specify a state for the periods of a CTL.

Following the **statechart** approach, a state can be a simple value, or can embed a scalar automaton, a vector of automata (the vector elements are instances of the same enumeration), or a Cartesian product of scalar automata (which are instances of possibly different enumerations)[2]. The embedded automata must be instances of other enumerations: there cannot be recursion. They may themselves have states that embed other automata. Thus, an enumeration or automaton can be viewed as a tree (see an example in Figure 3-2) where the root represents the top-level enumeration or automaton, the intermediate nodes represent the embedded automata, and the leaves represent simple values.

When an automaton does not contain any vector or product of embedded automata, then its state at a given instant is represented by a single path from the root of the tree to one single leaf, possibly through intermediate nodes. Such a state is an **atomic state**. When the automaton does contain at least one vector or product, then some of its states may be represented by a single leaf (which are thus atomic states), and others by multiple leaves. Such states are **composite states**.

```
enumeration B = (f, g, h);
enumeration D = (i, j);
enumeration K = (m, n, o);
enumeration L = (p, q);
enumeration E = (K k & L l);              // Cartesian product
enumeration A = (B b, c, D[3] d, E e);   // d is a vector, e a product
A a;   // declaration of automaton a as an instance of enumeration A
```

[1] Theoretically, they could also be instantiated into sets, but as operations on states are very limited, this is probably not very useful in practice.

[2] FORM-L does not allow for now Cartesian products where some of the factors are vectors. That might be considered in the future if necessary or useful.

In this artificial example, automaton *a* is an instance of complex enumeration *A*. Figure 3-2 shows the corresponding tree. Intermediate nodes are shown as rectangles, leaves are shown as circles. Hereafter are a few examples of states that automaton *a* can take:

- *b.f* // since there is only one leaf in the tree named *f*, this state could be simply denoted *f*
- *c*
- *d.(i & j & i)* // could be denoted *(i & j & i)*. In Figure 3-2, this composite state is shown in blue
- *e.(k.n & l.q)* // could be denoted *(n & q)*. In Figure 3-2, this composite state is shown in green

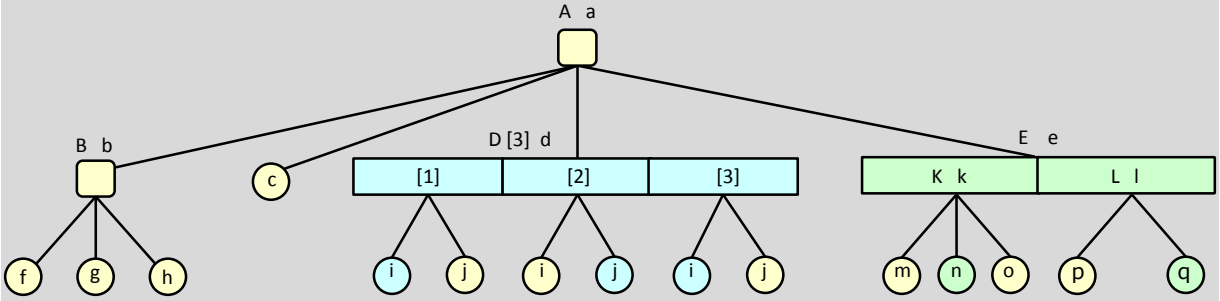


*Figure 3-2*
*Artificial example of complex automaton*

## 3.4. Functions

A FORM-L function calculates a variable from one or more parameters other than time. A parameter can be another variable, an event, an object or a property.

A number of predefined functions are provided through a standard library.

## 3.5. Events & Events Occurrences

An **event** states something of interest that can occur in physical time but has no duration. Each time that something occurs is an **occurrence** of the event. An occurrence can happen as the result of an expression (e.g., *when* A > 10.0 *becomes true*) or of a definition.

The occurrences of a **naked event** carry no other information than their time of occurrence. The occurrences of a **rich event** carry other information in the form of variables, as specified by the definition of the event.

## 3.6. Objects & Classes

A real-world entity is usually more complex than a simple value or event, and is characterised by multiple **features**. It is represented in FORM-L by an **object**, the features of which can be variables, events, time locators (see Section 3.2.5 *WHEN: Time Locators)*, properties (see Section 3.9 *Properties*), embedded objects or associations to other objects (see Section 0 *An* object is characterised by its features. A feature can be a variable, an event (naked or rich), an embedded object (i.e., an object that is an integral part the object being characterised and that has no independent existence), an associated object (i.e., an object that exists independently of the object being characterised) or a property.

Associations). When multiple objects have the same set of features and represent the same sort of real-world objects, it is often convenient to define a **class** that will bear all the similarities. These objects are then **instances** of the class. One-of-a-kind objects can be declared directly without having to go through a class.

## 3.7. Scalars, Vectors & Sets

### 3.7.1. Scalars

The notion of **scalar** in FORM-L is opposed to the notions of **vector** and **set**. Term *Real number* is used to designate a Real with no dimension. Term *quantity* is used to designate a Real with a dimension.

### 3.7.2. Vectors

The size of a vector is usually a constant or a fixed integer. However, some vectors may be of variable, increasing size: new elements may be appended to the vector during a case. Example: the occurrences of an event are represented by a vector of increasing size. The actual size of a vector is given by its *size* attribute. Indexing begins at 1.

### 3.7.3. Sets

When requirements and assumptions are stated at a very early stage of a project, the precise identification and number of objects constituting the system and its environment are often unknown. **Sets** are an essential ingredient of FORM-L, as they enable reasoning and the expression of properties on items with particular roles or characteristics but that are not known individually yet.

A FORM-L set is a finite collection of discrete items of the system under study (e.g., variables or objects), of its environment, or of the property model itself (e.g., properties). Thus, a continuous range of Reals is not a FORM-L set. An item may belong to several sets, but a set contains a given item only once. Also, a set is NOT a vector. In particular, a vector element cannot be an element of another vector, or cannot be an independent scalar. Also, whereas a vector has an inherent internal order, sets have no notion of internal order. However, a vector can always be viewed as a set.

If the precise membership of a set is known at the time of the writing of the model, then it can be defined *in extension* by listing all its members individually. Otherwise, it must either be defined *in intension* (stating the criteria that identify the members) or be declared as *external* to the model (in which case its membership needs to be defined by another model or by a query to an engineering database through an appropriate binding, see [12]).

FORM-L supports the classical operations on sets: union, intersection, substraction, Cartesian product, extraction of a subset, selection of a single member, application of an operation or calculation to each set members, calculations of Boolean, Integer or Real scalars based on all set members, calculation of cardinals, application of existential and universal quantifiers.

### 3.7.4. Identities

The notion of set implies the notion of **identity**, which is a fixed attribute the value of which is unique within a configuration. Identities can only be compared for equality or difference.

The nature of the identity attribute is implementation dependent, but here, it is assumed to be a String. It could be based on:

- The identity of an external item obtained from a behavioural model is determined by its binding. In a configuration, if the same information is used by different property models, this must be made explicit through the binding, so that the same identity is used.
- If an item is defined by a property model, then that model determines its identity, e.g., based on the full pathname of the item. The other property models declaring that item as external use that identity.
- The identity of a vector element combines the identity of the vector and the index of the element.
- The identity of an item attribute or feature combines the identity of the item and the name of the attribute or feature.
- The identity of an expression combines the formula of the expression and the identities of the arguments of the expression.
- The identity of a literal or literal expression is based on its value.

### 3.7.5. T-uples

To support the Cartesian product of sets, FORM-L has the notion of **t-uples**. If *t* is a t-uple, then *t(i)* designates the $i^{th}$ member of *t*.

## 3.8. Categories

A **category** is an expression that precisely specifies the nature (type, class or event class) and the arity of an item.

The rules concerning categories are as follows:

- Two categories are identical if and only if they both specify the same nature and the same arity.
- Category *A* is compatible with category *B* if and only if both its nature and its arity are compatible with those of *B*.
- Type *A* is compatible with type *B* if and only if *A* is the same as *B*, or is derived from *B*.
- Class *A* is compatible with class *B* if and only if *A* is the same as *B*, or is an extension of *B*.
- Event class *A* is compatible with event class *B* if and only if *A* is the same as *B*, or is an extension of *B*.
- Arity *A* is compatible with arity *B* if and only if *A* is the same as *B*, or if *B* is a vector of undetermined size.

## 3.9. Properties

A **property** expresses constraints called **checks**. Properties can be divided into two groups: **probabilistic properties**, and **non-probabilistic properties**. A non-probabilistic property specifies constraints that can be checked in the framework of individual operational cases: for each case, one can conclude whether the property has been tested, and if so, whether it has been violated or satisfied. A probabilistic property specifies a constraint on the probability that a non-probabilistic property is violated, and can be verified only by analysing the outcome of a sufficient number of cases. Thus, probabilistic properties need to be treated separately from non-probabilistic properties.

### 3.9.1. HOW WELL: Probabilistic Properties

A probabilistic priperty specifies HOW WELL a desirable property must be satisfied (given the fact that real-world systems are bound to have failures), in the form of a limit on the probability that the desirable property is violated.

### 3.9.2. Requirements, Assumptions & Guards

A **requirement** is a property that expresses something that is required, usually of the system under study. An **assumption** is a property that expresses something that is supposed to be satisfied, usually by the environment or the components of the system and by operational cases. A **guard** is a property that specifies when a model is valid: when the guard is violated, a replacement model the guard of which is satisfied should be used.

Note. A given property may be viewed as an assumption in one model, and as a requirement in another model. For example, during the verification of an overall design, design decisions are first expressed as assumptions made on components. Verification then aims at ensuring that these assumptions will satisfy the specified system requirements. Once this is done, the assumptions can then be considered as requirements for the components. This change is expressed in FORM-L by contracts, in the framework of a configuration.

## 3.10. Coordinated Snippets

### 3.10.1.Coordination Mechanisms

Sometimes, multiple definitions need to be coordinated in time. This could be done by specifying adequate time locators for each. However, to highlight the nature of the coordination, to simplify the expression of coordinated definitions, and to facilitate understanding, FORM-L offers five **coordination mechanisms** (see Section 14 *Coordination*):

- **Sequence**: the definitions are placed one after the other in time.
- **Concurrence**: the definitions are placed at the same instant in time.

- **Iteration**: the definition(s) are repeated sequentially in time while a condition holds.
- **Selection**: the definition(s) are selected depending on one or more conditions.
- **Procedure**: the definitions are organised as a parameterised template that can be instantiated multiple times. A parameter can be a variable, an event, an object or a property.

### 3.10.2.Relative Time Locators

In the framework of a coordination, to precisely place a definition in time, one uses **relative time locators** rather than the **absolute time locators** introduced in Section 3.2.5 *WHEN: Time Locators*. A relative time locator may be used to specify a **delay**, with respect to the preceding definition in the coordination or to the beginning of the coordination. When defining a variable or a property, a relative time locator may be used to specify a **time span**.

## 3.11. Time Domains

Perception of time and durations may be different in different (sub)systems. For example, digital control systems are sometimes designed to be synchronous: they see their environment and act upon it only at specific instants. Some are even composed of multiple locally synchronous, but globally asynchronous, subsystems. Such (sub)systems interact with a physical environment that generally operates in continuous time. Thus, a property model may be composed of multiple **time domains** (see Figure 3-3).

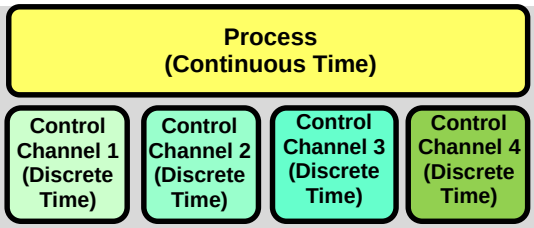


*Figure 3-3*
*Example of multiple time domains. The digital control system is constituted of 4 redundant, independent, and individually synchronous control channels. Each box is a separate time domain*

### 3.11.1.Continuous & Discrete Time Domains

In FORM-L, there is one and only one **continuous time domain**, and it does not need to be declared. One can declare and define multiple **discrete time domains**. An object belongs to one and only one time domain. By default, it belongs to the continuous time domain, unless its definition places it explicitly in a specific discrete time domain.

A discrete time domain has an associated **clock** that specifies, using a DTL, the instants in real physical time when the variables and statements attached to the discrete time domain are evaluated.

The clock is usually periodic with a fixed phase (generally random) with respect to the beginning of time. More complex clocks could be used, for example to introduce jitter and / or drift in an otherwise periodic clock.

Figure 3-4 illustrates the effects of discrete time domains. It considers a control system with two identical, redundant, and individually synchronous channels. As these channels are independent, they have different, random 'phases', and may perceive a process signal in continuous time differently.

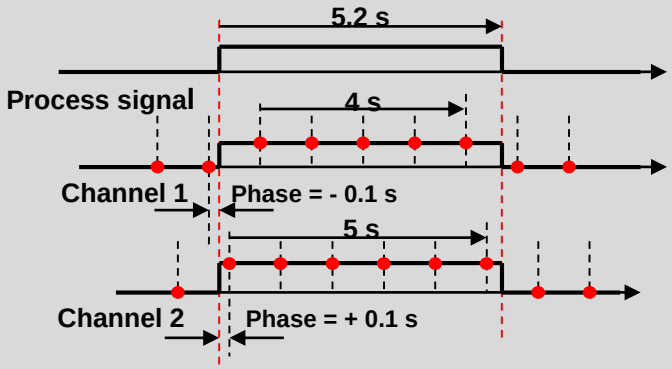


*Figure 3-4*
*Impact of independent time domains. The process signal is in the continuous time domain, whereas the two channels are in separate discrete time domains, both having the same period but different phases.*

### 3.11.2.Time Domain Interfaces

The events issued by objects not belonging to a discrete time domain cannot be perceived directly by the objects of that domain: they must go through **domain interfaces** that perform the necessary timing adaptations.

A domain interface 'sees' the other time domain(s) and, based on this perception, generate conditions and values perceptible at the time positions specified by the DTL of the discrete time domain. Various domain interface schemes are possible

Example 1: Holding an event occurrence for the duration of one period of the discrete time domain. If the objects of that domain do not react to the occurrence at the next instant, the occurrence is lost.

Example 2: Holding an event occurrence until the next occurrence of that event. If the objects of that domain do not react to the first occurrence before the next one, that first occurrence is lost.

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

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

### 3.11.3.Time Periods Boundaries

In the continuous time domain, whether the boundaries of a time period belong to it or not is usually irrelevant. In a discrete time domain, this could be significant, and FORM-L offers ways to precisely specify it.

## 3.12. Attributes

A FORM-L item has a fixed set of attributes that depends on its category:

- **identity** is a fixed character string that determines the identity of an item. It is automatically calculated and cannot be altered by a model.
- **start** is a fixed value attached to a type or a variable. It specifies a default initial value for variables for which no such value is given by the model. The value attached to a variable overrides the attribute value attached to its type.
- **value** is attached to a variable and specifies its current value. This name is used only in variable definitions.
- **dimension** is a constant character string attached to Real and its derived types that gives the normalised dimension equation of the type. For Real numbers, it is an empty string. An instance cannot override the attribute value specified by its type.

- **unit** is a character string attached to Real, its derived types and their instances. It specifies the unit in which the value is expressed (e.g., `"ms"` for a Duration expressed in milliseconds). By default, it is an empty string. The value attached to an instance overrides the value attached to its type and may be modified in time. However, this should be done with great care, in general in consistency with attributes *value*, *scale* and *offset*.
- **scale** is a number attached to Real, its derived types and their instances. It specifies a ratio with respect to the corresponding SI unit (e.g., 0.001 for a Duration expressed in milliseconds). By default, the value is 1. The value attached to an instance overrides the value attached to its type and may be modified in time. However, this should be done with great care, in general in consistency with attributes *value* and *unit*.
- **offset** is a number attached to Real, its derived types and their instances. It specifies a value expressed in the corresponding SI unit and to be added after the *scale* has been applied. By default, the value is 0. In practice, a non-zero offset is used only for temperatures. The value attached to an instance overrides the value attached to its type and may be modified in time. However, this should be done with great care, in general in consistency with attributes *value*, *unit* and *scale*.
- **memory** is a constant true-false value attached to an enumeration or to an automaton. It is meaningful only for a sub-automaton (i.e., an automaton placed inside a state of another automaton). It specifies whether the sub-automaton has memory or not (see Section 4.9 *Enumerations, States & Automata*). The value attached to an automaton overrides the value attached to its enumeration.
- **states** is a constant set of states attached to an enumeration. It contains all the states of that enumeration. It is automatically calculated and cannot be altered by a model. It is mainly intended for the specification of test coverage criteria.
- **transitions** is a constant set of state pairs attached to an automaton. It contains all the state transitions defined for that automaton. It is automatically calculated and cannot be directly altered by a model. It is mainly intended for the specification of test coverage criteria.
- **instantState** is an automaton attached to a check. Its states are **notTestedNow**, **satisfiedNow** and **violatedNow**. It specifies whether, at the current instant, the check is tested, and if so, whether it is satisfied or violated. As for all automata, it is associated with three true-false attributes bearing the names of the states. The automaton is automatically calculated and cannot be directly altered by a model.
- **cumulState** is also an automaton attached to a check. Its states are **notTested**, **satisfied** and **violated**. It specifies whether the check has been tested, and whether when tested it has always been satisfied or has been violated at least once. As for all automata, it is associated with three true-false attributes bearing the names of the states. The automaton is automatically calculated and cannot be directly altered by a model.
- **size** is a whole number attached to a vector, giving the size of the vector. It is automatically calculated and cannot be directly altered by a model.
- **index** is a domain of whole numbers attached to a vector, containing the possible indexes for the vector. It is automatically calculated and cannot be directly altered by a model.
- **count** is a whole number attached to an event. It states the number of occurrences of that event up to the current instant. It is automatically calculated and cannot be directly altered by a model.
- **time** is a dynamically sized vector attached to an event. Its $i^{th}$ element provides the time position of the $i^{th}$ occurrence of the event. It is automatically calculated and cannot be directly altered by a model.
- **clock** is a dynamically sized vector attached to an event. It is meaningful only in a discrete time domain. Its $i^{th}$ element is the position of the $i^{th}$ occurrence of the event, in terms of tick number of the time domain clock. It is automatically calculated and cannot be directly altered by a model.

Note. One can observe that the names of the primitive types (Boolean, Integer, Real, String) are not

used in the description of attributes, to highlight the fact that an attribute does not have attributes.

# 4. Variables & Types

## 4.1. References to a Type

The syntactic expression type specifies either the name of a primitive type, or a pathname to a defined enumeration or derived Real:

```
type =
   Boolean
| Integer
| Real
| String
| path(enumeration)
| path(derivedReal)
;
```

Note. A *path* is a sequence of dot-separated names and possibly of indexes or t-uple identifications that can be used to designate an item unambiguously.

## 4.2. Type Definitions

Only enumerations and derived types need to be defined:

```
typeDef = derivedRealDef | enumerationDef;
```

### 4.2.1. Derived Real

The definition of a derived Real specifies values for particular attributes.

```
derivedRealDef = (
   type name
     dimension = string(const);
     unit = string(const);
     [scale = real(const);]                    // Default: 1.0
     [offset = real(const);]                   // Default: 0.0
     [start = (real(fixed) | specific);]       // Default: specific
```

This defines a type directly derived from Real. Attributes *dimension* and *unit* must each be assigned a constant value. This style of definition is mainly intended for the base quantities and units of the SI system of units, for which attributes *scale* and *offset* have their default values (respectively 1.0 and 0.0). Sub-units (e.g., milli or kilo) should be defined via constants, as shown in the following example. Assigning a non-default value to *offset* is mainly intended for temperatures. Attribute *start* may be assigned a fixed value (i.e., a constant or a value that is determined for each operational case). It may also be defined using keyword *specific*: each instance of the type must then specify its own initial value. The order of attribute definition needs not be the one shown here.

```
type Duration
   dimension = "time";
   unit = "s";
   end Duration;

constant Duration s = 1.0;      // Allows notations such as 60*s

constant Duration ms            // Allows notations such as 50*ms
   unit = "ms";
   scale = 0.001;
   value = 1.0;
   end ms;

type Temperature
```

```
  dimension = "temperature";
  unit = "K";
  end Temperature;

constant Temperature K = 1.0;  // Allows notations such as 300*K

constant Temperature C         // Allows notations such as 20*C
  unit = "C";
  value = 1.0;
  offset = 273.16;             // TK = TC + offset
  end C;

constant Temperature F         // Allows notations such as 20*F
  unit = "F";
  value = 1.0;
  scale = 5./9.;
  offset = 273.16-32.*5./9.;   // TK = (TF * scale) + offset
  end F;
```

```
|  type name
     dimension = dimensionExpr;
     [unit = string(const);]                    // Default: deducted
     [scale = real(const);]                     // Default: deducted
     [offset = real(const);]                    // Default: 0.0
     [start = (real(fixed) | specific);]        // Default: specific
```

This defines a derived Real type using a dimension expression based on already defined derived Real types, with multiplications, divisions and non-zero constant integer powers. If not assigned a value, attribute *unit* is calculated by replacing each quantity in the normalised dimension equation by its *unit* attribute. If not assigned a value, attribute *scale* takes a value calculated by replacing each type in the type expression by their *scale* attribute. In principle, non-zero *offset* attributes are used only for non SI temperatures (e.g., Fahrenheit) that should not participate in dimension expressions. The order of attribute definition needs not be the one shown here.

This style of definition should be used for quantities derived from the base quantities of the SI system of units.

```
)    end name;
;

dimensionExpr =
   path(realType)
|  (dimensionExpr)
|  dimensionExpr1 (* | /) dimensionExpr2
|  dimensionExpr ^ integerLiteral    // The literal must be different from 0
;
```

```
type Mass
  dimension = "mass";
  unit = "kg";
  end Mass;

constant Mass kg = 1.0;

type Length
  dimension = "length";
  unit = "m";
  end Length;

constant Length m = 1.0;

type Force
  dimension = (Mass*Length) / (Duration^2);
  unit = "N";
  end Force;
```

```
constant Force N = 1.0;
```

## 4.2.2. Enumerations (and Automata)

### State Literals

When expressing properties, one must be able to name a specific state in the tree representing a complex enumeration (or automaton) (see Section 3.3.7 *Enumerations & Automata*):

- For an atomic state, only one leaf is implied. If no other leaf bears the same name as the desired one, then in the name scope of the enumeration (or automaton) definition, the name of the leaf is sufficient to unambiguously identify the desired state. If other leaves bear the name, then an unambiguous path (using a dot notation) must be used to remove the ambiguity.
- For a composite state, multiple leaves are implied, and one must use a parenthesised, &-separated notation, with one item per vector or product element.
- Outside of the name scope of the definition, if there is no ambiguity, then one can use the same notation as within the name scope. Otherwise, one must prefix it with a pathname for the enumeration or automaton, using a dot notation.

The syntactic expression for state literals is given in Section 4.9.1 *State Literals*.

### Attribute *start*

The default initial state of an automaton can be explicitly specified by its definition with the *start* attribute. If not, it is specified by the *start* attribute of its enumeration. The initial state for an embedded automaton is determined as follows, in decreasing order of priority:

- The *start* state explicitly specified by the definition of the embedding automaton.
- The *start* state explicitly specified by the definition of the embedding enumeration.
- The *start* state specified by the embedded enumeration.

The default initial state can be specified by a fixed state expression: this could be one particular state of the enumeration, or a randomly chosen state (see Section 0

*Primary State* Expressions). Keyword *specific* indicates that the enumeration has no default initial state and that each instance must specify its own. This is what is assumed if *start* is left undefined.

```
startState = start = (state(fixed) | specific);
```

There are two styles for specifying the start state for vectors of automata. With the global style, all the vector elements have the same start state. The itemised style is based on multiple snippets: each specifies a start state, and an index domain `ints` specifying which vector elements have that start state. If one uses a random formula for the start state, all vector elements denoted by the snippet will have the same start state. If one wishes to use the same random formula but different start states, then one must specify different snippets. Like all such forms of definition, all the vector indexes must be covered once and only once.

```
startStates =
  startState                                          // Global style
| {start[ints] = (state(fixed) | specific);}+         // Itemised style
;
```

### Attribute *memory*

Attribute *memory* is significant only when an instance of the enumeration is embedded in a state of another enumeration (or automaton), either directly or through a product.

When a transition to the embedding state does not specify a state for the embedded automaton:

- When the embedding state is entered for the first time, the state used for the embedded automaton is the one specified by the applicable *start* attribute.

- When it is not the first time and the embedded automaton has no memory, its state is again the one specified by the applicable *start* attribute. If it has memory, its state is the one it had at the last exit from the embedding state.

The effective value for an automaton is determined as follows, in decreasing order of priority:

- The value explicitly specified by the definition of the embedding automaton.
- The value explicitly specified by the definition of the embedding enumeration.
- The value specified by the embedded enumeration.

There are two possible ways to specify whether an enumeration has memory by default:

- This can be specified by a fixed Boolean expression. *False* is assumed if *memory* is left undefined.
- Keyword *specific* requires that each embedded instance of the enumeration specifies explicitly whether it has memory or not.

```
memory = memory = (bool(fixed) | specific);

memories =
   memory                                                    // Style 1
|  {memory[ints] = (bool(fixed) | specific);}+               // Style 2
;
```

### Composite States

A state in an enumeration or an automation is either atomic or composite. An atomic state is specified simply by its name. A composite state is specified by its name, by the name of another enumeration, and possibly by attributes *start* and *memory*. When the composite state is the initial state, then the *start* attribute specifies which state of embedded enumeration is the initial state.

```
compositeStateDef =
   name(state) = path(enumeration)                           // Case 1, scalar
      [  (startState; memory)
      |  (startState)
      |  (memory)
      ]
|  name(state) = path(enumeration)[size]                     // Case 2, vector
      [  (startStates; memories)
      |  (startStates)
      |  (memories)
      ]
;
```

### Enumeration Definition

An enumeration can be either a of list states or a Cartesian product of two or more scalar automata. In the first case, a state in the list can either be an atomic state or embed a scalar or vector automaton. In both cases, the definition can optionally specify a value for attributes *start* and *memory*.

```
enumerationDef =
   enumeration name = ({name_i(state),}2+);                  // Style 1
```

With this style, the enumeration is defined simply as a list of atomic states. The names of the states must be different from one another, and there must be at least two states. Attributes *start* and *memory* have their default value.

```
enumeration SystemState = (normal, maintenance, test);
```

There are three possible values for any instance of *SystemState*: *normal*, *maintenance* and *test*.

| ***enumeration*** ***name*** **=** **(**{***name***$_i$**(state)**,}2+**)** **// Style 2**
  **[startState;]**
  **[memory;]**
  ***end*** ***name***;

With this style, the enumeration is also defined as a list of atomic states, but attributes *start* and / or *memory* can now be given a value.

```
enumeration SystemState2 = (normal, maintenance, test)
   start = normal;
   end SystemState2;
```

This refined example specifies that the default initial state for instances of *SystemState2* is *normal*.

| ***enumeration*** ***name*** **=** **(**{ ***name***$_i$**(state)** | **compositeStateDef**,}2+**);** **// Style 3**

This style is similar to style 1, but some states may be composite.

```
enumeration Degraded = (slow, rebound);
enumeration Failed = (stuckOpen, stuckClose, stuckAsIs);
enumeration OpState =
   (nominal, degraded = Degraded (memory = true), failed = Failed);
```

This example illustrates two ways for declaring an embedded automaton. State *nominal* is an atomic state and does not embed an automaton. State *degraded* is specified by a pathname to a separately defined enumeration (named *Degraded*). As degradation could be intermittent, *Degraded* is specified to have memory. State *failed* is specified by a pathname to another enumeration (named *Failed*).

| ***enumeration*** ***name*** **=** **(**{ ***name***$_i$**(state)** | **compositeStateDef**,}2+**)** **// Style 4**
  **[defaultStartState;]**
  **[defaultMemory;]**
  ***end*** ***name***;

This style combines styles 2 and 3.

```
enumeration Degraded = (slow, rebound);
enumeration Failed = (stuckOpen, stuckClose, stuckAsIs);
enumeration OpState2 =
   (nominal, degraded = Degraded (memory = true), failed = Failed)
   start = nominal;
   end OpState2;
```

This example is a variation to the previous one, where a default initial state is specified for *OpState2*.

| ***enumeration*** ***name*** **=** **(**{ **compositeStateDef&**}2+**);** **// Style 5**

With this style, the enumeration is defined simply as a Cartesian product of two or more composite states.

| ***enumeration*** ***name*** **=** **(**{ **compositeStateDef&**}2+**);** **// Style 6**
  **[defaultStartState;]**
  **[defaultMemory;]**
  ***end*** ***name***;

This style combines styles 2 and 5.

```
enumeration Sequencing = (standby, active, finished);
enumeration Mode = (normal, test);
enumeration Operational = (sequencing = Sequencing & mode = Mode);
enumeration State = (operational = Operational, maintenance, failure}
   start = operational.(standby & normal);
   end State;
```

Enumeration *State* identifies the main states of a system, which can be:

- *operational*, which is the normal state where the system is ready to perform, or is performing, its mission.
- *maintenance* is the state taken when the system is under maintenance.

- *failure* is the state taken 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*): the instructions to be performed by the system depend on it. *Sequencing* indicates the current phase of system operation:

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

;

## 4.3. Variables

A variable is characterised by:

- Its name.
- Its type.
- An optional parameter list.
- Its **variability**: it can be a constant, a fixed value or a variable.
- Its **arity**: it can be a **scalar**, a **vector** (of a fixed or variable size) or a **set**.
- Its value.
- Its other attributes values.

## 4.4. Value Domains

**Value domains** have been introduced for convenience, to facilitate models understanding. They specify a possibly infinite collection of values for a given type.

A Boolean, state or String domain is composed of a finite number of individual values. Thus, it is indistinguishable from a set. An Integer or Real domain is a collection of individual values and intervals.[1] An interval can be upwardly and / or downwardly unbounded, and a Real interval can include or not its boundaries. The values and the boundaries in a domain may be constants or fixed values.

## 4.5. Primary Expressions

**Primary expressions** are the basic way to specify values for variables. They do not directly specify time locators: that is the role of **snippets** and **properties**. The result of a primary expression can be a scalar, a vector or a set. It can also be a constant, a fixed, a variable or an **in-period** value. An in-period value is evaluated for each time period of a CTL or an STW, and depends on time within the period. A variable is a particular case of in-period value, where the period is the total time.

In the following, a general description for primary expressions is given for each primitive type. When an expression is based only on constants (resp. fixed values), then its result is a constant (resp. a fixed value). When it includes one or more in-period values, then its result is an in-period value.

Primary expressions must not exhibit circular, recursive dependencies.

Access to the value of a primary expression at a particular instant in the past can be obtained in the following ways:

[1] 2D, 3D and more generally n-D domains might be added in the future.

**expr *at* [*now* -] duration**
Value of expr at *t0* + duration or now - duration. That instant must be neither in the future of the current instant nor before *t0*.

**expr *at* [*now* -] clockDuration**
In a discrete time domain, value of expr at *t0* + clockDuration or now - clockDuration. That instant must be neither in the future of the current instant nor before *t0*.

***previous*(expr)**
Value of expr just before *now*.

## 4.6. Booleans

As they are very widely used, to enhance legibility, FORM-L offers a variety of means and syntactic shortcuts for Boolean expressions.

### 4.6.1. Primary Boolean Expressions

**boolTerm =**

**(bool)**

**| (*true* | *false*)**
Boolean literals.

**| *path*(Boolean)**
Pathname to a declared or defined instance of Boolean.

**;**

**bool =**

**boolTerm**

**| ctl**
True at instants belonging to the CTL, false otherwise.

**| functionCall(Boolean)**
Call of a function returning a Boolean.

**| vector(Boolean) [int]**
Indexation in a vector of Booleans.

**| *path*(tuple) (int(const))**
Extraction of the int(const)th element of a t-uple, where that element is a Boolean. (T-uples result from the Cartesian product of two or more sets.)

**| *not* bool**
**| boolTerm (*and* | *or* | *xor* | *excl* | =) bool**
Classical Boolean operations.

**| $int_1$ (< | ≤ | = | ≠ | ≥ | >) $int_2$**
**| $real_1$ (< | ≤ | = | ≠ | ≥ | >) $real_2$**
**| $fsa_1$ (= | ≠) $fsa_2$**
**| $string_1$ (= | ≠) $string_2$**
Reals must have the same dimension, and the comparison takes the scales and offsets into account.

**| *if* $bool_1$ *then* $bool_2$ {*elseIf* $bool_i$ *then* $bool_j$;}* *else* $bool_3$**
Choice of one Boolean among several, based on successive Boolean tests.

**| *switch* scalar(x) {*case* $domain_i$(x) *then* $bool_i$;}+ *else* $bool_3$**
Choice of one Boolean among several, based on successive membership tests on scalar(x).

**;**

### 4.6.2. Predefined Booleans and Boolean Functions

***path*(property).**
**(*notTested*|*satisfied*|*violated*|*notTestedNow*|*satisfiedNow*|*violatedNow*)]**
One of the six Boolean attributes associated with a property.

***path*(property).**
**(*provisionallyOK*|*provisionallyNotOK*|*definitivelyOK*|*definitivelyNotOK*)**
One of the four in-period Boolean attributes associated with a property. In-period expressions are valid only under the scope of a CTL or an STW.

***path*(fsa).state**
***path*(fsa).states**
These expressions are true when and only when the automaton is in the specified state, or in one of the specified states. They are shortcuts for fsa = state, or fsa *isIn* states.

**state**
**states**
These expressions are valid only when there is one and only one automaton in the model that can have the specified state or states. They are true when and only when that automaton is in the specified state, or in one of the specified states.

**bool *isIn* bools**
**int *isIn* ints**
**real *isIn* reals**
**fsa *isIn* states**
**string *isIn* strings**
**x *isIn* set(y)**
Test of membership to a domain or a set. x must be consistent with y, the category of the set.

**$set_1$(x) (*isSubset* | *isStrictSubset*) $set_2$(y)**
test of set inclusion, where category x is compatible with category y.

### 4.6.3. Boolean Domains

A Boolean domain is a set of Booleans:

**bools = set(Boolean);**

## 4.7. Integer

### 4.7.1. Primary Integer Expressions

**int =**

**(int)**

**| *integerLiteral***
Integer literal.

**| *path*(Integer)**
Pathname for a declared or defined instance of Integer.

**| functionCall(Integer)**
Call of a function returning an Integer.

**| vector(Integer) [int]**
Indexation in a vector of Integers.

**| *path*(tuple) (int(const))**
Extraction of the int(const)th element of a t-uple, where that element is an Integer. (T-uples result from the Cartesian product of two or more sets.)

**| $int_1$ (+ | - | * | / | *modulo* | ^) $int_2$**

| **- int**
Classical Integer operations.

| ***if*** **bool**$_1$ ***then*** **int**$_1$ **{*elseIf*** **bool**$_i$ ***then*** **int**$_i$**;}* *else*** **int**$_2$
Choice of one Integer value among several, based on successive Boolean tests.

| ***switch*** **scalar(x)** **{*case*** **domain**$_i$**(x)** ***then*** **int**$_i$**;}+ *else*** **int**$_3$
Choice of one Integer among several, based on successive membership tests on scalar(x).

;

### 4.7.2. Predefined Integers and Integer Functions

***Integer*** **(real)**
Conversion of real into an Integer. The decimal part of real, if any, is truncated. Rounding must be explicit or through a library function.

***cardinal*** **set**
Number of members of a set, which could be a calculated set.

***abs*** **(int)**
Absolute value of int.

***min*** **(int)**
***max*** **(int)**
Min / max of int up to current instant.

***inPMin*** **(int)**
***inPMax*** **(int)**
Min / max of int in the period up to current instant. In-period expressions are valid only under the scope of a CTL or an STW.

***count*** **(event)**
Number of occurrences of an event up to current instant.

***inPCount*** **(event)**
Number of occurrences of an event in the period up to current instant. In-period expressions are valid only under the scope of a CTL or an STW.

***clockDuration*** **(bool)**
In a discrete time domain, number of clock ticks up to current instant where bool has been true.

***inPClockDuration*** **(bool)**
In a discrete time domain, number of clock ticks in the period up to current instant where bool has been true. In-period expressions are valid only under the scope of a CTL or an STW.

### 4.7.3. Integer Domains

As [ ] are already used for indexing, intervals are denoted with a [[x , y]] notation. An Integer interval has the following form:

**intInterval = (-%|int**$_1$**) .. (int**$_2$**|+%);**

Boundaries are always included. The interval is empty if **int**$_2$ strictly less than **int**$_1$. If **int**$_2$ is equal to **int**$_1$, then the interval has just one value.

An Integer domain has the following form:

**ints =**
**int**
An Integer domain can be a single Integer.

| **intInterval**
An Integer domain can be a single Integer interval.

| **{(int | intInterval),}*}**
It can be a (possibly empty) set of Integers and Integer intervals.

| **set(Integer)**
It can also be a set of Integers (see Section 4.12 *Sets*).

;

## 4.8. Reals

### 4.8.1. Primary Real Expressions

The dimension attribute of the Reals involved in an expression follow the normal rules for quantities.

**real =**

**(real)**

| ***realLiteral***
Real literal.

| ***path*(real)**
Pathname for a declared or defined instance of Real.

| **functionCall(Real)**
Call of a function returning a Real.

| **vector(Real) [int]**
Indexation in a vector of Reals.

| ***path*(tuple) (int(const))**
Extraction of the int(const)th element of a t-uple, where that element is a Real. (T-uples result from the Cartesian product of two or more sets.)

| **$real_1$ (+ | - | * | / | ^) $real_2$**
| **real ^ int**
| **- real**
Classical Real operations. For binary + and -, the arguments have the same quantity. For ^, $real_1$ and $real_2$ must have no dimension.

| ***if* $bool_1$ *then* $real_1$ {*elseIf* $bool_i$ *then* $real_i$;}* *else* $real_2$**
Choice of one Real among several, based on successive Boolean tests.

| ***switch* scalar(x) {*case* $domain_i$(x) *then* $real_i$;}+ *else* $real_3$**
Choice of one Real among several, based on successive membership tests on scalar(x).

;

### 4.8.2. Predefined Reals and real Functions

***Real* (int)**
Conversion of Integer int number into a primitive Real.

***time***
Real time duration since the beginning of the operational case. The type is *Duration*.

***inPTime***
Real time duration since the beginning of the period. The type is *Duration*. In-period expressions are valid only under the scope of a CTL or an STW.

***duration* (bool)**
Real time duration where bool has been true since the beginning of the operational case. The type is *Duration*.

***inPDuration*** **(bool)**
Real time duration where bool has been true in the period. The type is *Duration*. In-period expressions are valid only under the scope of a CTL or an STW.

***abs*** **(real)**
Absolute value of real.

***min*** **(real)**
***max*** **(real)**
Min / max of real up to current instant.

***inPMin*** **(real)**
***inPMax*** **(real)**
Min / max of real in the period up to current instant. In-period expressions are valid only under the scope of a CTL.

***derivative*** **(real)**
***integral*** **(real)**
In the continuous time domain, derivative / integral of real. The dimension is the dimension of real divided / multiplied by time, in seconds.

**random (real);**
Selection of a random Real between 0.0 and the argument value, with equal probability weights. More complex random Real functions will be defined later.

**conversion (real$_1$, real$_2$);**
Conversion of the first argument to the unit of the second argument. Both arguments must be of the same dimension. The conversion is based on the scale and offset attributes: the value of the second argument is not taken into consideration.

**log (real);**
**exp (real);**
**exp10 (real);**
**sin (real(angle));**
**cos (real(angle));**
**tan (real(angle));**
**arcSin (real);**
**arcCos (real);**
**arcTan (real);**
Classical mathematical functions.

### 4.8.3. Real Domains

A Real interval has the following form:

**realInterval = ([|])(-%|real$_1$), (real$_2$|+%)([|]);**

The conventions are the same as for Integer intervals.

A Real domain has the following form:

**reals =**
**real**
A Real domain can be a single Real.

**| realInterval**
A Real domain can be a single Real interval.

**| {(real | realInterval),}***
It can be a possibly empty set of Reals and Real intervals.

**| set(Real)**
It can also be a set of Reals (see Section 4.12 *Sets*).

**;**

## 4.9. Enumerations, States & Automata

There are three types of primary expressions for states and automata:

- A primary state expression determines a single, well-defined state in a particular enumeration or automaton. That state can be a constant, a fixed state or a variable state.
- A primary target state expression determines the target of a state transition. Such a target can be specified by a normal primary state expression, but for an enumeration or automaton that contains one or more vectors or Cartesian products, it can specify that some elements of a vector or some members of a product remain in their current state, and that some others get into the *start* state.
- An automaton expression designates an automaton that is always active (i.e., an automaton that is in a well-defined defined state at all times). This is the case for a top-level automaton, i.e., an automaton not embedded in another one. On the contrary, an automaton embedded in a state of another automaton is active only when the parent automaton is in that state. If an always active automaton is a vector or a Cartesian product, then each element of the vector or each member of the product is also always active.

All states in a state expression must belong to the same enumeration / automaton.

### 4.9.1. State Literals

```
parentName = name(state);
```
Name of a composite state (i.e., of a state containing either an automaton or a vector or Cartesian product of automata).

```
atomName = name(state);
```
Name of an atomic state (i.e., of a state the contains neither an automaton nor a vector or Cartesian product of automata).

```
stateLiteral = {parentName.}* (atomName | ({stateLiteral&}2+));
```
A state literal identifies a complete state constant in a possibly complex enumeration.

```
stateExtract =
   {parentName.}* (start | current | atomName | ({stateExtract&}2+));
```
This expression designates a single, complete state of a possibly complex automaton, with respect to the current state. For each leaf of the automaton tree, it specifies whether the value is the start value, the current value or a literal value.

### 4.9.2. Primary State Expressions

A state expression must result in a state in the enumeration determined by the context of the expression.

```
state =
```

```
   (state)
```

```
| stateLiteral
```
A state literal (a constant) identifies a single state in the enumeration.

```
| path(fsa) [.stateExtract]
```
State extracted from an always active automaton. If no extraction is specified, then this is the complete current state of the automaton.

```
| stateExtract
```
State extracted from the always active automaton implied by context.

```
| if bool1 then state1 {elseIf booli then statei;}* [else state2]
```
Choice of one state among several, based on successive Boolean tests.

| ***switch*** **scalar(x)** **{*case*** **domain$_i$(x)** ***then*** **state$_i$;}+** ***else*** **state$_3$**
Choice of one state among several, based on successive membership tests on scalar(x).

;

```
enumeration SwitchState = (on, off, inBetween);
SwitchState[3] systState;
```

In this example, the state of a system (*systState*) is defined by the *on* / *off* / *inBetween* states of its three switches. Primary expression

```
systState[2]
```

designates an automaton (a variable) representing the state of the second switch of the system, whereas the following expression designates a particular state (a constant) of the system

```
systState.(on & on & off)
```

That expression is valid due to the fact that a vector of automata is the Cartesian product of its elements, in the order of the index.

### 4.9.3. State Domains

All the states in a state domain must belong to the same enumeration.

**statesLiteral = {parentName.}* (atomName | *any* | ({stateLiteral&}2+));**
This defines a states domain containing all the states below the points marked by keyword *any*.

**states = *states* | statesLiteral | {{statesLiteral,}*};**

;

## 4.10. Strings

### 4.10.1.String Primary Expressions

**string =**

**(string)**

| ***stringLiteral***
String literal.

| ***path*****(string)**
Pathname for a declared or defined instance of String.

| **vector(String) [int]**
Indexation of a vector of Strings.

| ***path*****(tuple) (int(const))**
Extraction of the int(const)$^{th}$ element of a t-uple, where that element is a String. (T-uples result from the Cartesian product of two or more sets.)

| ***if*** **bool$_1$** ***then*** **state$_1$** **{*elseIf*** **bool$_i$** ***then*** **string$_i$;}*** ***else*** **string$_2$**
Choice of one String among several, based on successive Boolean tests.

| ***switch*** **scalar(x)** **{*case*** **domain$_i$(x)** ***then*** **string$_i$;}+** ***else*** **string$_3$**
Choice of one String among several, based on successive membership tests on scalar(x).

;

### 4.10.2.String Domains

A String domain is a set of Strings:

```
strings =
|  string
|  (strings)
|  {{string,}*}
;
```

## 4.11. Vectors of Variables

### 4.11.1.Vector Primary Expressions

In FORM-L, some primary expressions apply globally to one or more vectors. Vectors in the same vector primary expression must be of the same size. Also, as a vector can be viewed as a set, set primary expressions also apply.

```
vector(x) =
   (vector(x))
```

```
|  path(vector(x))
```
Pathname to a declared or defined vector of the right category.

```
|  ({expression(x),}2+)
```
Parenthesised list of expressions of the right category.

```
|  vector(y).path(x)
```
Derived vector of features or attributes, where x is an attribute or feature of y, possibly through a multistage *path*.

```
|  vector1(x)  [vector2(Integer)]
```
Indexation of a vector of X by a vector of Integers.

```
|  vector1(y)  binOp  vector2(z)
|  vector(y)  binOp  expression(z)
|  term(y)  binOp  vector(z)
```
Application of a binary operation to corresponding vector elements, or to each vector element with a scalar, where binOp is a legitimate operation such that x = y binOp z.

```
|  unOp vector(y)
```
Application to each vector element of a unary operation, where x = unOp y.

```
|  if (bool1 | vector1(Boolean)) then vector2(x)
     {elseIf (booli | vectori(Boolean)) then vectorj(x);}*
     else vector3(x)
```
Choice of one vector among several, based on successive Boolean tests. At each stage, tests may be made globally, or individually for each vector element.

```
|  switch scalar(y) {case domaini(y) then vectori(x);}+ else vector3(x)
```
Choice of one vector among several, based on successive membership tests made with the value of scalar(y).

```
;
```

### 4.11.2.Predefined Vectors

*event*
An event can be viewed as a vector of variable size, its members being the times of its occurrences.

# 4.12. Sets

## 4.12.1.Set Definitions

| **[*private*] [*constant*] type{} *name* = {{path(type),}*};**
| **[*private*] [*constant*] ((*constant* | *fixed*) type){} *name* = {{path(type),}*};**
Definition in extension of a set of scalar variables with constant membership. The members must have been declared or defined elsewhere in the model, and must be of a category consistent with the type. With the second style, one can constrain the members to be constants or fixed.

| **[*private*] [*fixed*] type{} *name* = {sl(type)};**
| **[*private*] [*fixed*] ((*constant* | *fixed*) type){} *name* = {sl(type)};**
Definition in intention of a set of scalar variables with fixed membership, using a spatial locator. The members must be of a category consistent with the type. With the second style, one can constrain the members to be constants or fixed.

## 4.12.2.Set Primary Expressions

In FORM-L, some primary expressions apply globally to one or more sets.

**set(x) =**
**(set(x))**

| **vector(x)**
A vector can be considered as the set of its elements.

| ***path*(set(x))**
Path to a declared or defined set of the right category.

| **{{X,}*}**
Curly-bracketed list of expressions of the right category.

| **set(y).*path*(x)**
Derived set of features or attributes, where x is an attribute or feature of y, possibly through a multistage *path*. This is a shortcut to *{e.path forAll e in set}.*

| **set(y) *binOp* term(z)**
Application to each set member of a binary operation with a scalar item on the right-hand side, where binOp is a legitimate operation such that x = y binOp z. This is a shortcut to *{e binOp scalar forAll e in set}.*

| **term(y) *binOp* set(z)**
Application to each set member of a binary operation with a scalar item on the left-hand side, where binOp is a legitimate operation such that x = y binOp z. This is a shortcut to *{scalar binOp e forAll e in set}.*

| ***unOp* set(y)**
Application to each set member of a unary operation, where x = unOp y. This is a shortcut to *{unOp e forAll e in set}.*

| **{*all name in* set(x) *suchThat* bool(*name*)}**
Selection of the subset of elements that satisfy a given Boolean condition.

| **{scalar(x, *name*) sl(y, *name*)}**
Derived set, based on a computation made on each element determined by spatial locator sl (see Section 12 *Spatial Locators*).

| ***if* $bool_1$ *then* $set_1$(x) {*elseIf* $bool_i$ *then* $set_j$(x);}* *else* $set_2$(x)**
Choice of one set among several, based on successive Boolean tests.

| ***switch* scalar(y) {*case* $domain_i$(y) *then* $set_i$(x);}+ *else* $set_3$(x)**
Choice of one set among several, based on successive tests made with the value of scalar(y).

;

### 4.12.3.Predefined Sets

FORM-L has a number of predefined sets.

***all requirement***
Set of all the requirements of the model.

***all assumption***
Set of all the assumptions of the model.

***all guard***
Set of all the guards of the model.

***all path(class)***
Set of all instances of a class.

***path(enumeration|fsa).states***
Set containing all the states of an enumeration or automaton.

***path(fsa).transitions***
Set containing all the defined transitions of an automaton.

### 4.12.4.Predefined Set Functions

**$set_1$(x) *union* $set_2$(y)**
**$set_1$(x) *inter* $set_2$(y)**
**$set_1$(x) *excl* $set_2$(y)**
Set operations, where category y is compatible with category x.

**$set_1$(y) & $set_2$(z)**
Cartesian product of two sets, where x = (y, z).

**&&({set(y),}2+)**
Cartesian product of multiple sets, where x = ($y_1$, $y_2$,...).

***randomIn*** **set**
Returns a randomly chosen member of a set, with equal probability weights. This expression is valid only at fixed or under the scope of a DTL and is evaluated at each instant of the DTL.

***AND*** **set(bool)**
Returns *true* if and only if all the members of a vector or set are *true*, or if the vector or set is empty.

***OR*** **set(bool)**
Returns *false* if and only if all the members of a vector or set are *false*, or if the vector or set is empty.

***SUM*** **set(int)**
***SUM*** **set(real)**
Returns the sum of all the members of a vector or set, or 0. if the vector or set is empty.

***PROD*** **set(int)**
***PROD*** **set(real)**
Returns the product of all the members of a vector or set, or 1. if the vector or set is empty.

***SMALLEST*** **set(int)**
***LARGEST*** **set(int)**
***SMALLEST*** **set(real)**
***LARGEST*** **set(real)**
Returns the smallest or largest member of a vector or of a set. If the vector or set is empty, returns the largest (SMALLEST) or lowest (LARGEST) representable number.

### 4.12.5.Quantifiers

The universal quantifier ∀ can be expressed in several manners (where bool(name) is a Boolean expression involving name):

```
cardinal {all name in set suchThat bool(name)} = cardinal set;
cardinal {all name in set suchThat not bool(name)} = 0;
AND {bool(name) forAll name in set};
```

The existential quantifier ∃ can be expressed as follows:

```
cardinal {all name in set suchThat bool(name)} ≥ 1;
OR {bool(name) forAll name in set};
```

## 4.13. Variable Declarations

`size = int(fixed);`
This defines the size of a fixed-size vector. It must be evaluated as a strictly positive number.

`arity = [[] | [size] | {}];`
This defines whether the instance is a vector or a set. If absent, then the instance is a scalar. The arity for a vector of variable or yet unknown size is of the form `[]`. The arity for a vector of fixed size is of the form `[size]`. The arity for a set is of the form `{}`.

The general form of a variable declaration is:

```
varDecl =
   [external | private] [fixed | constant] type [arity] name;
| fsaDecl
;
```

where:

- Keyword *external* specifies that the model does not have a definition for the declared instance, and that its value must be obtained from another model through a binding.
- Keyword *private* specifies that the declared instance cannot be accessed through its name outside of the current name scope. However, it could be accessed outside of that scope if included in a non-private set.
- Keyword *fixed* specifies that the instance is a fixed value that does not change in time. However, the value is not necessarily the same for different operational cases.
- Keyword *constant* specifies that the instance does not change value, neither in time nor across operational cases. If neither *fixed* nor *constant* are present, then the instance can be variable. (It is not necessarily so: it could be given a constant or fixed value.)
- `type` is the type of the variable.
- `arity` specifies either a vector or a set. If absent, the variable is a scalar.
- `name` is the name of the variable.

A variable can be declared several times, but the declarations and the possible definition must all be consistent.

An automaton can be declared directly without an intermediate enumeration. However, the declaration cannot specify the *start* and *memory* attributes: this is left to the definition.

```
fsaDecl =
   [external | private] automaton [arity] name =        // Style 1
     ({name_i(state),}2+);
|  [external | private] automaton [arity] name =        // Style 3
     ({name_i(state) [= path(enumeration)],}2+);
|  [external | private] automaton [arity] name =        // Style 5
     ({name_i(state) = path(enumeration)&}2+);
;
```

## 4.14. Variable Definitions

There are two styles of variable definitions.

The global style applies only to scalars and sets. It is a declaration associated with a single expression that specifies the value of the scalar or the membership of the set at all times. This style is closed and cannot specify the value of other attributes. The members of a set must exist independently of the set, and the expression specifying the membership specification is given in extension in the form of a list of names.

The itemised style is a declaration associated with one or more definition snippets. A snippet is composed of three parts:

- An optional spatial locator that identifies the vector element(s) concerned. This part is absent in the case of a scalar or if all vector elements are concerned by the snippet.
- An optional time locator that specifies at which times the snippet applies. If absent, the snippet covers all times. Special time locator *else* covers all times not covered by the other snippets. For a scalar, an *else* clause can be used at most once. For a vector, a given element can be subject to at most one *else* clause.
- An expression that specifies the value of the variable, of the vector elements selected by the spatial locator, or of an attribute other than *value*, at the times specified by the time locator. If the variable is a set, the expression specifies the membership of the set at the times specified by the time locator. If the snippet concerns an attribute other than *value*, then the time locator must be omitted and the snippet must cover all times. Special expression *fixed* specifies that during the time locator, the value or membership does not change.

For a variable or a set, the definition is closed if and only if it covers all times. If not, it is open. For a vector, the fact that the definition is open or closed is determined element by element: it can be open for some, and closed for the others.

```
varDef =
   [private] constant type name = expr(const);
```
Global definition of a scalar constant.

```
|  [private] fixed type name = expr(fixed);
```
Global definition of a fixed scalar variable. The expression must be evaluated at the beginning of each operational case.

```
constant Duration ms = 0.001*s;
constant Duration cycleTime = 50*ms;
fixed Duration phase = random(cycleTime);
```

The discrete time domain of a synchronous control system is characterised by two durations. One named *cycleTime* is a constant of 50 milliseconds. The second named *phase* is a fixed variable of random value between 0 and 50 milliseconds: it does not change in the course of an operational case, but it might different for each case. All three declarations use the global style.

| `[private] type name = expr;`
Global definition of a scalar variable in the continuous time domain.

```
Real maxFailureProbability = 1.0 - exp(-k*time);
```

This is a closed variable definition in a global style: it covers all times and does not allow assignments elsewhere in the property model.

| `[private] constant type name`
`((attribute | name) = expr(const);)+`
`end name;`
Itemised definition of a scalar constant. It is used to specify attributes *value*, *unit*, *scale* and *offset*. All expressions must be constant. The name of the constant may be used in the place of attribute name *value*.

| `[private] fixed type name`
`((attribute | name) = expr(fixed);)+`
`end name;`
Itemised definition of a fixed scalar variable. It is used to specify attributes *value*, *unit*, *scale* and *offset*. The expression for *value* must be fixed. The name of the fixed variable may be used in the place of keyword *value*. The expression for the other attributes must be constant.

| `[private] type name`
`[timeDomain = (continuous | path(timeDomain);]`
`((attribute | name) = expr;)+`
`end name;`
Itemised definition of a scalar variable in the specified time domain. By default, it is in the continuous time domain. This style of definition is also used to specify attributes *value*, *start*, *unit*, *scale* and *offset*. If present, the expression for *value* is a global assignment. The name of the variable may be used in the place of keyword *value*. If present, the expression for attribute *start* must be a fixed value. *value* takes precedence over *start*. The expression for the other attributes must be constant.

| `[private] type name`
`[timeDomain = (continuous | path(timeDomain);]`
`(attribute = expr (fixed);)*` `// Except attribute value`
`((dtl`$_i$ `then [(value | name) =] expr`$_i$`);`
`|(ctl`$_j$ `then [(value | name) =] (expr`$_j$ `| fixed));`
`)*`
`[else [(value | name) =] (expr`$_k$ `| fixed);]`
`end name;`
Itemised definition of a scalar variable, attribute *value* being defined using time locators. Value must be consistent in time: there must be no contradiction. The *else* clause covers the time instants not covered by any time locators. Under the scope of a *ctl* or an *else* clause, keyword *fixed* may be used to indicate that the value does not change during the corresponding time periods. If the *else* clause is present or if the time locators cover the complete time, then the definition if closed. Otherwise, it is open.

```
Power maxPowerDemand
  during not powered then 0.0*w;
  during powered and (after powered for settleTime) then initialPower;
  else power;
  end maxPowerDemand;
```

Variable *maxPowerDemand* determines an upper bound for the power demand of an electrical component, taking into account the initial spike when the component is connected to power (see Figure 4-1). The definition is of the itemised style and is closed (due to the *else* clause): i.e., it does not allow any assignments elsewhere in the property model. Note that the formula avoids inconsistencies when the *powered* period is shorter than *settleTime*.

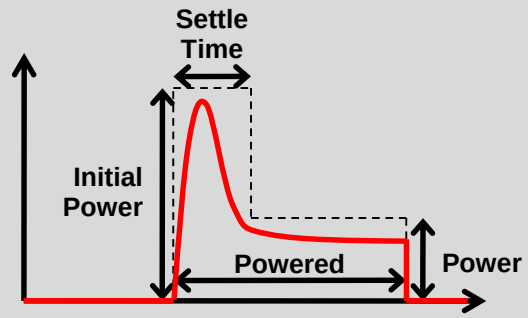


*Figure 4-1*
*Maximum expected demand for power for a backed-up equipment*

```
| [private] constant type [size] name
    ((attribute | name) [[ints]] = expr(const);)+
    end name;

| [private] fixed type [size] name
    ((attribute | name) [[ints]] = expr(fixed);)+
    end name;

| [private] type [size] name
    [timeDomain = (continuous | path(timeDomain);]
    ((attribute | name) [[ints]] = expr;
    | dtl_i then (value | name) [[ints_i]] = expr_j;
    | ctl_j then (value | name) [[ints_j]] = (expr_j | fixed);
    | else (value | name) [[ints_k]] = (expr_k | fixed);
    )+
    end name;
```

Itemised definition of a vector of variables. All vector elements must be in the same time domain When, for a given snippet, no index is specified, then the snippet concerns all the elements of the vector. The expressions for attributes other than *value* must be fixed.

One can mix global value definitions (through the variable name or attribute *value*) and itemised value definitions (through time locators and possibly through an *else* clause). However, for a given vector element, it must be either one or the other, not both. The itemised value definitions for each vector element must be consistent: i.e., at each time instant, at most one value can be specified for that element. An *else* clause can be used to cover the time instants not covered by any time locators. Keyword *fixed* (to be used only with a CTL or an *else* clause) specifies that the value of the element does not change during the corresponding time periods. Itemised value definitions for which no index is specified concern all the elements of the vector, except those that have a global value definition.

If an element has a global value definition, or has an itemised value definition that covers the complete time (e.g., with an *else* clause), then the definition is closed for that element. Otherwise, it is open. This decision is made element by element.

```
| [private] [constant] type{} name = {{path(varType),}*};
| [private] [constant] ((constant|fixed) type){} name ={{path(varType),}*};
```

Definition in extension of a set of scalar variables with constant membership. The members must have been declared or defined elsewhere in the model, and must be scalars of a type consistent with type. With the second style, one can constrain the members to be constants or fixed.

```
| [private] [fixed] type{} name = {sl(type)};
| [private] [fixed] ((constant | fixed) type){} name = {sl(type)};
```

Definition in intention of a set of scalar variables, possibly with fixed membership, using a spatial locator. The members must be scalars of a type consistent with type. With the second style, one can constrain the members to be constants or fixed.

```
| [private] automaton name =
    [timeDomain = (continuous | path(timeDomain);]
    ({name_i(state) [= path(enumeration)],}2+)
    [defaultStartState;]
    [defaultMemory;]
    ((dtl_i then [(value | name) =] stateExtract_i;)
    |(ctl_j then [(value | name) =] (stateExtract_j | fixed);)
    )*
    [else [(value | name) =] (stateExtract_k | fixed);]
    end name;
```

Direct definition of a scalar automaton as a list of states and state transitions.

```
| [private] automaton [size] name =
    [timeDomain = (continuous | path(timeDomain);]
    ({name(state) [= path(enumeration)],}2+)
    [defaultStartStates;]
    [defaultMemories;]
    ((dtl_i then (value | name) [[ints_i]] = stateExtract_i;)
    |(ctl_j then (value | name) [[ints_j]] = (stateExtract_j| fixed);)
    |(else (value | name) [[ints_k]] = (stateExtract_k | fixed);)
    )*
    end name;
```

Direct definition of a vector of automata as a list of states and state transitions.

```
;
```

## 4.15. Definition Snippets for Variables

Only variables with an open definition can have definition snippets outside their definition.

```
varSnippet =
```

`tl then path(variable) = expr;`
A snippet includes a time locator and a value definition. expr must be compatible with the type (*Real*, *Integer*, *Boolean*, *automaton*,...) of the variable. All snippets for the same variable must be consistent, i.e., at any instant, there must be one and only one value. If the *path* leads to a vector or a set, then all elements are concerned.

`| tl then path(variable) [ints] = expr;`
The *path* must lead to a vector, and the index specifies which elements are concerned, with ints being a sub-domain of [1, *size*], where *size* is the number of elements of the vector.

`| sl tl then name = expr;`
*name* is the name used by the spatial locator sl.

```
;
```

Note. Inconsistencies arise when two snippets give different values at the same time to the same variable. Inconsistencies also arise when the order of multiple snippets is not specified and is yet significant (different orders yielding different values for the same variable at the same time). It is the responsibility of model authors to avoid such inconsistencies, possibly with the help of a static analysis tool.

```
Power maxPowerDemand
   start := 0.0*w;
   end maxPowerDemand;
```

This is an alternative, open definition for *maxPowerDemand*. It allows (and indeed, needs) assignments elsewhere in the property model.

```
after powered for settleTime then maxPowerDemand = initialPower;
after powered + settleTime until not powered then maxPowerDemand = power;
```

```
during not powered then maxPowerDemand = 0.0*w;
```

# 5. Functions & Function Calls

A **function** is a formula that calculates a scalar Boolean, Integer or Real (primitive or derived) based on one or more **arguments**. The function is characterised by its **signature**, composed of its pathname and of the ordered list of its arguments categories. Different functions may have the same name, but then their lists of arguments categories must be different. Also, in a given space name, a function cannot have the same name as a procedure. Argument values are provided whenever the function is called: they must match the category specified by the signature. A signature may specify that the function can take a variable number of arguments (using keyword *etc*): any additional argument beyond the signature must then match the last argument category of the signature.

```
functionType =
   Boolean
| Integer
| Real
| path(derivedReal)
;
```

```
classPath = path(class);
```
Pathname to a class definition.

```
eventPath = path(event);
```
Pathname to an event definition.

```
category = ([constant |fixed] type | classPath | eventPath) [arity];
```
The category of a function argument can be a type, a class or an event. If an arity is specified, then the argument has that arity, else it is a scalar.

```
argumentName = name;
```
Name of an argument. All argument names for a given function must be different from one another.

```
functionDecl = functionType name ({category,}+ [,etc]); ;
```
Function declaration, where:

- functionType is the type of the result of the function, restricted to Boolean, Integer, Real and derived Reals.
- name is the name of the function.
- category is the category of the $i^{th}$ argument.
- If present, keyword *etc* specifies that the function has a variable number of arguments: the key word repeats the preceding category 0 to n times.

```
functionDef = functionType name ({(category argumentName),}+ [,etc]) [
```
argumentName are the names that can be used in the expressions defining the value of the function to represent the arguments of the function. Keyword *etc* at the end of the argument list specifies that a call to the function may provide more arguments. Any supplementary argument must match the last category in the signature. In the expressions defining the value of the function, one gets access to these arguments through a vector bearing the name of the last argument in the signature, and the elements of which being the argument corresponding to that name and all supplementary arguments, in the order provided by the call. Its size can be obtained through the *size* attribute of the vector.

```
= expr;
```
In this style of function definition, expr is an expression based on the arguments that specifies the value of the function at all times.

```
|  ((dtl_j | ctl_j) then expr_j;)+
   [else expr_k;]
   end name;
```

In this style of function definition:

- $tl_j$ is a time locator, possibly expressed in terms of the arguments.
- $expr_j$ is an expression based on the arguments that specifies the value of the function at the instants specified by time locator $tl_j$.
- $expr_k$ is an expression based on the arguments that specifies the value of the function at the instants not covered by any time locator.

```
;
```

```
functionCall = name ({expr,}+);
```

Function call, where the: expr are the values of the arguments of the function. The number and the categories of the arguments must be consistent with one function signature.

# 6. Events & Event Occurrences

An **event** characterises particular facts that have no duration: each of these facts is an **occurrence** of the event. This notion is closely related to, but different from, the notion of **discrete time locator** (DTL). A DTL is attached to each event and states the instants of occurrence of the event. Positions in time are relative to the beginning of time, which is represented by a predefined event named *t0* that has only one instant. Wherever a DTL can or must appear, an event can replace it (more precisely, it will be its DTL).

By default, the instants of occurrence are the only feature attached to an event. That event is then said to be a **naked event**. However, a **rich event** may have additional features, in the form of variables and associated objects (see Section 0 *An* object is characterised by its features. A feature can be a variable, an event (naked or rich), an embedded object (i.e., an object that is an integral part the object being characterised and that has no independent existence), an associated object (i.e., an object that exists independently of the object being characterised) or a property.

Associations). The additional features of a rich event are specified by an **event class**. An event class always extends another event class (which could be the predefined event class *event*, which corresponds to the naked events).

```
eventDecl = nakedEventDecl | richEventDecl;
```

```
eventDef = nakedEventDef | richEventDef;
```

## 6.1. Naked Event Declaration

A naked event declaration states:

- The name of the event.
- Whether the event is private or not.
- Whether the event is external or not.
- Whether the event is a scalar, a vector or a set. If the event is a vector, the declaration may specify the vector size. As an event is itself a vector of occurrences of variable size, a vector of events cannot be of variable size.

```
nakedEventDecl =
   [external] [private] event [arity] name;
```
Declaration of a naked event in the continuous time domain. If the event is a vector or a set, all elements are in the continuous time domain.

```
|  [external] [private] event [arity] name =
      [timeDomain = (continuous | path(timeDomain);]
      end name;
```
Declaration of a naked event in a specified time domain. If the event is a vector or a set, all elements are in the same time domain.

```
;
```

```
event eActivateMotor;
```

This is a declaration of scalar naked event *eActivateMotor* that signals that the motor must be started. An event declaration can also serve as an open definition.

```
when motorNeeded becomes true then raise eActivateMotor;
```

This is a snippet that raises *eActivateMotor* when the motor is needed.

## 6.2. Naked Event Definition

A naked event definition states:

- The name of the event.
- Whether the event is private or not.
- Whether the event is a scalar, a vector or a set. If the event is a vector, the definition specifies the vector size. If the event is a set, the definition identifies its members.
- The occurrences of the event (scalar) or events (vector).
- Whether the definition is open or closed. In the case of a vector, whether the definition is open or final can be specific to each vector member.

```
nakedEventDef =
   [private] event name = dtl;
```
This style of definition for a naked scalar event just states when it occurs. The event is in the continuous time domain. This is a closed definition.

```
external Boolean button;
event eRequest = when button becomes true;
```

To signal a request, a human operator has an on/off device named *button*, which is modelled as a Boolean. The request is issued by pressing the button and is represented by event *eRequest*.

```
|  [private] event name
      [timeDomain = (continuous | path(timeDomain);]
      (ctl_i then [name =] (dtl_i | fixed);)+
      [else [name =] (dtl_j | fixed);]
      end name;
```
This style of definition for a naked scalar event specifies that during a given ctl, the event occurs as specified by the corresponding dtl, or has no occurrences (keyword *fixed*) The CTLs must not overlap. It is a closed definition if the *else* clause is present or if the CTLs cover the complete time.

A definition equivalent to the *eRequest* example could be:

```
event eRequest2
   when button becomes true;
   end eRequest2;
```

```
| [private] event [size] name
    [timeDomain = (continuous | path(timeDomain);]
    ((ctl_i then [name [[size]] =] (dtl_i | fixed);)
    |(else [name [[size]] =] (dtl_i | fixed);)
    )*
    end name;
```

This style of definition applies to vector of naked events. It is similar to the previous one, except that an index may be used to specify which vector elements are concerned.

```
| [private] [constant] event {} name = {{path(event),}*};
```

Definition in extension of a set with a constant membership of naked events. The members must have been declared or defined elsewhere in the model and must be in the same time domain.

```
| [private] [constant] event {} name
    [timeDomain = (continuous | path(timeDomain);]
    (name | value) = {{path(event),}*};
    end name;
```

Definition in extension of a set with a constant membership of naked events. The members must have been declared or defined elsewhere in the model and must be in the specified time domain.

```
| [private] [fixed] event {} name = {sl(event)};
```

Definition in intention of a set of naked events in the same time domain, possibly with fixed membership, using a spatial locator.

```
| [private] [constant] event {} name
    [timeDomain = (continuous | path(timeDomain);]
    (name | value) = {sl(event) };
    end name;
```

Definition in intention of a set of naked events in the specified time domain, possibly with fixed membership, using a spatial locator.

```
;
```

## 6.3. Event Class Definition

An **event class** directly or indirectly extends predefined event class *event*. It may specify a time domain, or specify that its instances are to be placed in a discrete time domain (not necessarily the same for all). Also, when an event class provides a definition for a feature (instead of just a declaration), then that definition must be closed.

```
eventClassDef = class name extends [event | path(eventClass)]
   [timeDomain = (continuous | discrete | path(timeDomain);]
    (varDecl|varDef(closed)|associationDecl|associationDef(closed))*
    end name;
;
```

## 6.4. Rich Event Declaration

A rich event declaration is similar to a naked event declaration, except that it specifies a defined event class in the place of keyword *event*.

```
richEventDecl = [external | private] path(eventClass) [arity] name;;
```

## 6.5. Rich Event Definition

A rich event definition is similar to a naked event definition, with the addition that it provides a definition (open or closed) for each feature declared but not defined by the event class.

```
richEventDef =
   [private] path(eventClass) name = dtl;
```

This style of definition for a rich scalar event applies when the event class defines all the event features. The event occurs at the instants specified by the dtl. This is a closed definition.

| `[private] path(eventClass) name`
`[timeDomain = (continuous | path(timeDomain);]`
`(dtl then sequence (raise name; | path(feature) = expr;)2+ end;)+`
`[else fixed;]`
`end name;`

This style of definition for a rich scalar event applies when some of the event features are not defined by the event class. The event occurs at the instants specified by one or more dtl, and the sequence associated with each specifies the values of those features not defined by the event class. The raise clause must appear once and only once in a sequence. This is a closed definition if the *else* clause is present, an open definition if it is not. There is a contradiction if different dtl have an occurrence at the same instant and specify different values for the same feature.

```
class EMsg extends event
   Integer nb
      start = 0;
      end nb;
   end EMsg;

EMsg eMsg
   when ... then sequence
      nb = previous(nb) + 1;
      raise eMsg;
      end;
   end eMsg;
```

*eMsg* is a rich event representing the emission of a data communication message. Attached to each *eMsg* occurrence is the identification number of the message. The definition is open.

*nb* counts the number of messages already emitted, and is used to identify the next message to be emitted.

| `[private] path(eventClass) [size] name = dtl;`

This style of definition for a vector of rich events applies when the event class defines all the event features. All the vector elements occur at the instants specified by the dtl. This is a closed definition.

| `[private] path(eventClass) [size] name`
`[timeDomain = (continuous | path(timeDomain);]`
`(name[ints] = dtl)+`
`end name;`

This style of definition for a vector of rich events applies when the event class defines all the event features. A vector element occurs at the instants specified by the associated dtl. Each index of the vector must appear once and only once. This is a closed definition.

| `[private] path(eventClass) [size] name`
`[timeDomain = (continuous | path(timeDomain);]`
`(name[ints] =`
`(dtl then sequence (raise name; | path(feature) = expr;)2+ end;)+`
`[else fixed;]`
`)+`
`end name;`

This style of definition for a rich scalar event applies when some of the event features are not defined by the event class. A vector element occurs at the instants specified by one or more associated dtl, and the sequence associated with each specifies the values of those features not defined by the event class. The raise clause must appear once and only once in a sequence. For a given vector element, this is a closed definition if there is an *else* clause, an open definition if it is not. There is a contradiction if different dtl have an occurrence at the same instant and specify different values for the same feature. Each index of the vector must appear once and only once.

| `[private] [constant] path(eventClass) {} name = {{path(event),}*};`

Definition in extension of a set of rich scalar events with constant membership. The members must have been declared or defined elsewhere in the model, and must be scalars of the event class specified.

```
| [private] [constant] path(eventClass) {} name
    [timeDomain = (continuous | path(timeDomain);]
    (name | value) = {{path(event),}*};
    end name;
```
Definition in extension of a set with a constant membership of rich events. The members must have been declared or defined elsewhere in the model and must be in the specified time domain.

```
| [private] [fixed] path(eventClass) {} name = {sl(event)};
```
Definition in intention of a set of rich scalar events, possibly with fixed membership, using a spatial locator. The members must be scalars of the event class specified.

```
| [private] [constant] path(eventClass) {} name
    [timeDomain = (continuous | path(timeDomain);]
    (name | value) = {sl(event) };
    end name;
```
Definition in intention of a set of rich events in the specified time domain, possibly with fixed membership, using a spatial locator.

```
;
```

## 6.6. Definition Snippets for Events, Event Signalling

Only events with an open definition can have definition snippets outside their definition. An event definition snippet is composed of a DTL and an event signalling. The latter specifies which event is to be signalled, and may provide definition snippets for the features of a rich event. There is an occurrence of the event at each instant specified by the DTL, in addition to the those already specified by the event definition. In the case of a rich event, the value assigned to a feature must be compatible with the type (*Real*, *Integer*, *Boolean*, *fsa*,...) of the feature. All declarations, definitions and assignments for the same feature must be consistent.

```
eventSnippet =
```

```
  dtl raise path(event)
```
This style is used for naked events, and for rich events the features of which are all defined by the event class. If path(event) leads to a vector or a set, then all elements are concerned.

```
| dtl raise path(event) [ints]
```
This style is used for vectors of naked events, and for vectors of rich events the features of which are all defined by the event class. The index specifies which elements are concerned, with ints being a sub-domain of [1, *size*], where *size* is the number of elements of the vector.

```
| sl dtl then raise name;
```
*name* is the name used by the spatial locator sl.

```
| dtl sequence
    (raise path(richEvent); | path(feature) = expr;)2+
    end;
```
This style is used for rich events some features of which are not defined by the event class. If path leads to a vector or a set, then all elements are concerned.

```
| dtl sequence
    (raise path(richEvent) [ints]; | path(feature) [ints] = expr;)2+
    end;
```
This style is used for rich events some features of which are not defined by the event class. path must lead to a vector, and the index specifies which elements are concerned, with ints being a sub-domain of [1, *size*], where *size* is the number of elements of the vector. It must be the same for all features and the signalling.

```
| sl dtl then sequence
    (raise name1; | name1.name2(feature) = expr;)2+
    end;
```
$name_1$ is the name used by the spatial locator sl. $name_2$ is the name of a feature.

```
;
```

## 6.7. Event Expressions

```
event =
```

```
   dtl [alias name] [suchThat bool]
```

An event expression can be a dtl, in which case it produces a naked event.

```
| path(eventScalar) [alias name] [suchThat bool]
| path(eventVector) [int] [alias name] [suchThat bool]
```

An event expression can also be a pathname to a defined or declared event (naked or rich). A scalar index must be provided if the pathname leads to a vector.

```
;
```

### 6.7.1. Aliases

The occurrences of an event are usually accessed in the framework of a property or a snippet that uses the event name to implicitly identify an occurrence. However, there are cases where one needs to deal with different occurrences of the same event. **Aliases** are used to give a specific name to each of these occurrences.

```
requirement r =
   after eMsg alias first until eMsg alias second
   check second.id = first.id+1;
   end r;
```

Continuing with the messaging example, a requirement could be that messages are received (event *eMsg*) in their original sequence. I.e., two messages received consecutively must have consecutive identification values.

### 6.7.2. Event Filters

Sometimes, one needs to refer to occurrences of an event that satisfy a given Boolean condition. This usually concerns rich events, but the Boolean condition may put constraints on the timing of naked events. Event filtering is expressed with keyword *suchThat*.

## 6.8. Event Occurrences

Reference to arbitrarily chosen occurrences can be done using the following:

- Event attribute **count**, a variable whole number, is the number of occurrences of the event since the beginning of time. It is automatically calculated and cannot be directly specified by a model.
- Event attribute **time**, a dynamically sized vector (of size *count*). Its $i^{th}$ element provides the time of the $i^{th}$ occurrence of the event. It is automatically calculated and cannot be directly specified by a model.
- Event attribute **clock**, a dynamically sized vector (of size *count*). In a discrete time domain, its $i^{th}$ element provides the number of clock ticks since the beginning of time to the $i^{th}$ occurrence of the event. It is automatically calculated and cannot be directly specified by a model.

# 7. Classes & Objects

An object is characterised by its features. A feature can be a variable, an event (naked or rich), an embedded object (i.e., an object that is an integral part the object being characterised and that has no independent existence), an associated object (i.e., an object that exists independently of the object being characterised) or a property.

## 7.1. Associations

An associated object, also called target object, has a name and an existence of its own. The association can be defined by assigning the pathname of the target object to the feature name. The feature name is then a surrogate name for the target object, and a path to the feature is in fact a path to the target object. Thus, the notion of association is very similar to the notion of pointer in programming languages. For a given association, the target object may be always the same (then the association is constant) or can vary in time.

```
associationDecl =
   [external | private] [constant]
     association [arity1] name to path(class) [arity2];
```

*name* is the feature name, and $arity_1$ is the arity of the feature. Each of member of the association is associated to one target object of a class consistent with class and having the arity specified by $arity_2$. The same object may be the target of several associations.

```
associationDef =
```

```
   [private] [constant] association name = path(object)
```

In this case, the association is a scalar, and has the same target object at all times and in all cases.

```
|  [private] association name
     [start = path1(object);]
     (dtli then [(name | value) =]  pathi(object);
     |ctlj then [(name | value) =] (pathj(object) | fixed);
     )*
     [else [(name | value) =] (pathk(object) | fixed);]
     end name;
```

In this case, the association is a scalar, and its target object may vary in time as specified.

```
|  [private] [constant] association [size] name = path(object)
```

In this case, the association is a vector, all elements of which have the same target object at all times and in all cases.

```
|  [private] [constant] association [size] name
     ((name | value) [ints] = path(object);)2+
   end name;
```

In this case, the association is a vector, and different elements may have different target objects. A given element has the same target at all times and in all cases.

```
|  [private] association [size] name
     ((start | name) [[ints]] = path1(object);
     | dtli then (value | name) [[intsi]] =  pathi(object);
     | ctlj then (value | name) [[intsj]] = (pathj(object) | fixed);
     | else (value | name) [[intsk]] = (pathk(object) | fixed);
     )+
   end name;
```

In this case, the association is a vector, and the target object of each element may vary in time as specified.

```
|  [private] [constant] association {} name = {{pathi(object),}*};
```

In this case, the association is a constant set defined in extension.

```
|  [private] [fixed] association {} name = {sl(object)};
```

In this case, the association is a set, possibly of fixed membership, defined in intension with a spatial locator.

```
;
```

## 7.2. Class Definition

A class definition specifies:

- The name of the class being defined.
- Whether the class is partial or not. If partial, it cannot be directly instantiated, but only through non-partial extension classes. A class is partial when it is so astract that instantiating it would be meaningless.
- Whether the class extends another one or not. If so, its instances will be also be those of the extended class.
- An optional discrete time domain. That time domain may be defined by the model, or may be declared external and defined by another model. A time domain is allowed only if the class is not an extension to a class that already has an attached time domain. When a time domain is attached to a class, all the instances of the class are placed in that time domain. When no time domain is attached to the class, each instance can decide in which time domain it is placed, the default being the continuous time domain.
- A list of features for the class. A feature can be attached to the instances of the class (by default) or to the class itself. Features attached to the class must be defined. Features attached to the instances can be either defined by the class or just declared, in which case each instance must separately provide a definition.

A class or an event class may be embedded in another one: then, it can be instantiated only in the framework of the embedding class. The advantage though is that it has direct access to the features of the embedding class.

A class can also specify or constrain the time domains of its instances and global features:

- It may force all of them to be in the continuous time domain, or in the same particular discrete time domain.
- It may force each of its instances to specify a discrete time domain, not necessarily the same, by using keyword *specific*. Its global features must then be in the continuous time domain.
- If no time domain constraints are specified, then instances may be placed in any time domain, but global features are placed in the continuous time domain.
- When the class is a derived class, it inherits and must be consistent with the choices, if any, made by the class it extend. This means that if the extended class specifies that its instances must be placed in an undetermined discrete time domain, the derived class may specify a specific one. If the extended class specifies no time domain constraints (and thus its global features are in the continuous time domain), the derived class may specify some that may place its additional global features in a specific discrete time domain.

```
featureDecl = [global | specific]
   (varDecl | eventDecl | objectDecl | associationDecl | propertyDecl
    classDecl | eventClassDecl)
```

If present, optional keyword *global* specifies that the feature is attached to the class. If absent, the feature is attached to the instances. If present, optional keyword *specific* specifies that the feature is not defined by the class and is to be defined by by extension classes or by each instance.

```
featureDef = [global]
   (varDef | eventDef | objectDef | associationDef | propertyDef |
    classDef | eventClassDef);
```

If present, optional keyword *global* specifies that the feature is attached to the class. If absent, the feature is attached to the instances.

```
classDecl= [partial] class name [extends path1(class);]
   [(timeDomain = (continuous | path2(timeDomain));) | timeDomainDef)]
   featureDecl+
   end name;
```

```
classDef = [partial] class name [extends path1(class);]
   [(timeDomain = (continuous | path2(timeDomain));) | timeDomainDef)]
   (featureDecl | featureDef)+
   end name;
;
```

where:

- *name* is the name of the class being defined.
- Keyword *partial* specifies that the class cannot be instantiated directly.
- Keyword *extends* specifies that the class is a subclass of another class. A partial class can only extend another partial class.
- path1(class) is a pathname to the class that is extended.
- Keyword *continuous* specifies that all instances and global features will be placed in the continuous time domain.
- Keyword *discrete* specifies that all instances will have to specify a discrete time domain, and that global features will be placed in the continuous time domain.
- path2(timeDomain) is a pathname to a discrete time domain where all instances and global features will be placed.

## 7.3. Object Declaration

An object can be declared (and not defined) only through a class instantiation, which specifies:

- A pathname for the non-partial class being instantiated.
- The name of the object instantiating the class.
- Whether the object is defined by the model or not.
- Whether the object can be accessed through its name outside the current name scope.
- The arity of the object (i.e., whether it is a scalar, a vector or a set).

Regarding time domains:

- If the class already specifies a specific time domain, then the object declaration does not need to specify one. If it does, it must be the same as the one specified by the class.
- If the object is external (i.e., it is not defined by the model), and if the class leaves the choice between the continuous or a discrete time domain, then the object declaration must specify whether the time domain is continuous, discrete but not determined, or discrete and fully determined.
- If the object is defined by the model, and if the class leaves the choice between the continuous or a discrete time domain, then the object declaration must specify a specific time domain. All declarations and the definition for the object must specify the same time domain.
- If the object is a set, then each set member must be in a time domain consistent with the constraints set by the object.

```
objectDecl =
|  [external | private] path1(class) [arity] name;
|  [external | private] path1(class) [arity] name
   [timeDomain = (continuous | discrete | path2(timeDomain));]
   end name;
;
```

where:

- path1 (class) is a pathname for the class of the object being declared.
- *name* is the name of the object.

- Keyword *external* specifies that the model does not have a definition for the declared instance, and that its value must be obtained from another model through a binding.
- Keyword *private* specifies that the declared instance cannot be accessed through its name outside the current name scope. However, it could be accessed outside that scope if included in a non-private set.
- arity specifies whether the object is a scalar, a vector of given or unknown size, or a set.
- Keyword *continuous* specifies that if the object is a scalar, it is in the continuous time domain. If the object is a vector, all its elements are also in the continuous time domain. If the object is a set, all its members must be in the continuous time domain.
- Keyword *discrete* specifies that if the object is a scalar, it is in a discrete time domain that will be specified by the object definition. If the object is a vector, all its elements are in a discrete time domain, not necessarily the same, that will be specified by the object definition. If the object is a set, all its members must be in a discrete time domain, not necessarily the same.
- path$_2$(timeDomain) is a pathname to a discrete time domain where the object is placed if it is a scalar. If it is a vector, all its elements are in that time domain. If it is a set, all its members must be in that time domain.

## 7.4. Object Definition

There are two styles of object definition:

- A direct object definition can be used when the object is the only one of its kind and scalar. It is similar to a class definition, except that keyword *object* is used instead of *class*, and that keywords *global*, *specific*, *partial* and *same* are not used.
- A class instantiation can always be used, but is necessary when there are other objects of that kind. A class instantiation must comply with the class definition and for example cannot introduce features not already in the class or change their category.

If it not already done by the class definition, an object definition places the object in a time domain:

- If the object is a scalar, an optional discrete time domain may be specified. If no time domain is specified, neither by the class nor by the object definition, then the object is in the continuous time domain.
- If the object is a vector of fixed size, an optional discrete time domain may be specified, globally for all elements, or individually for each element. If no time domain is specified, neither by the class nor by the object definition, then all the vector elements are in the continuous time domain.
- If the object is a vector of variable size, an optional discrete time domain may be specified for all elements. If no time domain is specified neither by the class nor by the object definition,, then all the vector elements are in the continuous time domain.
- If the object is a set, then the time domain of each member is to be specified by that member but must be consistent with the constraints set by the set.

```
objectDef =
   [private] object name
      [extends path1(class);]
      [(timeDomain = (continuous | path2(timeDomain));) | timeDomainDef)]
      (varDecl| eventDecl | associationDecl | contractExtract | featureDef)+
      end name;
```

Direct object definition. Features that are just declared (only variables, associations and events) have open definitions and will need assignments or signalling elsewhere.

```
|  [private] path(class) name
      [(timeDomain = (continuous | path2(timeDomain));) | timeDomainDef)]
      (assignment | signalling | statementDef)*
      end name;
```

Class instantiation as a scalar. Only variables and events declared or given an open definition by the class (either directly or through object features) can be adjusted by assignments or signalling.

```
| [private] path(class) [size] name
    [(timeDomain = (continuous | path2(timeDomain));) | timeDomainDef)]
    (varDecl | eventDecl | associationDecl | featureDef)+
  end name;
```

Class instantiation as a vector. The time locators and properties definitions, and the time domain clauses, can specify an index identifying the vector element(s) concerned. If no index is specified, all elements are.

```
| [private] [constant] path(class) {} name = {{path(instOfClass),}*};
```

Definition in extension of a set of scalar objects with constant membership. The members must have been declared or defined elsewhere in the model, and must be objects of a class consistent with class.

```
| [private] [fixed] path(class) {} name = {sl(instOfClass)};
```

Definition in intention of a set of scalar objects, possibly with fixed membership, using a spatial locator. The members must be objects declared or defined elsewhere and of a class consistent with class.

```
;
```

```
object motor
   external Boolean active;
   external Boolean ready;
   assumption a = when not active check not ready;

   external automaton mode = (workToDeath, protection);

   end motor;
```

The *motor* modelled here can be *active* or not: it becomes *active* when it starts operating, and remains so until it stops completely. It needs some time between the instant it starts operating and the instant it is *ready* for use. Both are represented by Booleans that will be determined later by a design. Thus, they are here declared *external*.

When *active*, the *motor* can be either in *workToDeath* or *protection* mode:

- In the *workToDeath* mode, it will continue operating even when its internal checks signal problems.
- In the *protection* mode, it will stop operating when its internal checks signal a problem to avoid damaging it any further.

The *mode* is also determined by the design and declared *external*.

# 8. Durations

A **duration** measures the length of a stretch of time. In FORM-L, there are different ways to express a duration:

- A duration with respect to physical time is expressed as an instance of type *Duration* derived from Real, with *dimension* attribute **"time"**.
- In a discrete time domain, a duration can also be expressed as a number of clock ticks.
- A duration may be expressed as a number of instants of a DTL.

Figure 8-1 summarises the legitimate ways of expressing a duration, depending on whether the time locator is expressed in a continuous or discrete time domain, and on whether the time locator itself is continuous or discrete.

| | Continuous Time Domain | Discrete Time Domain |
|---|---|---|
| CTL | Physical time | Physical time |
| | | Number of clock ticks |
| DTL | Physical time | Physical time |
| | | Number of clock ticks |
| | Number of DTL instants | Number of DTL instants |

*Figure 8-1*
*Legitimate ways of expressing a duration.*

# 9. Discrete Time Locators (DTL)

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

```
dtl =
```

(dtl)

## 9.1. Basic DTLs

```
| when bool becomes true
| when bool becomes false
| when bool changes
```

The DTL has a time position for each instant where the Boolean condition becomes true (see Figure 9-1), becomes false (see Figure 9-2), or changes value (see Figure 9-3). When evaluated in a discrete time domain, the Boolean condition changes value when its value at a given instant is different from its value at the preceding instant.

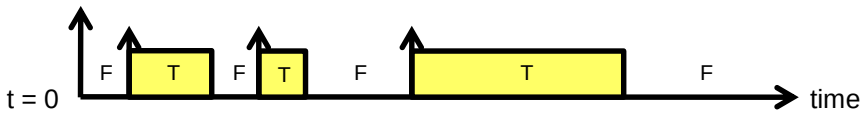


*Figure 9-1*
***when bool becomes true***

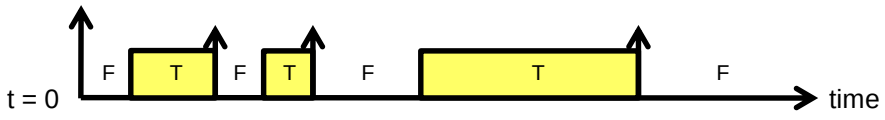


*Figure 9-2*
***when bool becomes false***

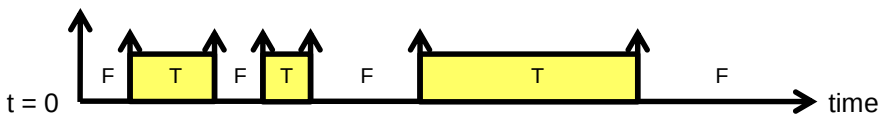


*Figure 9-3*
***when bool changes***

| ***when*** **path(fsa)** ***changes***
| ***when*** **path(fsa)** ***becomes*** **(state$_1$ | states$_1$)**
| ***when*** **path(fsa)** ***leaves*** **(state$_1$ | states$_1$)**
| ***when*** **path(fsa)** ***leaves*** **(state$_1$ | states$_1$)** ***for*** **(state$_2$ | states$_2$)**
| ***when*** **int** ***changes***
| ***when*** **int** ***becomes*** **(int$_1$ | ints$_1$)**
| ***when*** **int** ***leaves*** **(int$_1$ | ints$_1$)**
| ***when*** **int** ***leaves*** **(int$_1$ | ints$_1$)** ***for*** **(int$_2$ | ints$_2$)**
A DTL can be defined when an automaton changes state or transitions between two given (sets of) states. They can also be defined when an integer variable changes value, or transitions between two given (sets of) values.

| ***when*** **event**
This expression simply denotes the DTL associated with the event.

| ***at*** **(duration(fixed) | clockDuration(fixed))**
This expression defines a DTL with a single instant that occurs at the specified duration after the beginning of time. That duration must be a fixed value. It can be expressed either as a physical time duration (to be used only in the continuous time domain) or as a number of clock ticks (clockDuration, to be used only in a discrete time domain).

| ***every*** **(duration(fixed) | clockDuration(fixed))**
This expression defines a periodic DTL, the period of which is the specified duration (see Figure 9-4), which must be a fixed value. It can be expressed either as a physical time duration (to be used only in the continuous time domain) or as a number of clock ticks (clockDuration, to be used only in a discrete time domain).

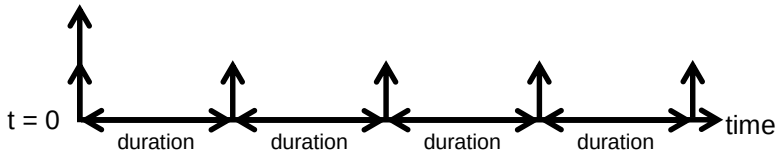


*Figure 9-4*
***every*** **duration**

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

This example defines the discrete time domain associated with a digital control system that has a so-called synchronous design: it sees its environment and acts upon it only at the instants defined by the *td* time domain. The period is defined by *cycleTime* (50 milliseconds). The time domain has a random phase defined by *phase*.

## 9.2. Combining & Transforming DTLs

FORM-L offers various means for deriving new DTLs from already existing ones.

| **dtl$_1$** ***or*** **dtl$_2$**
The instants of the resulting DTL are the union of those of dtl$_1$ and dtl$_2$ those of. If a dtl$_2$ instant occurs at the same time as a dtl$_1$ instant, only one instant is accounted (see Figure 9-5).

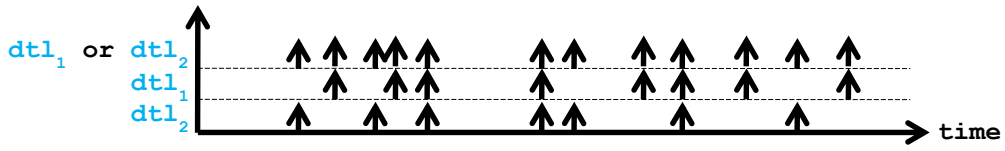

*Figure 9-5*
**dtl₁ *or* dtl₂**

| **dtl₁ *and* dtl₂**
The instants of the resulting DTL are obtained by selecting only those of dtl₁ occurring at the same time as a dtl₂ instant (see Figure 9-6).

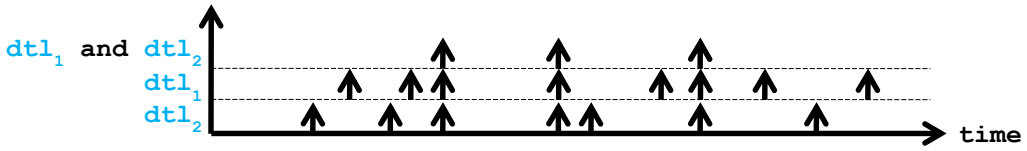


*Figure 9-6*
**dtl₁ *and* dtl₂**

| **dtl₁ *excl* dtl₂**
The instants of the resulting DTL are obtained by selecting only those of dtl₁ not occurring at the same time as a dtl₂ instant (see Figure 9-7).

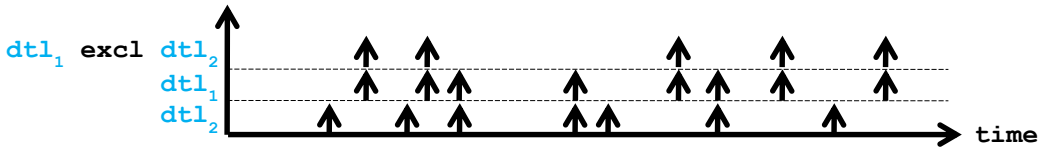


*Figure 9-7*
**dtl₁ *excl* dtl₂**

| **dtl₁ *xor* dtl₂**
This is equivalent to *(*dtl₁ *excl* dtl₂*) or (*dtl₂ *excl* dtl₁*)* (see Figure 9-8).

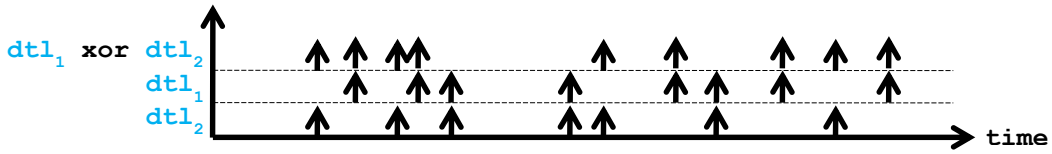


*Figure 9-8*
**dtl₁ *xor* dtl₂**

| **dtl₁ *selecting* dtl₂**
The *selecting* operator selects the dtl₂ instants that are simultaneous to, or that immediately follow, a dtl₁ instant. If between two dtl₁ instants there are no dtl₂ instant, then no dtl₂ instant will correspond to the first of these dtl₁ instants (see Figure 9-9).

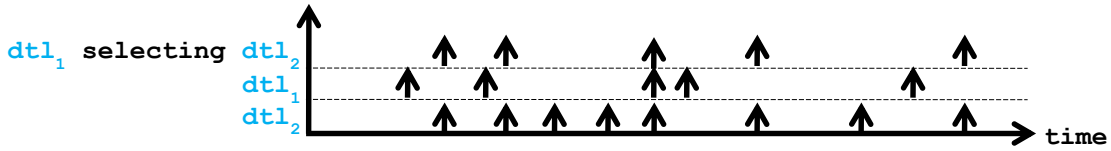


*Figure 9-9*
**dtl₁ *selecting* dtl₂**

| ***first* dtl**
| ***first*(n) dtl**
This selects the first or the n first instants of dtl (see Figure 9.8). n must be a strictly positive constant or parameter Integer.

| ***dropFirst* dtl**
| ***dropFirst*(n) dtl**
This selects all the instants of *dtl* except the first or the n first ones (see Figure 9-10). n must be a

strictly positive constant or parameter Integer.

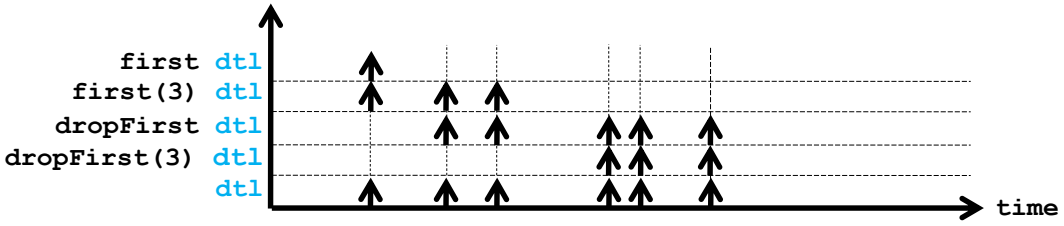


*Figure 9-10*
***first, dropFirst***

| **dtl *trunc* (duration | clockDuration)**
This selects all the instants of dtl occurring after a specified duration after the first instant (see Figure 9-11). The duration is evaluated at the first instant of dtl. It can be expressed either as a physical time duration or as a number of clock ticks (clockDuration, to be used only in a discrete time domain).

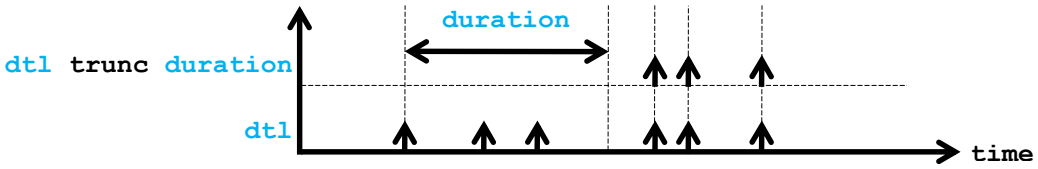


*Figure 9-11*
**dtl *trunc* duration**

| **dtl + (duration | clockDuration)**
This delays all the instants of dtl by a specified duration (see Figure 9-12). The duration is evaluated at each instant of dtl. It can be expressed either as a physical time duration (to be used only in the continuous time domain) or as a number of clock ticks (clockDuration, to be used only in a discrete time domain).

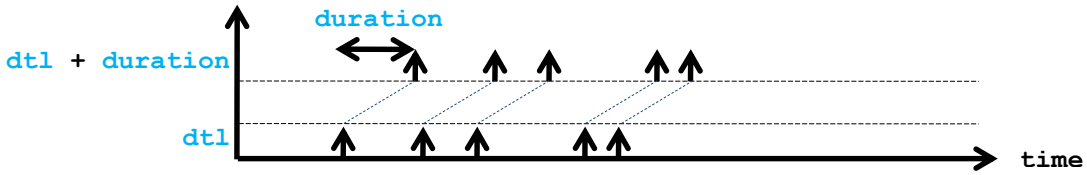


*Figure 9-12*
**dtl + duration**

| **dtl *and* ctl**
| **ctl *and* dtl**
These select all the instants of dtl that occur during a time period of ctl (see Figure 9-13).

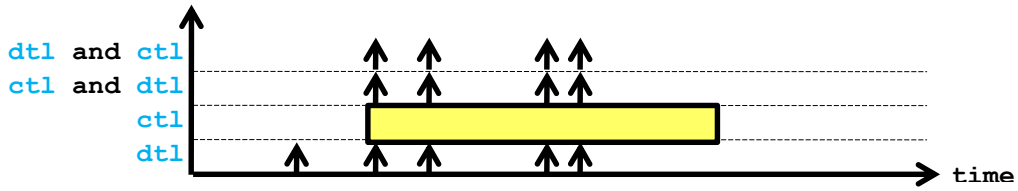


*Figure 9-13*
**dtl *and* ctl**

| ***randomAfter* dtl *within* (duration | clockDuration)**
This selects a random instant within each time period beginning with an instant of dtl and lasting the specified duration (see Figure 9-14). There is a one-to-one correspondence between the instants of the original dtl and those of the resulting DTL. The duration is evaluated at the beginning of each

period. It can be expressed either as a physical time duration or as a number of clock ticks (clockDuration, to be used only in a discrete time domain).

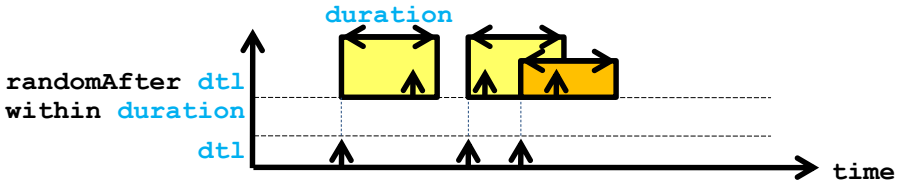


*Figure 9-14*
***randomAfter*** **dtl** ***within*** **duration**

```
timeDomain quasiPeriodic
   constant Duration cycleTime = 50*ms;
   constant Duration jitter = 1*ms;
   private parameter Duration phase = random(0.0, cycleTime);
   clock = (randomAfter every cycleTime within jitter) + phase;
   end quasiPeriodic;
```

This example refines the previous example of synchronous discrete time domain by introducing a "jitter": the resulting time domain is "quasi periodic".

```
;
```

# 10. Sliding Time Windows (STW)

A sliding time windows (or STW) of a given duration designates any time period of that duration (see Figure 10-1). In a discrete time domain, the duration must be specified as a number of clock ticks.

```
stw = duringAny (duration | clockDuration);
```

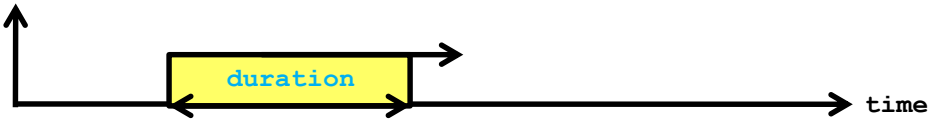


*Figure 10-1*
***duringAny*** **duration**

```
property p = duringAny 6*s check off;
```

Property *p* is satisfied at the end of each 6 second time window during which Boolean *off* has been continuously true. Conversely, it is violated at the end of a 6 second time window where *off* has not remained continuously true.

# 11. Continuous Time Locators (CTL)

A CTL defines one or more time periods that have a strictly positive duration. Time periods may overlap. In some cases, it is important to be able to specify whether the beginning or ending boundary belongs or not to a period. As a general rule:

- With keywords *always*, *during* and *duringAny*, both boundaries belong to the period.
- With keyword *after*, the beginning boundary does not. With keyword *from*, it does.
- With keywords *for* and *within*, the ending boundary belongs to the period. With keyword *before*, it does not.

```
ctl =

   (ctl)
```

## 11.1. Basic DTLs

| ***always***
If no time locator is specified in a property, then the default time locator is *always*, a single time period covering the complete time (see Figure 11-1).

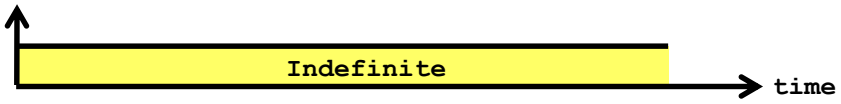


*Figure 11-1*
***always***

| ***during*** bool
The time periods defined are those where the bool condition is true (see Figure 11-2).

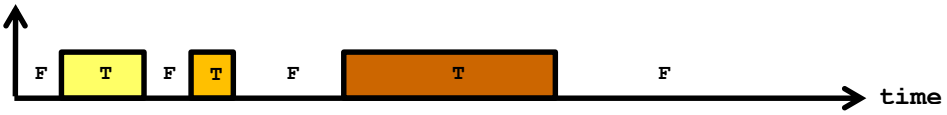


*Figure 11-2*
***during*** bool

```
requirement r =
   during (not needed) and tolerance check not active;
```

In this example, the CTL is *during (not needed) and tolerance*, the Boolean condition being *(not needed) and tolerance*. This requirement specifies that when the system is in a fault tolerance condition, it must not be spuriously activated.

| (***after*** | ***from***) event
The time periods defined begin with each occurrence of the event (possibly subject to a filtering Boolean condition) and last until the end of time (see Figure 11.3). There are as many periods as there are event occurrences: if there are more than one occurrence, they will overlap. If one does not wish to have overlapping periods, one can use the *after first event* idiom, where the CTL is based only on the first occurrence of event. An alternative approach is to use the *flat after event* idiom, where overlapping periods are merged into a single one.

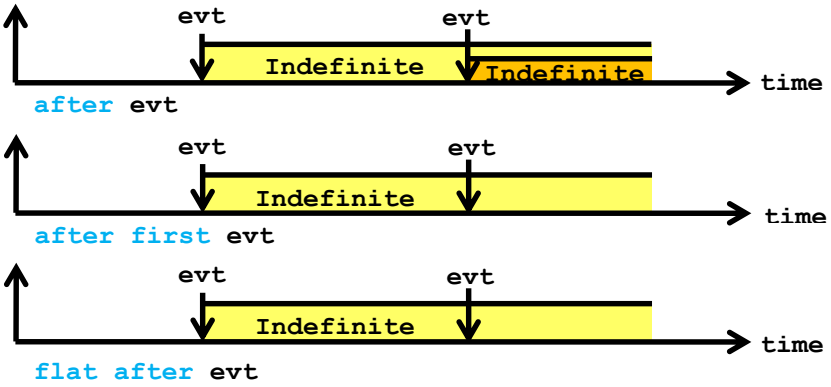


*Figure 11-3*
***after*** event

| **(*after* | *from*)** event₁ **(*until* | *before*)** event₂

A time period begins with each occurrence of event₁ (possibly subject to a first filtering Boolean condition) and lasts until the first strictly following occurrence of event₂ (also possibly subject to a second filtering Boolean condition). In the case where there are several occurrences of event₁ before the next occurrence of event₂, all the corresponding time periods are closed by that occurrence of event₂ (see Figure 11-4).

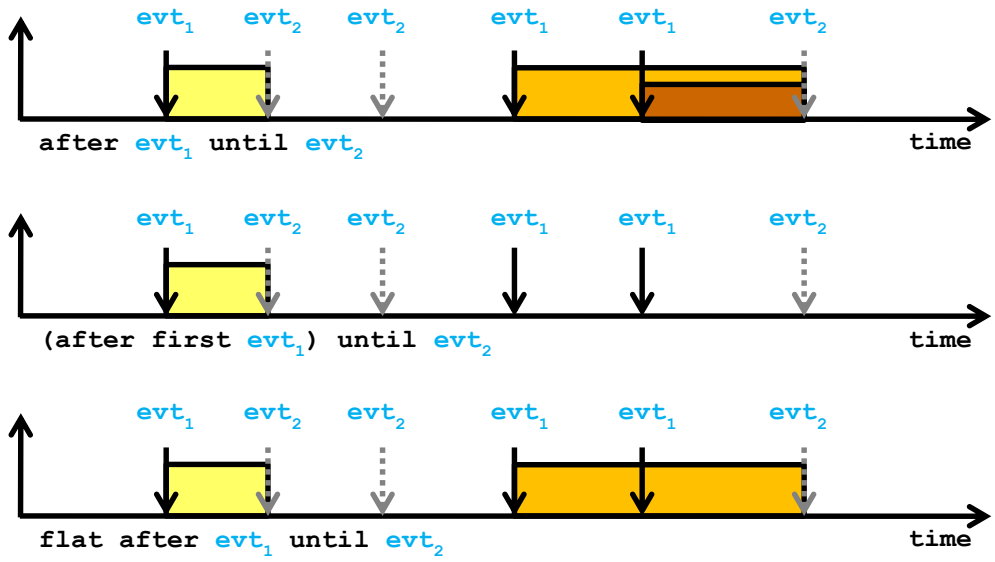


*Figure 11-4*
***after*** event₁ ***until*** event₂

The first filtering condition can only refer to features of event₁. The second filtering condition can refer to features of both event₁ and event₂. If the two refer to successive occurrences of the same event, aliases may be used to distinguish the occurrences.

If one does not wish to have overlapping periods, one can use the *after first(event₁) until event₂* idiom, where the CTL is based only on the first occurrence of *event₁*. The *flat(after event₁ until event₂)* idiom merges the overlapping periods into a single one. Note that the two are not equivalent.

```
Boolean running = after eStart until eStop;
```

This statement concerns the motor of previous examples. It can signal several events, including *eStart* (it has started) and *eStop* (it has stopped). The statement defines the time periods where the motor is *running*. Note that a CTL can be converted into a Boolean.

| **(*after* | *from*)** event **(*for* | *within* | *before*]**
**(duration | clockDuration | durationExpr)**

The time periods defined begin with each occurrence of the event (possibly subject to a filtering Boolean condition) and last for the specified duration (see Figure 11-5, which also illustrates how various idioms could be used to precisely determine the desired CTL).

Keyword *for* has exactly the same meaning as keyword *within*, but whereas *for* is mainly used for condition-based properties (the condition must be true *for* a certain time period), *within* are mainly intended for event-based properties (an event must occur *within* a certain delay).

The duration can be expressed as a physical time duration, as a number of clock ticks (clockDuration, to be used only in a discrete time domain), or as a duration expression (durationExpr).

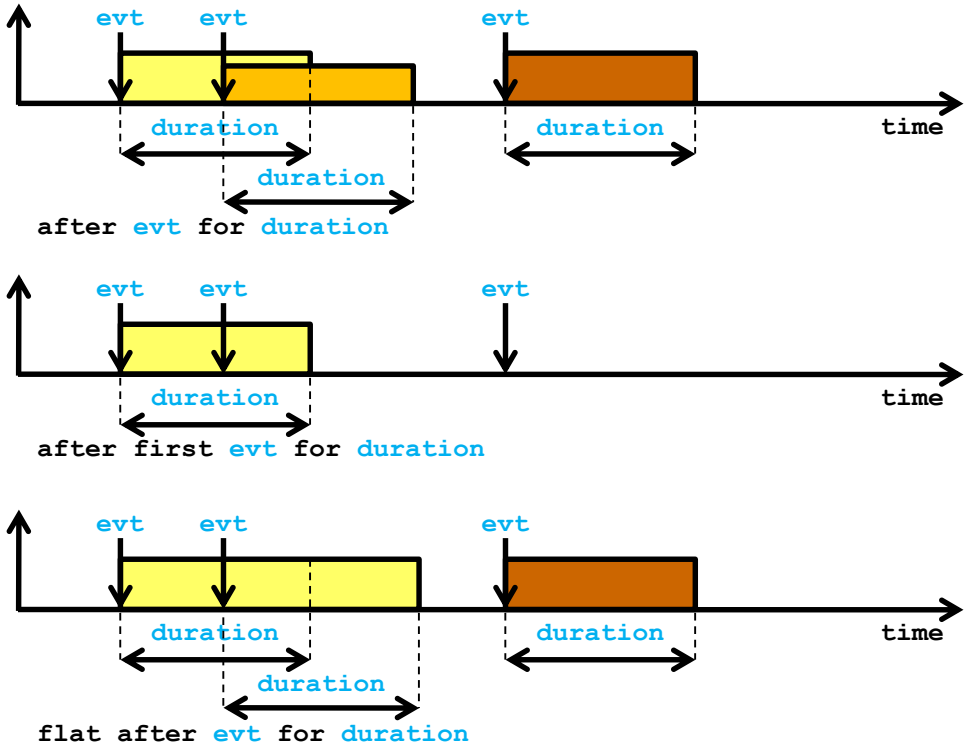


*Figure 11-5*
***after*** event ***for*** duration

```
assumption
   after powered for settleTime check demand ≤ initialPower;
```

This statement makes an assumption on the demand for power of an electrical component just after it has been connected to power (*powered*). It states that for a duration of *settleTime* after connection to power, the demand is less or equal to *initialPower*.

| ***(until | before)*** event

The time period defined starts at the beginning of time and ends at the first occurrence of the event, possibly subject to a filtering Boolean condition (see Figure 11-6).

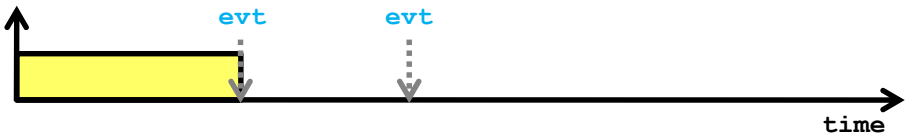


*Figure 11-6*
***until*** event

| ***every*** duration₁**(fixed)** [***for*** | ***within***] duration₂**(fixed)**
| ***every*** clockDuration₁**(fixed)** [***for*** | ***within***] clockDuration₂**(fixed)**
These expressions define periodic time periods (see Figure 11-7). Durations are expressed either as physical time durations (for duration₁, to be used only in the continuous time domain), or as numbers of clock ticks (clockDurations, to be used only in a discrete time domain). Both durations must be fixed values. If duration₂ is longer than duration₁, there will be time period overlaps.

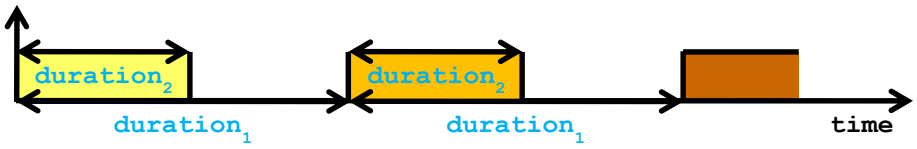


*Figure 11-7*
***every*** duration₁ ***for*** duration₂

```
requirement every 10*s within 2*s check eCheck;
```

This statement requires that periodically, every 10 seconds, event *eCheck* should occur in the first 2 seconds.

## 11.2. Combining & Transforming CTLs

| ***not*** ctl
The resulting CTL is composed of the time periods not covered by the time periods of the original ctl (see Figure 11-8). 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*.

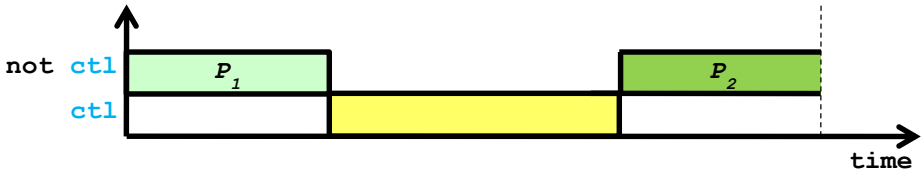


*Figure 11-8*
***not*** ctl

| ***flat*** ctl
In the resulting CTL, the overlapping time periods of the original ctl are merged. Time instants not covered by the original ctl are also not covered by the resulting CTL (see Figure 11-9).

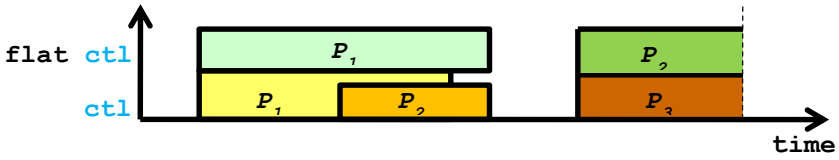


*Figure 11-9*
***flat*** ctl

| **ctl + (duration | clockDuration)**
The resulting CTL is composed of the time-shift of each of the individual time periods of the original ctl (see Figure 11-10). The duration of the time-shift is evaluated at the beginning of each of the time periods to be shifted and must be positive. It can be expressed either as a physical time duration (to be used only in the continuous time domain) or as a number of clock ticks (clockDuration, to be used only in a discrete time domain).

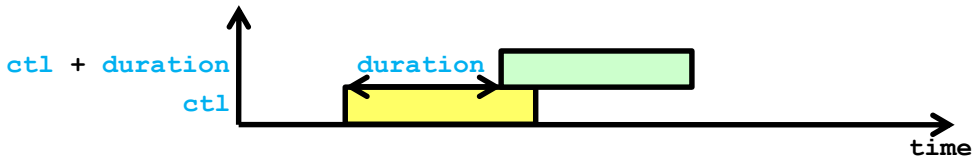


*Figure 11-10*
**ctl + duration**

| **ctl *trunc* (duration | clockDuration)**
The resulting CTL is composed of the time-truncation of each 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 11-11). The duration of the truncation is evaluated at the beginning of each of the time periods to be truncated and must not be negative. It can be expressed either as a physical time duration or as a number of clock ticks (only in a discrete time domain).

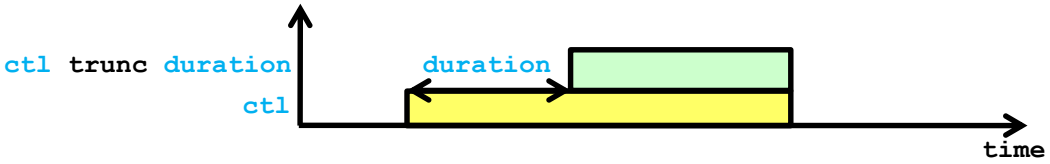


*Figure 11-11*
**ctl *trunc* duration**

| **$ctl_1$ *or* $ctl_2$**
The time periods of the resulting CTL are the union of those of $ctl_1$ and $ctl_2$ (see Figure 11.12).

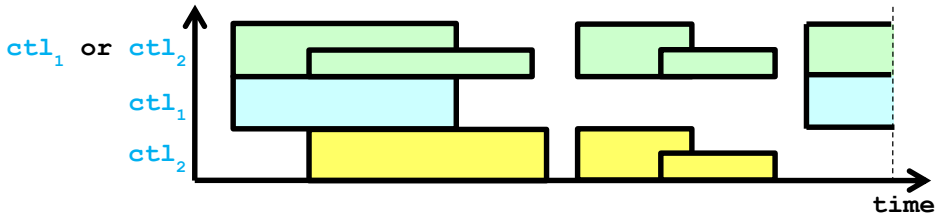


*Figure 11-12*
**$ctl_1$ *or* $ctl_2$**

| **$ctl_1$ *and* $ctl_2$**
The time periods of the resulting CTL are obtained by excluding from each time period of $ctl_1$ any instant not belonging to $ctl_2$ (see Figure 11-13). Empty periods are eliminated. It is to be noted that *$ctl_1$ and $ctl_2$* is not necessary equal to *$ctl_2$ and $ctl_1$*.

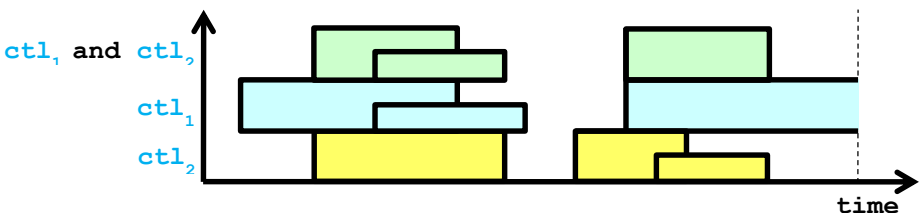


*Figure 11-13*
**$ctl_1$ *and* $ctl_2$**

| $ctl_1$ ***excl*** $ctl_2$
The time periods of the resulting CTL are obtained by excluding from each time period of $ctl_1$ any instant belonging to $ctl_2$ (see Figure Figure 11-14). Empty periods are eliminated.

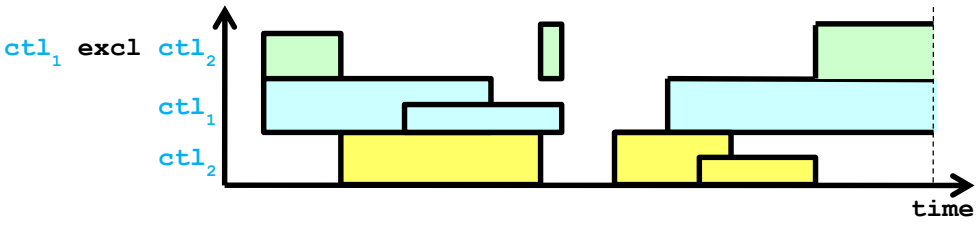


*Figure 11-14*
$ctl_1$ ***excl*** $ctl_2$

| $ctl_1$ ***xor*** $ctl_2$
This is equivalent to *($nsCTL_1$ excl $nsCTL_2$) or ($nsCTL_2$ excl $nsCTL_1$)* (see Figure 11-15).

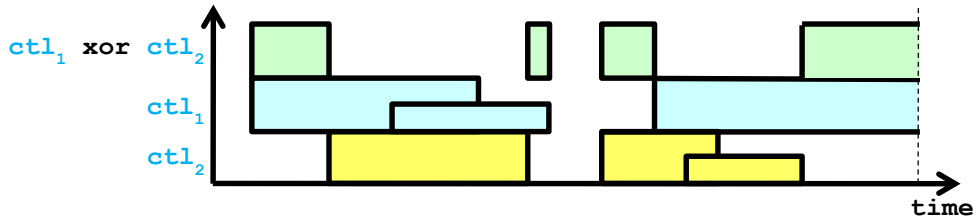


*Figure 11-15*
$ctl_1$ ***xor*** $ctl_2$

# 12. Spatial Locators

The role of a spatial locator is to scan a set, and possibly apply a filtering criterion. The general form is:

```
sl = forAll name in set [suchThat bool(name)];
```

The Boolean condition bool (which might be time-dependent) defines the filter. It usually mentions the associated scanning *name*. When several sets are concerned, one can apply the spatial locator to their Cartesian product, and *name* then designates the t-uples that satisfy the condition.

# 13. Properties

A **property** is a combination of an optional time locator, an optional spatial locator and a constraint verification, called a **check**. When no time locator is specified, CTL *always* is assumed. When no spatial locator is specified, all the items concerned must be explicitly named. In principle, the time locator and the spatial locator do not interfere with one another and may specified in any order. The property is evaluated at the times specified by the time locator, on the items specified by the spatial locator.

```
propertyDecl = (property | requirement | assumption | guard) name; ;

propertyDef = [private] (
   (property|requirement|assumption|guard) name = [sl] [dtl|stw|ctl] check;
```

This defines a non-probabilistic property.

```
| (requirement|assumption) name =
      [ctl] checkSpecial probability event (< | ≤ | ≥ | >) real;
```

Property constraining the probability of event occurring at least once during the specified time locator should satisfy the specified constraint. real must be a dimensionless Real number in the [0., 1.] range. This may be used to express conditional probabilities (then the time locator specifies the condition, and real is usually a fixed value) or probabilities in continuous modes of operation (then the time locator specifies the operation times, and real is usually a variable).

**);**

Note. The verdict for a probabilistic property cannot be based on a single operational case, and needs to be judged based on statistical analyses of the outcomes of a sufficient number of operational cases, or on specific probabilistic analyses. Such analyses are out of the scope of this document, and term *check* as used here does not include probabilistic verifications.

Ultimately, the check for a non-probabilistic property is expressed as a Boolean. However, the algorithm for deciding the status of a property under a CTL depends on the nature of its check. Thus, different types of checks have been identified:

**check =**

***check*** **bool**
This specifies that Boolean condition bool must be satisfied at the instants or during the periods of the time locator.

| ***checkSpecial inPDuration*****(bool) (< | ≤ | = | ≠ | ≥ | >) duration**
This check is not valid under a DTL. It specifies that for each time period of the time locator, the in-period duration where Boolean condition bool is true must satisfy the constraint with respect to duration, which must be strictly positive.

| ***checkSpecial inPClock*****(bool) (< | ≤ | = | ≠ | ≥ | >) int**
This check is not valid under a DTL and in the continuous time domain. It specifies that for each time period of the time locator, the in-period count of clock ticks where Boolean condition bool is true must satisfy the constraint with respect to int, which must be strictly positive.

| ***checkSpecial inPCount*****(event) (< | ≤ | = | ≠ | ≥ | >) int**
This check is not valid under a DTL. It specifies that for each time period of the time locator, the in-period count of the occurrences of event must satisfy the constraint with respect to int, which must be strictly positive.

In addition to the general forms, some shortcut idioms may be used:

| ***check*** **event**
This check is not valid under a DTL. It specifies that the event must occur at least once during each period of the time locator. This is equivalent to *checkSpecial inPCount(event) ≥ 1*

| ***check once*** **event**
This check is not valid under a DTL. It specifies that the event must occur once and only once during each period of the time locator. This is equivalent to *checkSpecial inPCount(event) = 1*

| ***check no*** **event**
This check is not valid under a DTL. It specifies that the event must not occur during any period of the time locator. This is equivalent to *checkSpecial inPCount(event) = 0*

**;**

## 13.1. Properties Attributes

One main goal of properties modelling and analysis or simulation is to determine whether the specified properties (and most particularly, requirements) have been tested or not, and then violated or not.

In time, the instantaneous state of a non-probabilistic check is represented by an automaton (named *instantState*) that can be in one of three states:

- *notTestedNow* (initial state).
- *satisfiedNow*.
- *violatedNow*.

The cumulated state of the check is represented by another automaton (named *cumulState*) that can also be in one of three states:

- *notTested* (initial state).
- *satisfied*.
- *violated*.

Note 1. *satisfied* here does not mean that the check is satisfied in all cases. It does not mean either that it will not be violated later: it just means that it is satisfied by the current case, up to now. In the same manner, *satisfiedNow* just means that it is satisfied by the current case at the current instant. It might have been violated earlier in the case, and might be violated later in the case.

Note 2. As with other states of finite state automata, six Boolean attributes (bearing the names of the states) are attached to a check, determining whether the check is in that state or not.

Thus, one can write:

```
requirement r1 = after evt for 5*s check condition;
event e1 = when r1.violated becomes true;
```

And with a composite instruction:

```
after evt for 5*s sequence
   r1a: within 2*s check once evt2;
   r1b: check condition;
   end;
event e2 = when r1b.violated becomes true;
```

# 14. Coordination

Variable, event and property definitions specify precisely placed (in time and in space) effects, using the notions of snippets, time locators and possibly spatial locators. Sometimes, the effects concerning different items need to be tightly coordinated in time. Segregating the specification of such effects in closed definitions not only complicates the specification, it also hides the coordination from view. To make coordination explicit, FORM-L has the notions of **statement**, **instruction**, **relative time locator**, and **coordination**.

## 14.1. Statements

A **statement** is a combination of an optional time locator, an optional spatial locator and an instruction. When no time locator is specified, CTL *always* is assumed. When no spatial locator is specified, all the items concerned must be explicitly named. In principle, the time locator and the spatial locator do not interfere with one another and may specified in any order. The instruction is performed at the times specified by the time locator, on the items specified by the spatial locator.

```
statement = [sl] [tl] instruction;;
```

## 14.2. Elementary Instructions

An instruction is either an **elementary instruction** or a **composite instruction**.

```
instruction = (elementaryAction | compositeAction);
```

There are four types of elementary instructions:

- A **check** specifies a verification. It may be performed at specific instants or during time periods. Checks must be uniquely named.
- A **value definition** for a variable specifies its value at specific instants or during time periods. The notion of open definition allows placing some of the value definitions for a variable anywhere in the model.

- An event **signalling** specifies instants when the event occurs. It is always instantaneous. The notion of open definition allows placing some of the signallings for an event anywhere in the model.
- A **stop** signals the end of time, when the operational case has reached its conclusion. It is always instantaneous and occurs only once.

```
elementaryAction = (name: check) | valueDef | signalling | stop;
```

A statement the instruction of which is a value definition constitutes a definition snippet for a variable. A statement the instruction of which is an event signalling constitutes a definition snippet for an event. A statement the instruction of which is a check is constitutes a property bearing the name of its check.

## 14.3. Relative Time Locators

In a composite instruction, a relative time locator may be associated with a member instruction so that the member instruction can be precisely placed in time either with respect to an absolute time locator (a CTL or a DTL), or to other member instructions. The relative time locator specifies an optional **delay** and an optional **time span**:

```
rtl = delay then | timeSpan | delay then timeSpan;
```

When the instruction associated with a relative time locator is initiated:

- The delay, if any, is first initiated.
- At the end of the delay, the instruction is then taken into account.
- If no time span is specified, checks and value definitions are instantaneous instructions.
- If no time span is specified and the instruction is composite, the instruction is taken into account without any duration constraint other than those already imposed by the context.
- If a time span is specified and the instruction is composite, the instruction is taken into account only for the duration of the time span. It means that it can be aborted if it lasts longer. If it is shorter, there will be an additional delay to complete the specified duration.

### 14.3.1.Duration Expressions

```
duration = real(time);
```

A duration represents a length of physical time. It is expressed as an instance of Duration (derived from Real with a "time" dimension).

```
clockDuration = int(clockTime);
```

A clockDuration represents a length of time as a number of clock ticks. It should be used only in a discrete time domain.

```
durationExpr = [timeSpan | shortestSpan | longestSpan] (
   duration
|  clockDuration
|  until  event
|  durationExpr
;)2+ end
;
```

A duration expression is a list composed of physical time durations, of numbers of clock ticks (clockDurations, to be used only in a discrete time domain), of time periods lasting until an event occurs (this includes events such as *bool(inP) becomes true*), and of embedded duration expressions. There are three types of duration expression. With the first one (keyword *timeSpan*), the duration elements of the list are evaluated one after the other, in the order of the list. With the second one (keyword *shortestSpan*), the duration elements of the list are evaluated in parallel, and the duration expression terminates as soon as one of the duration elements terminates. With the third one (keyword *longestSpan*), the duration elements of the list are evaluated in parallel, and the duration expression terminates as soon as all the duration elements have terminated.

### 14.3.2.Delays

In a discrete time domain, the evaluation of what follows the delay is evaluated at the clock tick immediately following the expiration of the delay.

```
delay =
```

***wait*** **duration**
The duration of the delay is expressed in physical time and is evaluated when the delay is initiated. This type of delay can be used only in the continuous time domain.

| ***wait*** **clockDuration**
The clockDuration of the delay is expressed as a number of clock ticks of a discrete time domain and is evaluated when the delay is initiated. This type of delay can be used only in a discrete time domain.

| ***wait*** **event**
The delay lasts until the next occurrence of the event.

| ***wait*** **durationExpr**
The delay lasts until the durationExpr terminates.

```
;
```

### 14.3.3.Time Spans

```
timeSpan =
```

[***for*** | ***within***] **duration**
The duration of the time span is expressed in physical time and is evaluated when the time span is initiated. This type of time span can be used only in the continuous time domain.

| [***for*** | ***within***] **clockDuration**
The clockDuration of the time span is expressed as a number of clock ticks of a discrete time domain and is evaluated when the time span is initiated. This type of time span can be used only in a discrete time domain.

| ***until*** **event**
The time span lasts until the next occurrence of the event.

| [***for*** | ***within***] **durationExpr**
The time span lasts until the durationExpr terminates.

```
;
```

## 14.4. Composite Instructions

A composite instruction coordinates multiples member instructions. A member instruction can be an elementary instruction or another composite instruction. FORM-L proposes five types of composite instructions:

- **Sequential instructions**
- **Concurrent instructions**
- **Iterative instructions**
- **Conditional instructions**
- **Procedure calls**

```
compositeAction = sequence | simultaneous | iterative | conditional;
```

When a statement specifies a composite instruction, its time locator controls the execution as follows:

- If the time locator is a DTL, the instruction is initiated at each instant of the DTL and is executed to completion. This means executions may overlap.
- If the time locator is a CTL, the instruction is initiated at the beginning of each period of the CTL.

If at the end of the period the execution is not complete, it is aborted. Executions may overlap if the time periods overlap.

As the time locator of the statement controls globally the composite instruction, individual member instructions may need their own **relative time locator**. The overall position in time of a member instruction is determined by the type of the composite instruction to which it belongs.

As already mentioned, event signallings and stops are always instantaneous. Checks and value definitions can be instantaneous or not. A composite instruction is instantaneous if and only if it is composed only of instantaneous member instructions not specifying any delays.

### 14.4.1.Sequential Instructions

The member instructions of a sequential instruction (or **thread**) are performed one after the other, possibly with a relative time locator ahead of each member instruction (see Figure 14-1).

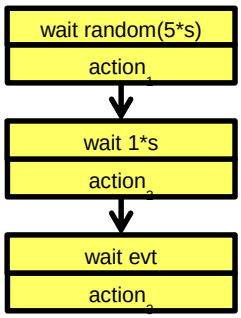


*Figure 14-1*
*Example of sequential instructions.*

The general form of a sequence is:

```
sequence = sequence ([rtl] instruction;)* end;
```

### 14.4.2.Concurrent Instructions

The member instructions of a concurrent instruction are all initiated in parallel when the concurrent instruction is initiated. The concurrent instruction terminates as soon as each of its member instructions has terminated (see Figure 14-2).

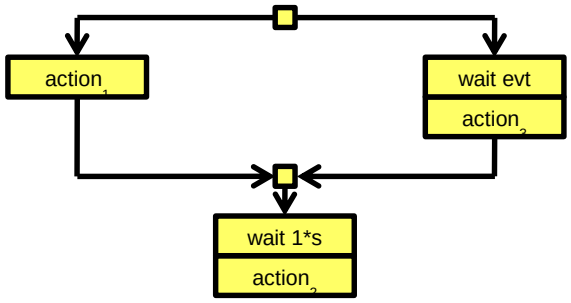


*Figure 14-2*
*Example of parallel instructions*

The general form of a parallel instruction is:

```
simultaneous = simultaneous (
   [rtl] instruction;
|  [rtl(p)] instruction(p) sl(p);
```

In this case, spatial locator sl (see Section 12 Spatial Locators) identifies a number of items p, and the instruction is performed for each p, possibly after relative time locator rtl, which may also depend on p.

)* *end*;

### 14.4.3.Iterative Instructions

The member instruction of an iterative instruction is sequentially executed multiple times. There are four kinds of iterations.

**iterative =**

***repeat*** **[rtl] instruction** ***forever***
In this case, instruction is sequentially repeated until the end of time or until abortion occurs, which ever comes first.

| ***repeat*** **[rtl] instruction** ***for*** **integer** ***times***
In this case, instruction is sequentially repeated the specified number of times, unless abortion occurs. The number is evaluated when the iterative instruction is initiated.

| ***repeat*** **[rtl] instruction** ***while*** **bool**
In this case, instruction is sequentially repeated while the condition is true, unless abortion occurs.

| ***repeat*** **[rtl] instruction** ***until*** **event**
In this case, instruction is sequentially repeated until the next occurrence of event, unless abortion occurs.

;

### 14.4.4.Conditional Instructions

**conditional =**
***if*** **bool$_1$** ***then*** **[rtl$_1$] action$_1$**
**[*elseIf* bool$_i$ *then* [rtl$_i$] action$_i$]+**
**[*else* [rtl$_n$] action$_n$]**
In this case, the Boolean conditions are evaluated one after the other when the conditional instruction is initiated, as specified by the left-right order, until one evaluates to true. Its associated instruction is then performed. If no condition evaluates to true, and if an *else* clause is provided, then the instruction of the else clause is performed. Else, no instruction is performed.

| ***switch*** **scalar(x)**
**{*case* domain$_i$(x) *then* [rtl$_i$] action$_i$;}+**
**[*else* [rtl$_n$] action$_n$]**
Choice of one instruction among several, based on successive tests on scalar(x).

;

## 14.5. Procedures & Procedure Calls

***Procedures***

A **procedure** is an instruction template that can be based on a number of arguments. The procedure is characterised by its signature, composed of its pathname and of the ordered list of its arguments categories. In a given name space, different procedures may have the same name, but their signatures must be different. Argument values are provided whenever the procedure is called: then, they must match the category specified by the signature. A signature may specify that the procedure can take a variable number of arguments (using keyword *etc*): any additional argument beyond the signature must then match the last argument category of the signature.

The notion of procedure serves two main purposes:

- It is first an abstraction mechanism, to organise a complex sequence of instructions into

meaningful and more easily understandable subsets.

- It is also a reuse mechanism, when the same pattern of instructions can be used in different models, at different times and / or at different places, possibly with variations thanks to the arguments.

***Procedure Definitions***

A procedure definition specifies a procedure signature, i.e., its name and an optional argument list, each argument being characterised by a name and a category. All the argument names in a procedure definition must be different. The definition also specifies an instruction to be performed when the procedure is called. In principle, the instruction depends on the arguments, if any. Keyword *etc* at the end of the argument list specifies that a call to the procedure may provide more arguments than there are in the signature. Any supplementary argument must match the last category in the signature. In the expression of the instruction, one gets access to these arguments through a vector bearing the name of the last argument in the signature, and the elements of which being the argument corresponding to that name and all supplementary arguments, in the order provided by the call. Its size can be obtained through the *size* attribute of the vector.

```
procedureDef = procedure name [({(category argumentName),}+ [,etc])]
   instruction;
   end name;;
```

***Procedure Calls***

A procedure call activates the procedure. It specifies the name of the procedure, and possibly a list of argument values. The name of the called procedure and the number and the categories of the values must match with one and only one procedure signature, and thus one and only one procedure definition.

There are two styles of procedure calls. With the first one, the argument values are provided in the form of a parenthesised, comma-separated list. The matching of the call with a procedure signature, and the association of each argument value with an argument name, are based only on the position of the argument value in the list and its category. With the second style, each argument value in the call is explicitly associated with an argument name. The argument values can be provided in any order, except for any supplemental argument value, which must be placed at the end of the list.

```
procedureCall =
   path(procedure) [({expr,}+)]
|  path(procedure) [({argumentName = expr,}+ {expr,}*)]
;
```

```
when needed then sequence
   within 0.5*s check eStartMotor;
   wait ready then within 0.5*s check eOpenBrk;
   simultaneous
      wait 1*s then check brk.open;
      wait 5*s then within 0.5*s check eProceed;
      end;
   end;
```

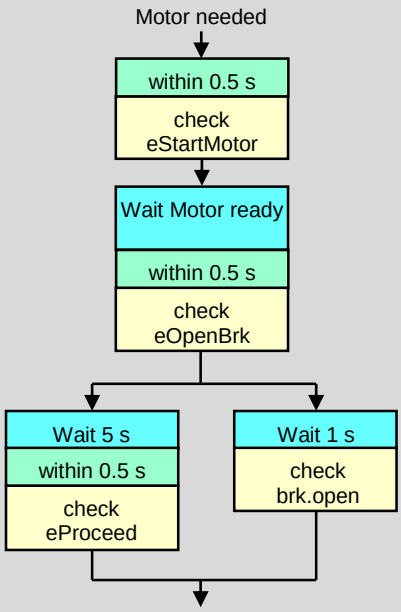


*Figure 14-3*
*Example of composite instruction*

This statement is related to the motor example. It specifies the following instructions (see *Figure 14-3*):

- When the motor becomes *needed*, then within 0.5 seconds an order to start the motor should be issued. In this model, this is represented by event *eStartMotor*.
- After the motor becomes *ready*, then within 0.5 seconds an order to open circuit breaker *brk* should be issued. In this model, this is represented by event *eOpenBrk*.
- Within 1 second after event *eOpenBrk*, the state of *brk* should become *open*.
- Between 5 and 5.5 seconds after event *eOpenBrk*, an order to proceed should be issued. In this model, this is represented by event *eProceed*.

# 15. Time Domains

In FORM-L, a discrete time domain is declared as follows:

```
timeDomainDecl = [external} timeDomain name;;
```

A discrete time domain is defined as follows:

```
timeDomainDef = timeDomain name
   (varDef)*
   ((clock = dtl;) | statement+)
   end name;;
```

```
timeDomain pseudoPeriodic
   constant Duration cycleTime = 50*ms;
   constant Duration jitter = 1*ms;
   private parameter Duration phase = random(0*ms, cycleTime);
   sequence
      wait phase;
      repeat
         wait cycleTime - jitter/2.0 + random(0*ms, jitter) then raise clock;
         forever;
      end;
   end pseudoPeriodic;
```

This example is another refinement of the synchronous example, where the time interval between two clock ticks is slightly variable and generate a drift.

# 16. Models Organisation

The notions presented in this section (property and non-property models, object models, libraries, bindings, contracts and configurations) have been defined to promote models modularity and composability, which are both necessary when addressing complex systems and multidisciplinary approaches.

## 16.1. Property Models

A property model is composed of declarations and definitions for all types of FORM-L items, except those related to models organisation. In addition, it can import the non-private declarations and definitions made in other models or in libraries (see Section 16.2 *Libraries*), and the terms specified by contracts (see Section 16.5 *Contracts*).

```
declaration = (
   varDecl           // Variable declaration
|  eventDecl         // Event declaration
|  objectDecl        // Object declaration
|  functionDecl      // Function declaration
|  procedureDecl     // Procedure declaration
|  propertyDecl      // Property declaration
|  timeDomainDecl    // Time domain declaration
);;

definition = (
   typeDef           // Type definition
|  varDef            // Variable definition
|  eventDef          // Event definition
|  eventClassDef     // Event class definition
|  classDef          // Class definition
|  objectDef         // Object definition
|  functionDef       // Function definition
|  procedureDef      // Procedure definition
|  propertyDef       // Property definition
|  statementDef      // Statement definition
|  timeDomainDef     // Time domain definition
);;

pModel = objectDef | classDef | eventClassDef |
|  property model name
   (contractExtract | declatation | definition)+
   end name;
;
```

## 16.2. Libraries

A library is a restricted property model that only provides definitions. In particular, it cannot declare external items. The definitions can be reused in other property models. Typical examples would be libraries supporting the SI system of units, providing standard mathematical functions, or useful domain-specific functions. The syntax for libraries is the same as for property models, except that libraries should not allow declarations, and that keyword *library* is used instead of *property model*. When a library is included in a configuration, then all its definitions are made available to all the other FORM-L models of the configuration. They may refer to the defined items as if they had defined them themselves. However, if such a model defines an item of the same name, then the library item must be referred to with a pathname.

```
library = library name
   (definition)+
   end name;;
```

## 16.3. Behavioural Models

In FORM-L, a model that is not a property model is a reference to a model in another language (e.g., a Modelica model). It specifies the name under which the model is known in FORM-L, and the modelling language used, in the form of a constant String. The mapping between FORM-L notions and those provided by the language must be established (by means that are language and implementation dependent), so that items in the original model can be referenced as FORM-L items. If naming rules are different from those of FORM-L, a transposition convention also needs to be established.

```
bModel = behavioural model name stringLiteral;
```

## 16.4. Bindings

A binding takes information from one model (which could be a property model or not), transforms it if necessary, and transfers it to a property model.

The syntax for bindings is the same as for property models, except that keyword *binding* is used instead of *property model*, and that a binding requests **access** to the models it binds.

```
binding = binding name
   (access |declaration | definition)+
   end name;;
```

An access to a model specifies whether it is for **importation** (from the accessed model to the binding), or **exportation** (from the binding to the accessed model). Importation and exportation are restricted to non-private variables, objects and events. An import (resp. export) clause may be restrict the importation (resp. exportation) to specific items.

```
access =
   import name(model);
```

In this case, the binding imports and can read all non-private variables, objects and events of the model from which the import is made. It also imports all the type, class and event class definitions.

```
| export name(model);
```

In this case, the binding can export to (i.e., define the values of) all external variables, objects and events of the model to which the export is made.

```
| import name(model)
      {((type | class | event) name | categoryi pathi(item));}+
      end;
```

In this case, the binding imports only the items (non-private variables, objects and events) that are cited. For each item, the category mentioned in the importation clause must match the category in the model from which the import is made. It also imports the type, class and event class definitions that are cited.

```
| export name(model)
    {categoryi pathi(item);}+
    end;
```

In this case, the binding can export only to the items (external variables, objects and events) that are cited. For each item, the category mentioned in the exportation clause must match the category in the model to which the export is made.

```
;
```

Extensions to the notion of binding (e.g., binding to engineering databases through queries, generic bindings with arguments, etc.) could be introduced in the future.

## 16.5. Contracts

A **contract** is expressed in three parts:

- First, it identifies the **parties** of the contract, which must be FORM-L property models or objects (exclusively). Within the contract, they can be referred to either directly by their pathname (as determined by the configuration, see Section 16.6 *Configurations*), or indirectly via the notion of **role**.

  A contract may for example have a *client* and a *supplier*. The notion of role facilitates the reuse of the contract, when the same *supplier* has multiple *clients*

- Then, it identifies the **deliverables** of the contract, i.e., the items (variables, objects and events) that each party is responsible for (a party is responsible for an item when it provides a definition for it[1]) and provides to the other parties. A deliverable is under the responsibility of one and only one party.

- Lastly, it states the **obligations** and **rights** of each party. The obligations of a party are the requirements that it must satisfy, under the assumption (which constitutes its rights) that all the other parties will satisfy their own obligations. In other terms, a requirement for one party can be an assumption for another. Again, each property (requirement or assumption) must be defined by the parties concerned in a manner consistent with how it is declared in the contract.

Like in real life, a contract can be informal and be based just on mutual understanding: the concerned property models could simply include their part of the "moral "contract, from their own standpoint. However, also like in real life, there is a risk of misunderstanding, as different parties might have non-consistent views regarding some of the deliverables and / or obligations and rights. Ensuring consistency among all parties after modification might also be problematic. Thus, FORM-L provides support to ensure consistency (all parties do have the same understanding of the contract) and completeness (obligations and rights must be stated in terms of the deliverables, each deliverable is provided by one and only one party, each right from one party is matched by one obligation from another party).

Similar contracts (but with different parties) may be based on a **standard contract** that can be reused and instantiated in as many contracts as necessary. Instantiation of standard contracts is done in the framework of a configuration. Also, a (standard) contract can be extended (much like a class can be extended) by adding parties, deliverables, obligations and rights.

To facilitate understanding and the legibility of property models, a party in a contract may (but is not obliged to) include an **extract** of the contract, generally the part of the contract it is concerned with. Each extract must be consistent with the contract, which could be verified by supporting tools.

[1] A party can also be responsible for an item that it has declared external, if the definition of the item is provided to the party by a binding to models that are not parties in the contract.

The FORM-L syntax for contracts and standard contracts is the following:

```
contract = contract name
   [({nameᵢ(role | parameter),}+)] [extends nameₐ(contract)]
   (typeDef | classDef | eventClassDef)*;
   (party nameᵢ(propertyModel | objectModel | role)
      ( varDecl
      | objectDecl
      | eventDecl
      | (requirement | assumption) nameⱼ = [sl] [dtl | stw | ctl] check;
      | (requirement | assumption) nameₖ;
      end nameᵢ;
      )+
   end name;
;
```

where:

- name is the name of the contract.
- $name_a$ is the name of the extended contract.
- $name_i$ are the names of the roles and other parameters in a standard contract. If the list of roles is absent, then the contract is a regular contract, not a standard contract. When a standard contract is instantiated, a property model (belonging to the same configuration as the instantiation) is assigned to each role.
- A number of types, constants, classes and class events may be imported or defined to specify the categories of the i deliverables of the contract. These are shared by all parties.
- Each party (identified either as a role or directly by its name in the configuration) then declares the items for which it takes responsibility, and its obligations and rights.
- An item that is a deliverable of the contract is declared and named by one and only one party. All other parties must use the same name to refer to it.
- In the same spirit, a property (expressed as an obligation / requirement, or as a right / assumption) is defined and named by one and only one party. All other parties use that name to refer to it.
- Each property is viewed as a requirement by one and only one party. All other parties either ignore it or view it as an assumption.

For the syntax for instantiating standard contracts, see Section 16.6 *Configurations*.

The FORM-L syntax for a contract extract to be cited in a property model is the following:

```
contractExtract =
   contract name(contract) [as name₁(role)];
```

This just specifies that the property or object model is a party of the named contract. The role played by the model may be mentioned for information, but this is not strictly necessary since this is stated in the contract itself. The obligations and rights of the model are to be found in the contract.

```
|  contract name(contract) [as name₁(role)]
      ( varDecl
      | objectDecl
      | eventDecl
      | (property | requirement | assumption) nameⱼ =
          [sl] [dtl | stw | ctl] check;
      )+
      end name;
```

This similar to the previous case, but all the obligations and rights of the model are cited. These must be consistent with the terms of the contract, mutatis mutandis.

```
;
```

## 16.6. Configurations

A configuration is a set of models (property and object models, non-property models, libraries, bindings and contracts) that constitutes a significant whole that can be analysed, simulated or optimised. For a configuration, each of its models is identified by a pathname in the file system. A model in a configuration is known to the other models in the same configuration by a local name. By default, this is the name declared by the model definition. However, if there are name conflicts (two or more models, that by chance, have the same name), then they must be given a specific local name (an alias) to remove the ambiguity.

```
configuration = configuration name
   ( [reference] property model filepathi(property model) [alias namei];
   | [reference] object model filepathi(object model) [alias namei];
   | library filepathi(library) [alias namei];
   | behavioural model filepathi(behavioural model) [alias namei];
   | binding filepathi(binding) [alias namei];
   | contract filepathi(contract) [alias namei];
   | contract namea =
        nameb(standardContract) ({namei(role) = namej(property model),}+);
```

This is an instantiation named $name_a$ of a standard contract named $name_b$, where each role of the standard contract is explicitly associated with a property model in the configuration. The order in which the associations are made is not significant.

```
   | contract namea = nameb(standardContract) ({namej(property model),}+);
```

This is an instantiation named $name_a$ of a standard contract named $name_b$, where each role of the standard contract is associated with a property model in the configuration. Since the role names are not given, the order is significant, and must be the one of the standard contract definition.

```
   )+
   end name;;
```

# 17. Conclusion

This document is a preliminary proposal for the FORM-L. It has been developed based on a few case studies. Further steps will be following:

- Other case studies should be developed to confirm that the notions presented here are appropriate and sufficient to specify properties in a legible, understandable manner.
- Joint work with tool developers should aim at verifying that the final, concrete syntax is appropriate and parsable, and that the semantics are unambiguous and implementable.

# 18. References

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

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

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

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

[5] SPEEDS: Contract Specification Language (CSL)

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

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

[8] Duriaud Y., “EuroSysLib – Deliverable 7.1a – Collected needs within EDF”, EDF technical report, H-P1A-2007-01712-FR, 2008.

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

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

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

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

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

[14] Thuy Nguyen, “MODRIO D8.1.3 The Backup Power Supply (BPS) Case Study”, EDF technical report, H-P1A-2014-00550-EN.

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

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

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