D2.1	CRML specification
Access[1]:	**PU**
Type[2]:	**Report**
Version:	**1.2**
Due Dates[3]:	**M12, M24**

Environment for model-based rigorous adaptive co-design and operation of CPS

Executive summary[4]:

CRML (Common Requirement Modelling Language) is being developed out of the desire to express formal multidisciplinary requirements in the form of spatiotemporal constraints on the behaviour of cyber-physical systems, in particular energy systems.

This document constitutes the formal specification of CRML as defined within the EMBrACE project.

It is made of two main parts: the formal semantics, written in mathematical notation, and the syntax, completed by numerous examples.

It also contains the expression of two CRML libraries:

- The ETL library for the evaluation of requirements, i.e. to decide whether requirements are satisfied or not over a given time period.
- The FORM-L library that provides standard functions (CRML operators) for the expression of requirements in the form of time-dependent constraints. This library is inspired by the work of Thuy Nguyen of EDF Lab Chatou.

The first version (version 0.1) of this report is available in the form of PowerPoint slides.

The current version (version 1.2) includes some minor updates compared to the official release delivered for the final ITEA review of EMBrACE (version 1.1).

[1] Access classification as per definitions in PCA; PU = Public, CO = Confidential. Access classification per deliverable stated in FPP.

[2] Deliverable type according to FPP, note that all non-report deliverables must be accompanied by a deliverable report.

[3] Due month(s) according to FPP.

[4] It is mandatory to provide an executive summary for each deliverable.

Deliverable Contributors:

	Name	Organisation	Primary role in project	Main Author(s)[5]
Deliverable Leader[6]	Daniel Bouskela	EDF	WP2 Leader	X
Contributing Author(s)[7]	Audrey Jardin	EDF	WP2 Contributor	
Internal Reviewer(s)[8]				

[5] Indicate Main Author(s) with an "X" in this column.

[6] Deliverable leader according to FPP, role definition in PCA.

[7] Person(s) from contributing partners for the deliverable, expected contributing partners stated in FPP.

[8] Typically person(s) with appropriate expertise to assess deliverable structure and quality.

Document History:

Version	Date	Reason for Change	Status[9]
0.1	30/01/2021	First Draft Version (PowerPoint slides)	Draft
0.2	18/01/2022	First Report Version	Draft
0.3	27/01/2022	Improvements	Draft
0.4	11/02/2022	Removal of keywords operator, template and category. New type: Probability.	Draft
0.5	15/02/2022	New operator for types Real, Integer and Boolean: at. Correction of categories varying1 and varying2.	Draft
0.6	08/03/2022	Improvements	Draft
0.7	11/03/2022	New operator for Boolean and Events: or. Correction of semantics of operators before and after. Correction of the ETL operator ‘becomes false’.	Draft
0.8	15/04/2022	Simplification of the definition of a requirement. The differential operator is not used anymore. The logical difference, pre and trim operators are removed.	Draft
1.0	22/08/2022	First final version released.	Released
1.1	06/12/2022	Addition of keyword new. Removal of keyword nil. Use of keyword external is modified.	Released
1.2	05/12/2023	Addition of keyword time. Generalization of the use of keyword new. Some typo fixes. Correction in the definition of FORM-L operators which imply elapsed time periods.	Released

[9] Status = “Draft”, “In Review”, “Released”.

AVERTISSEMENT / CAUTION

Executive Summary

CRML (Common Requirement Modelling Language) is being developed out of the desire to express formal multidisciplinary requirements in the form of spatiotemporal constraints on the behaviour of cyber-physical systems, in particular energy systems.

This document constitutes the formal specification of CRML as defined within the EMBrACE project.

It is made of two main parts: the formal semantics, written in mathematical notation, and the syntax, completed by numerous examples.

It also contains the expression of two CRML libraries:

- The ETL library for the evaluation of requirements, i.e. to decide whether requirements are satisfied or not over a given time period.
- The FORM-L library that provides standard functions (CRML operators) for the expression of requirements in the form of time-dependent constraints. This library is inspired by the work of Thuy Nguyen of EDF Lab Chatou.

The first version (version 0.1) of this report is available in the form of PowerPoint slides.

The current version (version 1.2) includes some minor updates compared to the official release delivered for the final ITEA review of EMBrACE (version 1.1).

Summary

List of Figures

List of Tables

Acronyms

CRML	Common Requirement Modelling Language

1. Rationale for CRML

1.1. Motivation

It is important to ensure that complex cyber-physical systems such as energy systems comply with their objectives and constraints, because they are systems of important socio-economic nature for the common good.

The acronym of CRML stands for Common Requirement Modelling Language.

The purpose of CRML is to offer to the numerous stakeholders involved in the design, operation and maintenance of cyber-physical systems a common language to negotiate and agree upon a common set of requirements in order to ensure that they can comply with their mutual commitments while meeting their own objectives.

The idea is to write the requirements using a formal language to provide a solution to following challenges:

- Provide comprehensive descriptions of all spatiotemporal assumptions and constraints that bear on the system under study. Constraints can be of all kinds: physical, performance (reliability, availability, economical…), regulatory (safety, security, environmental, reserve capacity for grid balancing, grid access and priority dispatch…).
- The requirement models must be easily legible and understandable by all stakeholders whatever their expertise and business domain. It is expected that a common requirement language will improve the productivity of studies. To that end, the syntax of the language must be close to natural language.

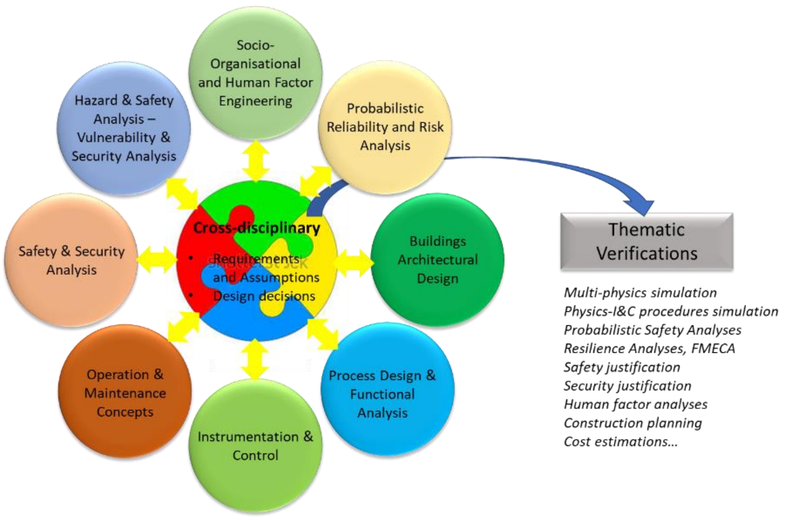
Socio-Organisational and Human Factor Engineering
Hazard & Safety Analysis – Vulnerability & Security Analysis
Probabilistic Reliability and Risk Analysis
Cross-disciplinary
Requirements and Assumptions
Design decisions
Safety & Security Analysis
Buildings Architectural Design
Thematic Verifications
Multi-physics simulation
Physics-I&C procedures simulation
Probabilistic Safety Analyses
Resilience Analyses, FMECA
Safety justification
Security justification
Human factor analyses
Construction planning
Cost estimations…
Operation & Maintenance Concepts
Process Design & Functional Analysis
Instrumentation & Control

Fig. 1. Some stakeholders' disciplines

- The systems which are the targets of the language exhibit strong physical aspects. Therefore, particular attention must be paid to physical aspects: physical units, real time, events, synchronism and asynchronism, components and objects, failures and uncertainties. Time dependent continuous and discrete variables must be dealt with in an asynchronous framework. This goes well beyond finite automata that are the reference in model checking.
- Perform automatic verifications by coupling requirement models with behavioral models of any kind and complexity: min and max limits (that represent authorized operation domains), finite state automata (that represent logical system operation), multidomain physical 0D/1D/2D/3D models (that represent detailed physical behavior), … Verifications are performed by co-

simulating requirement models with behavioral models. Requirement models act as observers to detect possible requirement violations of the behavioral models.

Behavioral models can be deterministic or stochastic. For the latter, probabilistic criteria are added to requirements and Monte Carlo techniques are used to simulate the verification model, which consists of the requirement model, the behavioral models, the observers of the behavioral models, and the links between observers and requirements which are called "bindings" (cf. Fig. 2).

- Generate automatically optimal architectures that comply with the requirements (constraint optimization, the constraints being given by the requirement model, and the cost function corresponding in general to economic objectives). This remains a largely unexplored aspect (at least for energy systems) and should be useful to tackle the new system architectures for the energy transition (energy mix with higher shares of renewables, complex energy markets, increased environmental constraints, decentralized grid architectures…).

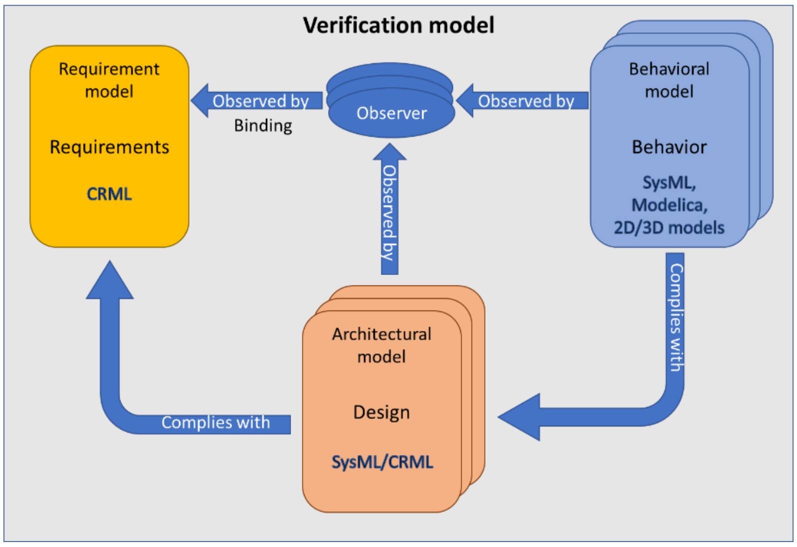

Fig. 2. Architecture of the verification model

1.2. Basic principles

1.2.1. Structure of the requirements

The language introduces the new concept of requirement made of four parts:

1. Spatial locator (WHERE): it defines the objects that are subject to the requirement. The word "spatial" means that the objects are selected depending on some criteria that can be time dependent.
2. Time locator (WHEN): it defines the time periods when the requirements should be satisfied. A time period is initiated and terminated when events occur. An event occurs when a Boolean becomes true. Therefore, a time locator can be composed of multiple time periods that can overlap.
3. Condition to be fulfilled (WHAT): it is the condition to be verified by the objects selected with the spatial locator within the bounds of the time periods selected by the time locator.
4. Probabilistic constraint (HOW_WELL): it defines a probabilistic constraint on the condition to be fulfilled.

Time periods and probabilistic constraints constitute the novelty of the approach. They are required to handle *realistic requirements*, because realistic requirements cannot be satisfied anytime at any cost. Time periods define when requirements are in effect and the time delay to satisfy them. Probabilistic constraints define some tolerance for the system to fail complying with the requirements. These two aspects have profound technical and economic impact on the design and operation of the system.

Example of a requirement: “While in service, the system should always stay within its operating domain. However, if the system fails to stay within its operating domain, it should not stay outside of its operating domain for more than ten minutes more than three times per year, with a probability of success of 99.99 %.”

In this requirement, three time locators and two conditions are involved. The three time locators are: a fixed time period “while the system is in service”, time periods of ten minutes starting when the event “leaving the operating domain” occurs, and a sliding time period of one year starting when the system is started and finishing when the system is stopped. The two conditions are “stay within the operating domain”, and “number of operating domain departures less than three”. The probabilistic constraint is the constant 99.99 %. However, it could depend on time for, e.g., requirements invoking probabilities of failure that use the Weibull law. The spatial locator is “the system”.

It turns out that this requirement is in fact a combination of three elementary requirements that are related by Boolean conjunction (∧) and inference (⟹) operators. It is therefore important to be able to combine requirements using Boolean operators. Also, a distinction must be made between the elementary requirements and the full requirement: the elementary requirements can fail, but the full requirement must be satisfied.

Example of a requirement: “While they are in operation, all pumps in the system should not cavitate.”

In this requirement, the spatial locator is “all pumps in the system”. It is a set that is only defined in intention, because its elements are only known by their property of being pumps. Therefore, the number of elements of the set is not known. The set could even be empty. The time locator is “While they are in operation”, which is attached to each pump because pumps can be operating at different time periods. The condition is “should not cavitate”. This kind of requirement is generic because it applies to any system, even without pumps. It is therefore important to be able to build libraries of generic requirements that can be used on different systems and use the notion of objects to structure the requirements.

1.2.2. Mathematical foundation of the language

The language builds on four pillars of mathematics: Boolean algebra, set theory, function theory and probability theory.

Boolean algebra

Boolean algebra is used to handle the time locators (WHEN) and the conditions to be fulfilled (WHAT). The idea is to use a Boolean algebra made of four values { *true*, *false*, *undecided*, *undefined* }.

To understand the meaning of the 4-valued Boolean algebra, let us take the simple example of a project report that must be written before a given deadline. The time locator for this requirement is a time period that extends from the start of the project until the deadline which is often the end of the project. Therefore, the requirement is: “The project report must be completed before the end of the project”. Before the start of the project, the value of the requirement is *undefined*, which means that the requirement is not applicable. After the start of the project and before the end of the project, the value of the requirement is *undecided* which means that the requirement is applicable but its outcome is uncertain until either the report is completed before or at the end of the time period (thus before the deadline or just in time), in such case the requirement is satisfied and its value is *true*, or until the end of the time period, in such case the requirement is not satisfied and its value is *false* (the report is late or cancelled).

The three values { *true*, *false*, *undecided* } are necessary to compute the value of the requirement, taking into account the fact that the decision cannot be made instantly: one must wait either until the document is completed, or until the deadline to decide whether the requirement is satisfied or not. This event is called the *decision event*. The three values { *true*, *false*, *undecided* } constitute by themselves a Boolean algebra which is similar to the 3-valued Kleene logic, but with different semantics.

The value *undefined* is necessary to combine requirements using Boolean operators. When a requirement is outside of its time locator, it is not applicable. Therefore, if it is combined with a second requirement, at time periods outside of its time locator it should have no effect on the other requirement. As an illustrating example, let us consider the requirement “While they are in operation, all pumps in the system should not cavitate.” Obviously, when a given pump is not in operation, the fact that it cavitates or not should not affect the outcome of the requirement on the pumps that are in operation.

The production and evaluation of requirements builds on the 4-valued Boolean logic (cf. Fig. 3):

- An event occurs when a Boolean becomes *true* (whatever its past value is *false*, *undecided*, or *undefined*).
- Time periods open when an event occurs, and close when an event occurs (the closing event can be equal to or different from the opening event).
- Conditions are Boolean expressions.
- Requirements result from the association of conditions with multiple time periods.
- The value of a requirement is a Boolean. Therefore, a requirement can be mathematically seen as a function that associates a couple (condition, multiple time period) to a 4-valued Boolean. This is denoted $(\varphi, P) \longrightarrow \varphi \otimes P$, where φ is the condition and P is the time locator that consists of multiple time periods (i.e., a set of time periods).
- The value of a requirement can be reused to form other events and other conditions, and hence other requirements.

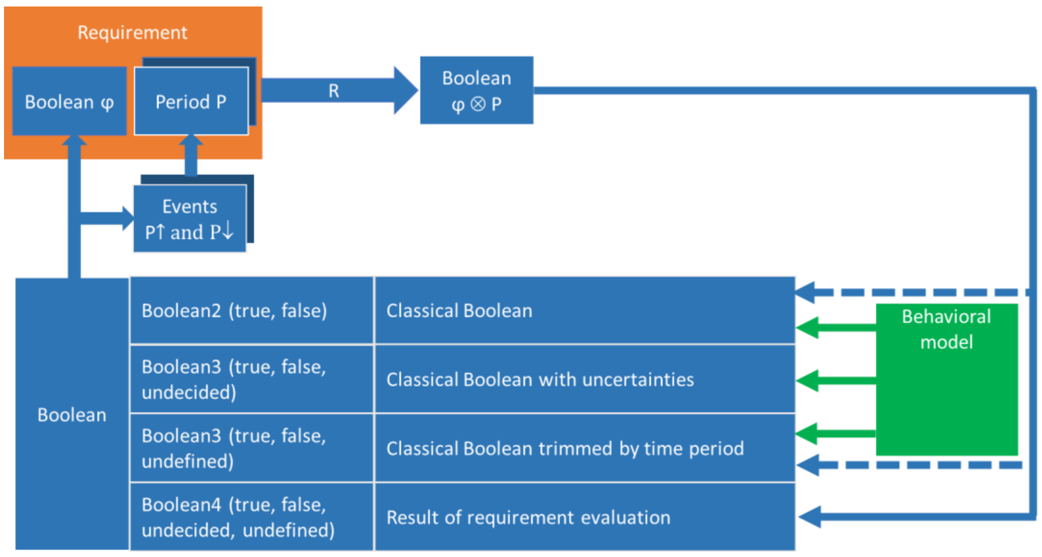
Requirement
Boolean φ
Period P
R
Boolean φ ⊗ P
Events P↑ and P↓
Boolean
Boolean2 (true, false)
Classical Boolean
Boolean3 (true, false, undecided)
Classical Boolean with uncertainties
Boolean3 (true, false, undefined)
Classical Boolean trimmed by time period
Boolean4 (true, false, undecided, undefined)
Result of requirement evaluation
Behavioral model

Fig. 3. General architecture of requirements

Booleans have different types depending on their origin:

- 2-valued Booleans { *true*, *false* } are the classical Booleans in general issued from the behavioral model that captures the behavior of the system under study.

- 3-valued Booleans { *true*, *false*, *undecided* } can be produced by the behavioral model to represent classical Booleans with uncertainties. For instance, Fig. 4 represents an operating domain whose limits are known with uncertainties: if the operating point is within the uncertainty of the boundary, then its position is *undecided*, otherwise it is *true* (inside the domain) or *false* (outside of the domain). They also correspond to time periods with no delay between them, i.e. such that time periods cover the whole arrow of time.

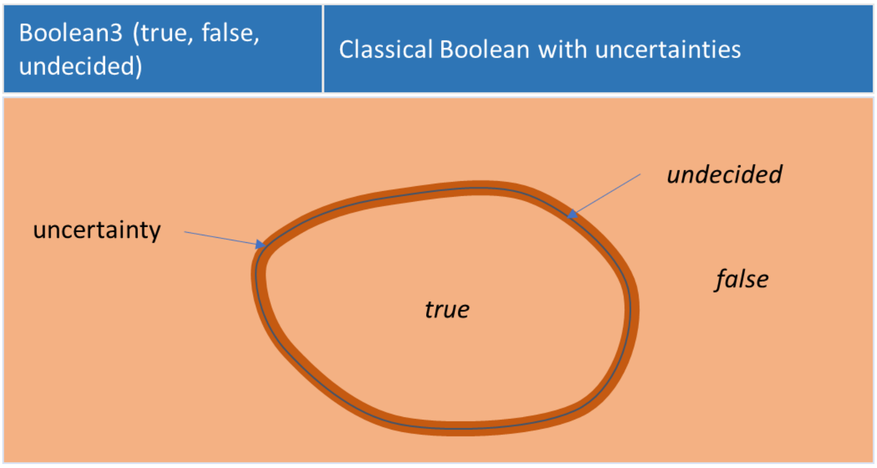
Boolean3 (true, false, undecided)
Classical Boolean with uncertainties
undecided
uncertainty
false
true

Fig. 4. Operating domain with uncertainty on the boundary

- 3-valued Booleans { *true*, *false*, *undefined* } correspond to the values of requirements such that their associated time periods have their opening and closing events equal (i.e. such that the duration of the time periods is zero, cf. Fig. 5). Then the value of such requirements can never be *undecided*, and consequently the decision is instantly made (the opening, decision and closing events are the same).

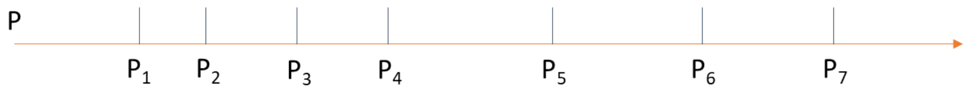

Fig. 5. Multiple time period with zero length single time periods

- 4-valued Booleans { *true*, *false*, *undecided*, *undefined* } are always the result of the evaluation of requirements. Using such Booleans to build requirements allow to issue requirements on requirements (i.e., meta-requirements) such as “10 seconds after requirement R1 fails, requirement R2 must be satisfied”, where requirements R1 and R2 are not specified (i.e., are considered as dummy variables in the expression of the meta-requirement).

In limiting situations where the length of time periods and the delay between time periods go to zero, Boolean variables can only take the values *true* or *false* (because *undecided* can only be found strictly inside time periods, and *undefined* can only be found strictly outside time periods). Thus the 2-valued Boolean algebra { *true*, *false* } corresponds in continuous time to limiting unrealistic situations when decisions can be made instantly anytime.

<u>Set theory</u>

The CRML language consists of elements which belong to sets, which can also be set elements (but not all elements are sets). It is assumed that there exists a unique universal set that contains all sets.

Each element is a typed couple (*variable*, *value*), where *variable* stands for the name of the element, and *value* is the value of the element. The type of the element is the domain that it belongs to, a domain being the set of all elements of the same kind, such as real numbers, Booleans, class definitions, classes (a class contains all objects instances of that class), etc.

Sets are therefore used to structure models as arrays of objects of different types and relations between objects, like in relational databases.

Function theory

CRML functions are very simple and are called operators. They are of the form $y = f(x_1, \ldots, x_n)$, where y and $x_1, \ldots, x_n$ are language elements, as defined above. f is the name of the operator, and y is its value, which is computed by a unique expression that combines operators on $x_1, \ldots, x_n$.

For instance, the Boolean disjunction $x_1\ or\ x_2$ can be defined from the Boolean conjunction $x_1\ and\ x_2$ and the Boolean negation $not\ x$: $x_1\ or\ x_2 := not\ (not\ x_1\ and\ not\ x_2)$.

Operators could be recursive, but this is of no obvious use in the context of requirement modelling.

Although $x_1, \ldots, x_n$ could be operators (like in λ-calculus), this is not required in CRML.

A requirement can be mathematically viewed as a function that associates the couple (condition, set of time periods) to its Boolean value. Therefore, the framework of λ-calculus could be more appropriate (at least theoretically) to handle requirements (because everything is function in λ-calculus), however this has not been investigated.

1.2.3. Syntax of the language

The ambition of CRML is to be close to natural language. This is obtained by replacing the usual functional notation $y = f(x_1, \ldots, x_n)$ by generic (or boilerplate) sentences with words that represent the placeholders of the operator's arguments.

For instance, for function $n = count\ (C, P)$ that counts the number of events generated by a clock C within a given time period P, the CRML operator is defined as

operator [Integer] count Clock C inside Period P = *expr* (C,P);

where [Integer] is the return type of the operator, and C and P are the two arguments of the operator of respective types Clock (a clock is a set of events) and Period (a period is a time period). The words count and inside represent the placeholders for the two arguments C and P. *expr* (C,P) denotes the expression that computes the value of the operator from C and P. Then the expression

Integer n is count C inside P;

is equivalent to $n = count\ (C, P)$.

2. CRML architecture

The ability to write CRML models for the expression and simulation of requirements builds on two main parts:

- The CRML language itself.
- Two CRML libraries:
  - The ETL library for the evaluation of requirements, whose purpose is to decide whether requirements are satisfied or not over a given time period.
  - The FORM-L library that provides standard functions (CRML operators) for the expression of requirements in the form of time-dependent constraints. This library is inspired by the work of Thuy Nguyen of EDF Lab Chatou.

The CRML language is divided into several chapters, each chapter corresponding to a language type.

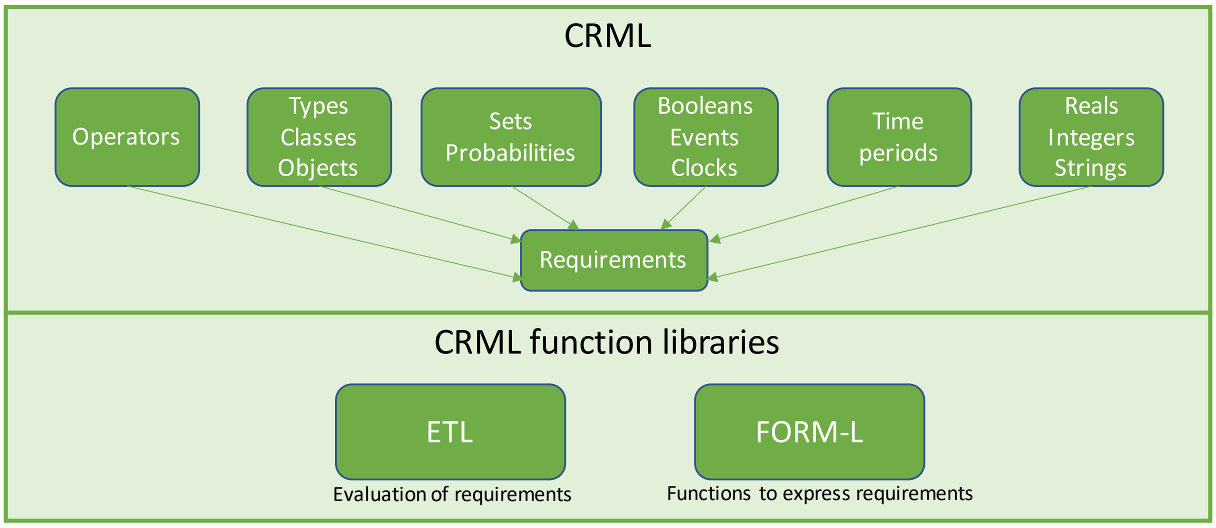

Fig. 6. CRML architecture

3. Formal semantics

3.1. Continuous clock

There is one continuous clock denoted $\mathfrak{C}$. It represents the Newtonian time. All time instants t in $\mathfrak{C}$ are real numbers: $t \in \mathbb{R}$, where $\mathbb{R}$ is the domain of real numbers. Therefore, time has no upper or lower bounds.

The time scale of $\mathfrak{C}$ depends on an arbitrary time origin t_0. Therefore, the value of t_0 can have any value (i.e., can be chosen arbitrarily). Only time differences (or delays) $t - t_0$ between any time instant t and the time origin t_0 have unique values (i.e., they cannot be chosen arbitrarily). It is customary to take $t_0 = 0$ so that $t = t - t_0$ has a unique value, but the choice of t_0 on the time axis is still arbitrary.

In the following figures, the continuous clock is represented by a continuous horizontal arrow.

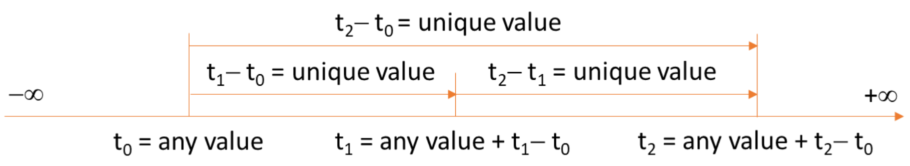
$t_2 - t_0$ = unique value
$t_1 - t_0$ = unique value
$t_2 - t_1$ = unique value
$-\infty$
$+\infty$
t_0 = any value
t_1 = any value + $t_1 - t_0$
t_2 = any value + $t_2 - t_0$

Fig. 7. The continuous clock

In the sequel, all variables are functions of time $t \in \mathfrak{C}$: $x = x(t)$ where x denotes any variable.

Let f be a function of several variables:

$$\begin{aligned} f\colon \mathbb{D}_1 \times \mathbb{D}_2 \times \ldots \times \mathbb{D}_n &\longrightarrow \mathbb{D} \\ (x_1, x_2, \ldots, x_n) &\longmapsto y = f(x_1, x_2, \ldots, x_n) \end{aligned}$$

where the $\mathbb{D}_i$ and $\mathbb{D}$ stand of any domain.

Unless specified otherwise, the values of all variables x_i are taken at the same instant $t \in \mathfrak{C}$.

3.2. 4-valued Booleans

4-valued Booleans φ satisfy the following algebra:

$$\varphi := \varphi \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \tag{1}$$

The logical negation (¬) and conjunction (∧) operators are defined with the following truth tables:

Table 1. Truth table for the logical negation

not	**Logical negation**			
φ	*true*	*false*	*undecided*	*undefined*
$\neg\varphi$	*false*	*true*	*undecided*	*undefined*

Table 2. Truth table for the logical conjunction

and	**Logical conjunction**			
$\varphi_1 \wedge \varphi_2$	***true***	***false***	***undecided***	***undefined***
true	*true*	*false*	*undecided*	*true*
false	*false*	*false*	*false*	*false*
undecided	*undecided*	*false*	*undecided*	*undecided*
undefined	*true*	*false*	*undecided*	*undefined*

The rationale for Table 1 and Table 2 is the following:

1. The truth table reduced to the values *true* and *false* are the same as the one for the classical 2-valued Booleans.
2. φ is *undecided* means that it not known whether φ is *true* or *false*. Hence, if φ is *undecided*, then it is not known whether $\neg\varphi$ is *true or false*, and therefore $\neg\varphi$ is *undecided*. If φ_1 is *undecided* and φ_2 is *true*, then it is not known whether $\varphi_1 \wedge \varphi_2$ is *true* or *false*, because it is *true* if φ_1 is *true*, and *false* if φ_1 is *false*. Therefore $\varphi_1 \wedge \varphi_2$ is *undecided*. However, if φ_2 is *false*, then $\varphi_1 \wedge \varphi_2$ is *false* whatever the value of φ_1.
3. φ is *undefined* means that φ should not have any influence on the result of the Boolean operation because it is not applicable (cf. Section 1.2.2). Therefore, *undefined* is the neutral element for all Boolean operators.

The negation and conjunction operators verify the following properties of Boolean algebra:

- Involution: $\forall\varphi,\ \neg(\neg\varphi) = \varphi$
- Commutativity: $\forall\varphi_1, \forall\varphi_2,\ \varphi_1 \wedge \varphi_2 = \varphi_2 \wedge \varphi_1$
- Associativity: $\forall\varphi_1, \forall\varphi_2, (\varphi_1 \wedge \varphi_2) \wedge \varphi_3 = \varphi_1 \wedge (\varphi_2 \wedge \varphi_3)$
- Idempotence: $\forall\varphi,\ \varphi \wedge \varphi = \varphi$

But the following property is not verified:

- Contradiction: $\forall\varphi,\ \varphi \wedge \neg\varphi = false$

All other logical operators are defined using the Morgan laws:

- Logical disjunction:

$$\varphi_1 \vee \varphi_2 := \neg(\neg\varphi_1 \wedge \neg\varphi_2) \tag{2}$$

- Logical exclusive disjunction (or parity):

$$\varphi_1 \oplus \varphi_2 := (\varphi_1 \wedge \neg\varphi_2) \vee (\neg\varphi_1 \wedge \varphi_2) \tag{3}$$

- Logical inference:

$$\varphi_1 \Longrightarrow \varphi_2 := \neg\varphi_1 \vee \varphi_2 \qquad (4)$$

It is easy to verify that the operators defined with the Morgan laws make sense considering the meaning given to *undecided* and *undefined*. For instance, for the logical disjunction, the truth table obtained using Eq. (2) and the truth table obtained using the rationale given for the logical conjunction operator are equal, cf. Table 3.

Table 3. Truth table for the logical disjunction

or	**Logical disjunction**			
$\varphi_1 \vee \varphi_2$	***true***	***false***	***undecided***	***undefined***
true	*true*	*true*	*true*	*true*
false	*true*	*false*	*undecided*	*false*
undecided	*true*	*undecided*	*undecided*	*undecided*
undefined	*true*	*false*	*undecided*	*undefined*

For clarity, the truth table of the equality operator ($=$) is given in the following table.

Table 4. Truth table for the logical equality

equal	**Logical equality**			
$\varphi_1 = \varphi_2$	***true***	***false***	***undecided***	***undefined***
true	*true*	*false*	*false*	*false*
false	*false*	*true*	*false*	*false*
undecided	*false*	*false*	*true*	*false*
undefined	*false*	*false*	*false*	*true*

The logical difference operator is defined as

$$\varphi_1 \neq \varphi_2 := \neg(\varphi_1 = \varphi_2) \qquad (5)$$

Contrary to 2-valued Boolean algebra, in the 4-valued Boolean algebra, the negation operator is different from the difference operator: $\varphi_1 \neq \varphi_2$ does not imply $\varphi_1 = \neg\varphi_2$ and reciprocally. Therefore, one must be cautious not to confound negation with difference when using 4-valued Boolean algebra.

The domain of 4-valued Booleans { *true*, *false*, *undecided*, *undefined* } is denoted $\mathbb{B}$ or $\mathbb{B}_4$.

The domain of 2-valued Booleans { *true*, *false* } is denoted $\mathbb{B}_2$.

It is worthwhile noting that the two domains of 3-valued Booleans { *true*, *false*, *undecided* } and { *true*, *false*, *undefined* } are indeed Boolean algebras (they satisfy the Morgan laws). They are respectively denoted $\mathbb{B}_3$ and $\mathbb{B}'_3$.

Thus:

- $\mathbb{B} = \mathbb{B}_4 = \{true, false, undecided, undefined\}$
- $\mathbb{B}_3 = \{true, false, undecided\}$
- $\mathbb{B}'_3 = \{true, false, undefined\}$
- $\mathbb{B}_2 = \{true, false\}$

3.3. Events

An event E corresponds to the single occurrence of a 4-valued Boolean φ becoming *true*, which is denoted $\varphi \uparrow$:

$$E \ := \ \varphi \uparrow \tag{6}$$

Notice that before the event, the value of φ can be *false*, *undecided* or *undefined*.

Events always occur in the continuous time domain $\mathfrak{C}$.

The instant t of occurrence of event $\varphi \uparrow$ is denoted $@\varphi \uparrow$:

$$t := \ @\varphi \uparrow \tag{7}$$

where t is the instant of occurrence of $\varphi \uparrow$.

The event that corresponds to the current time $t \in \mathfrak{C}$ is denoted $\mathfrak{C} \uparrow$. Therefore $t = @\mathfrak{C} \uparrow$.

The event corresponding to the current instant in the continuous clock $\mathfrak{C}$ is denoted $\mathfrak{C} \uparrow$. Therefore, the current instant is $t = @\mathfrak{C} \uparrow$.

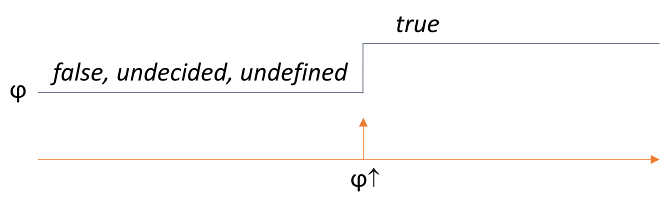

Fig. 8. Event generated by a Boolean

The domain of events is denoted $\mathcal{E}$.

At instant time t, $\mathcal{E}$ contains all events that have occurred before t, including t. It does not contain future events that will occur after t excluding t because it is not decidable at t whether future events will actually occur. Therefore, the domain of events $\mathcal{E}$ is time dependent and the number of its elements increases in time as the number of occurred events increases in time. Before the occurrence of the first event, $\mathcal{E}$ is the empty set.

The domain of non-empty events is denoted $\mathcal{E}^*$. Therefore $\mathcal{E}^*$ always contains the first event.

$$\begin{cases} |\mathcal{E}| \geq 0 \\ |\mathcal{E}^*| > 0 \end{cases} \tag{8}$$

where $|S|$ denotes the number of elements in set S.

$\mathcal{E}^*$ is an ordered set because the events in $\mathcal{E}$ are ordered by increasing occurrences in time:

$$\mathcal{E}^*(t) := \{\varphi_1\uparrow,\ \varphi_2\uparrow, \dots,\ \varphi_i\uparrow, \dots,\ \varphi_n\uparrow\} \text{ such that } \forall i, \forall j > i,\ @\varphi_i\uparrow \leq @\varphi_j\uparrow \tag{9}$$

where the $\varphi_i\uparrow$ are all the events that have occurred before t.

Because $\varphi_n\uparrow$ is the last event in $\mathcal{E}^*(t)$, the following relation holds:

$$\mathcal{E}^*(t) = \mathcal{E}^*(\varphi_n\uparrow) \tag{10}$$

where $\mathcal{E}^*(\varphi_n\uparrow)$ denotes the value of $\mathcal{E}^*$ at the instant of occurrence of $\varphi_n\uparrow$ ($\mathcal{E}^*(\varphi_n\uparrow) = \mathcal{E}^*(@\varphi_n\uparrow)$). .

Therefore

$$\mathcal{E}^*(\varphi_n\uparrow) := \{\varphi_1\uparrow,\ \varphi_2\uparrow, \dots,\ \varphi_i\uparrow, \dots,\ \varphi_n\uparrow\} \text{ such that } \forall i, \forall j > i,\ @\varphi_i\uparrow \leq @\varphi_j\uparrow \tag{11}$$

Hence:

- $\varphi_i\uparrow$ occurs before $\varphi_j\uparrow$ can be formally expressed as $\varphi_j\uparrow \notin \mathcal{E}^*(\varphi_i\uparrow)$ if it is not known whether φ_j will occur or not, or $\varphi_i\uparrow \in \mathcal{E}^*(\varphi_j\uparrow)$ or $@\varphi_i\uparrow \leq @\varphi_j\uparrow$ if φ_j has occurred.
- $\mathcal{E}^* = \mathcal{E}^*(\varphi_{|\mathcal{E}^*|}\uparrow)$.

The set of events obtained after the disjunction of two events $\varphi_1\uparrow$ and $\varphi_2\uparrow$ (i.e., after at least one of the events $\varphi_1\uparrow$ or $\varphi_2\uparrow$ has occurred) is denoted $\mathcal{E}^*(\varphi_1\uparrow \vee \varphi_2\uparrow)$. Therefore:

$$\mathcal{E}^*(\varphi_1\uparrow \vee \varphi_2\uparrow) := \begin{cases} \mathcal{E}^*(\varphi_1\uparrow) \text{ if } \varphi_2\uparrow \notin \mathcal{E}^*(\varphi_1\uparrow) \\ \mathcal{E}^*(\varphi_2\uparrow) \text{ if } \varphi_1\uparrow \notin \mathcal{E}^*(\varphi_2\uparrow) \end{cases} \tag{12}$$

The set of events obtained after the conjunction of two events $\varphi_1\uparrow$ and $\varphi_2\uparrow$ (i.e., after the two events $\varphi_1\uparrow$ and $\varphi_2\uparrow$ have occurred) is denoted $\mathcal{E}^*(\varphi_1\uparrow \wedge \varphi_2\uparrow)$. Therefore:

$$\mathcal{E}^*(\varphi_1\uparrow \wedge \varphi_2\uparrow) := \begin{cases} \mathcal{E}^*(\varphi_1\uparrow) \text{ if } \varphi_2\uparrow \in \mathcal{E}^*(\varphi_1\uparrow) \\ \mathcal{E}^*(\varphi_2\uparrow) \text{ if } \varphi_1\uparrow \in \mathcal{E}^*(\varphi_2\uparrow) \end{cases} \tag{13}$$

3.4. Multiple events

A multiple event Ω is a subset of $\mathcal{E}$: $\Omega \in 2^{\mathcal{E}}$.

It is therefore a set of events ordered by increasing occurrences in time:

$$\Omega := \{\varphi_1\uparrow,\ \varphi_2\uparrow, \dots,\ \varphi_i\uparrow, \dots\} \text{ such that } \forall i, \forall j > i,\ @\varphi_i\uparrow \leq @\varphi_j\uparrow \tag{14}$$

Let us consider a Boolean ψ that becomes true at each event $\varphi_i \uparrow$ (there are infinitely many of such Booleans). Then Ω is generated by ψ. This is denoted $\Omega(\psi)$:

$$\Omega(\psi) := \{\psi(t_1)\uparrow, \psi(t_2)\uparrow, \dots, \psi(t_i)\uparrow, \dots\} \tag{15}$$

Where the $t_i = @\varphi_i \uparrow$ are the time instants when ψ becomes *true* (i.e., such that $\psi(t_i)\uparrow = \varphi_i \uparrow$) for all $i \in \mathbb{N}^*, 1 \leq i \leq |\Omega(\psi)|$.

The chosen Boolean ψ that generates $\Omega(\psi)$ is denoted $\psi(\Omega)$.

Note that the number of elements $|\Omega(\psi)|$ is time dependent as the set $\Omega(\psi)$ increases in time like any subset of $\mathcal{E}$:

$$|\Omega(\psi)| = \begin{cases} 0 \text{ if } t < t_1 \\ i \text{ if } i \leq t < i+1 \end{cases} \tag{16}$$

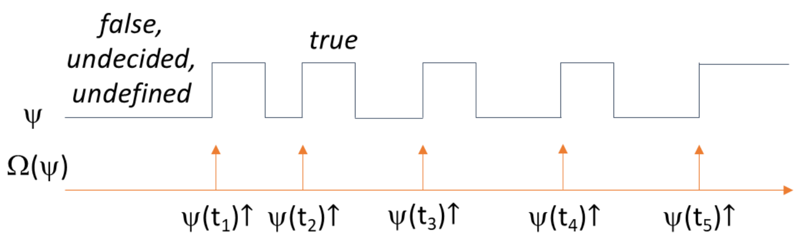

Fig. 9. Multiple event generated by a Boolean

The domain of multiple events is denoted $\mathcal{D}$. Then $\mathcal{D} = 2^{\mathcal{E}}$ (the set of the subsets of $\mathcal{E}$).

3.5. Operators on events

3.5.1. Equal operator

The equal operator tells whether two events $\varphi_1 \uparrow$ and $\varphi_2 \uparrow$ are equal, i.e. whether they occur at the same time instant. It is formally defined as follows:

$$\begin{gathered} = : \mathcal{E} \times \mathcal{E} \longrightarrow \mathbb{B} \\ (\varphi_1 \uparrow, \varphi_2 \uparrow) \mapsto \varphi_1 \uparrow = \varphi_2 \uparrow := \begin{cases} true \text{ if } \varphi_2 \uparrow \in \mathcal{E}^*(\varphi_1 \uparrow) \wedge \varphi_1 \uparrow \in \mathcal{E}^*(\varphi_2 \uparrow) \\ false \text{ if } \neg(\varphi_2 \uparrow \in \mathcal{E}^*(\varphi_1 \uparrow) \wedge \varphi_1 \uparrow \in \mathcal{E}^*(\varphi_2 \uparrow)) \\ undecided \text{ if } \varphi_1 \uparrow \notin \mathcal{E} \text{ and } \varphi_2 \uparrow \notin \mathcal{E} \end{cases} \end{gathered} \tag{17}$$

Eq. (17) means that $\varphi_1 \uparrow = \varphi_2 \uparrow$ is $true$ if $\varphi_1 \uparrow$ and $\varphi_2 \uparrow$ occur at the same time instant. It is $undecided$ if none of the two events $\varphi_1 \uparrow$ or $\varphi_2 \uparrow$ occurs. It is $false$ otherwise. Therefore, the value of $\varphi_1 \uparrow = \varphi_2 \uparrow$ is always known, even if $\varphi_1 \uparrow$ and/or $\varphi_2 \uparrow$ have not occurred.

Also notice that $\varphi_1 \uparrow = \varphi_2 \uparrow$ does not imply that $\varphi_1 = \varphi_2$ (but the converse is true, cf. Fig. 10).

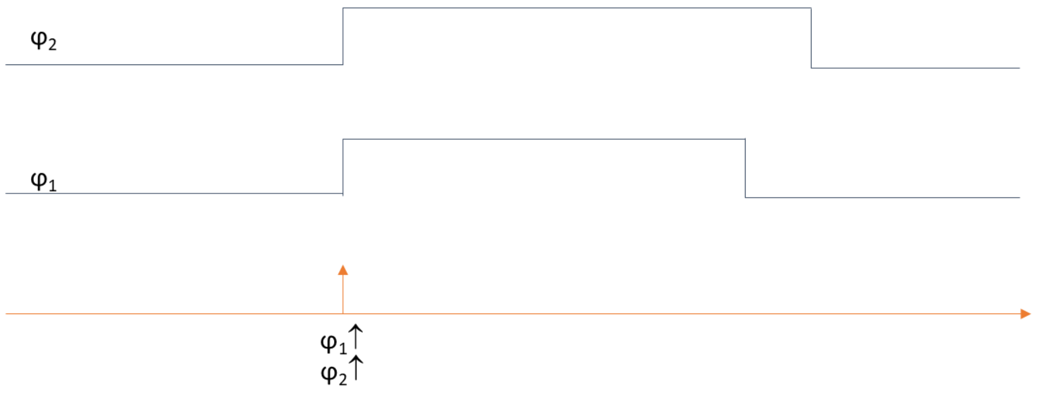

Fig. 10. Event equal operator

3.5.2. Different operator

The different operator tells whether two events $\varphi_1\uparrow$ and $\varphi_2\uparrow$ are different, i.e. whether they occur at different time instants. It is formally defined as follows:

$$\begin{gathered}\neq : \mathcal{E} \times \mathcal{E} \longrightarrow \mathbb{B} \\ (\varphi_1\uparrow, \varphi_2\uparrow) \mapsto \varphi_1\uparrow \neq \varphi_2\uparrow := \neg(\varphi_1\uparrow = \varphi_2\uparrow)\end{gathered} \tag{18}$$

The value of $\varphi_1\uparrow \neq \varphi_2\uparrow$ is always known, even if $\varphi_1\uparrow$ and/or $\varphi_2\uparrow$ have not occurred.

Notice that Eq. (18) is consistent with the definition of the different operator from first principles:

$$\varphi_1\uparrow \neq \varphi_2\uparrow := \begin{cases} true \text{ if } \neg(\varphi_2\uparrow \in \mathcal{E}^*(\varphi_1\uparrow) \wedge \varphi_1\uparrow \in \mathcal{E}^*(\varphi_2\uparrow)) \\ false \text{ if } \varphi_2\uparrow \in \mathcal{E}^*(\varphi_1\uparrow) \wedge \varphi_1\uparrow \in \mathcal{E}^*(\varphi_2\uparrow)\ ) \\ undecided \text{ if } \varphi_1\uparrow \notin \mathcal{E} \text{ and } \varphi_2\uparrow \notin \mathcal{E} \end{cases}$$

3.5.3. Before operator

The before operator tells whether a first event $\varphi_1\uparrow$ occurs before a second event $\varphi_2\uparrow$. It is formally defined as follows:

$$\begin{gathered}\leq : \mathcal{E} \times \mathcal{E} \longrightarrow \mathbb{B} \\ (\varphi_1\uparrow, \varphi_2\uparrow) \mapsto \varphi_1\uparrow \leq \varphi_2\uparrow := \begin{cases} true \text{ if } \varphi_2\uparrow \notin \mathcal{E}^*(\varphi_1\uparrow) \vee \varphi_1\uparrow = \varphi_2\uparrow \\ false \text{ if } \varphi_1\uparrow \notin \mathcal{E}^*(\varphi_2\uparrow) \\ undecided \text{ if } \varphi_1\uparrow \notin \mathcal{E} \text{ and } \varphi_2\uparrow \notin \mathcal{E} \end{cases}\end{gathered} \tag{19}$$

Eq. (19) means that $\varphi_1\uparrow \leq \varphi_2\uparrow$ is:

- $undecided$ if neither $\varphi_1\uparrow$ nor $\varphi_2\uparrow$ have occurred,
- $true$ if $\varphi_1\uparrow$ has occurred first, or if $\varphi_1\uparrow$ and $\varphi_2\uparrow$ occur simultaneously,
- $false$ if $\varphi_2\uparrow$ has occurred first.

Therefore, the value of $\varphi_1\uparrow \leq \varphi_2\uparrow$ is always known, even if $\varphi_1\uparrow$ and/or $\varphi_2\uparrow$ have not occurred.

The strictly before operator is defined as follows:

$$\begin{gathered}< : \mathcal{E} \times \mathcal{E} \longrightarrow \mathbb{B} \\ (\varphi_1\uparrow, \varphi_2\uparrow) \mapsto \varphi_1\uparrow < \varphi_2\uparrow := (\varphi_1\uparrow \leq \varphi_2\uparrow) \wedge (\varphi_1\uparrow \neq \varphi_2\uparrow)\end{gathered} \tag{20}$$

The value of $\varphi_1\uparrow < \varphi_2\uparrow$ is always known, even if $\varphi_1\uparrow$ and/or $\varphi_2\uparrow$ have not occurred.

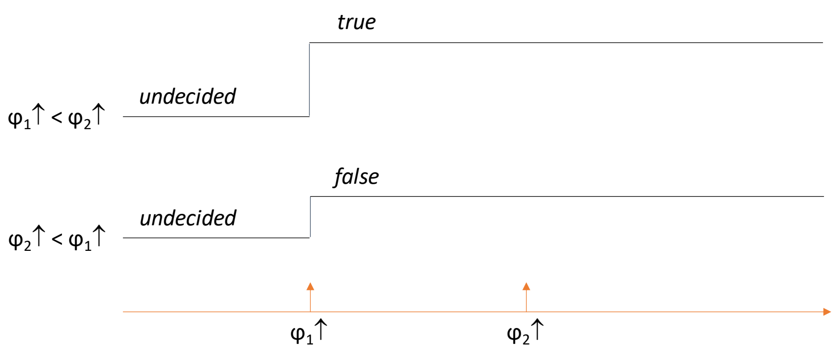

Fig. 11. Event before operator

3.5.4. After operator

The after operator tells whether a first event $\varphi_1\uparrow$ occurs after a second event $\varphi_2\uparrow$. It is formally defined as follows:

$$\begin{gathered} \geq : \mathcal{E} \times \mathcal{E} \longrightarrow \mathbb{B} \\ (\varphi_1\uparrow, \varphi_2\uparrow) \longmapsto \varphi_1\uparrow \geq \varphi_2\uparrow := \varphi_2\uparrow \leq \varphi_1\uparrow \end{gathered} \tag{21}$$

Notice that Eq. (21) is consistent with the definition of the after operator from first principles:

$$\begin{gathered} \geq : \mathcal{E} \times \mathcal{E} \longrightarrow \mathbb{B} \\ (\varphi_1\uparrow, \varphi_2\uparrow) \longmapsto \varphi_1\uparrow \geq \varphi_2\uparrow := \begin{cases} true \ \text{ if } \varphi_1\uparrow \notin \mathcal{E}^*(\varphi_2\uparrow) \vee \varphi_2\uparrow = \varphi_1\uparrow \\ false \ \text{ if } \varphi_2\uparrow \notin \mathcal{E}^*(\varphi_1\uparrow) \\ undecided \text{ if } \varphi_1\uparrow \notin \mathcal{E} \text{ and } \varphi_2\uparrow \notin \mathcal{E} \end{cases} \end{gathered}$$

Eq. (21) means that $\varphi_1\uparrow \geq \varphi_2\uparrow$ is:

- $undecided$ if neither $\varphi_1\uparrow$ nor $\varphi_2\uparrow$ have occurred,
- $true$ if $\varphi_2\uparrow$ has occurred first or if $\varphi_2\uparrow$ and $\varphi_1\uparrow$ occur simultaneously,
- $false$ if $\varphi_1\uparrow$ has occurred first.

Therefore, the value of $\varphi_1\uparrow \geq \varphi_2\uparrow$ is always known, even if $\varphi_1\uparrow$ and/or $\varphi_2\uparrow$ have not occurred.

The strictly after operator is defined as follows:

$$\begin{gathered} > : \mathcal{E} \times \mathcal{E} \longrightarrow \mathbb{B} \\ (\varphi_1\uparrow, \varphi_2\uparrow) \longmapsto \varphi_1\uparrow > \varphi_2\uparrow := \varphi_2\uparrow < \varphi_1\uparrow \end{gathered} \tag{22}$$

The value of $\varphi_1\uparrow > \varphi_2\uparrow$ is always known, even if $\varphi_1\uparrow$ and/or $\varphi_2\uparrow$ have not occurred.

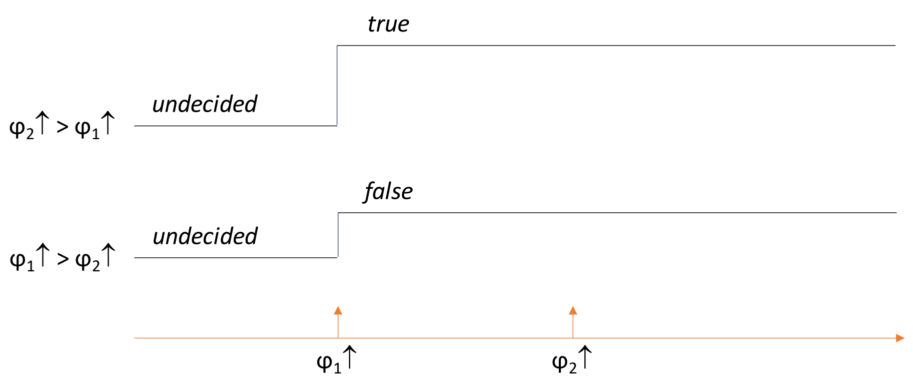

Fig. 12. Event after operator

3.5.5. Min operator

The min operator computes the first occurring event between two events $\varphi_1\uparrow$ and $\varphi_2\uparrow$, $\varphi_2\uparrow$ occurring before or after $\varphi_1\uparrow$. It is formally defined as follows:

$$\begin{gathered}\min\colon \mathcal{E}\times\mathcal{E}\longrightarrow 2^{\mathcal{E}}\\ (\varphi_1\uparrow,\varphi_2\uparrow)\longmapsto \min(\varphi_1\uparrow,\varphi_2\uparrow):=\begin{cases}\{\varphi_1\uparrow\}\text{ if }(\varphi_1\uparrow\le\varphi_2\uparrow)=true\\ \{\varphi_2\uparrow\}\text{ if }(\varphi_2\uparrow\le\varphi_1\uparrow)=true\\ \emptyset\text{ else}\end{cases}\end{gathered}\tag{23}$$

Eq. (23) produces a multiple time event to cope with the possibility of non-occurrence of $\varphi_1\uparrow$ and $\varphi_2\uparrow$.

Notice that Eq. (23) is consistent with the definition of the min operator from first principles:

$$\min(\varphi_1\uparrow,\varphi_2\uparrow)\;:=\;\begin{cases}\{\varphi_1\uparrow\}\text{ if }\varphi_2\uparrow\notin\mathcal{E}^*(\varphi_1\uparrow)\vee\varphi_1\uparrow=\varphi_2\uparrow\\ \{\varphi_2\uparrow\}\text{ if }\varphi_1\uparrow\notin\mathcal{E}^*(\varphi_2\uparrow)\vee\varphi_2\uparrow=\varphi_1\uparrow\\ \emptyset\text{ else}\end{cases}$$

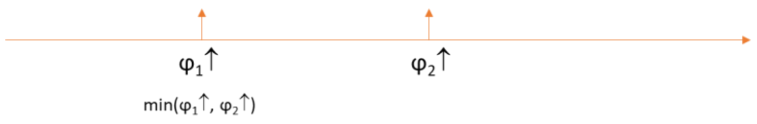

Fig. 13. Event min operator

3.5.6. Max operator

The max operator computes the last occurring event between two events $\varphi_1\uparrow$ and $\varphi_2\uparrow$, $\varphi_2\uparrow$ occurring before or after $\varphi_1\uparrow$. It is formally defined as follows:

$$\begin{gathered}\max\colon \mathcal{E}\times\mathcal{E}\longrightarrow 2^{\mathcal{E}}\\ (\varphi_1\uparrow,\varphi_2\uparrow)\longmapsto \max(\varphi_1\uparrow,\varphi_2\uparrow):=\begin{cases}\{\varphi_1\uparrow\}\text{ if }\varphi_2\uparrow\in\mathcal{E}^*(\varphi_1\uparrow)\\ \{\varphi_2\uparrow\}\text{ if }\varphi_1\uparrow\in\mathcal{E}^*(\varphi_2\uparrow)\\ \emptyset\text{ else}\end{cases}\end{gathered}\tag{24}$$

Eq. (24) produces a multiple time event to cope with the possibility of non-occurrence of $\varphi_1\uparrow$ and $\varphi_2\uparrow$.

Notice that the following definition is not correct because $\varphi_1\uparrow$ and $\varphi_2\uparrow$ must have occurred to provide a non-empty value for $\max(\varphi_1\uparrow,\varphi_2\uparrow)$:

$$\max(\varphi_1\uparrow,\varphi_2\uparrow)\;:\neq\;\left\{\begin{cases}\{\varphi_1\uparrow\}\text{ if }\varphi_1\uparrow\ge\varphi_2\uparrow\\ \{\varphi_2\uparrow\}\text{ if }\varphi_2\uparrow\ge\varphi_1\uparrow\\ \emptyset\text{ else}\end{cases}\right.$$

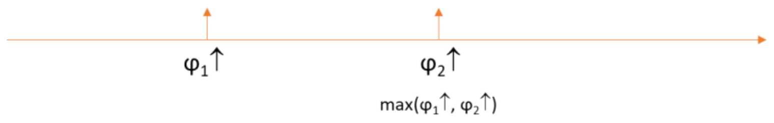

Fig. 14. Event max operator

3.5.7. Elapsed operator

The elapsed operator computes the elapsed time between two events $\varphi_1\uparrow$ and $\varphi_2\uparrow$, $\varphi_2\uparrow$ occurring after $\varphi_1\uparrow$, which means that $\varphi_1\uparrow\in\ \mathcal{E}^*(\varphi_2\uparrow)$. It is denoted $-$:

$$\begin{aligned} -\colon\ \mathcal{E}^*(\varphi_2\uparrow)\times\ \mathcal{E}^*(\varphi_2\uparrow) &\longrightarrow\ \mathbb{R}^+\ \text{such that}\ \varphi_2\uparrow\ \geq \varphi_1\uparrow \\ (\varphi_1\uparrow,\varphi_2\uparrow) &\longmapsto \varphi_2\uparrow - \ \varphi_1\uparrow = \ d \end{aligned} \tag{25}$$

where $d\in\mathbb{R}^+$. d is the elapsed time, or delay between $\varphi_1\uparrow$ and $\varphi_2\uparrow$. The value of $\varphi_2\uparrow - \varphi_1\uparrow$ is not known until $\mathcal{E}^*(\varphi_2\uparrow)$ exists, i.e. until the second event $\varphi_2\uparrow$ has occurred.

Notice that Eq. (25) is consistent with Eq. (17), (19) and (20), but not equivalent since they have different definition domains:

$$\varphi_1\uparrow - \ \varphi_2\uparrow = 0 \ \Leftrightarrow\ \ \varphi_1\uparrow = \ \varphi_2\uparrow$$

$$\varphi_2\uparrow - \ \varphi_1\uparrow \geq 0 \ \Leftrightarrow\ \varphi_2\uparrow \geq \varphi_1\uparrow$$

$$\varphi_2\uparrow - \ \varphi_1\uparrow > 0 \ \Leftrightarrow\ \varphi_2\uparrow > \varphi_1\uparrow$$

If $\varphi_1\uparrow$ occurs at time t_1 and $\varphi_2\uparrow$ occurs at time t_2, then the elapsed time between the two events is given by $\varphi_2\uparrow - \varphi_1\uparrow$.

$$t_2 - t_1 = @\varphi_2\uparrow - @\varphi_1\uparrow = \ \varphi_2\uparrow - \varphi_1\uparrow \tag{26}$$

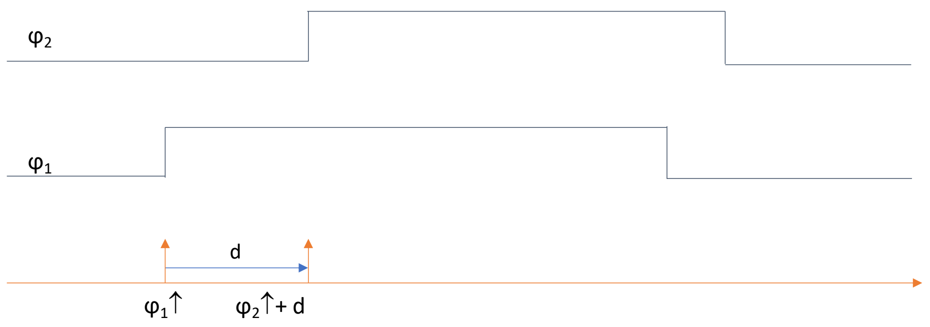

Fig. 15. Event elapsed operator

3.5.8. Delay operator

The delay operator applied on event $\varphi_1\uparrow$ generates a second event $\varphi_2\uparrow = \varphi_1\uparrow + d$ after $\varphi_1\uparrow$ where $d\in\mathbb{R}^+$ is the delay between $\varphi_1\uparrow$ and $\varphi_2\uparrow$. It is formally defined as follows:

$$\begin{aligned} +\colon\ \mathcal{E}\times\mathbb{R}^+ &\longrightarrow\ 2^{\mathcal{E}} \\ (\varphi_1\uparrow,\ d) &\longmapsto\ \varphi_1\uparrow + d\ := \{\varphi_2\uparrow \text{such that}\ @\varphi_2\uparrow = @\varphi_1\uparrow + d\,\} \end{aligned} \tag{27}$$

Therefore $\varphi_1 \uparrow + d$ is the empty set as long as $\varphi_1 \uparrow + d$ has not occurred, and a set that contains one event when $\varphi_1 \uparrow + d$ has occurred.

Notice that Eq. (27) is consistent with Eq. (25) because:

$$@\varphi_2 \uparrow \; = \; @\varphi_1 \uparrow + d \;\Leftrightarrow\; \varphi_2 \uparrow - \; \varphi_1 \uparrow \; = \; d$$

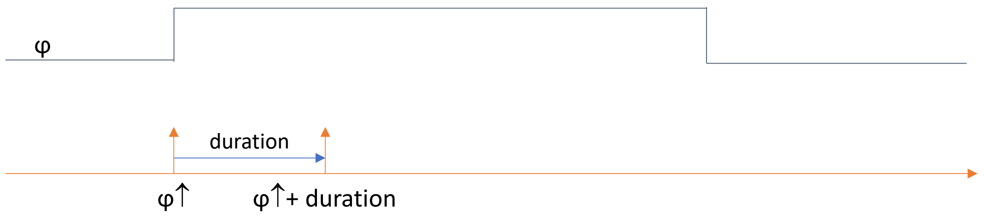

Fig. 16. Event delay operator

3.5.9. Conjunction operator

The conjunction operator ∧ applied on two events $\varphi_1 \uparrow$ and $\varphi_2 \uparrow$ generates the event $\varphi \uparrow = \varphi_1 \uparrow = \varphi_2 \uparrow$ if $\varphi_1 \uparrow$ and $\varphi_2 \uparrow$ occur at the same instant in time. It is formally defined as follows:

$$\begin{gathered} \wedge : \mathcal{E} \times \mathcal{E} \longrightarrow \; 2^{\mathcal{E}} \\ (\varphi_1 \uparrow, \varphi_2 \uparrow) \;\longmapsto \varphi_1 \uparrow \;\wedge\; \varphi_2 \uparrow := \{\, \varphi \uparrow \text{such that } \varphi \uparrow = \varphi_1 \uparrow = \varphi_2 \uparrow \} \end{gathered} \tag{28}$$

Therefore $\varphi_1 \uparrow \;\wedge\; \varphi_2 \uparrow$ is the empty set if $\varphi_1 \uparrow$ or $\varphi_2 \uparrow$ have not occurred, or if $\varphi_1 \uparrow \neq \varphi_2 \uparrow$, and a set that contains the event $\varphi_1 \uparrow$ if $\varphi_1 \uparrow$ has occurred and $\varphi_1 \uparrow = \varphi_2 \uparrow$.

The conjunction operator ∧ applied on an event $\varphi \uparrow$ and a multiple event Ω yields a multiple event $\{\, \varphi \uparrow \}$ if $\varphi \uparrow$ is an event of Ω. It is formally defined as follows:

$$\begin{gathered} \mathcal{E} \times \mathcal{D} \longrightarrow \; \mathcal{D} \\ (\varphi \uparrow, \Omega) \;\longmapsto \varphi \uparrow \;\wedge\; \Omega \; := \{\varphi \uparrow\} \cap \Omega \end{gathered} \tag{29}$$

3.5.10. Disjunction operator

The disjunction operator ∨ applied on two events $\varphi_1 \uparrow$ and $\varphi_2 \uparrow$ yields a multiple event Ω that contains the two events $\varphi_1 \uparrow$ and $\varphi_2 \uparrow$. It is formally defined as follows:

$$\begin{gathered} \vee : \mathcal{E} \times \mathcal{E} \longrightarrow \; \mathcal{D} \\ (\varphi_1 \uparrow, \varphi_2 \uparrow) \;\longmapsto \varphi_1 \uparrow \;\vee\; \varphi_2 \uparrow := \{\, \varphi_1 \uparrow, \varphi_2 \uparrow \} \end{gathered} \tag{30}$$

Therefore $\varphi_1 \uparrow \;\vee\; \varphi_2 \uparrow$ is the empty set if $\varphi_1 \uparrow$ and $\varphi_2 \uparrow$ have not occurred, and a set that contains the event $\varphi_1 \uparrow$ if $\varphi_1 \uparrow$ has occurred and the event $\varphi_2 \uparrow$ if $\varphi_2 \uparrow$ has occurred.

The disjunction operator ∨ applied on an event $\varphi \uparrow$ and a multiple event Ω yields a multiple event Φ that contains $\varphi \uparrow$ and the events of Ω. It is formally defined as follows:

$$\begin{gathered} \mathcal{E} \times \mathcal{D} \longrightarrow \; \mathcal{D} \\ (\varphi \uparrow, \Omega) \longmapsto \varphi \uparrow \;\vee\; \Omega \; := \{\varphi \uparrow\} \cup \Omega \end{gathered} \tag{31}$$

3.5.11. Composition of Booleans with events

A Boolean φ can be composed with and event $\psi \uparrow$ as follows:

$$\begin{gathered} op\colon \mathbb{B} \times \mathcal{E} \longrightarrow \; \mathbb{B} \\ (\varphi, \psi \uparrow) \longmapsto \varphi \;\; op \;\; \psi \uparrow := \;\; \varphi \;\; op \begin{cases} \quad true \text{ if } t = @\psi \uparrow \\ undefined \text{ if } t \neq @\psi \uparrow \end{cases} \end{gathered} \tag{32}$$

where op denotes the binary logical operators ∧ or ∨.

A Boolean φ can be composed with a multiple event Ω as follows:

$$\begin{gathered} op\colon\ \mathbb{B}\times\mathcal{D}\longrightarrow\ \mathbb{B} \\ (\varphi,\Omega)\longmapsto\varphi\ op\ \Omega\ :=\big((\varphi\ op\ \Omega^1)\ op\ \Omega^2\big)\ldots\ op\ \Omega^{|\Omega|} \end{gathered} \tag{33}$$

where op denotes the binary logical operators ∧ or ∨.

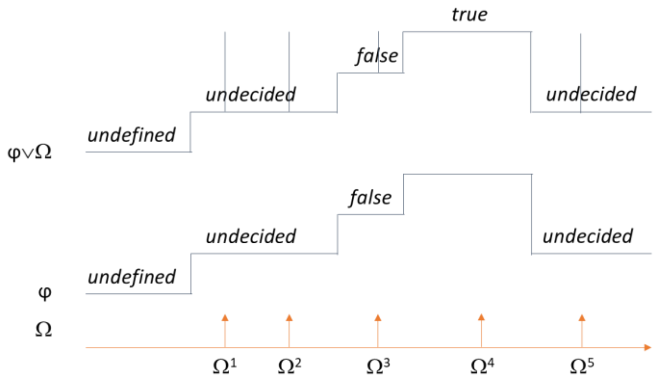

Fig. 17. Composition of Booleans with events

3.6. Discrete clocks

A discrete clock Ω is a multiple event Ω. Therefore, the terms ‘discrete clock’ and ‘multiple event’ are synonymous.

It is therefore a set of events ordered by increasing occurrences in time:

$$\Omega\ :=\ \{\varphi_1\uparrow,\ \varphi_2\uparrow,\ldots,\ \varphi_i\uparrow,\ldots\}\ \text{such that}\ \forall i,\forall j>i,\ @\varphi_i\uparrow\ \leq\ @\varphi_j\uparrow \tag{34}$$

The $\varphi_i\uparrow$ in Eq. (34) are called the clock ticks. The rank of the clock tick $\varphi_i\uparrow$ in Ω is denoted $\mathrm{rank}(\varphi_i\uparrow\ \in\Omega)$.

Let us consider a Boolean ψ that becomes true at each event $\varphi_i\uparrow$ (e.g., $\psi=false\vee\Omega$). Then Ω is generated by ψ. This is denoted $\Omega(\psi)$:

$$\Omega(\psi)\ :=\ \{\psi(t_1)\uparrow,\ \psi(t_2)\uparrow,\ldots,\ \psi(t_i)\uparrow,\ldots\} \tag{35}$$

where the $t_i=@\varphi_i\uparrow$ are the time instants when ψ becomes *true* (i.e., such that $\psi(t_i)\uparrow\ =\ \varphi_i\uparrow$) for all $i\in\mathbb{N}^*, 1\leq i\leq|\Omega(\psi)|$. When no confusion is possible between clocks and events, it is also possible to write $\Omega(\psi)=\psi\uparrow$.

The chosen Boolean ψ that generates $\Omega(\psi)$ is denoted $\psi(\Omega)$. If the clock $\Omega(\psi)$ is written $\Omega(\psi)=\psi\uparrow$, then $\psi(\Omega)=\psi$.

Note that $\Omega(false)=\emptyset$, which can be denoted $false\uparrow=\emptyset$.

The discrete time associated to a discrete clock is given by the sequence number i of the clock tick, but each clock tick is associated with its time occurrence in the continuous clock domain. The ith tick of clock Ω is denoted $\Omega^i=\psi(t_i)\uparrow$**.** It is occurring at discrete time $i\in\mathbb{N}^*$ in the discrete clock domain Ω and at continuous time $t_i\in\mathbb{R}$ in the continuous clock domain $\mathfrak{C}$.

$$\Omega(\psi) := \{\Omega^1, \Omega^2, \dots, \Omega^i, \dots\} \tag{36}$$

where $\Omega^i = \psi(t_i)\uparrow$ for all $i \in \mathbb{N}^*, 1 \le i \le |\Omega(\psi)|$.

If the notation $\psi\uparrow$ is used to denote clock $\Omega(\psi)$, then $\Omega^i = (\psi\uparrow)^i$. Then the event $\psi\uparrow$ can be considered as the first tick of clock $\psi\uparrow$:

$$\psi\uparrow \equiv (\psi\uparrow)^1 \tag{37}$$

The rank function is defined as:

$$\begin{gathered}\text{rank: } \Omega \longrightarrow \mathbb{N} \\ \Omega^i \longmapsto \text{rank}(\Omega^i) = i\end{gathered} \tag{38}$$

The discrete time associated to clock Ω is given by the number of ticks $|\Omega|$ of Ω. $|\Omega|$ increases in time by one unit at each new tick $\Omega^{|\Omega|}$. $\Omega^{|\Omega|}$ is thus called the *current tick* of clock Ω. $|\Omega| = 0$ before the first tick (the set Ω is empty). The discrete time associated to Ω is not defined when Ω is empty because when $|\Omega| = 0$, $\Omega^{|\Omega|}$ is not defined (Ω^i is defined for $i \ge 1$).

$$|\Omega| = \begin{cases} 0 \text{ if } t < @\Omega^1 \\ i \text{ if } @\Omega^i \le t < @\Omega^{i+1} \end{cases} \tag{39}$$

where $@\Omega^i$ denotes the instant of occurrence of tick Ω^i.

$\Omega(i)$ denotes the value of Ω at instant $@\Omega^i$:

$$\Omega(i) := \Omega(\Omega^i) = \{\Omega^1, \Omega^2, \dots, \Omega^i\} \tag{40}$$

Therefore, $\Omega \neq \emptyset \Longrightarrow \Omega = \Omega(|\Omega|)$.

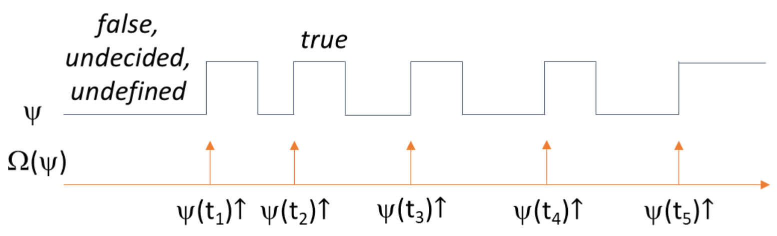

Fig. 18. Discrete clock generated by a Boolean

The domain of discrete clocks is denoted $\mathcal{D}$. Then $\mathcal{D} = 2^{\mathcal{E}}$ (the set of the subsets of $\mathcal{E}$).

The set of the subsets of $\mathcal{E}(\varphi\uparrow)$ is denoted $\mathcal{D}(\varphi\uparrow)$: $\mathcal{D}(\varphi\uparrow) = 2^{\mathcal{E}(\varphi\uparrow)}$.

3.7. Operators on discrete clocks

3.7.1. Projection operator

The projection $\varphi\uparrow/\Omega$ of an event $\varphi\uparrow$ on a discrete clock Ω is the first tick of Ω that follows $\varphi\uparrow$. It is formally defined as follows:

$$\begin{aligned} / : \mathcal{E} \times \mathcal{D} &\longrightarrow \mathcal{D} \\ (\varphi\uparrow, \Omega) &\longmapsto \varphi\uparrow/\Omega := \begin{cases} \left\{ \Omega^i = \min_{j \geq i}(\Omega^j \geq \varphi\uparrow) \text{ if } \exists k \text{ such that } \Omega^k \geq \varphi\uparrow \right\} \\ \emptyset \text{ if } \nexists k \text{ such that } \Omega^k \geq \varphi\uparrow \end{cases} \end{aligned} \quad (41)$$

The projection operator Ω_1/Ω_2 of a discrete clock Ω_1 on a discrete clock Ω_2 is defined as follows:

$$\begin{aligned} / : \mathcal{D} \times \mathcal{D} &\longrightarrow \mathcal{D} \\ (\Omega_1, \Omega_2) &\longmapsto \Omega_1/\Omega_2 := \cup_{i=1}^{|\Omega_1|} \Omega_1^i/\Omega_2 \end{aligned} \quad (42)$$

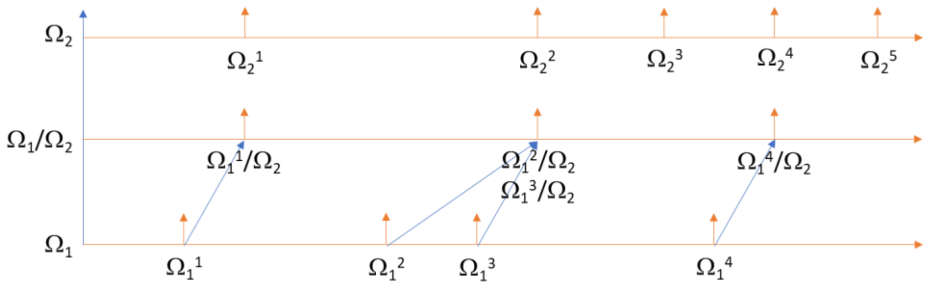

Fig. 19. Discrete clock projection operator

The projection t/Ω of a time instant $t \in \mathfrak{C}$ on a discrete clock Ω is the first tick of Ω that follows t. It is formally defined as follows:

$$\begin{aligned} / : \mathfrak{C} \times \mathcal{D} &\longrightarrow \mathcal{D} \\ (t, \Omega) &\longmapsto t/\Omega := \begin{cases} \left\{ \Omega^i = \min_{j \geq i}(@\Omega^j \geq t) \text{ if } \exists k \text{ such that } @\Omega^k \geq t \right\} \\ \emptyset \text{ if } \nexists k \text{ such that } @\Omega^k \geq t \end{cases} \end{aligned} \quad (43)$$

It follows from this definition that the projection of the continuous clock $\mathfrak{C}$ on a discrete clock Ω and that the projection of a discrete clock Ω on the continuous clock $\mathfrak{C}$ are the discrete clock Ω itself:

$$\mathfrak{C}/\Omega = \Omega \quad (44)$$

$$\Omega/\mathfrak{C} = \Omega \quad (45)$$

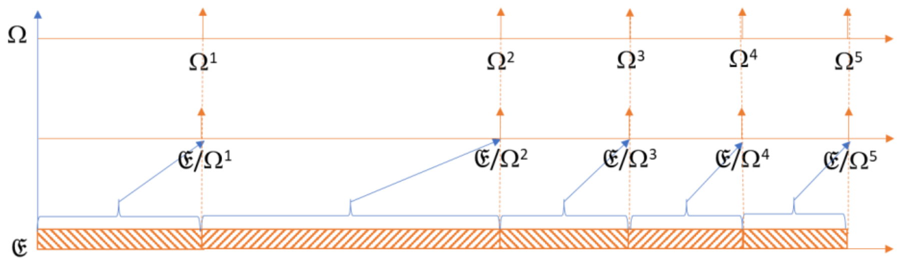

Fig. 20. Continuous clock projection operator on a discrete clock

3.7.2. Bounded projection operator

The bounded projection $\varphi\uparrow/{}^d\Omega$ of an event $\varphi\uparrow$ on a discrete clock Ω is the first tick of Ω that follows $\varphi\uparrow$ if it is delayed by at most $d \in \mathbb{R}^+$ from $\varphi\uparrow$. Otherwise, the projection is lost. It is formally defined as follows:

$$\begin{aligned} /^{d}:\ & \mathcal{E}\times\mathcal{D}\times\mathbb{R}^{+}\longrightarrow\ \mathcal{D} \\ (\varphi\uparrow,\Omega,d)\ \mapsto\ \varphi\uparrow/^{d}\Omega\ & := \begin{cases} \left\{\Omega^{i}=\min_{j\geq i}(\varphi\uparrow+d\geq\Omega^{j}\geq\varphi\uparrow)\ \text{if}\ \exists k\ \text{such that}\ \varphi\uparrow+d\geq\Omega^{k}\geq\ \varphi\uparrow\right\} \\ \emptyset\ \text{if}\ \nexists k\ \text{such that}\ \varphi\uparrow+d\geq\Omega^{k}\geq\ \varphi\uparrow \end{cases} \end{aligned} \quad (46)$$

The projection operator $\Omega_1/^{d}\Omega_2$ of a discrete clock Ω_1 on a discrete clock Ω_2 is defined as follows:

$$\begin{aligned} /^{d}:\ & \mathcal{D}\times\mathcal{D}\longrightarrow\ \mathcal{D} \\ (\Omega_1,\Omega_2)\ \mapsto\ & \Omega_1/^{d}\Omega_2 := \bigcup_{i=1}^{|\Omega_1|}\Omega_1^{i}/^{d}\Omega_2 \end{aligned} \quad (47)$$

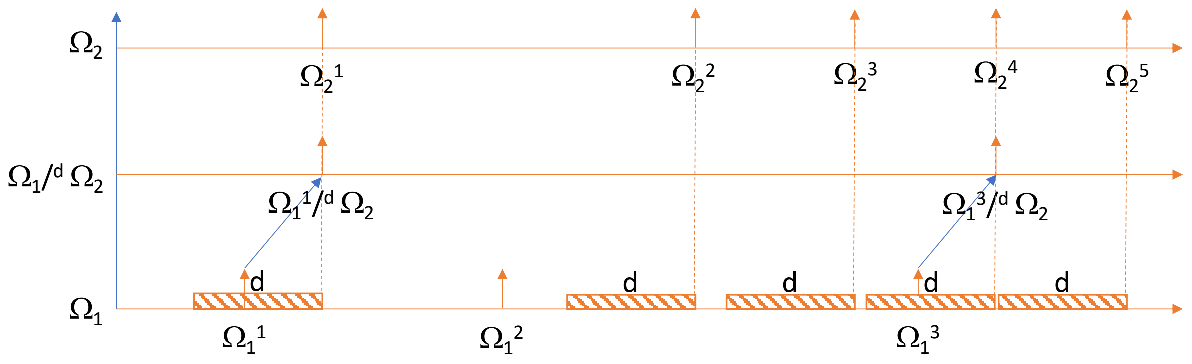

Fig. 21. Discrete clock bounded projection operator

The bounded projection $t/^{d}\Omega$ of a time instant $t\in\mathfrak{C}$ on a discrete clock Ω is the first tick of Ω that follows t if it is delayed by at most $d\in\mathfrak{C}$ from t. Otherwise, the projection is lost. It is formally defined as follows:

$$\begin{aligned} /:\ & \mathfrak{C}\times\mathcal{D}\times\mathbb{R}^{+}\longrightarrow\ \mathcal{D} \\ (t,\Omega,d)\ \mapsto\ t/^{d}\Omega\ & := \begin{cases} \left\{\Omega^{i}=\min_{j\geq i}(t+\ d\geq @\Omega^{j}\geq t)\ \text{if}\ \exists k\ \text{such that}\ t+\ d\geq @\Omega^{k}\geq\ t\right\} \\ \emptyset\ \text{if}\ \nexists k\ \text{such that}\ t+\ d\geq @\Omega^{k}\geq\ t \end{cases} \end{aligned} \quad (48)$$

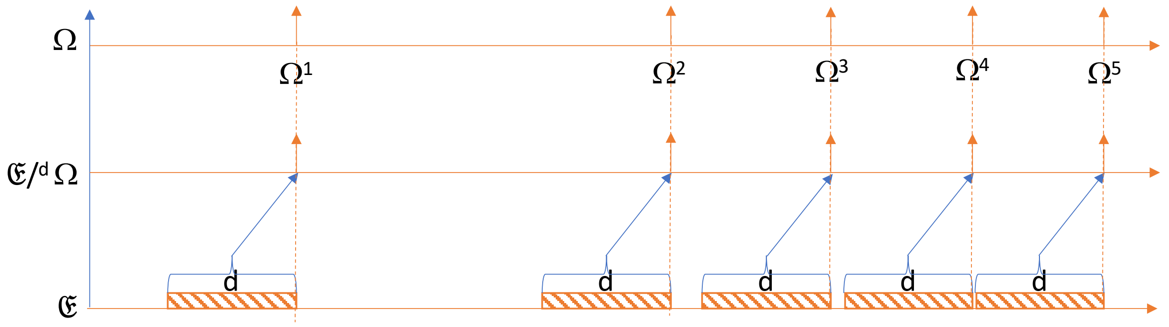

Fig. 22. Continuous clock bounded projection operator on a discrete clock

Note that d is a function of time (it is not a fixed quantity).

3.7.3. Delay operator

The delay operator denoted $+$ applied on tick Ω^i of clock Ω delays Ω^i by n ticks. It is formally defined as follows:

$$\begin{aligned} +:\ \Omega \times \mathbb{N} &\longrightarrow\ \ \Omega \\ (\Omega^i, n) &\longmapsto\ \Omega^i + n\ :=\ \Omega^{i+n} \end{aligned} \tag{49}$$

where $n \in \mathbb{N}$ is the delay expressed in number of ticks of Ω.

Note that tick Ω^{i+n} can never occur.

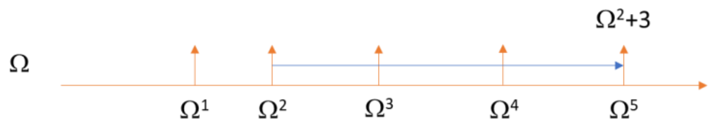

Fig. 23. Clock tick delay operator

The delay operator denoted $+$ applied on clock $\Omega_1(\Omega_1^i)$ generates a second clock Ω_2 whose ticks are delayed by n ticks of Ω_1. It is formally defined as follows:

$$\begin{aligned} +:\ \mathcal{D} \times \mathbb{N} &\longrightarrow\ \ \mathcal{D} \\ \left(\Omega_1(\Omega_1^i), n\right) &\longmapsto \Omega_2 = \Omega_1 + n \coloneqq \left\{ \Omega_1^j + n,\ \ i \le j \le |\Omega_1| \right\} \end{aligned} \tag{50}$$

where $n \in \mathbb{N}$ is the delay expressed in number of ticks of Ω_1.

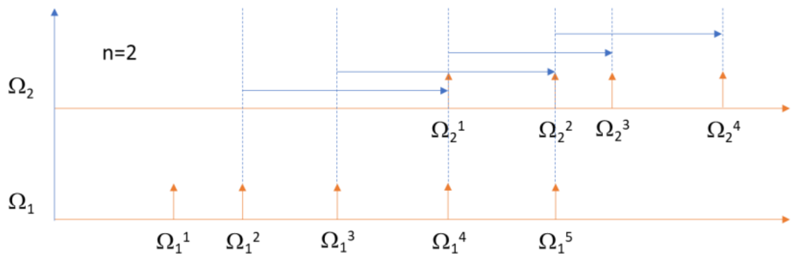

Fig. 24. Discrete clock delay operator

3.7.4. Elapsed operator

The elapsed operator denoted $-$ applied on two ticks Ω^i and Ω^j of clock Ω, Ω^j occurring after Ω^i, yields the elapsed time (or delay) between Ω^i and Ω^j. It is formally defined as follows:

$$\begin{aligned} -:\ \Omega(\Omega^i) \times \Omega(\Omega^j) &\longrightarrow\ \mathbb{N} \text{ such that } j \ge i \\ (\Omega^i, \Omega^j) &\longmapsto\ \Omega_j - \Omega_i := j - i \end{aligned} \tag{51}$$

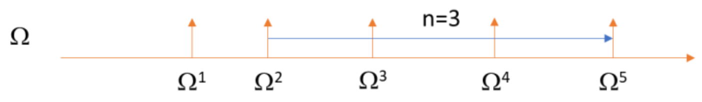

Fig. 25. Discrete clock elapsed operator

3.7.5. Conjunction operator

The conjunction operator ∧ applied on two clocks Ω_1 and Ω_2 yields a third clock Ω that contains all ticks of Ω_1 and Ω_2 that are equal. It is formally defined as follows:

$$\begin{gathered} \wedge : \mathcal{D} \times \mathcal{D} \longrightarrow \ \mathcal{D} \\ (\Omega_1, \Omega_2) \ \longmapsto \Omega = \Omega_1 \wedge \ \Omega_2 \coloneqq \Omega_1 \cap \ \Omega_2 \end{gathered} \tag{52}$$

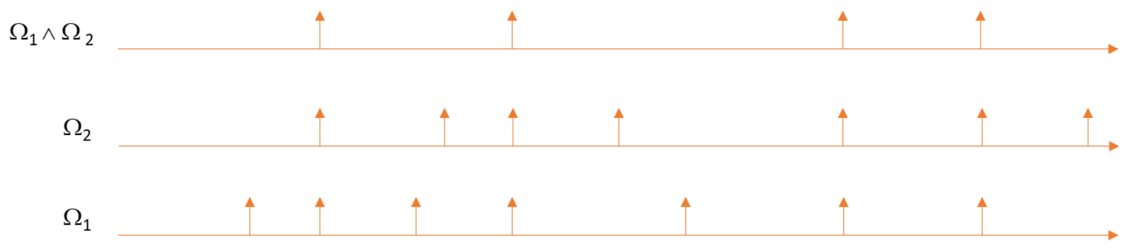

Fig. 26. Discrete clock conjunction operator

3.7.6. Disjunction operator

The disjunction operator ∨ applied on two clocks Ω_1 and Ω_2 yields a third clock Ω that contains all ticks of Ω_1 and Ω_2. It is formally defined as follows:

$$\begin{gathered} \vee : \mathcal{D} \times \mathcal{D} \longrightarrow \ \mathcal{D} \\ (\Omega_1, \Omega_2) \ \longmapsto \Omega = \Omega_1 \vee \ \Omega_2 \coloneqq \Omega_1 \cup \ \Omega_2 \end{gathered} \tag{53}$$

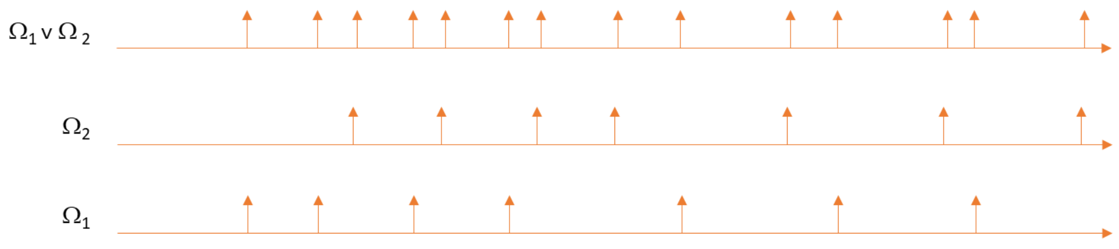

Fig. 27. Discrete clock disjunction operator

3.7.7. Filter operator

The filter operator applied on clock Ω filters (i.e., removes) all ticks that do not satisfy a Boolean condition. It is formally defined as follows:

$$\begin{gathered} \text{filter: } \mathcal{D} \times \mathbb{O}(\Omega \longrightarrow \mathbb{B}) \longrightarrow \ \mathcal{D} \\ (\Omega_1, \text{cond}) \longmapsto \Omega_2 = \Omega_1\big(\text{cond}(*)\big) \coloneqq \left\{ \Omega_1^i \ \in \Omega_1 \text{ such that cond}\left(\Omega_1^i \right) = true \right\} \end{gathered} \tag{54}$$

where $*$ denotes a dummy variable that represent any element $\Omega_1^i \in \Omega_1$, and cond is a Boolean condition on Ω_1^i.

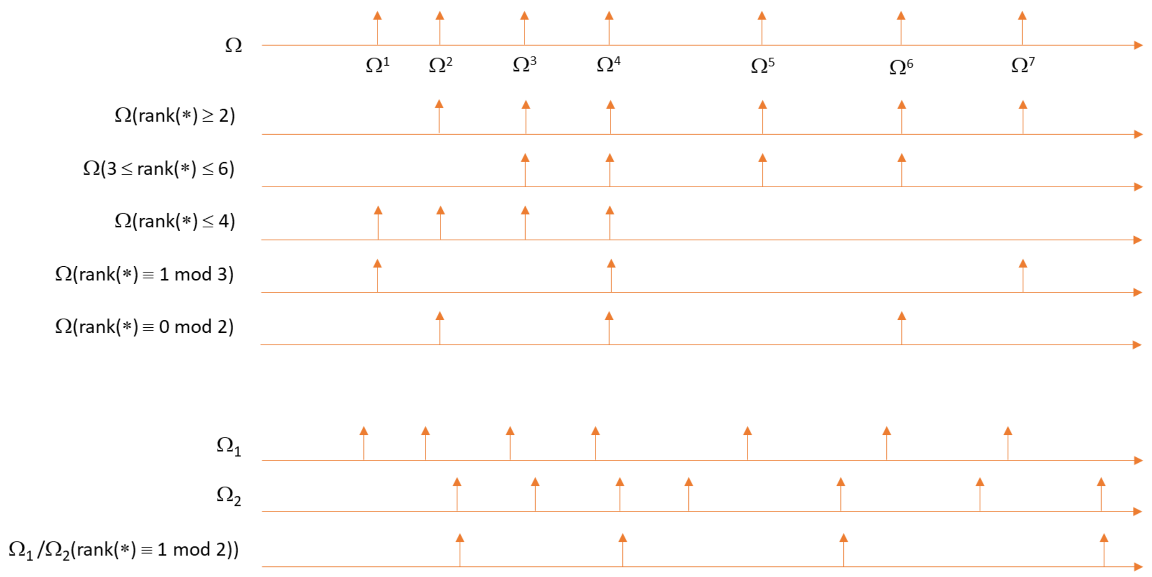

Fig. 28. Discrete clock filter operator

3.7.8. Extending operators on a discrete clock to the continuous time domain

To be simulated, operators defined on a discrete clock must be extended to the continuous time domain $\mathfrak{C}$.

Let $f(\Omega)$ be a function that takes the clock Ω as argument:

$$\begin{aligned} f : \mathcal{D} &\longrightarrow \mathbb{D} \\ \Omega &\longmapsto f(\Omega) \end{aligned} \tag{55}$$

where $\mathbb{D}$ can be any domain.

The extension $\bar{f}$ of f to the continuous time domain $\mathfrak{C}$ is defined as follows:

$$\begin{aligned} [\, @\Omega^1, +\infty\, [&\longrightarrow \mathbb{D} \\ t \longmapsto \bar{f}(t) &= \begin{cases} f(\Omega^i) \text{ for } @\Omega^i \le t < @\Omega^{i+1} \text{ if } 1 \le i < |\Omega| \\ f\left(\Omega^{|\Omega|}\right) \text{ for } t \ge @|\Omega| \end{cases} \end{aligned} \tag{56}$$

The definition given by Eq. (56) can be extended to functions $f(\Omega_1, \dots, \Omega_i, \dots, \Omega_n)$ defined on multiple clocks $\Omega_1, \dots \Omega_i, \dots \Omega_n$ as follows:

$$\begin{aligned} [\, @\Omega_1^1, +\infty\, [\times \dots \times\ [\, @\Omega_i^1, +\infty\, [\times \dots \times [\, @\Omega_n^1, +\infty\, [&\longrightarrow \mathbb{D} \\ (t_1, \dots t_i, \dots t_n) \longmapsto \bar{f}(t_1, \dots t_i, \dots t_n) &= f\left(\Omega_1^{j_1}, \dots, \Omega_i^{j_i}, \dots, \Omega_n^{j_n}\right) \text{ for } @\Omega_i^{j_i} \le t < @\Omega_i^{j_i+1} \end{aligned} \tag{57}$$

with the convention that $@\Omega_i^{j_i+1} = +\infty$ if $j_i = |\Omega_i|$. Note that $\bar{f}$ is undefined until all clocks have their first tick.

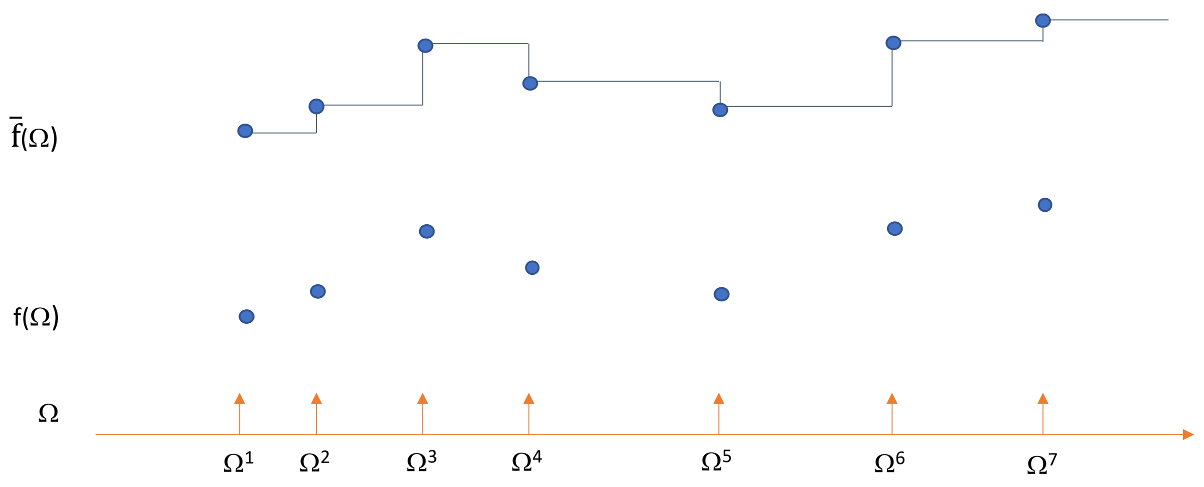

Fig. 29. Extending operators on discrete clocks to the continuous time domain

3.8. Single time periods

A single time period P is a time interval between two events $\varphi_1 \uparrow$ and $\varphi_2 \uparrow$. The left and right boundaries can be included or excluded (this is indicated by the notation [|]).

$$P \;:=\; [\,|\,]\, \varphi_1 \uparrow,\ \varphi_2 \uparrow\ [\,|\,] \tag{58}$$

$\varphi_1 \uparrow$ is called the opening event. It is denoted $P \uparrow$.

$\varphi_2 \uparrow$ is called the closing event. It is denoted $P \downarrow$.

Therefore, Eq. (58) can be rewritten as follows:

$$P \;:=\; [\,|\,]\, P \uparrow,\ P \downarrow\ [\,|\,] \tag{59}$$

The domain of single time periods is denoted $\mathcal{P}$.

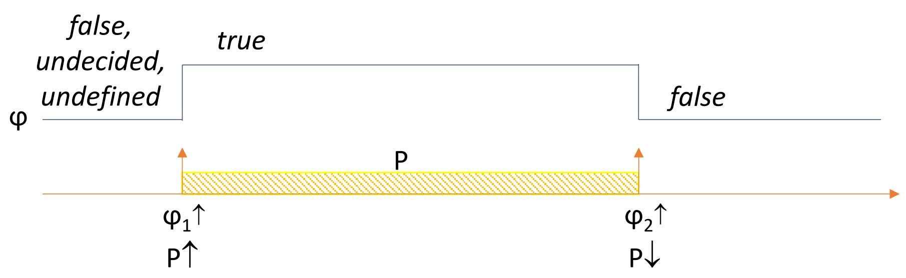

Fig. 30. Single time period

3.9. Multiple time periods

A multiple time period P is a set of single time periods P_i ordered by opening events increasing in time:

$$P \;:=\; \{\, P_1,\ P_2, \ldots ,\ P_i, \ldots \}\ \text{such that}\ \forall i, \forall j,\ j > i\ \Rightarrow P_j \uparrow\ \geq\ P_i \uparrow \tag{60}$$

Multiple time periods P containing single time periods opened at all occurrences of $\varphi_1 \uparrow$ and closed at all occurrences of $\varphi_2 \uparrow$ are denoted

$$P \ := \ \Pi([\,|\,]\,\varphi_1 \uparrow,\ \varphi_2 \uparrow\ [\,|\,]) \tag{61}$$

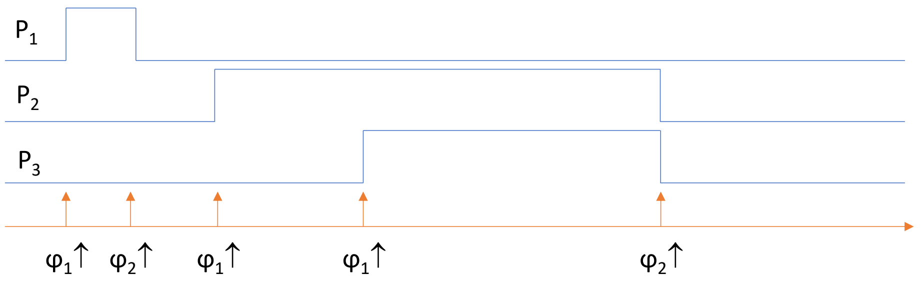

Fig. 31. Multiple time period

The domain of multiple time periods is denoted $2^{\mathcal{P}}$.

A non-overlapping time period P is a multiple time period such that there is no time overlaps between the single time periods P_i of the multiple time period P. It is formally defined as follows:

$$P \ := \ \{\,P_1,\ P_2, \dots ,\ P_i, \dots\} \ \text{ such that } \forall i \in [1, |P| - 1],\ P_i \downarrow \ < \ P_{i+1} \uparrow \tag{62}$$

The non-overlapping time period $P(\psi)$ generated by the Boolean ψ is defined as:

$$P(\psi) \ := \ \Pi([\,|\,]\,\psi \uparrow,\ \psi \downarrow\ [\,|\,]) \tag{63}$$

where $\psi \downarrow := \ \neg\psi \uparrow$.

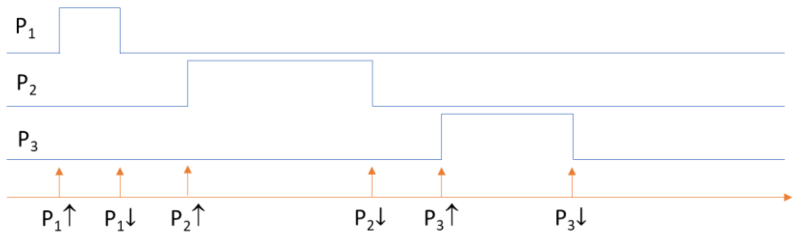

Fig. 32. Non-overlapping time period

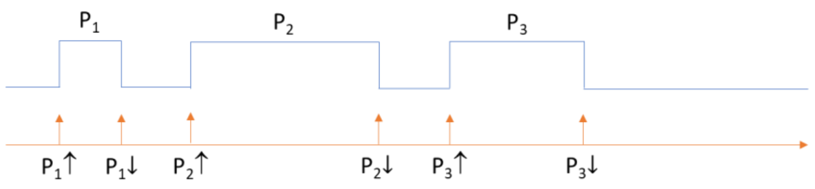

Fig. 33. Reduced graphical representation of a non-overlapping time period

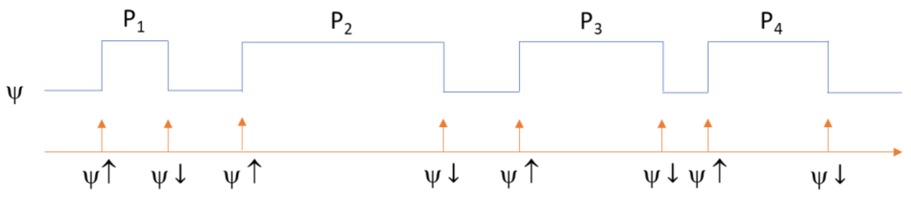

Fig. 34. Non-overlapping time period generated by a Boolean

3.10. Operators on multiple time periods

3.10.1. Intersection operator

The intersection operator between two single time periods P_1 and P_2 produces a multiple time period P that contains a single time period made of all time instants common to P_1 and P_2. It is formally defined as follows:

$$\begin{array}{c} \cap : \mathcal{P} \times \mathcal{P} \longrightarrow 2^{\mathcal{P}} \\ (P_1, P_2) \longmapsto P = P_1 \cap P_2 := \begin{cases} \{[\max(P_1 \uparrow, P_2 \uparrow), \min(P_1 \downarrow, P_2 \downarrow)]\} \text{ if } \max(P_1 \uparrow, P_2 \uparrow) \leq \min(P_1 \downarrow, P_2 \downarrow) \\ \emptyset \text{ if } \max(P_1 \uparrow, P_2 \uparrow) > \min(P_1 \downarrow, P_2 \downarrow) \end{cases} \end{array} \quad (64)$$

$P_1 \cap P_2$ is a multiple time period which can be empty or contains one single time period. Therefore:

$$0 \leq |P_1 \cap P_2| \leq 1 \quad (65)$$

3.10.2. Truncation operator

The truncation operator between a non-overlapping multiple time period $P_1 = \{P_{1,i}\}_{1 \leq i \leq |P_1|}$ and a multiple time period $P_2 = \{P_{1,j}\}_{1 \leq j \leq |P_2|}$ produces a multiple time period $P = \{P_k\}_{1 \leq k \leq |P|}$ such that all its single time periods P_k are the single time periods $P_{2,j}$ of P_2 truncated by the single time periods $P_{1,i}$ of P_1. It is formally defined as follows:

$$\begin{array}{c} \supset : 2^{\mathcal{P}} \times 2^{\mathcal{P}} \longrightarrow 2^{\mathcal{P}} \\ (P_1, P_2) \longmapsto P = P_1 \supset P_2 := \bigcup_{1 \leq i \leq |P_1|, 1 \leq j \leq |P_2|} P_{1,i} \cap P_{2,j} \end{array} \quad (66)$$

From Eq. (66), in general:

$$0 \leq |P_1 \supset P_2| \leq |P_1||P_2| \quad (67)$$

The non-overlapping time period P_1 such that $P_1 \supset P_2$ is called the frame time period of P_2.

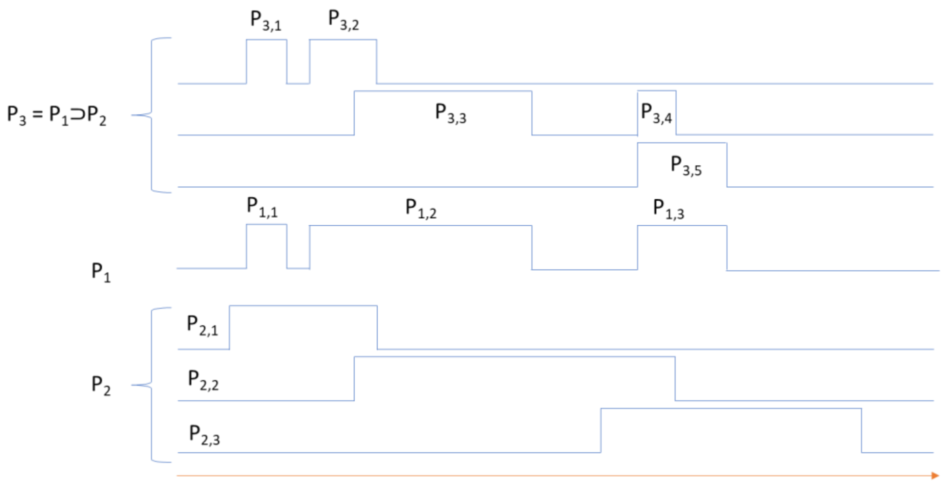

Fig. 35. Truncation of a multiple time period

3.11. Temporal operators on Booleans

3.11.1. Accumulation operator

The accumulation operator, denoted $+$, is applied on two different values of the same Boolean φ at two different instants t_1 and t_2:

$$\begin{aligned} +\colon \ \mathbb{B} \times \mathbb{B} &\longrightarrow \ \mathbb{B} \\ \big(\varphi(t_1), \varphi(t_2)\big) &\mapsto \ \varphi(t_1) + \varphi(t_2) \end{aligned} \tag{68}$$

The values of the accumulation operator are given by the following truth table:

Table 5. Truth table for the Boolean accumulation operator

$+$	**Accumulation operator**			
$\boldsymbol{\varphi(t_1) + \varphi(t_2)}$	***true***	***false***	***undecided***	***undefined***
true	*true*	*false*	*true*	*true*
false	*false*	*false*	*false*	*false*
undecided	*true*	*false*	*undecided*	*undecided*
undefined	*true*	*false*	*undecided*	*undefined*

The purpose of the accumulation operator is to define the policy of requirement satisfaction (cf. Section 3.12).

The accumulation operator verifies the following properties:

- Commutativity: $\varphi(t_1) + \varphi(t_2) = \varphi(t_2) + \varphi(t_1)$
- Associativity: $\big(\varphi(t_1) + \varphi(t_2)\big) + \varphi(t_3) = \varphi(t_1) + \big(\varphi(t_2) + \varphi(t_3)\big)$

- Idempotence: $\varphi(t) + \varphi(t) = \varphi(t)$

3.11.2. Filter operator

The filter operator is formally defined as follows:

$$\begin{aligned} \times : \mathbb{B} \times \mathbb{B} &\longrightarrow \mathbb{B} \\ (a, \varphi) &\longmapsto a \times \varphi \end{aligned} \tag{69}$$

where a and φ are two Booleans at the same instant in time. a is the filter.

The truth table of the filter operator is given below.

Table 6. Truth table for the Boolean filter operator

$\times$	**Filter operator**			
a* × *φ	***true***	***false***	***undecided***	***undefined***
true	*true*	*false*	*undecided*	*undefined*
false	*undecided*	*undecided*	*undecided*	*undefined*
undecided	*undecided*	*undecided*	*undecided*	*undefined*
undefined	*undefined*	*undefined*	*undefined*	*undefined*

The purpose of the filter operator is to filter out events that are not decision events.

3.11.3.Duration operator

The duration of a Boolean φ over a single time period P is the elapsed time while φ equals *true* inside P. It is formally defined as follows:

$$\begin{aligned} \text{duration:}\ \mathbb{B} \times \mathcal{P} &\longrightarrow \mathbb{R}^{+} \\ (\varphi, P) \longmapsto \text{duration}(\varphi, P) &\coloneqq \int_{t \in P} [\![\varphi]\!]\, dt \end{aligned} \tag{70}$$

where $[\![\ ]\!]$ is the indicator function:

$$\begin{aligned} [\![\ ]\!]\colon \mathbb{B} &\longrightarrow \{0,\ 1\} \\ \varphi \longmapsto [\![\varphi]\!] &= \begin{cases} 1 \text{ if } \varphi = true \\ 0 \text{ if } \varphi \neq true \end{cases} \end{aligned} \tag{71}$$

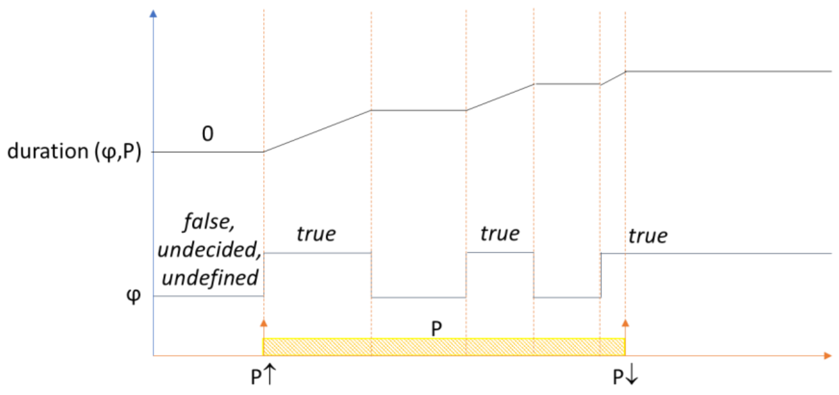

Fig. 36. Boolean duration operator

3.11.4. Delay operator

The delay operator applied on Boolean φ_1 generates a second Boolean φ_2 as formally defined below:

$$\begin{aligned} +: \mathbb{B} \times \mathbb{R}^+ &\longrightarrow \mathbb{B} \\ (\varphi_1, d) &\mapsto \varphi_2 = (\varphi_1 + d)(t) := \varphi_1(t - d) \end{aligned} \tag{72}$$

where $d \in \mathbb{R}^+$ is the delay or elapsed time between φ_1 and φ_2.

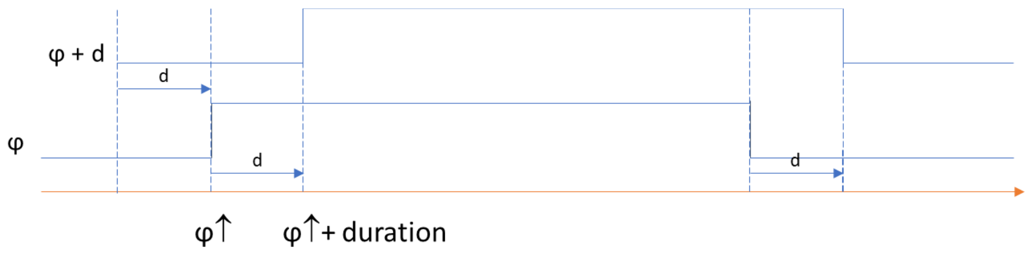

Fig. 37. Boolean delay operator

3.11.5. Sum operator

The sum operator of Boolean φ over the single time period P is formally defined for a discrete clock Ω as follows:

$$\begin{aligned} \Sigma : \mathbb{B} \times \mathcal{P} \times \mathcal{D} &\longrightarrow \mathbb{B} \\ (\varphi, P, \Omega) &\mapsto \Sigma_{P,\Omega}\, \varphi := \begin{cases} undefined & \text{if } \Omega = \emptyset \\ \sum_{@\Omega^{|\Omega|} \in P} \varphi\left(@\Omega^{|\Omega|}\right) & \text{if } \Omega \neq \emptyset \end{cases} \end{aligned} \tag{73}$$

where $\sum_{@\Omega^{|\Omega|} \in P} \varphi\left(@\Omega^{|\Omega|}\right) = \varphi(t_1) + \cdots + \varphi(t_i) + \cdots$ is an accumulator that accumulates the values of $\varphi(t_i)$ such that $t_i = @\Omega^{|\Omega|}$ and $t_i \in P$. Its value is $undefined$ when the accumulator is empty, and varies at each time instant $t_i \in P$ when $|\Omega|$ increases by one unit.

The term *undefined* is introduced to provide a value to the sum even if there are no ticks of Ω within P. Therefore, the value of the sum is *undefined* if there are no ticks of Ω within P, and the sum has a value for any value of $t \in \mathfrak{C}$.

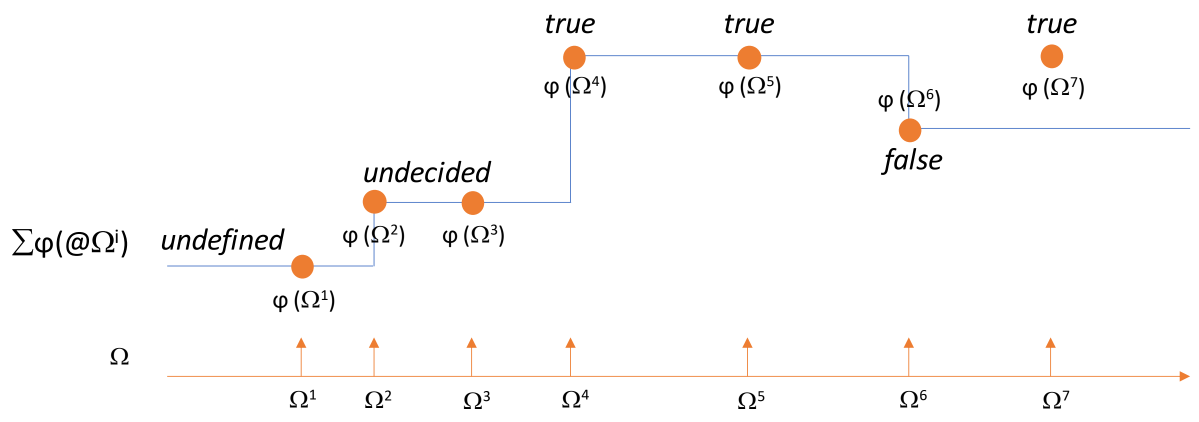

Fig. 38. Sum operator

3.11.6. Integral operator

Let us consider a Boolean $\varphi(t) \in \mathbb{B}$ which is a function of time $t \in \mathfrak{C}$ (like any other variable). Let us consider the single time period P, a time quantum $dt \in \mathbb{R}^{*+}$ and the discrete clock Ω such that

$$\Omega \neq \emptyset \Longrightarrow \Omega^1 = P \uparrow \text{ and } \forall i \in [1, |\Omega| - 1],\ \Omega^{i+1} = \Omega^i + dt \tag{74}$$

Then the integral operator of $\varphi(\tau)$ over the single time period P is formally defined as follows:

$$\begin{aligned} \int : \mathbb{B} \times \mathcal{P} &\longrightarrow \mathbb{B} \\ (\varphi, P) &\mapsto \int_P \varphi := \lim_{dt \to 0} \textstyle\sum_{P,\Omega} \varphi \end{aligned} \tag{75}$$

It corresponds to the extension of the definition of the sum operator for the continuous clock $\mathfrak{C}$. The integral operator sums φ at any time instant $t \in P$. Before $P \uparrow$, $\int_P \varphi = undefined$. Note that the value of $\int_P \varphi$ varies in time between $P \uparrow$ and $P \downarrow$, and stays constant after $P \downarrow$.

3.12. Requirements

3.12.1. Definition

A requirement R corresponding to a Boolean φ associated to a single time period P is denoted $R = \varphi \otimes \{P\}$ and formally defined as the following function:

$$\begin{aligned} \otimes : \mathbb{B} \times \mathcal{P} &\longrightarrow \mathbb{B} \\ (\varphi, P) &\mapsto \varphi \otimes \{P\} := \int_P a(\varphi, P) \times \varphi \end{aligned} \tag{76}$$

The filter $a(\varphi, P) \in \mathbb{B}$ depends on $\varphi \in \mathbb{B}$ and $P \in \mathcal{P}$. $R = \varphi \otimes P$ varies with time $t \in \mathfrak{C}$ when $t \leq P \downarrow$ and stays constant for $t > P \downarrow$.

A decision event is an event that occurs when the decision whether R is satisfied or not can be made, cf. Section 3.12.3. The decision events for condition $\varphi \in \mathbb{B}$ and single time period $P \in \mathcal{P}$ constitute a clock whose ticks are denoted $\mathrm{dev}(\varphi, P) \uparrow$. Then

$$\mathrm{dev}(\varphi, P) \uparrow = \big(a(\varphi, P) \times \varphi \vee \neg(a(\varphi, P) \times \varphi)\big) \uparrow \tag{77}$$

Eq. (77) states that decision events occur when $a(\varphi, P) \times \varphi$ becomes true or false.

In principle, the value $\varphi \otimes P$ should be the same at all decision events, but there may be some situations where the value of $\varphi \otimes P$ can differ from one decision event to the other. If the decision over the satisfaction of $\varphi \otimes P$ is final at the first decision event (i.e., is not reversed at the next decision event):

$$\forall t \geq @(\mathrm{dev}(\varphi, P) \uparrow)^1,\ \varphi \otimes P(t) = \varphi \otimes P\ (@(\mathrm{dev}(\varphi, P) \uparrow)^1) \tag{78}$$

A requirement R corresponding to a Boolean φ associated to a multiple time period $P = \{P_1, P_2, \ldots, P_{|P|}\}$ is denoted $R = \varphi \otimes P$ and formally defined as the following function:

$$\begin{gathered} \otimes :\ \mathbb{B} \times 2^{\mathcal{P}} \longrightarrow\ \mathbb{B} \\ (\varphi, P) \longmapsto\ \varphi \otimes P\ :=\ \bigwedge_{i=1}^{|P|} \varphi \otimes \{P_i\} = \varphi \otimes \{P_1\} \wedge \varphi \otimes \{P_2\} \wedge \ldots \wedge \varphi \otimes \{P_{|P|}\} \end{gathered} \tag{79}$$

$\varphi \in \mathbb{B}$ is called the *condition* of the requirement.

$P \in 2^{\mathcal{P}}$ is called the *time period* of the requirement (the term 'multiple' is eventually dropped as all time periods of requirement are considered to be multiple time periods, even if the multiple time period contains only one single time period, i.e. if $|P| = 1$).

$R(\varphi, P) = \varphi \otimes P \in \mathbb{B}$ is called the *value of the requirement.* It depends on time $t \in \mathfrak{C}$.

The multiple time period P associated with $R = \varphi \otimes P$ is denoted $\otimes R$. Similarly, the condition φ associated with $R = \varphi \otimes P$ is denoted $R \otimes$. Thus, $P = \otimes R$, $\varphi = R \otimes$ and $R = R \otimes \otimes R$.

The filter $a(\varphi, P)$ filters out the events of φ which are not decision events and ensures that the closing event of P is always a decision event if $\varphi \in \mathbb{B}_2$. The filter is composed of two terms, one that depends only on φ and the other that depends only on P:

$$\begin{gathered} a\!:\ \mathbb{B} \times \mathcal{P} \longrightarrow\ \mathbb{B} \\ a(\varphi, P) = a'(\varphi) \vee P \downarrow \end{gathered} \tag{80}$$

$\mathrm{dev}'(\varphi, P) \uparrow = \big(a'(\varphi, P) \times \varphi \vee \neg(a'(\varphi, P) \times \varphi)\big) \uparrow$ is called the *early decision event* because if it occurs, it occurs before $P \downarrow$.

There is no general algorithm for computing $a'(\varphi)$ from the condition φ, but a general procedure to write $a'(\varphi)$ is given in Section 3.19. Therefore, $a'(\varphi)$ must be given together with φ. Examples of $a'(\varphi)$ are $a'(\varphi) = \varphi$, $a'(\varphi) = \neg\varphi$, $a'(\varphi) = true$ or $a'(\varphi) = false$. If $a'(\varphi) = \varphi$, then the requirement is satisfied as soon as the condition φ is true. If $a'(\varphi) = \neg\varphi$, then the requirement is violated as soon as the condition φ is true. If $a'(\varphi) = true$, then all events generated by φ can be considered as decision events: as soon as φ becomes true or false, the decision is made. If $a'(\varphi) = false$, then no event generated by φ can be considered as decision event: the decision can only be made at the end of P.

If at the end of a single time period P, $a'(\varphi) \in \mathbb{B}_2$, then $a(\varphi, P) \in \mathbb{B}_2$.If in addition $\varphi \in \mathbb{B}_2$, then $\varphi \otimes P \in \mathbb{B}_2$:

$$a'(\varphi(@P \downarrow)) \in \mathbb{B}_2 \text{ and } \varphi(@P \downarrow) \in \mathbb{B}_2 \implies \varphi \otimes P \in \mathbb{B}_2 \tag{81}$$

Example 1 with $a'(\varphi) = \neg\varphi$. Let us consider a requirement $R = \varphi \otimes P$ that specifies that the number of events generated by a Boolean ψ inside P must be less than a given number N.

The number of events of ψ inside P is given by:

$$\begin{gathered} \mathrm{count}\!:\ \mathbb{B} \times \mathcal{P} \longrightarrow\ \mathbb{N} \\ \mathrm{count}\ (\psi, P)\ =\ \left|\{(\psi \uparrow)^i \text{ such that } P \uparrow \leq (\psi \uparrow)^i\ \leq \mathrm{P} \downarrow, i \in [1, |\psi \uparrow|]\}\right| \end{gathered} \tag{82}$$

The condition φ is given by:

$$\varphi = (\text{count}\,(\psi, P) \leq N) \tag{83}$$

where $N \in \mathbb{N}$ is a constant.

Then $a'(\varphi) = \neg\varphi$ because the decision that the requirement is not satisfied can be made as soon as $\text{count}(\psi, P) > N$. $\varphi \in \mathbb{B}_2 \implies a'(\varphi) \in \mathbb{B}_2$. Because Eq. (81) is satisfied, the value of $\varphi \otimes P$ is *true* or *false* (it cannot be *undecided* or *undefined*).

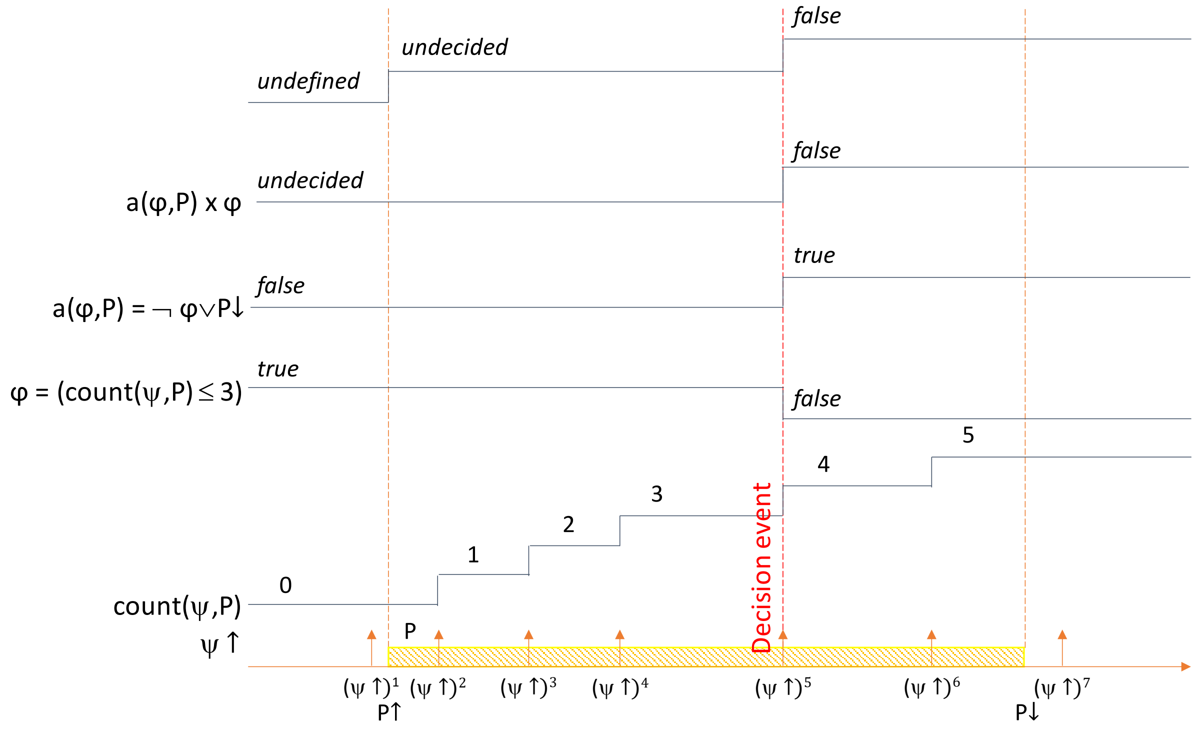

Fig. 39. Example 1 with the decision event inside the time period

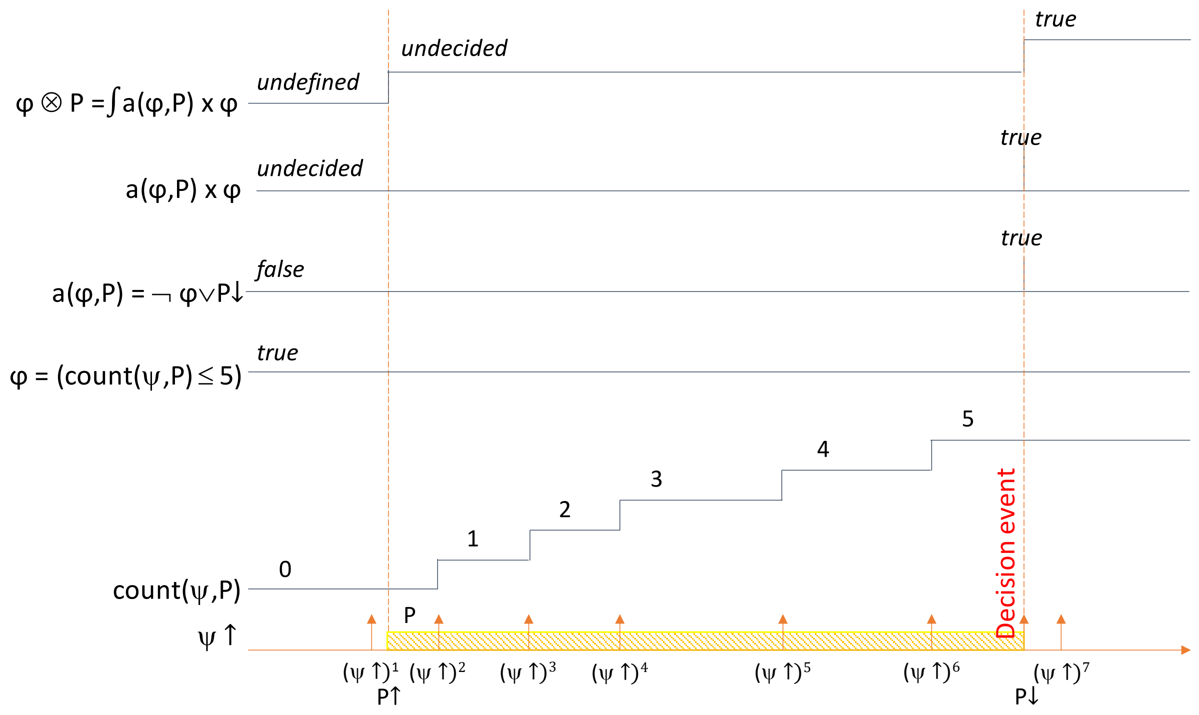

Fig. 40. Example 1 with the decision event at the end of the time period

The domain of requirements is denoted $\mathcal{R}$.

3.12.2. Composition of requirements with logical operators

The composition of a requirement $R = \varphi \otimes P$ with a unary logical operator op is given by:

$$\begin{aligned} op\colon \mathcal{R} &\longrightarrow \mathbb{B} \\ R &\longmapsto op\ R \end{aligned} \tag{84}$$

where op denotes any logical unary operator (¬ ...).

The composition of two requirements $R_1 = \varphi_1 \otimes P_1$ and $R_2 = \varphi_2 \otimes P_2$ with a binary logical operator op is given by:

$$\begin{aligned} op\colon \mathcal{R} \times \mathcal{R} &\longrightarrow \mathbb{B} \\ (R_1, R_2) &\longmapsto R_1\ op\ R_2 \end{aligned} \tag{85}$$

where op denotes any binary logical operator (∧, ∨, ⊕, ⟹, …).

$R = R_1\ op\ R_2$ does not satisfy the definition of requirements given by Eq. (79), because two time periods being involved (one for R_1 and the other for R_2) that cannot be combined into one, it does not belong to $\mathbb{B} \times 2^{\mathcal{P}}$ (it belongs to $\mathbb{B}$). R will be called a *composed requirement*, or somewhat imprecisely but more conveniently, a *requirement* when there is no ambiguity to do so.

The composition of multiple requirements follows the same rule:

$$\begin{aligned} \mathcal{R}^n &\longrightarrow \mathbb{B} \\ (R_1, R_2, \dots, R_n) &\longmapsto R_1\ op_1\ R_2 \dots op_{n-1}\ R_n \end{aligned} \tag{86}$$

where $op_1 \dots op_{n-1}$ denote any logical binary operators (∧, ∨, ⊕, ⟹, …).

Note that in general,

$$t \in [P\uparrow, P\downarrow[\ \nRightarrow\ (\varphi_1\ op\ \varphi_2) \otimes P = \varphi_1 \otimes P\ op\ \ \varphi_2 \otimes P \qquad (87)$$

although there are cases where $(\varphi_1\ op\ \ \varphi_2) \otimes P = \varphi_1 \otimes P\ \ op\ \ \varphi_2 \otimes P$ holds within $[P\uparrow, P\downarrow[$, depending on the logical operator op.

However,

$$t \notin [P\uparrow, P\downarrow[\ \Longrightarrow (\varphi_1\ op\ \varphi_2) \otimes P = \varphi_1 \otimes P\ op\ \ \varphi_2 \otimes P \qquad (88)$$

Example 1: $\forall t, (\text{count}\ (\psi, P) = N) \otimes P = \big((\text{count}\ (\psi, P) \leq N) \wedge (\text{count}\ (\psi, P) \geq N)\big) \otimes P$

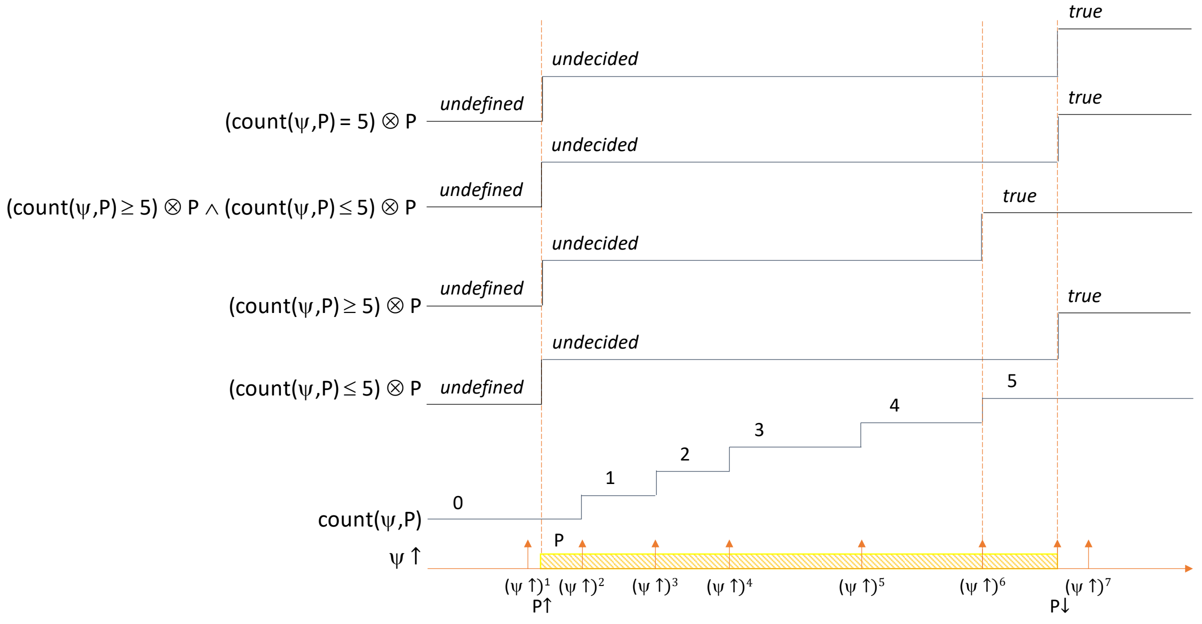

Fig. 41. Example 1 with *N* = 5

Example 2: $(\text{count}\ (\psi, P) \geq N) \otimes P \neq \big((\text{count}\ (\psi, P) >\ N) \vee (\text{count}\ (\psi, P) = N)\big) \otimes P$ within P

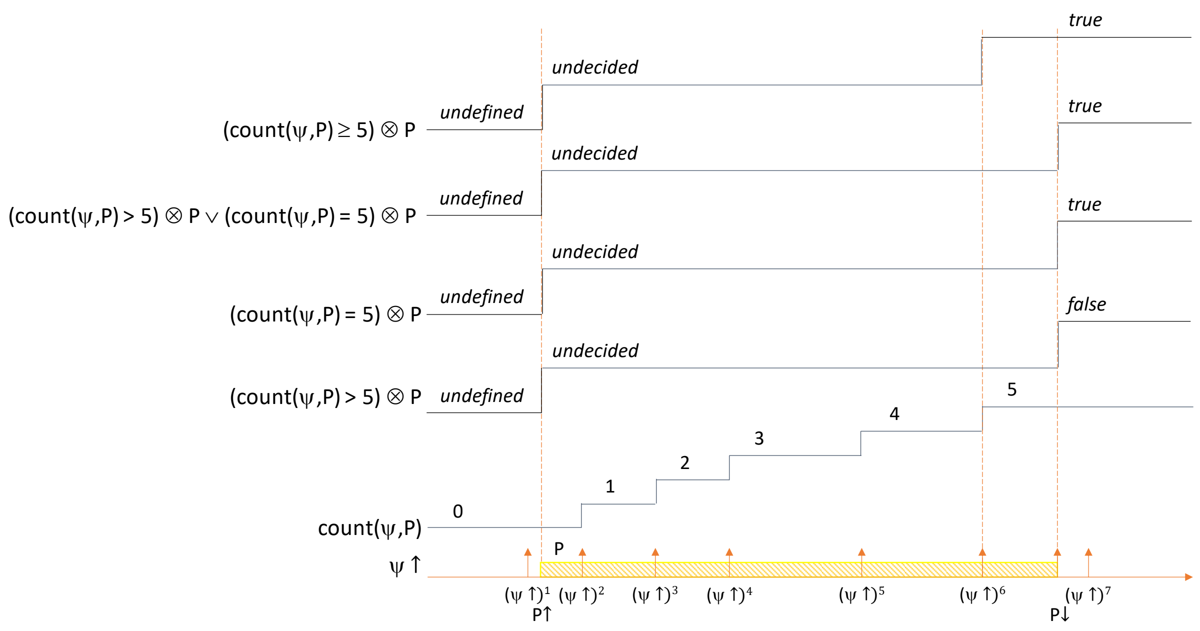

Fig. 42. Example 2 with *N* = 5

3.12.3. Satisfaction of requirements

A requirement $R = \varphi \otimes P$ is satisfied iff $\varphi \otimes P = true$ at the decision event $\mathrm{dev}(\varphi, P) \uparrow$. The satisfaction of $R = \varphi \otimes P$ is denoted $P \vDash \varphi$ which means that condition φ is satisfied over time period P:

$$\begin{gathered} \vDash: \ 2^{\mathcal{P}} \times \mathbb{B} \longrightarrow \ \mathbb{B}_2 \\ (P,\ \varphi) \longmapsto P \vDash \varphi := (\varphi \otimes P)(@\mathrm{dev}(\varphi, P) \uparrow) = true) \end{gathered} \qquad (89)$$

Because $\varphi = \ R \otimes$ and $P = \otimes R$, the satisfaction of R can be denoted $\otimes R \vDash R \otimes$.

From Eq. (89), $P \vDash \varphi$ is a 2-valued Boolean:

- If $R = \varphi \otimes P$ is satisfied, then $(\varphi \otimes P)(@\mathrm{dev}(\varphi, P) \uparrow) = true)$ and $P \vDash \varphi = true$.
- If $R = \varphi \otimes P$ is not satisfied, then $(\varphi \otimes P)(@\mathrm{dev}(\varphi, P) \uparrow) = false)$ and $P \vDash \varphi = false$.

Therefore

$$P \vDash \varphi = (\varphi \otimes P)(@\mathrm{dev}(\varphi, P) \uparrow)$$

The value of $(\varphi \otimes P)(t)$ is:

$$(\varphi \otimes P)(t) = \begin{cases} undefined \ \text{ if } t < @P_1 \uparrow \\ undecided \ \text{ if } @P_1 \uparrow \leq t < @\mathrm{dev}(\varphi, P) \uparrow \\ P \vDash \varphi \ \text{ if } t \geq @\mathrm{dev}(\varphi, P) \uparrow \end{cases} \qquad (90)$$

with $P = \{P_1,\ P_2, \dots, P_{|P|}\}$.

3.12.4. Applying frame time periods to requirements

Applying the frame time period M to requirement $R = \varphi \otimes P$ consists in applying M to P. It is formally defined as:

$$\begin{aligned} \supset :\ 2^{\mathcal{P}} \times \mathcal{R} &\longrightarrow\ \mathcal{R} \\ (M,\ R) &\longrightarrow M \supset R := \varphi \otimes (M \supset P) \end{aligned} \tag{91}$$

3.12.5. Simulating requirements

The continuous time domain $\mathfrak{C}$ spans from $-\infty$ to $+\infty$. However, the simulation time window spans from t_0 to t_f.

Therefore, the user must ensure that all events to be verified occur between t_0 and t_f, excluding t_0 and t_f because the behavior of the simulator is uncertain at instant time $t \leq t_0$ and instant time $t \geq t_f$. This can be done by encapsulating all events in a single frame time period, that defines the beginning and the end of the experiment.

According to Eq. (90), before the beginning of the frame time period, all requirements are evaluated to *undefined*: testing has not started. After the end of the frame time period, all requirements are evaluated in general to *true* or *false*, and sometimes to *undecided*. If some requirements are still undefined, that means that not all events are within the frame time period and testing is incomplete. If some requirements are still undecided, it means that not all events are within the frame time period and testing is incomplete, or more rarely that decisions could not be made due to uncertainties on thresholds.

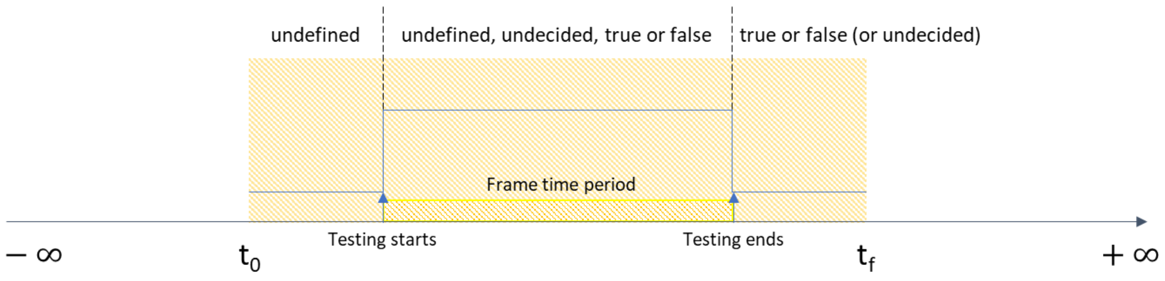
undefined
undefined, undecided, true or false
true or false (or undecided)
Frame time period
Testing starts
Testing ends
− ∞
t0
tf
+ ∞

Fig. 43. Frame time period to define the simulation period

3.13. Stochastic requirements

The satisfaction x of requirement $R = \varphi \otimes P$ is defined as $x = P \vDash \varphi$, cf. Eq. (89). x is a 2-valued Boolean that takes the value $true$ if R is satisfied, or $false$ otherwise.

A realistic requirement cannot be satisfied with absolute certainty. To that end, the satisfaction of a probabilistic requirement R is defined as a condition to be satisfied on the probability p that $x = true$, e.g. $p \geq 99.9\%$, or $p \geq f(t)$ where $f(t)$ is a function of time.

Then $x \in \mathbb{B}_2$ is replaced by a random variable X that associates probabilistic events to the outcomes $true$ or $false$:

$$X: \Omega \longrightarrow \mathbb{B}_2 \tag{92}$$

where Ω denotes the domain of probabilistic events. Ω is the randomized version of domain $2^{\mathcal{P}} \times \mathbb{B}$.

Examples of probabilistic events can be whether a tank level subject to random fluctuations exceeds a maximum level within a given time period, or whether a system goes out of its authorized operating domain according to measurements subject to random errors.

The probability that X takes the value $x \in \mathbb{B}_2$ at time t is denoted $\mathbb{P}(X = x|t)$. If time t is the instant of occurrence of an event $b \uparrow$, i.e. if $t = @b \uparrow$, then the probability that X takes the value $x \in \mathbb{B}_2$ when event $b \uparrow$ occurs is denoted $\mathbb{P}(X = x|b \uparrow)$ (it can also be denoted $\mathbb{P}(X = x|@b \uparrow)$). The Boolean b that triggers the event $b \uparrow$ can itself be the outcome of a random variable B.

The probability that requirement $R = \varphi \otimes P$ is satisfied is then $\mathbb{P}(X = true|\mathrm{dev}(\varphi, P) \uparrow)$, the outcomes of X being $x = P \vDash \varphi$.

We are now interested in computing $p = \mathbb{P}(X = true|t)$. This equivalent to writing $p = \mathbb{E}([\![X = true]\!]|t)$ where $[\![\cdot]\!]$ is the indicator function:

$$[\![x]\!] = \begin{cases} 1 \text{ if } x = true \\ 0 \text{ if } x = false \end{cases}$$

X is a Bernoulli distributed random variable:

$$\mathbb{P}(X = x|t) = \begin{cases} p \ \text{ if } \ [\![x]\!] = 1 \\ 1 - p \ \text{ if } \ [\![x]\!] = 0 \end{cases}$$

or equivalently

$$\mathbb{P}(X = x|t) = p^{[\![x]\!]} \cdot (1 - p)^{1-[\![x]\!]}, x \in \{true, false\}$$

An estimate $\hat{p}$ of p is given by an estimate $\hat{\mu}_X$ of $\mu_X = \mathbb{E}([\![X = true]\!]|t)$. An estimate of the error on $\hat{\mu}_X$ is given by an estimate $\hat{\sigma}_X^2$ of $\sigma_X^2 = \mathbb{V}\mathrm{ar}[[\![X = true]\!]|t] = \mathbb{E}[([\![X = true]\!]|t)^2] - \mathbb{E}[[\![X = true]\!]|t]^2$.

For a normally distributed variable $X \sim \mathcal{N}(\mu_X, \sigma_X^2)$, there is 95% probability that the true value $p = \mu_X$ lies in the uncertainty range $[\hat{\mu}_X - \lambda_{95\%} \cdot \sigma_X, \hat{\mu}_X + \lambda_{95\%} \cdot \sigma_X]$, with $\lambda_{95\%} = 1.96$ (but we only have an estimate $\hat{\sigma}_X$ of σ_X, which implies some uncertainty on the uncertainty range which is not computed).

The estimator $\hat{\mu}_X$ is given by

$$\hat{\mu}_X = \frac{1}{n} \cdot \sum_{i=1}^{n} y_i \tag{93}$$

where $y_i = [\![x_i]\!]$, x_i being the outcome of X in the i^{th} Monte Carlo simulation of R, and where n is the number of simulations.

The estimator $\hat{\sigma}_X^2$ is given by:

$$\hat{\sigma}_X^2 = \frac{1}{n-1} \cdot \sum_{i=1}^{n} (y_i - \hat{\mu}_X)^2 \tag{94}$$

Because Monte Carlo simulations generate random deviates from the same distribution, $\mathbb{E}[y_i] = \mu_X$ and $\mathbb{V}\mathrm{ar}[y_i] = \sigma_X^2$ for all y_i.

Then, Eq. (93) is an unbiased estimator because

$$\mathbb{E}[\hat{\mu}_X] = \frac{1}{n} \cdot \sum_{i=1}^{n} \mathbb{E}[y_i] = \frac{1}{n} \cdot \sum_{i=1}^{n} \mu_X = \mu_X$$

Because the generated deviates are independent, $\mathbb{C}\mathrm{ov}[y_i, y_j] = \mathbb{E}[y_i \cdot y_j] - \mathbb{E}[y_i] \cdot \mathbb{E}[y_j] = 0$ for $i \neq j$, which implies that $\mathbb{E}[y_i \cdot y_j] = \mu_X^2$ for all y_i and y_j such that $i \neq j$.

Then, Eq. (94) is an unbiased estimator because

$$\begin{aligned}
\mathbb{E}[\hat{\sigma}_X^2] &= \frac{1}{n-1} \cdot \mathbb{E}\left[\sum_{i=1}^{n} \left(y_i - \frac{1}{n} \cdot \sum_{i=1}^{n} y_i\right)^2\right] \\
&= \frac{1}{n-1} \cdot \mathbb{E}\left[\sum_{i=1}^{n} \left(y_i^2 - \frac{2}{n} \cdot y_i \cdot \sum_{j=1}^{n} y_j + \frac{1}{n^2} \cdot \left(\sum_{i=1}^{n} y_i\right)^2\right)\right] \\
&= \frac{1}{n-1} \cdot \mathbb{E}\left[\sum_{i=1}^{n} y_i^2 - \frac{2}{n} \cdot \sum_{j=1}^{n} y_j \cdot \sum_{i=1}^{n} y_i + n \cdot \frac{1}{n^2} \cdot \left(\sum_{i=1}^{n} y_i\right)^2\right] \\
&= \frac{1}{n-1} \cdot \mathbb{E}\left[\sum_{i=1}^{n} y_i^2 - \frac{2}{n} \cdot \sum_{j=1}^{n} y_j \cdot \sum_{k=1}^{n} y_k + \frac{1}{n} \cdot \sum_{j=1}^{n} y_j \cdot \sum_{k=1}^{n} y_k\right] \\
&= \frac{1}{n-1} \cdot \mathbb{E}\left[\sum_{i=1}^{n} y_i^2 - \frac{1}{n} \cdot \sum_{j=1}^{n} y_j \cdot \sum_{k=1}^{n} y_k\right]
\end{aligned}$$

$$= \frac{1}{n-1} \cdot \mathbb{E}\left[\left(1-\frac{1}{n}\right) \cdot \sum_{i=1}^{n} y_i^2 - \frac{1}{n} \cdot \sum_{j \neq k} y_j \cdot y_k\right]$$
$$= \frac{1}{n-1} \cdot \left(\left(1-\frac{1}{n}\right) \cdot \sum_{i=1}^{n} \mathbb{E}[y_i^2] - \frac{1}{n} \cdot \sum_{j \neq k} \mathbb{E}[y_j \cdot y_k]\right)$$
$$= \frac{1}{n-1} \cdot \left(\left(1-\frac{1}{n}\right) \cdot n \cdot (\sigma_X^2 + \mu_X^2) - \frac{1}{n} \cdot (n^2 - n) \cdot \mu_X^2\right)$$
$$= \sigma_X^2$$

$\hat{\mu}_X$ and $\hat{\sigma}_X^2$ are not defined for $n \leq 1$.

We are now interested in assessing the number of n of simulations to compute the estimator $\hat{\mu}_X$.

According to the central limit theorem applied to Eq. (93),

$$\lim_{n \to \infty} \hat{\mu}_X = \lim_{n \to \infty} \frac{1}{n} \cdot \sum_{i=1}^{n} y_i \sim \mathcal{N}\left(\mu_X, \frac{\sigma_X}{\sqrt{n}}\right)$$

where $\sigma_X^2 = \mu_X \cdot (1 - \mu_X)$ is the variance of the Bernoulli distribution.

Because $\hat{\mu}_X \sim \mathcal{N}\left(\mu_X, \frac{\sigma_X}{\sqrt{n}}\right)$ for n sufficiently large,

$$\mathbb{P}\left(|\hat{\mu}_X - \mu_X| < \lambda_{95\%} \cdot \frac{\sigma_X}{\sqrt{n}}\right) = 95\%$$

Then

$$\mathbb{P}\left(|\hat{\mu}_X - \mu_X| < \lambda_{95\%} \cdot \sqrt{\frac{\mu_X \cdot (1 - \mu_X)}{n}}\right) = 95\%$$

which yields the 95% probability confidence range for $\hat{\mu}_X$ for n sufficiently large.

We are now interested in assessing a condition for the value of n. When dealing with the satisfaction of requirements, the expected probability μ_X is close to 1, or equivalently, $1 - \mu_X$ is small (close to zero). The precision of the computation of the confidence range should increase when $1 - \mu_X$ decreases. For small values of $1 - \mu_X$, the maximum relative precision error ε of the estimator $\hat{\mu}_X$ is defined such that $|\hat{\mu}_X - \mu_X|/(1 - \mu_X) < \varepsilon$.

Then, the condition giving the confidence range for the estimator is satisfied with precision ε if

$$\lambda_{95\%} \cdot \frac{1}{1 - \mu_X} \cdot \sqrt{\frac{\mu_X \cdot (1 - \mu_X)}{n}} < \varepsilon$$

Consequently, n is given by the condition

$$n > \frac{\lambda_{95\%}^2}{\varepsilon^2} \cdot \frac{\mu_X}{1 - \mu_X}$$

For instance, computing an estimator for an almost sure event of expected probability $\mu_X \sim 1 - 10^{-3}$ requires $n \sim 3.84 \cdot 10^5$ simulations to be in the 95% confidence range with relative precision error $\varepsilon = 10\%$.

3.14. Domains

3.14.1. Definition

A domain is a finite or infinite set of elements of the same kind.

List of domains:

- $\mathfrak{T}$: type definitions
- $\mathfrak{C}$: continuous clock
- $\mathbb{N}$: natural numbers (positive integers including zero)
- $\mathbb{Z}$: positive or negative integers
- $\mathbb{R}$: real numbers
- $\mathbb{C}$: complex numbers
- $\mathbb{B}_2$: 2-valued Booleans { *true*, *false* }
- $\mathbb{B}_3$: 3-valued Booleans { *true*, *false*, *undecided* }
- $\mathbb{B}_3'$: 3-valued Booleans { *true*, *false*, *undefined* }
- $\mathbb{B}$, $\mathbb{B}_4$: 4-valued Booleans { *true*, *false*, *undecided*, *undefined* }
- $\mathbb{S}$: character strings
- $\mathcal{E}$: events
- $2^{\mathcal{E}}$, $\mathcal{D}$: discrete clocks
- $\mathcal{P}$: single time periods
- $2^{\mathcal{P}}$: multiple time periods
- $\mathcal{R}$: requirements. $\mathcal{R} = \mathbb{B} \times 2^{\mathcal{P}} \longrightarrow \mathbb{B}$
- $\mathbb{O}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2)$: operators from $\mathbb{D}_1$ to $\mathbb{D}_2$
- $\mathcal{C}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2)$: categories from $\mathbb{D}_1$ to $\mathbb{D}_2$
- $\mathcal{S}$: sets
- $\mathcal{C}$: class definitions
- $\mathcal{C}_C$: class C
- $2^{\mathcal{C}_C}$: sets of objects of class C
- $\mathcal{M}$: models
- $\mathcal{L}$: librairies
- $\mathcal{T}$: packages

Any domain is denoted $\mathbb{D}$.

To emphasize that $x \in \mathbb{D}$, x can be denoted $\mathbb{D}\colon x$.

The domain of x is denoted $\mathrm{domain}(x)$. Thus $\mathrm{domain}(\mathbb{D}\colon x) = \mathbb{D}$.

3.14.2. Domain extensions

Existing domains can be extended to create new domains. For instance, $\mathbb{R}$ can be extended to define the domain of physical quantities.

A domain $\mathbb{D}'$ that extends domain $\mathbb{D}$ is denoted $\mathbb{D}' \subset \mathbb{D}$ because if x is an element of $\mathbb{D}'$, then it is an element of $\mathbb{D}$.

$$x \in \mathbb{D}' \text{ and } \mathbb{D}' \subset \mathbb{D} \Longrightarrow x \in \mathbb{D} \quad (95)$$

A domain $\mathbb{D}''$ that extends a domain $\mathbb{D}' \subset \mathbb{D}$ also extends $\mathbb{D}$:

$$\mathbb{D}'' \subset \mathbb{D}' \text{ and } \mathbb{D}' \subset \mathbb{D} \Longrightarrow \mathbb{D}'' \subset \mathbb{D} \quad (96)$$

If $\mathbb{D}'$ extends $\mathbb{D}$, then all operators on $\mathbb{D}$ can be applied on $\mathbb{D}'$, unless specified otherwise (e.g., it is not possible to add or multiply temperatures which are quantities that extend real numbers).

An operator op defined on $\mathbb{D}$ and forbidden on $\mathbb{D}'$ is denoted $op \notin \mathbb{O}(\mathbb{D}' \longrightarrow *)$.

Then:

$$op \in \mathbb{O}(\mathbb{D} \longrightarrow *) \text{ and } \mathbb{D}' \subset \mathbb{D} \Longrightarrow op \in \mathbb{O}(\mathbb{D}' \longrightarrow *) \quad (97)$$

unless it is specified that $op \notin \mathbb{O}(\mathbb{D}' \longrightarrow *)$.

The definition $\widehat{\mathbb{D}}'$ of the extension $\mathbb{D}'$ of a domain $\mathbb{D}$ is given as follows:

$$\widehat{\mathbb{D}}' = \mathbb{D}: z(a_1, \dots a_i, \dots a_n) + \{\mathbb{D}_1: a_1, \dots \mathbb{D}_i: a_i, \dots \mathbb{D}_n: a_n\} \quad (98)$$

where the a_i denote the attributes of $\widehat{\mathbb{D}}'$. z is a variable of domain $\mathbb{D}$ that represents the definition of the variables of domain $\mathbb{D}'$ as a function of the attributes a_i. If no such function is necessary for the definition (i.e., if z has the same definition in $\mathbb{D}$ and $\mathbb{D}'$), then it can be omitted. Therefore $\widehat{\mathbb{D}}' = \mathbb{D}: z \Longleftrightarrow \widehat{\mathbb{D}}' = \mathbb{D}$. In the following, when there is no confusion between $\widehat{\mathbb{D}}'$, the definition of the domain, and $\mathbb{D}'$, the domain itself, the hat above the name of the domain will be omitted.

Example 1: defining the domain $\mathbb{Z}/n\mathbb{Z}$ of residual integers r modulo n.

$$\mathbb{Z}_n = \mathbb{Z}: r = p \bmod n + \{\mathbb{Z}: p, \mathbb{Z}: n\}$$

In this example, r is a dummy variable that represents any element of $\mathbb{Z}_n$. r is an integer that is computed from p and n: r is equal to p modulo n.

For instance, for $n = 7$ and $p = 22$, $r = 1$.

$\mathbb{D}''$ can in tun extend $\mathbb{D}'$:

$$\mathbb{D}'' = \mathbb{D}': z\left(a_1, \dots a_i, \dots a_n, a'_1, \dots a'_i, \dots a'_{n'}\right) + \left\{\mathbb{D}_1: a'_1, \dots \mathbb{D}_i: a'_i, \dots \mathbb{D}_{n'}: a'_{n'}\right\} \quad (99)$$

where the a'_i denote the additional attributes of $\mathbb{D}''$. z represents the variable of $\mathbb{D}'$ that is a function of the attributes a_i and a'_i.

However, a domain $\mathbb{D}$ cannot extend two different domains $\mathbb{D}_1$ and $\mathbb{D}_2$:

$$\mathbb{D} \subset \mathbb{D}_1 \text{ and } \mathbb{D} \subset \mathbb{D}_2 \Longrightarrow \mathbb{D}_1 = \mathbb{D}_2 \quad (100)$$

A variable $x \in \mathbb{D}'$ is obtained by assigning values to the attributes a_i, hence yielding a value for z:

$$\mathbb{D}': x = z(a_i = v_i, \ i \in [1, n]) \quad (101)$$

Example 2: finding the residual of 22 modulo 7

$$\mathbb{Z}_n: r(p = 22, n = 7)$$

3.14.3. Domain specializations

A domain $\mathcal{D}$ can specialize another domain $\mathbb{D}'$ that extends a domain $\mathbb{D}$ by assigning fixed values to some attributes of $\mathbb{D}'$:

$$\mathcal{D} = \mathbb{D}'(a_i = v_i,\ \ i \in F \subset [1,n]) \tag{102}$$

where F denotes the indices of the fixed attributes.

Example 1: defining the domain of integer residuals modulo 7

$$\mathbb{Z}_7 = \mathbb{Z}_n(n = 7)$$

$\mathcal{D}$ specializes $\mathbb{D}'$ is denoted $\mathcal{D} \prec \mathbb{D}'$. Then a variable $x \in \mathcal{D}$ is obtained by assigning values to the remaining non-fixed attributes $a_i \in \bar{F} = [1,n] - F$:

$$\mathcal{D}: x(a_i = v_i,\ \ i \in \bar{F} = [1,n] - F) \tag{103}$$

Example 2: finding the residual of 22 modulo 7

$$\mathbb{Z}_7: r(p = 22)$$

A domain $\mathcal{D}'$ can in turn specialize $\mathcal{D}$ by assigning fixed values to some non-fixed attributes of $\mathcal{D}$.

$$\mathcal{D}' = \mathcal{D}(a_i = v_i,\ \ i \in F' \subset \bar{F}) \tag{104}$$

where F' denotes the indices of the fixed attributes.

Then a variable $x \in \mathcal{D}'$ is obtained by assigning values to the remaining non-fixed attributes $a_i \in \bar{F'} = \bar{F} - F'$:

$$\mathcal{D}': x(a_i = v_i,\ \ i \in \bar{F'} = \bar{F} - F') \tag{105}$$

If there is only one free attribute left, i.e. if $|\bar{F'}| = 1$, then it is also possible to write without ambiguity

$$\mathcal{D}': x = v \tag{106}$$

to assign a value to x.

Example 3: finding the residual of 22 modulo 7 can be written

$$\mathbb{Z}_7: r = 22$$

because p is the only attribute in $\mathbb{Z}_7$.

If there are no free attribute left, i.e. if $|\bar{F'}| = 0$, then $\mathcal{D}'$ contains a single element.

Example 4: $\mathbb{Z}_{7:21} = \mathbb{Z}_7(p = 21)$ has no attribute. It is the domain that contains the single integer $\{1\}$.

A domain $\mathcal{D}''$ that specializes domain $\mathcal{D}' \prec \mathbb{D}'$ extends $\mathbb{D}'$:

$$\mathcal{D}'' \prec \mathcal{D}' \text{and}\, \mathcal{D}' \prec \mathbb{D}' \Longrightarrow \mathcal{D}'' \subset \mathbb{D}' \tag{107}$$

If a domain $\mathcal{D}(s)$ with an alias s specializes a domain $\mathbb{D}'$, then any variable $x \in \mathcal{D}(s)$ can be denoted $\mathbb{D}': x\ s$. The alias s must be unique for domain $\mathbb{D}'$ which means that two domains $\mathcal{D}_1(s_1)$ and $\mathcal{D}_2(s_2)$ that extend the same domain $\mathbb{D}'$ must have different aliases s_1 and s_2.

$$\mathcal{D}_1(s_1)\ \subset \mathbb{D}' \text{ and } \mathcal{D}_2(s_2)\ \subset \mathbb{D}' \Longrightarrow s_1 \neq s_2 \tag{108}$$

This feature is used to express physical units outside of the domain name.

3.14.4. Partial domains

A partial domain $\mathbb{D}$ is a domain that cannot have any elements because it is incompletely defined. It is then expected that its definition will be completed by extending or specializing $\mathbb{D}$. A partial domain $\mathbb{D}$ is denoted $\text{partial}\ \mathbb{D}$. Then:

$$\text{partial}\ \mathbb{D}\ \Longrightarrow \mathbb{D} = \emptyset \tag{109}$$

3.14.5. Domain aliases

An alias $s \in \mathbb{S}$ can be associated to a domain $\mathbb{D}$. This is denoted $\mathbb{D}(s)$. Then $x \in \mathbb{D}(s)$ can be denoted $x\ s$. In other words, $x\ s$ has the same meaning as $\mathbb{D}(s): x$.

If a domain $\mathbb{D}'(s)$ with an alias s extends or specializes a domain $\mathbb{D}$, then any variable $x \in \mathbb{D}'(s)$ can be denoted $\mathbb{D}: x\ s$. The alias s must be unique for any domain extension or specialization $\mathbb{D}'$ of domain $\mathbb{D}$ which means that two domains $\mathbb{D}'_1(s_1)$ and $\mathbb{D}'_2(s_2)$ that extend or specialize the same domain $\mathbb{D}$ must have different aliases s_1 and s_2.

$$(\mathbb{D}'_1(s_1)\ \subset \mathbb{D} \text{ or } \mathbb{D}'_1(s_1)\ \prec \mathbb{D}\ ) \text{ and } (\mathbb{D}'_2(s_2)\ \subset \mathbb{D} \text{ or } \mathbb{D}'_2(s_2)\ \prec \mathbb{D}) \Longrightarrow s_1 \neq s_2 \tag{110}$$

This feature is used to express physical units outside of the domain name.

Example 1: defining the domain Q of physical quantities in SI units by extending $\mathbb{R}$.

$$\text{partial}\ Q = (\mathbb{R}: q = r \times\ u + o) + \{\mathbb{S}: SIUnit,\ \mathbb{S}: userUnit,\ \mathbb{R}: u,\ \mathbb{R}: r,\ \mathbb{R}: o\} \tag{111}$$

where $SIUnit$ is the SI unit for quantity q, $userUnit$ is the user unit for quantity q, q is the quantity expressed in SI units, u is the quantity expressed in user units, r is the rate and o is the offset to convert the user unit into the SI unit. Partial indicates that this domain must be extended or specialized.

Example 2: defining physical units by specializing the domain of physical quantities Q.

The domain of pressures is obtained by specializing domain Q:

$$\text{partial Pressure} = Q(SIUnit = \text{"Pa"}) \tag{112}$$

The domain of pressures with user units in bars is obtained by specializing domain Pressure:

$$\text{PressureBar(bar)} = \text{Pressure}(userUnit = \text{"bar"},\ r = 10^5,\ o = 0) \tag{113}$$

Then a pressure of 3 bars can be expressed as $\text{PressureBar}: P(u = 3)$, $\text{PressureBar}: P = 3$ or, more conveniently, as $\text{Pressure}: P = 3\ \text{bar}$ (or even $\mathbb{R}: P = 3\ \text{bar}$). Note that the word bar is written without quotes because it cannot be confounded with the name of a variable.

The domain of pressures with user units in Pa is obtained by another specialization of domain Pressure:

$$\text{PressurePa}(\text{Pa}) = \text{Pressure}(userUnit = \text{"Pa"},\ r = 1,\ o = 0) \tag{114}$$

It then possible to write $\text{Pressure}: P = 3\ \text{bar} + 10000\ \text{Pa}$. The conversion of $3\ \text{bar}$ to SI units is made automatically using $3\ \text{bar} = r \times\ u + o = 10^5 \times 3 + 0 = 3.10^5\ \text{Pa}$. Then the value of P is $P = 3\ 10^5 + 10000 = 3.1\ 10^5\ \text{Pa}$.

If one omits the unit when writing $\text{Pressure}: P = 3$, then an error will be raised at variable definition time because Pressure is a partial domain that cannot have any elements.

If one writes a wrong unit such as $\text{Pressure}: P = 3\ \text{s}$, or an unknown unit such as $\text{Pressure}: P = 3\ \text{sz}$, then an error will be raised because neither s nor sz are aliases for Pressure.

The domain of absolute temperatures is obtained by specializing domain Q:

$$\text{partial AbsoluteTemperature} = Q(SIUnit = \text{"K"}) \tag{115}$$

To express the fact that absolute temperatures cannot be added, one can state that the + operator cannot be using on the $\text{AsoluteTemperature}$ domain: $+ \notin \mathbb{O}(\text{AsoluteTemperature}\ \longrightarrow *)$. Then a special domain should be created to express temperature differences that can be added to absolute temperatures to produce another absolute temperature.

The domain of temperatures with user units in Celsius is obtained by specializing domain Temperature:

$$\text{TemperatureCelsius}(\text{Celsius}) = \text{AbsoluteTemperature}(userUnit = \text{"Celsius"},\ r = 1,\ o = 273.15) \tag{116}$$

Then a temperature of 20 Celsius can be expressed as $\text{TemperatureCelsius}: t(u = 20)$, $\text{TemperatureCelsius}: t = 20$, $\text{AbsoluteTemperature}: t = 20\ \text{Celsius}$, $\mathbb{R}: t = 20\ \text{Celsius}$.

The domain of time is obtained by specializing domain Q:

$$\text{partial Time} = Q(SIUnit = \text{"s"}) \tag{117}$$

The domain of time with user units in seconds is obtained by specializing domain Time:

$$\text{TimeSecond}(\text{s}) = Q(userUnit = \text{"s"},\ r = 1,\ o = 0) \tag{118}$$

Then a duration of 2 seconds can be expressed as $\text{TimeSecond}: d(u = 2)$, $\text{TimeSecond}: d = 2$, $\text{Time}: d = 2\ s$, $\mathbb{R}: d = 2\ s$.

The domain of time with user units in hours is obtained by specializing domain Time:

$$\text{TimeHour}(\text{h}) = \text{Time}(userUnit = \text{"h"},\ r = 3600,\ o = 0) \tag{119}$$

Then a duration of 2 hours can be expressed as $\text{TimeHour}: d(u = 2)$, $\text{TimeHour}: d = 2$, $\text{Time}: d = 2\ \text{h}$, $\mathbb{R}: d = 2\ \times 3600\ \text{s}$, etc.

3.15. Types

A type $\mathbb{T}$ is defined as a domain $\mathbb{D}$, a domain extension $\mathbb{D}' \subset \mathbb{D}$, or a domain specialization $\mathcal{D} \prec \mathbb{D}'$.

As domain extensions and domain specializations can be considered as domains, the notion of type is equivalent to the notion of domain. Therefore, the words 'type' and 'domain' are synonymous.

If $x \in \mathbb{D}$ then x is of type $\mathbb{D}$. This can be denoted $\mathbb{D}: x$.

If x is of type $\mathbb{D}'$ and type $\mathbb{D}'$ extends type $\mathbb{D}$, then x is also of type $\mathbb{D}$:

$$x \in \mathbb{D}' \ \mathit{and} \ \mathbb{D}' \subset \mathbb{D} \Longrightarrow x \in \mathbb{D} \tag{120}$$

If x is of type $\mathcal{D}$ and type $\mathcal{D}$ specializes $\mathbb{D}'$, then x is also of type $\mathbb{D}'$:

$$x \in \mathcal{D} \ \text{and} \ \mathcal{D} \prec \mathbb{D}' \Longrightarrow x \in \mathbb{D}' \tag{121}$$

The domain of types (the domains of domains) is denoted $\mathfrak{T}$.

3.16. Variables

A variable is an element that belongs to the following domains, or extension of the following domains:

- $\mathfrak{C}$: continuous clock
- $\mathbb{N}$: natural numbers (positive integers)
- $\mathbb{Z}$: positive or negative integers
- $\mathbb{R}$: real numbers
- $\mathbb{C}$: complex numbers
- $\mathbb{B}_2$: 2-valued Booleans { *true*, *false* }
- $\mathbb{B}_3$: 3-valued Booleans { *true*, *false*, *undecided* }
- $\mathbb{B}'_3$: 3-valued Booleans { *true*, *false*, *undefined* }
- $\mathbb{B}$, $\mathbb{B}_4$: 4-valued Booleans { *true*, *false*, *undecided*, *undefined* }
- $\mathbb{S}$: character strings
- $\mathcal{E}$: events
- $2^{\mathcal{E}}$, $\mathcal{D}$: discrete clocks
- $\mathcal{P}$: single time periods
- $2^{\mathcal{P}}$: multiple time periods
- $\mathcal{R}$: requirements. $\mathcal{R} = \mathbb{O}(\mathbb{B} \times 2^{\mathcal{P}} \longrightarrow \ \mathbb{B})$

Any domain of variables is denoted $\mathbb{V}$.

Any variable $x \in \mathbb{V}$ is a function of time $t \in \mathfrak{C}$: $\mathbb{V}: x = x(\mathfrak{C}: t)$.

A variable that keeps the same value at all time instants t is called a fixed variable or a constant variable: $x = constant$.

When time is implicit in an equation, it is always implied that the values of all variables x are taken at the same instant in time $t \in \mathfrak{C}$.

Example 1:

$$x_1 \vee x_2 := \neg(\neg x_1 \wedge \neg x_2)$$

means

$$x_1(t) \vee x_2(t) := \neg\big(\neg x_1(t) \wedge \neg x_2(t)\big),\ t \in \mathfrak{C}$$

If the values of some variables are taken at different instants in time in an equation, then time is explicit in the equation, but not necessarily in the time definition domain of the equation. It is then implied that time belongs to the continuous clock $\mathfrak{C}$.

Example 2:

$$x(t_2) - x(t_1) := t_2 - t_1$$

means

$$x(t_2) - x(t_1) := t_2 - t_1,\ t_1 \in \mathfrak{C}, t_2 \in \mathfrak{C}$$

The fact that time t belongs to a discrete clock Ω is denoted by $t \in \Omega$. This means that the projection operator $/\Omega$ defined in Eq. (43) must be applied to all time instants.

Example 3:

$$x_1 \vee x_2 := \neg(\neg x_1 \wedge \neg x_2), t \in \Omega$$

means

$$x_1(@\,t/\Omega) \vee x_2(@\,t/\Omega) := \neg\big(\neg x_1(@\,t/\Omega) \wedge \neg x_2(@t/\Omega)\big),\ t \in \mathfrak{C}$$

Example 4:

$$x(t_2) - x(t_1) := t_2 - t_1, t_1 \in \Omega, t_2 \in \Omega$$

means

$$x(@\,t_2/\Omega) - x(@t_2/\Omega) := @t_2/\Omega - @t_1/\Omega,\ t_1 \in \mathfrak{C},\ t_2 \in \mathfrak{C}$$

3.17. Operators

An operator f is a function between a source domain $\mathbb{D}_1 \times \ldots \times \mathbb{D}_n$ that can be composed of several domains if $n \geq 2$ and a target domain $\mathbb{D}$:

$$\begin{aligned} f\colon \mathbb{D}_1 \times \ldots \times \mathbb{D}_n &\longrightarrow \mathbb{D} \\ (x_1, \ldots, x_n) \longmapsto y &:= f(x_1, \ldots, x_n) \end{aligned} \tag{122}$$

A unary operator op is a function from $\mathbb{D}_1$ to $\mathbb{D}$ (the source and the target can be different domains):

$$\begin{aligned} op\colon \mathbb{D}_1 &\longrightarrow \mathbb{D} \\ x \longmapsto y &:= op\, x \end{aligned} \tag{123}$$

A binary operator op on domain $\mathbb{D}$ is a function from $\mathbb{D} \times \mathbb{D}$ to $\mathbb{D}$ (the sources and the target are the same domain):

$$\begin{aligned} op\colon \mathbb{D} \times \mathbb{D} &\longrightarrow \mathbb{D} \\ (x_1, x_2) \longmapsto y &:= x_1\, op\, x_2 \end{aligned} \tag{124}$$

The domain of operators from $\mathbb{D}_1$ to $\mathbb{D}_2$ is denoted $\mathbb{O}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2)$.

The domain of operators from $\mathbb{D}_1$ to any domain is denoted $\mathbb{O}(\mathbb{D}_1 \longrightarrow *)$.

The domain of operators from any domain to $\mathbb{D}_2$ is denoted $\mathbb{O}(* \longrightarrow \mathbb{D}_2)$.

The domain of operators from any domain to any domain is denoted $\mathbb{O}(* \longrightarrow *)$.

3.18. Templates

Templates are operators taking Booleans as arguments and Booleans as values:

$$template\colon\ \mathbb{B}^n \ \longrightarrow\ \mathbb{B} \tag{125}$$

Example 1: mission change. If requirement R_1 fails, then switch to R_2: requirement R_2 should be satisfied at most one hour after R_1 fails.

$$\begin{gathered}\text{switch: } \mathbb{B}^2 \ \longrightarrow \mathbb{B} \\ \text{switch}(R_1, R_2) \ :=\ R_2 \otimes [R_1 \downarrow, R_1 \downarrow +1h]\end{gathered} \tag{126}$$

$$\begin{gathered}\text{missionChange: } \mathbb{B}^2 \ \longrightarrow \mathbb{B} \\ \text{missionChange}(R_1, R_2) \ :=\ \neg R_1 \Longrightarrow \text{switch}(R_1, R_2)\end{gathered} \tag{127}$$

Example 2: if mission change fails, then switch to requirement R_3: requirement R_3 should be satisfied at most one hour after the switching to R_2 fails

$$\neg\text{missionChange}(R_1, R_2) \Longrightarrow \text{missionChange}(\neg\text{switch}(R_1, R_2), R_3) \tag{128}$$

Example 3: elicitation rule. if assumption A is verified then R_1 should be satisfied, else R_2 should be satisfied.

$$\begin{gathered}\text{rule: } \mathcal{R} \times \mathcal{R} \times \mathcal{R} \ \longrightarrow \mathcal{R} \\ \text{rule}(A, R_1, R_2) \ :=\ (A \Longrightarrow R_1) \wedge (\neg A \Longrightarrow R_2)\end{gathered} \tag{129}$$

Example 4: definition of the disjunction of requirements from the conjunction and negation of requirements.

$$\begin{gathered}\vee : \mathcal{R} \times \mathcal{R} \ \longrightarrow \mathbb{B} \\ R_1 \vee R_2 \ :=\ \neg(\neg R_1 \wedge \neg R_2)\end{gathered} \tag{130}$$

3.19. Categories

Categories are operators on operators. They are introduced to handle the computation of the filter $a'(\varphi, P)$ in Eq. (80). They are formally defined as follows:

$$\begin{gathered}c\colon \mathbb{O}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2) \longrightarrow \mathbb{O}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2) \\ f \ \longmapsto g = c(f)\end{gathered} \tag{131}$$

where f and g are operators

$$f\colon \mathbb{D}_1 \longrightarrow \mathbb{D}_2$$

$$g\colon \mathbb{D}_1 \longrightarrow \mathbb{D}_2$$

The domain of categories c from $\mathbb{D}_1$ to $\mathbb{D}_2$ is denoted $\mathcal{C}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2)$. $\mathcal{C}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2)$ can be formally defined as $\mathcal{C}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2) \coloneqq \mathbb{O}\big(\mathbb{O}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2) \longrightarrow \mathbb{O}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2)\big)$

Let us consider two operators f and g:

$$f\colon \mathbb{D}_1 \longrightarrow \mathbb{D}_2 \\ g\colon \mathbb{D}_2 \longrightarrow \mathbb{D}_3 \tag{132}$$

and the category

$$c\colon \mathbb{O}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2) \longrightarrow \mathbb{O}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2) \\ f \longmapsto c(f) \tag{133}$$

Let us consider the operator

$$h\colon \mathbb{D}_1 \longrightarrow \mathbb{D}_3 \\ h = g \circ c(f) \tag{134}$$

and the category

$$c'\colon \mathbb{O}(\mathbb{D}_1 \longrightarrow \mathbb{D}_3) \longrightarrow \mathbb{O}(\mathbb{D}_1 \longrightarrow \mathbb{D}_3) \\ h \longmapsto c'(h) = \begin{cases} h \text{ if } c \in \mathcal{C}(h) \\ g \circ f \text{ if } c \notin C(h) \end{cases} \tag{135}$$

where $\circ$ denotes the composition of operators $g \circ f(x) = g\big(f(x)\big)$, and $\mathcal{C}(h)$ denotes the set of categories associated to operator h. c' is called the adjoint of c and is denoted $c' = \operatorname{adj} c$.

Then, if a category c is defined, the following procedure is applied:

1. The adjoint category $\operatorname{adj} c$ is applied to all operators h, whether $c \in \mathcal{C}(h)$ or $c \notin C(h)$.
2. If h is of the form $h = g \circ c(f)$, which means that category c is used in the expression of h, then
   a. If $c \in \mathcal{C}(h)$, then $(\operatorname{adj} c)(h) = h$: the effect of c on f is left unchanged and $\operatorname{adj} c$ has no effect on h.
   b. If $c \notin C(h)$, then $(\operatorname{adj} c)(h) = g \circ f$: the effect of c on f is cancelled and $\operatorname{adj} c$ has an effect on h.
3. If h is not of the form $h = g \circ c(f)$, which means that category c is not used in the expression of h, then $(\operatorname{adj} c)(h) = h$. $\operatorname{adj} c$ has no effect on h.

Then $a'(\varphi)$ in Eq. (80) is obtained by:

1. Writing the condition φ as:

$$\varphi = \lambda \circ c(\psi) \tag{136}$$

   where λ and ψ are conditions, and c is a category that depends on some properties of the condition φ (e.g., whether φ is monotonously increasing or decreasing in time). This allows to encode in φ the properties of φ that are relevant to compute $a'(\varphi)$.
2. Not associating the category c to φ:

$$c \notin C(\varphi) \tag{137}$$

   Then the introduction of c in φ does not modify the value of φ.
3. Writing $a'(\varphi)$ as:

$$a'(\varphi) = \varphi \tag{138}$$

   Then $a'(\varphi)$ is the condition φ.
4. Associating the category c to $a'(\varphi)$:

$$c \in \mathcal{C}\big(a'(\varphi)\big) \tag{139}$$

Then $a'(\varphi)$ is the condition φ with the right properties to raise the proper decision events.

5. Applying the above procedure.

A category c can be associated to several operators f_i. This is individually denoted $c \in \mathcal{C}(f_i)$ for each f_i, or collectively denoted $c \in \cap\{\mathcal{C}(f_i)\}_{1\leq i\leq n}$ for all f_i.

Several categories c_j can be associated to the same operator f. This is individually denoted $c_j \in \mathcal{C}(f)$ for each c_j, or collectively denoted $\cup\{c_j\}_{1\leq i\leq p} \subset \mathcal{C}(f)$ for all c_j.

Several categories c_j can be associated to several operators f_i. This is individually denoted $c_j \in \mathcal{C}(f_i)$ for each couple (c_j, f_i), or collectively denoted $\cup\{c_j\}_{1\leq i\leq p} \subset (\cap\{\mathcal{C}(f_i)\}_{1\leq i\leq n})$ for all (c_j, f_i).

Example 1: Generalization of Example 1 in Section 3.12.1. $\mathbb{O}(\mathbb{N}^2 \longrightarrow \mathbb{B}) = \{ >,\ \geq,\ =,\ \neq,\ <,\ \leq \}$

$$\begin{array}{c} \text{cat1}: \mathbb{O}(\mathbb{N}^2 \longrightarrow \mathbb{B}) \longrightarrow \mathbb{O}(\mathbb{N}^2 \longrightarrow \mathbb{B}) \\ > \ \longmapsto > \\ \geq \ \longmapsto \geq \\ = \longmapsto > \\ \neq \longmapsto > \\ < \longmapsto \geq \\ \leq \longmapsto > \end{array} \tag{140}$$

Let us consider requirement $R = \varphi \otimes P$ with $\varphi(\psi, P) = \text{count}\,(\psi, P)\ op\ N$, ψ being a Boolean, N being a fixed integer and op being one of the operators listed in Eq. (140).

Then φ is rewritten as

$$\varphi(\psi, P) = \ \text{count}\,(\psi, P)\ \text{cat1}(op)\ N$$

Because cat1 is not associated with φ, we have still $\varphi\ =\ \text{count}\,(\psi, P)\ op\ N$, but because cat1 is associated with $a'(\varphi)$, we have $a'(\varphi) = \text{count}\,(\psi, P)\ \text{cat1}(op)\ N$ where the properties of φ for the satisfaction of requirement R is embedded in $\text{cat1}(op)$. Then for $op = \leq$, the decision event is raised when $a'(\varphi) = \text{count}\,(\psi, P)\ >\ N$, therefore as soon as condition $\varphi =\ \text{count}\,(\psi, P)\ \leq N$ is violated. For $op = \neq$, the decision event is raised when $a'(\varphi) = \text{count}\,(\psi, P)\ >\ N$, therefore as soon as condition $\varphi =\ \text{count}\,(\psi, P)\ \neq N$ is satisfied again ($\varphi =\ \text{count}\,(\psi, P) > N$), after being violated ($\varphi = (\text{count}\,(\psi, P) = N)$), and after being initially satisfied ($\varphi =\ \text{count}\,(\psi, P) < N$), as the initial value for $\text{count}\,(\psi, P)$ is always zero. This exemplifies the difference between the satisfaction of condition φ and the satisfaction of requirement R: the satisfaction of φ is different from the satisfaction of R because $\text{cat1}(op) \neq op$.

Example 2: $\mathbb{O}(\mathbb{B} \longrightarrow \mathbb{B}) = \{\,\text{id},\ \text{false}\,\}$ where id is the identity function

$$\begin{array}{c} \text{id}:\ \mathbb{B}\ \longrightarrow \mathbb{B} \\ x \longmapsto\ x \end{array} \tag{141}$$

and false is the constant function

$$\begin{array}{c} \text{false}:\ \mathbb{B}\ \longrightarrow \mathbb{B} \\ x \longmapsto\ false \end{array} \tag{142}$$

$$\begin{array}{c} \text{cat2}:\ \mathbb{O}(\mathbb{B} \longrightarrow \mathbb{B})\ \longrightarrow \mathbb{O}(\mathbb{B} \longrightarrow \mathbb{B}) \\ \text{id} \longmapsto false \end{array} \tag{143}$$

Let us consider requirement $R = \varphi \otimes P$ with $\varphi(\psi) = \psi$, ψ being a Boolean.

Then φ is written as

$$\varphi(\psi) = \text{ cat2 } (\psi)$$

Because cat2 is not associated with φ, $\varphi = \psi$, and because cat2 is associated with $a'(\varphi)$, $a'(\varphi) = \text{cat2 } (\psi) = false$. Thus, no decision event is triggered by ψ (or φ), and the only decision event is the closing of the single time period P: the decision whether R is satisfied is made at the end of P, and R is always undecided within P before the end of P.

Example 3: $\mathbb{O}(\mathbb{B} \longrightarrow \mathbb{B}) = \{\text{ true, false, undecided, undefined }\}$.

$$\begin{gathered}\text{cat3: } \mathbb{O}(\mathbb{B} \longrightarrow \mathbb{B}) \longrightarrow \mathbb{O}(\mathbb{B} \longrightarrow \mathbb{B}) \\ true \longmapsto true \\ false \longmapsto true \\ undecided \longmapsto false \\ undefined \longmapsto false\end{gathered} \tag{144}$$

Let us consider requirement $R = \varphi \otimes P$ with $\varphi(\psi) = \psi$, ψ being a Boolean.

Then φ is written as

$$\varphi(\psi) = \text{ cat3 } (\psi)$$

Because cat3 is not associated with φ, $\varphi = \psi$, and because cat3 is associated with $a'(\varphi)$, $a'(\varphi) = \text{cat3 } (\psi)$. Thus, if $\varphi \in \mathbb{B}_2$ any value of ψ (or φ) generates a decision event, which means that decision events occur as soon as ψ (or φ) becomes *true* or *false*. Therefore R is always satisfied or violated within or after P and never undecided.

3.20. Sets

3.20.1. Definition

A set S is a finite collection of elements e_i of domains $\mathbb{D}_i$:

$$S = \{\, \mathbb{D}_1 : e_1, \dots \mathbb{D}_i e_i, \dots \mathbb{D}_{|S|} : e_{|S|} \,\} \tag{145}$$

where $|S|$ is the number of elements in S. The elements of the set can be separated using the comma (,) or the semicolon (;).

There are no duplicates in a set: $\forall i, j \in [1, |S|], i \neq j \Longrightarrow e_i \neq e_j$.

Two sets $S_1 = \{\, \mathbb{D}_{1,1} : e_{1,1}, \dots \mathbb{D}_{1,i} e_{1,i}, \dots \mathbb{D}_{1,|S_1|} : e_{1,|S_1|} \,\}$ and $S_2 = \{\, \mathbb{D}_{2,1} : e_{2,1}, \dots \mathbb{D}_{2,i} e_{2,i}, \dots \mathbb{D}_{2,|S_2|} : e_{2,|S_2|} \,\}$ are equal if they have the same elements, even if their elements are listed in different orders.

$$S_1 = S_2 \Longleftrightarrow |S_1| = |S_2| \text{ and } \exists \text{ permutation } \sigma \text{ such that } \forall i \in [1, |S_1|]\ e_{1,\sigma(i)} = e_{2,i} \tag{146}$$

The number of elements $|S|$ of a set S can be fixed or can vary in time. Typically, the number of elements in sets of events or time periods vary in time.

The usual set operators, union (∪), intersection (∩), difference (−), subset of (⊂), superset of (⊃), etc. can be applied to sets.

The domain of sets is denoted $\mathcal{S}$.

The set of subsets of $S = \{\, \mathbb{D}_1 : e_1, \dots \mathbb{D}_i : e_i, \dots \mathbb{D}_{|S|} : e_{|S|} \,\} \in \mathcal{S}$ is denoted 2^S. Therefore $S \in 2^S$ and $2^S \subset 2^{\bigcup_{i=1}^{|S|} \mathbb{D}_i}$.

The flattening $\Cup S$ of a set S that contains subsets $\{S_i\}_i$ that in turn contain subsets $\{S_{i,j}\}_j$, etc. is defined as the set that contains all elements in set S, subsets $\{S_i\}_i$, subsets $\{S_{i,j}\}_j$ brought up to the level of set S.

Example 1: flattening of $S = \{e_1, e_2, \{e_3, e_4, \{e_5, e_6\}\}, \{e_7, e_8\}, e_9\,\}$

$$\Cup S = \{e_1, e_2, e_3, e_4, e_5, e_6 e_7, e_8, e_9\}$$

3.20.2. Set namespace

Each element e_i of a set $S = \{\, \mathbb{D}_1 : e_1, \dots \mathbb{D}_i : e_i, \dots \mathbb{D}_{|S|} : e_{|S|} \,\}$ is given a name $a_i \in \mathbb{S}$ and possibly a value $v_i \in \mathbb{D}_i$: $e_i \equiv (a_i = v_i)$. Then

$$S = \{\, \mathbb{D}_1 : a_1 = v_1, \dots \mathbb{D}_i : a_i = v_i, \dots \mathbb{D}_{|S|} : a_{|S|} = v_{|S|} \,\}.$$

The names a_i given to elements e_i are unique within the namespace of S: $\forall i, j \in [1, |S|], i \neq j \implies a_i \neq a_j$. Two different sets S and S' have different namespaces.

The value v_i of element e_i in set S with name a_i is denoted $S.a_i$: $v_i = S.a_i$.

Let us consider two sets $S_1 = \{\mathbb{D}_{1,i} : a_{1,i} = v_{1,i}\}_{1 \le i \le |S_1|}$ and $S_2 = \{\mathbb{D}_{2,j} : a_{2,j} = v_{2,j}\}_{1 \le j \le |2|}$.

Let set $\mathcal{S}_1 = \{S_{1,i}\}_{1 \le i \le n_1}$ be the increasing ordered set of sets that contain S_1: $S_1 \in S_{1,1} \in \dots \in S_{1,n_1}$. Therefore, S_1 is in the namespace of $S_{1,1}$, and each $S_{1,i}$ is in the namespace of $S_{1,i+1}$.

Let set $\mathcal{S}_2 = \{S_{2,j}\}_{1 \le j \le n_2}$ be the increasing ordered set of sets that contain S_2: $S_2 \in S_{2,1} \in \dots \in S_{2,n_2}$. Therefore, S_2 is in the namespace of $S_{2,1}$, and each $S_{2,\mathrm{j}}$ is in the namespace of $S_{2,j+1}$.

Let S be the smallest set such that $S \in \mathcal{S}_1 \cap \mathcal{S}_2$.

If S exists, then $\exists\, m_2 \in [1, n_2]\ \mathrm{such\ that}\ S = S_{2,m_2}$. Then the value $v_{1,i}$ of an element $e_{1,i}$ of set S_1 can be set to the value of an element $e_{2,j}$ of set S_2 by writing $v_{1,i} = S_{2,m_2}.S_{2,m_{2}-1} \dots S_{2,1}.S_2.a_{2,j}$. $S_{2,m_2}.S_{2,m_{2}-1} \dots S_{2,1}$ is called the relative path of element $e_{2,j}$ within the namespace of S.

If S does not exist, then one assumes that there exists a universal set $\mathcal{U}$ that contains all sets except itself. Then the value $v_{1,i}$ of an element $e_{1,i}$ of set S_1 can be set to the value of an element $e_{2,j}$ of set S_2 by writing $v_{1,i} = \mathcal{U}.S_{2,n_2}.S_{2,n_{2}-1} \dots S_{2,1}.S_2.a_{2,j}$. $\mathcal{U}.S_{2,n_2}.S_{2,n_{2}-1} \dots S_{2,1}$ is called the absolute path of element $e_{2,j}$.

3.21. Operators on sets

3.21.1. Unary operators

Unary operators can be iteratively applied to all elements of a set S in the following way.

Let us consider the left-hand unary operator $op1$:

$$\begin{aligned} op1 \colon\ \mathbb{D}_1 &\longrightarrow \mathbb{D} \\ x &\longmapsto op1\ x \end{aligned} \qquad (147)$$

Then, the operator $op1$ can be applied to set $S = \{e_1, \dots e_i, \dots e_{|S|}\}$ such that $|S| \geq 1$ as follows:

$$\begin{gathered} op1\colon\ 2^{\mathbb{D}_1} \longrightarrow 2^{\mathbb{D}} \\ S \longmapsto op1\ S := \{op1\ e_1, \dots\ op1\ e_i, \dots\ op1\ e_{|S|}\} \end{gathered} \tag{148}$$

Let us consider the right-hand unary operator $op1$:

$$\begin{gathered} op1\colon\ \mathbb{D}_1 \longrightarrow \mathbb{D} \\ x \longmapsto x\ op1 \end{gathered} \tag{149}$$

Then, the operator $op1$ can be applied to set $S = \{e_1, \dots e_i, \dots e_{|S|}\}$ such that $|S| \geq 1$ as follows:

$$\begin{gathered} op1\colon\ 2^{\mathbb{D}_1} \longrightarrow 2^{\mathbb{D}} \\ S \longmapsto S\ op1 := \{e_1\ op1, \dots e_i\ op1, \dots e_{|S|}\ op1\} \end{gathered} \tag{150}$$

Let us now consider the operator $op2$ with two arguments:

$$\begin{gathered} op2\colon\ \mathbb{D}_1 \times \mathbb{D}_2 \longrightarrow \mathbb{D} \\ (x_1, x_2) \longmapsto x_1\ op2\ x_2 \end{gathered} \tag{151}$$

Let us consider the left-hand unary operator $op1 = x_1\ op2$:

$$\begin{gathered} x_1\ op2\colon\ \mathbb{D}_2 \longrightarrow \mathbb{D} \\ x_2 \longmapsto (x_1\ op2)\ x_2 = x_1\ op2\ x_2 \end{gathered} \tag{152}$$

Then $op1 = x_1\ op2$ can be applied to set $S = \{e_1, \dots e_i, \dots e_{|S|}\}$ following the rule given by Eq. (148):

$$\begin{gathered} x_1\ op2\colon\ 2^{\mathbb{D}_2} \longrightarrow 2^{\mathbb{D}} \\ S \longmapsto (x_1\ op2) S = x_1\ op2\ S\ = \{x_1\ op2\ e_1, \dots x_1\ op2\ e_i, \dots x_1\ op2\ e_{|S|}\} \end{gathered} \tag{153}$$

Let us consider the right-hand unary operator $op1 = op2\ x_2$:

$$\begin{gathered} op2\ x_2\colon\ \mathbb{D}_1 \longrightarrow \mathbb{D} \\ x_1 \longmapsto x_1\ (op2\ x_2) = x_1\ op2\ x_2 \end{gathered} \tag{154}$$

Then $op1 = op2\ x_2$ can be applied to set $S = \{e_1, \dots e_i, \dots e_{|S|}\}$ following the rule given by Eq. (150):

$$\begin{gathered} op2\ x_2\colon\ 2^{\mathbb{D}_1} \longrightarrow 2^{\mathbb{D}} \\ S \longmapsto\ S\ (op2\ x_2) = S\ op2\ x_2\ = \{e_1\ op2\ x_2, \dots e_i\ op2\ x_2, \dots e_{|S|}\ op2\ x_2\} \end{gathered} \tag{155}$$

It follows that unary operators $op1\colon\ \mathbb{D}_1 \longrightarrow \mathbb{D}$ are extended in the following manner:

$$\begin{gathered} op1 \in \mathbb{O}(\mathbb{D}_1 \longrightarrow \mathbb{D}) \Longrightarrow op1 \in \mathbb{O}(2^{\mathbb{D}_1} \longrightarrow 2^{\mathbb{D}}) \\ \Longrightarrow op1 \in \mathbb{O}(\mathbb{D}_1 \cup 2^{\mathbb{D}_1} \longrightarrow \mathbb{D} \cup 2^{\mathbb{D}}) \\ \Longrightarrow op1 \in \mathbb{O}\left(2^{\mathbb{D}_1 \cup 2^{\mathbb{D}_1}} \longrightarrow 2^{\mathbb{D} \cup 2^{\mathbb{D}}}\right) \\ \Longrightarrow op1 \in \mathbb{O}\left(\mathbb{D}_1 \cup 2^{\mathbb{D}_1} \cup 2^{\mathbb{D}_1 \cup 2^{\mathbb{D}_1}} \longrightarrow \mathbb{D} \cup 2^{\mathbb{D}} \cup 2^{\mathbb{D} \cup 2^{\mathbb{D}}}\right) \\ \dots \end{gathered} \tag{156}$$

Note that $\{x_i\}_i\ op2\ \{y\}_j$ is ambiguous as it could mean either $(\{x_i\}_i\ op2)\ \{y\}_j$ or $\{x_i\}_i\ \left(op2\ \{y\}_j\right)$, and that the two expressions are not equal:

$$\left.\begin{array}{l}(\{x_i\}_i\ op2)\ \{y\}_j = \left\{\{x_i\}_i\ op2\ y_j\right\}_j = \left\{\left\{x_i\ op2\ y_j\right\}_i\right\}_j \\ \{x_i\}_i\ \left(op2\ \{y\}_j\right) = \left\{x_i\ op2\ \{y\}_j\right\}_i = \left\{\left\{x_i\ op2\ \ y_j\right\}_j\right\}_i\end{array}\right\} \Longrightarrow (\{x_i\}_i\ op2)\ \{y\}_j \neq \{x_i\}_i\ \left(op2\ \{y\}_j\right)$$

Example 1: negation of a set of requirements $R = \left\{R_1, \dots R_i, \dots R_{|R|}\right\} \in 2^{\mathcal{R}}$:

$$\neg R = \left\{\neg R_1, \dots \neg R_i, \dots \neg R_{|R|}\right\} \tag{157}$$

Example 2: frame period $M \in 2^{\mathcal{P}}$ applied to a set of requirements $R = \left\{R_1, \dots R_i, \dots R_{|R|}\right\} \in 2^{\mathcal{R}}$:

$$M \supset R = \left\{M \supset R_1, \dots M \supset R_i, \dots M \supset R_{|R|}\right\} \tag{158}$$

Example 3: same condition $\varphi \in \mathbb{B}$ applied to a set of multiple time periods $P = \left\{P_1, \dots P_i, \dots P_{|P|}\right\} \in 2^{2^{\mathcal{P}}}$:

$$\varphi \otimes P = \left\{\varphi \otimes P_1, \dots \varphi \otimes P_i, \dots \varphi \otimes P_{|P|}\right\} \tag{159}$$

Example 4: same multiple time period $P \in 2^{\mathcal{P}}$ applied to a set of conditions $\Phi = \left\{\varphi_1, \dots \varphi_i, \dots \varphi_{|\Phi|}\right\} \in 2^{\mathbb{B}}$:

$$\Phi \otimes P = \left\{\varphi_1 \otimes P, \dots \varphi_i \otimes P, \dots \varphi_{|\Phi|} \otimes P\right\} \tag{160}$$

Example 5: getting the values of same attribute a_i for a set of objects $S = \left\{O_1, \dots O_i, \dots O_{|S|}\right\} \in 2^{\mathcal{C}_O}$ with $\mathcal{C}_O = \{\mathbb{D}_1 : a_1, \dots \mathbb{D}_i : a_i, \dots \mathbb{D}_n : a_n\}$:

$$\mathrm{S}.a_i = \left\{O_1.a_i, \dots O_i.a_i, \dots O_{|S|}.a_i\right\} \tag{161}$$

Example 6: negation of a set containing requirements and sets of requirements:

$$R = \{R_1, R_2, \{R_3, R_4\}, \{R_5, R_6\}\}$$

$$\neg R = \{\neg R_1, \neg R_2, \neg\{R_3, R_4\}, \neg\{R_5, \neg R_6\}\} = \{\neg R_1, \neg R_2, \{\neg R_3, \neg R_4\}, \{\neg R_5, \neg R_6\}\} \tag{162}$$

3.21.2. Binary operators

Binary operators can be iteratively applied to all elements of a set S in the following way.

Let us consider the binary operator $op2$:

$$\begin{array}{c} op2 : \mathbb{D} \times \mathbb{D} \longrightarrow \mathbb{D} \\ (x_1, x_2) \longmapsto x_1\ op2\ x_2 \end{array} \tag{163}$$

Then, the operator $op2$ can be applied to set $S = \left\{e_1, \dots e_i, \dots e_{|S|}\right\}$ such that $|S| \geq 2$ as follows:

$$\begin{array}{c} op2 :\ 2^{\mathbb{D}} \longrightarrow \mathbb{D} \\ S \longmapsto op2\ S := e_1\ op2\ e_2 \dots op2\ e_i\ \dots op2\ e_{|S|} \end{array} \tag{164}$$

From Eq. (163) and (164), it follows that binary operators $op2 : \mathbb{D} \times \mathbb{D} \longrightarrow \mathbb{D}$ are extended in the following manner:

$$\begin{array}{l} op2 \in \mathbb{O}(\mathbb{D} \times \mathbb{D} \longrightarrow \mathbb{D}) \Longrightarrow op2 \in \mathbb{O}(2^{\mathbb{D}} \longrightarrow \mathbb{D}) \\ \qquad\qquad\qquad\qquad\qquad \Longrightarrow op2 \in \mathbb{O}((\mathbb{D} \times \mathbb{D}) \cup 2^{\mathbb{D}} \longrightarrow \mathbb{D}) \end{array} \tag{165}$$

Because $\mathbb{D} \times \mathbb{D}$ and $2^{\mathbb{D}}$ involve different number of arguments (or because $\mathbb{D} \times 2^{\mathbb{D}} \not\subset (\mathbb{D} \times \mathbb{D}) \cup 2^{\mathbb{D}}$) it is not possible to mix elements and sets of elements with binary operators. Then $x\ op2\ \{y\}_j$ means necessarily $(x\ op2)\ \{y\}_j$ (indeed, it cannot mean $x\left(op2\ \{y\}_j\right)$ because $x\left(op2\ \{y\}_j\right) = x\ (y_1\ op2\ y_2 \ldots op2\ y_n)$ is meaningless). However, $x\ op2\ op2\ \{y\}_j$ makes sense because $x\ op2\ op2\ \{y\}_j = (x\ op2)\left(op2\ \{y\}_j\right) = (x\ op2)\ (y_1\ op2\ y_2 \ldots op2\ y_n) = x\ op2\ y_1\ op2\ y_2 \ldots op2\ y_n$.

Example 1: conjunction of a set of requirements $R = \{R_1, \ldots R_i, \ldots R_{|R|}\} \in 2^{\mathcal{R}}$:

$$\wedge R = R_1 \wedge R_2 \ldots \wedge R_i\ \ldots \wedge R_{|R|} \tag{166}$$

Example 2: inference chain of a set of requirements $R = \{R_1, \ldots R_i, \ldots R_{|R|}\} \in 2^{\mathcal{R}}$:

$$\Rightarrow R = R_1 \Rightarrow R_2 \ldots \Rightarrow R_i\ \ldots \Rightarrow R_{|R|} \tag{167}$$

3.21.3.Filter operator

Let us consider a set S of elements e_i of the same type $\mathbb{D}$:

$$S = \{\ \mathbb{D}: e_1, \ldots \mathbb{D}: e_i, \ldots \mathbb{D}: e_{|S|}\ \} \tag{168}$$

A filter on S is defined as the following function:

$$\begin{gathered}\text{filter: } 2^{\mathbb{D}} \times \mathbb{O}(\mathbb{D} \longrightarrow \mathbb{B}) \ \longrightarrow\ 2^{\mathbb{D}} \\ (S, \text{cond}) \longmapsto S(\text{cond}(*)) = \{\ e_i \in S \text{ such that } \text{cond}(e_i) = true\ \}\end{gathered} \tag{169}$$

where $cond$ is a Boolean condition on each element e_i and $*$ denotes a dummy variable that represents any element e_i of S:

$$\begin{gathered}\text{cond: } \mathbb{D} \ \longrightarrow\ \mathbb{B} \\ e \longmapsto \text{cond}(e)\end{gathered} \tag{170}$$

Example 1: all pumps that are started and close to their nominal regime.

Let us consider the class Pump defined by Eq. (189) and the following set of pumps:

$$Pumps = \{\ \text{Pump}: P_1, \text{Pump}: P_2, \text{Pump}: P_3, \text{Pump}: P_4\ \}$$

Then the set of pumps that are started and close to their nominal regime is:

$$PumpsInOperation = Pumps(*.s = true \wedge *.\eta > 0.90) \tag{171}$$

Example 2: all even ticks of a clock Ω (cf. also Eq. (54)).

$$EvenTicks =\ \Omega(\text{rank}(*) = 0 \bmod 2) \tag{172}$$

Example 3: all ticks of a clock Ω that occur after time *t*:

$$FutureTicks =\ \Omega(@ * \geq t) \tag{173}$$

Example 4: all requirements of a set of requirements R that are satisfied.

$$SatisfiedRequirements =\ R(\otimes * \vDash * \otimes) \tag{174}$$

Example 5: requirement satisfaction for all elements in a set of requirements R.

$$AllRequirementsSatisfied = \wedge\, R(\otimes * \vDash * \otimes) \tag{175}$$

Example 6: requirement satisfaction for at least one element in a set of requirements R.

$$AtLeastOneRequirementSatisfied = \vee\, R(\otimes * \vDash * \otimes) \tag{176}$$

3.22. Objects

An object O is a set of elements e_i such that the values v_i of all elements e_i can be calculated. The element e_i is also called an *attribute* of O. Each element $e_i \in O$ must have a:

- Declaration that states the name $a_i \in \mathbb{S}$ and the domain $\mathbb{D}_i$ of e_i. To declare an element e_i in object O, the following notations are possible: $\mathbb{D}_i: a_i \in O$ or $O.\mathbb{D}_i: a_i$. a_i must be unique within the name space of object O.

$$\forall i, j \in [1, |O|], i \neq j \Longrightarrow a_i \neq a_j \tag{177}$$

- Definition that provides a value v_i to e_i. To define an element e_i, one writes $a_i = v_i$, where v_i denotes the CRML expression that provides a value to e_i.

There is an exception to the above rule that concerns *external elements*. An external element is an element that is declared in object O and defined in another object $O' \neq O$ of any kind that is not necessarily expressed in CRML: requirement model, behavioral model, etc. The declaration of an external element e_i is denoted $\mathbb{D}_i: a_i \in O'$ or $O'.\mathbb{D}_i: a_i$ if object O' is known. Otherwise, it is denoted $\mathbb{D}_i: a_i \in *$ or $*.\mathbb{D}_i: a_i$ where $*$ indicates another object to be defined later via a *binding* mechanism.

An object O can contain the following elements e_i:

- Variables $x_i \in \mathbb{V}$ of any of the domains listed in Section 3.16.
- Other objects O_i.
- Sets S_i containing elements e_{ij} (recursive definition).
- At most one frame period F that truncates all time periods defined in the object.

According to the above definitions, and object O can be denoted as a set of valued attributes a_i:

$$O = \{\mathbb{D}_1: a_1 = v_1, \dots \mathbb{D}_i: a_i = v_i, \dots \mathbb{D}_{|O|}: a_{|O|} = v_{|O|}\} \tag{178}$$

where each attribute a_i denotes a variable $x_i \in \mathbb{V}$, an object $O_i \neq O$, a set S_i of variables or objects, or at most one frame period F. $\mathbb{D}_i$ is the domain of a_i, and each attribute a_i is being assigned a value v_i, which can be a fixed value, of a value obtained by an operator f_i which depends on the values $v_{j \neq i}$ and on the values v'_{kl} of attributes of external objects O'_k. The assignments of the values v_i must be done in explicit form (i.e., so that they do not generate implicit equations).

The value v_i of attribute a_i is denoted $O.a_i$: $v_i = O.a_i$.

Assigning a value v_i to attribute a_i of object O is denoted $O.a_i = v_i$ or $O(a_i = v_i)$.

Two objects $O_1 = \{\mathbb{D}_{1,1}: a_{1,1} = v_{1,1}, \dots \mathbb{D}_{1,i}: a_{1,i} = v_{1,i}, \dots \mathbb{D}_{1,|O_1|}: a_{1,|O_1|} = v_{1,|O_1|}\}$ and $O_2 = \{\mathbb{D}_{2,1}: a_{2,1} = v_{2,1}, \dots \mathbb{D}_{2,i}: a_{2,i} = v_{2,i}, \dots \mathbb{D}_{2,|O_2|}: a_{2,|O_2|} = v_{2,|O_2|}\}$ belong to the same class $\mathcal{C}_O$ if they have the same attributes with the same types.

$$\{\mathbb{D}_{1,1}: a_{1,1}, \dots \mathbb{D}_{1,i}: a_{1,i}, \dots \mathbb{D}_{1,|O_1|}: a_{1,|O_1|}\} = \{\mathbb{D}_{2,1}: a_{2,1}, \dots \mathbb{D}_{2,i}: a_{2,i}, \dots \mathbb{D}_{2,|O_2|}: a_{2,|O_2|}\} \Leftrightarrow O_1 \in \mathcal{C}_O \text{ and } O_2 \in \mathcal{C}_O \quad (179)$$

The two objects are not considered to be equal because at the same instant t in time, their respective attributes can have different values.

Therefore:

- The domain of object O is the class $\mathcal{C}_O$.Therefore, O belongs to $\mathcal{C}_O$: $O \in \mathcal{C}_O$. The objects $O \in \mathcal{C}_O$ are called the instances of $\mathcal{C}_O$. $O \in \mathcal{C}_O$ is also denoted $\mathcal{C}_O: O$.
- Two objects O_1 and O_2 of the same class $\mathcal{C}_O$ only differ in the values of their attributes: for all i, the value of $O_1.a_i$ can be different from the value of $O_2.a_i$.
- A class $\mathcal{C}_O$ is the domain of all objects O having the same attributes.
- $\mathcal{C}_O$ is a set: $\mathcal{C}_O \in \mathcal{S}$.

The domain of sets S of objects O_i that belong to the same class $\mathcal{C}_O$ is denoted $2^{\mathcal{C}_O}$.Therefore $\mathcal{C}_O \in 2^{\mathcal{C}_O}$.

Example 1: object Pump.

$\mathcal{C}_{\text{Pump}}: Pump = \{$

$*.\mathbb{R}: c, *.\mathbb{R}: \eta, *.\mathbb{B}_2: s, \mathbb{B}_2: *.cav;$

$\mathcal{R}: R_{noCav} = \neg cav \otimes [s \uparrow, \neg s \uparrow] \}$

where c is the pump characteristic, η is the pump efficiency, s is the state started or not of the pump and cav indicates whether the pump is cavitating or not. The requirement R_{noCav} states that the pump must not cavitate during operation. The values of the external attributes are to be provided via the binding mechanism.

In this example, all attributes are variables.

Example 2: object CoolingSystem that contains three pumps.

$\mathcal{C}_{\text{CoolingSystem}}: CoolingSystem = \{\mathcal{C}_{Pump}: P_1, \mathcal{C}_{Pump}: P_2, \mathcal{C}_{Pump}: P_3, \mathbb{B}_2: s\}$

where s is the state started or not of the cooling system.

In this example, three attributes are objects (P_1, P_2 and P_3), and one attribute is a variable (s). The requirement R_{noCav} applies automatically to all pumps in the cooling system.

Example 3: object CoolingSystem that contains an external set of pumps.

$\mathcal{C}_{\text{CoolingSystem}}: CoolingSystem = \{*.2^{\mathcal{C}_{Pump}}: pumps, \mathbb{B}_2: s\}$

where s is the state started or not of the cooling system. The value of the set of pumps, i.e. the pumps that are in the set, are to be provided via the binding mechanism. There are thus two binding mechanisms involved in forming the value of the set: one that gets the list of pumps that are in the set, and for each pump, one that gets the values of the external attributes of the pump.

In this example, one attribute is a set of objects, and one attribute is a variable. The requirement R_{noCav} applies automatically to all pumps in the cooling system.

3.23. Classes

3.23.1. Definition

A class $\mathcal{C}_O$ is the domain of all objects O having the same attributes $\mathbb{D}_i: a_i$ regardless of their values v_i. The definition $\hat{\mathcal{C}}_O$ of a class $\mathcal{C}_O$ is the set of the common attributes of all objects $O \in \mathcal{C}_O$. It is denoted as follows:

$$\hat{\mathcal{C}}_O = \{\mathbb{D}_1: a_1, \dots \mathbb{D}_i: a_i, \dots \mathbb{D}_n: a_n\} \tag{180}$$

This notation emphasizes that all objects $O \in \mathcal{C}_O$ (i.e., all instances of $\mathcal{C}_O$) have the same structure given by $\hat{\mathcal{C}}_O$ although the values of their variables (i.e., of the attributes that are not objects) can be different from one object to the other. Therefore, $\hat{\mathcal{C}}_O$ is the template for all objects $O \in \mathcal{C}_O$. In the following, the hat above the name of the class will be omitted when there is no confusion between the definition of the class, which is a set of attributes, and the class itself, which is a set of objects. $\hat{\mathcal{C}}_O \in \mathcal{S}$.

To emphasize that a_i is an attribute of class $\mathcal{C}_O$, a_i can be denoted $\mathcal{C}_O.a_i$.

The domain of class definitions is denoted $\mathcal{C}$. Therefore, $\hat{\mathcal{C}}_O$ can be denoted $\mathcal{C}: \hat{\mathcal{C}}_O$ to emphasize that it is a class definition.

3.23.2. Class extensions

Like any domain, a class $\mathcal{C}_{O'}$ can extend another class $\mathcal{C}_O \neq \mathcal{C}_{O'}$. This is denoted $\mathcal{C}_{O'} \subset \mathcal{C}_O$. If $\mathcal{C}_{O'}$ extends $\mathcal{C}_O$, then:

$$\mathcal{C}_{O'} = \mathcal{C}_O + \{\mathbb{D}'_1: a'_1, \dots \mathbb{D}'_i: a'_i, \dots \mathbb{D}'_{n'}: a'_{n'}\} \tag{181}$$

where the a'_i are the additional attributes of $\mathcal{C}_{O'}$.

If $\mathcal{C}_{O'}$ extends $\mathcal{C}_O$, it is possible to redeclare in $\mathcal{C}_{O'}$ an attribute $\mathbb{D}_i: a_i$ of $\mathcal{C}_O$ by adding in $\mathcal{C}_{O'}$ the attribute $\mathbb{D}'_i: a_i$, where $\mathbb{D}'_i$ is the new domain of a_i such that $\mathbb{D}'_i$ extends $\mathbb{D}_i$. Then the declaration $\mathbb{D}'_i: a_i$ is compatible with the initial declaration $\mathbb{D}_i: a_i$ because $a_i \in \mathbb{D}'_i \text{ and } \mathbb{D}'_i \subset \mathbb{D}_i \implies a_i \in \mathbb{D}_i$. It is also possible to modify the name of the redeclared attribute to a new name a'_i by writing $a_i \to a'_i$ to ensure traceability between the old name and the new name. Then the complete redeclaration of a_i in $\mathcal{C}_O$ writes $a_i \to \mathbb{D}'_i: a'_i$ in $\mathcal{C}_{O'}$. In the following equation, all attributes of $\mathcal{C}_O$ are redeclared, provided that $\forall i \in [1, n]\ \mathbb{D}'_i \subset \mathbb{D}_i$:

$$\mathcal{C}_{O'} = \mathcal{C}_O + \{\mathbb{D}'_1: a_1 \to a'_1, \dots \mathbb{D}'_i: a_i \to a'_i, \dots \mathbb{D}'_n: a_n \to a'_{n'}\} \tag{182}$$

It is possible to mix redeclared and non-redeclared attributes in the definition of $\mathcal{C}_{O'}$. The value v'_i of the redeclared attribute $\mathbb{D}'_i: a_i \to a'_i$ in an instance O' of $\mathcal{C}_{O'}$ is denoted $v'_i = O'.a'_i$, or $v'_i = O'.a_i \to a'_i$ if one wants to emphasize the traceability between a_i and a'_i in case $a'_i \neq a_i$.

Contrary to domains that are not classes, a class $\mathcal{C}_{O'}$ can extend several classes $\mathcal{C}_{O_i}$. This is denoted $\mathcal{C}_{O'} \subset \mathcal{C}_{O_1} + \cdots \mathcal{C}_{O_i} + \cdots \mathcal{C}_{O_n}$. Then

$$\mathcal{C}_{O'} = \mathcal{C}_{O_1} + \cdots \mathcal{C}_{O_i} + \cdots \mathcal{C}_{O_n} + \{\mathbb{D}'_1: a'_1, \dots \mathbb{D}'_i: a'_i, \dots \mathbb{D}'_{n'}: a'_{n'}\} \tag{183}$$

where the a'_i are the additional attributes of $\mathcal{C}_{O'}$.

3.23.3. Class specializations

Like any domain extension, a class $\mathcal{C}_{O'}$ can specialize another class $\mathcal{C}_O$ by assigning fixed values to some attributes of $\mathcal{C}_O$:

$$\mathcal{C}_{O'} = \mathcal{C}_O(a_i = v_i,\ \ i \in F_{O'} \subset [1, n]) \tag{184}$$

where $F_{O'}$ denotes the indices of the fixed attributes of O'. Here, the word 'fixed' means that a value has been assigned to an attribute of a class, but the value can still depend on time.

$\mathcal{C}_{O'}$ specializes $\mathcal{C}_O$ is denoted $\mathcal{C}_{O'} \prec \mathcal{C}_O$. Then an instance of $\mathcal{C}_{O'}$ is obtained by assigning values to the remaining non-fixed attributes $a_i \in \bar{F}_{O'} = [1,n] - F_{O'}$:

$$\mathcal{C}_{O'}: O'(a_i = v_i,\ \ i \in \bar{F}_{O'} = [1,n] - F_{O'}) \tag{185}$$

and the values of the fixed attributes in $\mathcal{C}_{O'}$ will be the same for all instances O' of $\mathcal{C}_{O'}$:

$$\forall i \in F_{O'}, \forall (O'_1, O'_2) \in \mathcal{C}_{O'} \times \mathcal{C}_{O'}, O'_1.a_i = O'_2.a_i \tag{186}$$

Contrary to domains that are not classes, it also possible to redefine a fixed attribute $O'.a_{i\in F_{O'}}$ of $O' \in \mathcal{C}_{O'}$ by changing the value of $O'.a_{i\in F_{O'}}$.

3.23.4. Partial classes

Like any domains, a class can be partial. A partial class $\mathcal{C}_O$ is a class that cannot have any instances because it is incompletely defined. It is then expected that its definition will be completed by extending or specializing $\mathcal{C}_O$. A partial class $\mathcal{C}_O$ is denoted $\text{partial}\ \mathcal{C}_O$. Then:

$$\text{partial}\ \mathcal{C}_O \implies \mathcal{C}_O = \emptyset \tag{187}$$

Example 1: class Equipment.

$$\text{partial}\ \mathcal{C}_{\text{Equipment}} = \{*.\mathbb{B}_2: s\} \tag{188}$$

where s is the equipment state in operation or not. Partial means that this class provides an attribute, the equipment state, that is common to all types of equipment, but does not give a sufficient description to create an equipment.

Example 2: class Pump that extends class Equipment.

$$\begin{aligned} \mathcal{C}_{\text{Pump}} = \mathcal{C}_{Equipment} + \{ & \\ & *.\mathbb{R}: c, *.\mathbb{R}: \eta, \mathbb{B}_2: *.cav; \\ & \mathcal{R}: R_{noCav} = \neg cav \otimes [s\uparrow, \neg s\uparrow]\ \} \end{aligned} \tag{189}$$

This definition states that a pump is an equipment with additional attributes:

- c: pump characteristic,
- η: pump efficiency,
- cav: indicates whether the pump is cavitating or not,
- R_{noCav} is a requirement that states that the pump must not cavitate when being in operation.

The values of the external attributes are to be provided via the binding mechanism.

In this example, all attributes are variables.

Example 3: class CoolingSystem that extends class Equipment and uses class Pump.

$$\mathcal{C}_{\text{CoolingSystem}} = \mathcal{C}_{Equipment} + \{\mathcal{C}_{\text{Pump}}: P_1, \mathcal{C}_{\text{Pump}}: P_2, \mathcal{C}_{\text{Pump}}: P_3\} \tag{190}$$

In this example, three attributes are objects (P_1, P_2 and P_3), and one attribute is a variable (s, defined in the class Equipment).

Example 4: class CentrifugalPump that extends class Pump with redeclaration.

$$\mathcal{C}_{\text{CentrifugalPump}} = \mathcal{C}_{\text{Pump}} + \{ *.\mathbb{R}: c \rightarrow c_f \} \tag{191}$$

In this example, only the name of the attribute c of the class Pump has been changed to c_f that means full characteristic.

Example 5: class $\text{CoolingSystem}'$ that extends class CoolingSystem to use class CentrifugalPump instead of class Pump.

$$\mathcal{C}_{\text{CoolingSystem}'} = \mathcal{C}_{\text{CoolingSystem}} + \{\mathcal{C}_{\text{CentrifugalPump}}: P_1, \mathcal{C}_{\text{CentrifugalPump}}: P_2, \mathcal{C}_{\text{CentrifugalPump}}: P_3\} \tag{192}$$

In this example, all pumps have been redeclared as centrifugal pumps without changing their names.

Example 6: class $\text{CoolingSystem}'$ that extends class CoolingSystem to use class CentrifugalPump instead of class Pump.

$$\mathcal{C}_{\text{CoolingSystem}'} = \mathcal{C}_{\text{CoolingSystem}} + \{\mathcal{C}_{\text{CentrifugalPump}}: P_1 \rightarrow P_1', \mathcal{C}_{\text{CentrifugalPump}}: P_2 \rightarrow P_2', \mathcal{C}_{\text{CentrifugalPump}}: P_3 \rightarrow P_3'\} \tag{193}$$

In this example, all pumps have been redeclared as centrifugal pumps with new names.

3.24. Models

3.24.1. Definition

A model M is a set of elements e_i such that the values v_i of all elements e_i can be calculated. To that end, each element $e_i \in M$ is composed of a:

- Declaration that states the name and the domain $\mathbb{D}_i$ of e_i. To declare an element e_i in model M, the following notations are possible: $\mathbb{D}_i: a_i \in M$ or $M.\mathbb{D}_i: a_i$, where $a_i \in \mathbb{S}$ is the name of e_i called the *identifier* of e_i. a_i must be unique within the name space of model M.
- Definition that provides a value v_i to e_i. To define an element e_i, one writes $a_i = v_i$, where v_i denotes the CRML expression that provides a value to e_i.

There is an exception to the above rule that concerns *external elements*. An external element is an element that is declared in model M and defined in another model $M' \neq M$ of any kind that is not necessarily expressed in CRML: requirement model, behavioral model, etc. The declaration of an external element e_i is denoted $\mathbb{D}_i: a_i \in M'$ or $M'.\mathbb{D}_i: a_i$ if model M' is known. Otherwise, it is denoted $\mathbb{D}_i: a_i \in *$ or $*.\mathbb{D}_i: a_i$ where $*$ indicates another model to be defined later via a *binding* mechanism.

A model M can contain the following elements e_i:

- Other models $M_i' \neq M$.
- Domain extensions $\mathbb{D}_i'$.
- Domain specializations $\mathcal{D}_i$.
- Class definitions $\mathcal{C}_i$.
- Operators f_i.
- Categories c_i.

- Objects O_i.
- Variables $x_i \in \mathbb{V}$ of any of the domains listed in Section 3.16.
- Sets S_i.
- At most one frame period F that truncates all time periods defined in the model. It is denoted $\sqsupset M = F$.

The domain of models is denoted $\mathcal{M}$. $\mathcal{M} \subset \mathcal{S}$.

Thus, the definition of models is very similar to the definition of objects, the main differences being that domains can be extended and specialized in models, categories and classes can be defined in models and that models are not instances of classes but belong to a dedicated domain $\mathcal{M}$.

3.24.2. Model extensions

In the same way as for classes (cf. Section 3.23), a model M' can extend a model M. This is denoted $M' \subset M$:

$$M' = M + \{\mathbb{D}'_1: a'_1, \dots \mathbb{D}'_i: a'_i, \dots \mathbb{D}'_{n'}: a'_{n'}\} \tag{194}$$

where the a'_i are the additional attributes of M'.

If M' extends M and if F is the frame period of M, then F is the frame period of M':

$$M' \subset M \text{ and } F = \sqsupset M \implies F = \sqsupset M' \tag{195}$$

In the same way as for classes (cf. Section 3.23), it is possible to redeclare in M' some or all of the attributes of M. The following equation redeclares all the attributes of M:

$$M' = M + \{\mathbb{D}'_1: a_1 \to a'_1, \dots \mathbb{D}'_i: a_i \to a'_i, \dots \mathbb{D}'_n: a_n \to a'_n\} \tag{196}$$

Example 1: a simple cooling system. This model expresses requirements on the pumps of the system only.

$\mathcal{M}$: CoolingSystem = {

Define the class $\mathcal{C}_{\text{Equipment}}$:

- id is a unique identifier for the equipment.
- s is the state of the equipment declared as an external variable: started ($s = true$) or not started ($s = false$).

$\mathcal{C}: \mathcal{C}_{\text{Equipment}} = \{\mathbb{S}: id, \ *.\mathbb{B}_2: s\};$

Define the class $\mathcal{C}_{\text{pump}}$ as an equipment with additional features that are declared as external variables that are elaborated by a behavioral model of the pump:

- ω is the rotational velocity of the pump.
- ω_n is the nominal rotational velocity of the pump.
- cav indicates whether the pump cavitates ($cav = true$) or not ($cav = false$). Cavitation occurs when the pressure P_i at the inlet of the pump is below a threshold P_{min}. The determination of cav includes a margin that considers the uncertainty ΔP_i on the measurement of P_i. There are two possibilities for defining cav:
  1. If $P_i \geq P_{min} + \Delta P_i$, then $cav = true$, else $cav = false$.

2. If $P_i \geq P_{min} + \Delta P_i$, then $cav = true$, else if $P_{min} - \Delta P_i < P_i < P_{min} + \Delta P_i$ then $cav = undecided$, else $cav = false$. This possibility yields more precise results by making the difference between cases where the pressure measurement is outside of the uncertainty margin or not.

$\mathcal{C}{:}\,\mathcal{C}_{\text{Pump}} = \mathcal{C}_{Equipment} + \{$

$*.\mathbb{R}{:}\,\omega,\ *.\mathbb{R}{:}\,\omega_n,\ *.\mathbb{B}_3{:}\,cav;$

$\mathcal{R}{:}\,R_{noCav} = [s\uparrow, \neg s\uparrow] \vDash \neg cav\ \}$

Define the class $\mathcal{C}_{\text{CoolingSystem}}$. The cooling system is defined as an equipment that contains a set of pumps that must not cavitate. The definition of the requirement $\mathcal{C}_{\text{CoolingSystem}}.R_{noCav}$"All pumps must not cavitate while they are in operation" is given as follows: when instantiating the objects pumps from the class $\mathcal{C}_{\text{Pump}}$, the individual requirement $\mathcal{C}_{\text{Pump}}.R_{noCav}$ will be automatically added to the model for each pump. Then the logical conjunction of all individual requirements $\mathcal{C}_{\text{Pump}}.R_{noCav}$ is taken to form the $\mathcal{C}_{\text{CoolingSystem}}.R_{noCav}$ requirement.

There are two possibilities:

1. Declare an external variable for the set of pumps. Then the set of pumps will be defined in another model. This way, the current model is insensitive to the number and characteristics of the pumps that will be chosen at the detailed design step.

   $\mathcal{C}{:}\,\mathcal{C}_{\text{CoolingSystem}} = \mathcal{C}_{\text{Equipment}} + \{*.2^{\mathcal{C}_{\text{Pump}}}{:}\,Pumps, \mathcal{R}{:}\,R_{noCav} = \wedge\ Pumps.R_{noCav}\};$

2. Define the set of pumps in the current model. The only attribute to be specified is the identifier id as it is the only attribute that is not external.

   $\mathcal{C}{:}\,\mathcal{C}_{\text{CoolingSystem}} = \mathcal{C}_{\text{Equipment}} + \{2^{\mathcal{C}_{\text{Pump}}}{:}\,Pumps, \mathcal{R}{:}\,R_{noCav} = \wedge\ Pumps.R_{noCav}\};$

Instantiate the object CoolingSystem from the class $\mathcal{C}_{\text{CoolingSystem}}$.

- If the set of pumps is an external variable, then:

  $\mathcal{C}_{\text{CoolingSystem}}{:}\,CoolingSystem;$

- Else:

  $\mathcal{C}_{\text{CoolingSystem}}{:}\,CoolingSystem(Pumps = \{\mathcal{C}_{\text{Pump}}{:}\,P_1(id = \text{"P1"}), \mathcal{C}_{\text{Pump}}{:}\,P_2(id = \text{"P2"}), \mathcal{C}_{\text{Pump}}{:}\,P_3(id = \text{"P3"})\});$

After instantiation of the object CoolingSystem, the requirement $\mathcal{C}_{\text{CoolingSystem}}.R_{noCav}$"All pumps must not cavitate while they are in operation" is automatically generated.

}

Example 2: refinement of the simple cooling system. This model expresses additional requirements on the pumps of the system following design choices regarding the pumps.

$\mathcal{M}{:}\,\text{CoolingSystem}' = \text{CoolingSystem} + \{$

Extend the class $\mathcal{C}_{\text{Pump}}$ defined in CoolingSystem to include the additional requirement due to pump technology limitations "The pump must not be started more than twice in a sliding time period of one day". The function count is given by Eq. (82).

$\mathcal{C}{:}\,\mathcal{C}_{\text{Pump}'} = \mathcal{C}{:}\,\mathcal{C}_{\text{Pump}} + \{\ 2^{\mathcal{P}}{:}\,P = [\ s\uparrow, s\uparrow +1\text{ d}\ [,\ \mathcal{R}{:}\,R_{noStart} = \text{P} \vDash (\text{count}(s, \text{P}) \leq 2)\ \};$

Extend the class $\mathcal{C}_{\text{CoolingSystem}}$ to redeclare the set of pumps as being instances of the new class $\mathcal{C}_{\text{Pump}'}$.

$\mathcal{C}: \mathcal{C}_{\text{CoolingSystem}'} = \mathcal{C}: \mathcal{C}_{\text{CoolingSystem}} + \left\{ 2^{\mathcal{C}_{\text{Pump}}'}: Pumps \right\}$

Redeclare the object CoolingSystem to be an instance the new class $\mathcal{C}_{\text{CoolingSystem}'}$. In the new design, only two pumps are considered.

- If the set of pumps is an external variable, then:

  $\mathcal{C}_{\text{CoolingSystem}'}: CoolingSystem;$

- Else:

  $\mathcal{C}_{\text{CCoolingSystem}'}: CoolingSystem\left(Pumps = \left\{\mathcal{C}_{\text{Pump}'}: P_1(id = \text{"P1"}), \mathcal{C}_{\text{Pump}'}: \text{P}_2(\text{id="P2"})\right\}\right)$

  Then in the refined model CoolingSystem′, the additional requirement on the pumps will be automatically taken into account.

}

3.25. Libraries

The purpose of libraries is to contain elements that can be reused in models.

A library L is a set that can contain the following elements e_i:

- Other libraries $L'_i \neq L$.
- Domain extensions $\mathbb{D}'_i$.
- Domain specializations $\mathcal{D}_i$.
- Class definitions $\mathcal{C}_i$.
- Operators f_i.
- Categories c_i.

Elements e_i that belong to library L are denoted $e_i \in L$ or $L.e_i$.

The domain of libraries is denoted $\mathcal{L}$. $\mathcal{L} \subset \mathcal{S}$.

Example 1: a library that contains the definition of the classes used in Example 1 of Section 3.24.

$\mathcal{L}: \text{CoolingLib} = \{$

$\mathcal{C}: \mathcal{C}_{\text{Equipment}} = \{\mathbb{S}: id, \ *.\mathbb{B}_2: s\};$

- id is a unique identifier for the equipment.
- s is the state of the equipment: started ($s = true$) or not started ($s = false$).

$\mathcal{C}: \mathcal{C}_{\text{Pump}} = \mathcal{C}_{Equipment} + \{$

$*.\mathbb{R}: \omega, \ *.\mathbb{R}: \omega_n, \ *.\mathbb{B}_3: cav;$

$\mathcal{R}: R_{noCav} = \neg cav \otimes [s \uparrow, \neg s \uparrow] \}$

- ω is the rotational velocity of the pump.
- ω_n is the nominal rotational velocity of the pump.
- cav indicates whether the pump cavitates ($cav = true$) or not ($cav = false$). Cavitation occurs

when the pressure P_i at the inlet of the pump is below a threshold P_{min}. The determination of cav includes a margin that considers the uncertainty ΔP_i on the measurement of P_i. There are two possibilities for defining cav:

1. If $P_i \geq P_{min} + \Delta P_i$, then $cav = true$, else $cav = false$.
2. If $P_i \geq P_{min} + \Delta P_i$, then $cav = true$, else if $P_{min} - \Delta P_i < P_i < P_{min} + \Delta P_i$ then $cav = undecided$, else $cav = false$. This possibility yields more precise results by making the difference between cases where the pressure measurement is outside of the uncertainty margin or not.

$$\mathcal{C}: \mathcal{C}_{CoolingSystem} = \mathcal{C}_{Equipment} + \left\{*.\, 2^{\mathcal{C}_{Pump}}: Pumps, \mathcal{R}: R_{noCavitation} = \wedge\ Pumps.R_{noCav}\right\};$$

or

$$\mathcal{C}: \mathcal{C}_{CoolingSystem} = \mathcal{C}_{Equipment} + \left\{2^{\mathcal{C}_{Pump}}: Pumps, \mathcal{R}: R_{noCavitation} = \wedge\ Pumps.R_{noCav}\right\};$$

}

Then the model CoolingSystem can be defined using the library CoolingLib:

$\mathcal{M}$: CoolingSystem = { CoolingLib.$\mathcal{C}_{CoolingSystem}$: $CoolingSystem$ }

or

$\mathcal{M}$: CoolingSystem = {

$$\text{CoolingLib}.\mathcal{C}_{CoolingSystem}: CoolingSystem\big(Pumps = \{\mathcal{C}_{\text{Pump}}: P_2(id = \text{"P2"}), \mathcal{C}_{\text{Pump}}: P_2(id = \text{"P2"}), \mathcal{C}_{\text{Pump}}: P_3(id = \text{"P3"})\big)$$

}

depending on whether the set of pumps is an external variable or not.

3.26. Packages

The purpose of a package is to contain all elements that are needed to compute a model.

A package P is a set that can contain the following elements e_i:

- Other packages $P'_i \neq P$.
- Libraries L_i..
- Models M_i.

Elements e_i that belong to package P are denoted $e_i \in P$ or $P.e_i$.

The domain of packages is denoted $\mathcal{T}$. $\mathcal{T} \subset \mathcal{S}$

Example 1: a package for the cooling system of Example 1 in Section 3.25.

$\mathcal{T}$: CoolingSystemPackage = { CoolingLib, CoolingSystem }

4. Syntax

4.1. Notation

[*expr*] denotes an optional expression *expr* when the color of the square brackets is black.

{ expr } denotes an expression that is repeated one or more times when the color of the curly braces is black.

*expr*_1 | *expr*_2 | … | *expr_n* denotes n possible alternatives between the expressions *expr*_1 to *expr_n*.

`'c'` denotes the character `c`.

`"keyword"` denotes the String `keyword`.

Built-in keywords are written in blue.

User-defined objects names are written in orange.

Categories names are written in red.

4.2. Keywords

4.2.1. Types

Keyword	Semantics	Comments
Boolean	$\mathbb{B}$	4-valued Booleans.
Category	$\mathcal{C}(\mathbb{O}_1 \longrightarrow \mathbb{O}_2)$	Categories.
class	$\mathcal{C}$	Class definitions.
Clock	$2^{\mathcal{E}}$ or $\mathcal{D}$	Clocks.
Event	$\mathcal{E}$	Events.
Integer	$\mathbb{Z}$	Positive and negative integers.
library	$\mathcal{L}$	Libraries.
model	$\mathcal{M}$	Models.
Operator	$\mathbb{O}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2)$	Operators. The names of the domains $\mathbb{D}_1$ and $\mathbb{D}_2$ are given in the declaration of the operator.
package	$\mathcal{T}$	Packages.
Period	$\mathcal{P}$	Single time periods.
Periods	$2^{\mathcal{P}}$	Multiple time periods.
Probability	$\mathbb{O}(\mathbb{B}_2 \longrightarrow \mathbb{R})$	Probabilities.
Real	$\mathbb{R}$	Real numbers.
String	$\mathbb{S}$	Strings.
Template	$\mathbb{O}(\mathbb{B}^n \longrightarrow \mathbb{B})$	Templates.
type	$\mathfrak{T}$	Type definitions.

4.2.2. Special values

Values	Semantics	Comments
false	$false$	
true	$true$	
undecided	$undecided$	
undefined	$undefined$	
time		

4.2.3. Special characters

Characters	Semantics	Comments
(	$($	
)	$)$	
[	$[$	
]	$]$	
{	$\{$	Opens sets.
}	$\}$	Closes sets.
,	$,$	Set element separator.
;	$;$	Set element separator.
.	$.$	Decimal point. Path element separator.
"	"	String delimiter.
'		Quote string delimiter.
E		Decimal exponent.
e		Decimal exponent.
//		Start of comment line.
/*		Begins comment.
*/		Ends comment.
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9		Digits

4.2.4. Operators

Operators	Comments
=	
+	
-	

Operators	Comments
*	
/	
<	
<=	
>	
>=	
==	
<>	
^	
acos	
alias	
and	
asin	
associate	
at	
card	
constant	
cos	
duration	
element	
else	
end	
estimator	
exp	
extends	
external	
filter	
flatten	
forbid	
if	
integrate	
is	
log	
log10	
mod	
new	

Operators	Comments
not	
on	
or	
parameter	
partial	
proj	
redeclare	
sin	
start	
then	
tick	
time from	
union	
variance	
while	
with	

4.3. Expressions

The CRML expression for the declaration and/or definition of any CRML element is of the form:

Natural language syntax	Mathematical syntax	Semantics	Reference
[[*type*] *ident* is] [*value* | external] [; | ,]	[[*type*] *ident* =] [*value* | external] [; | ,]	$type: ident = value$ $*.type: ident$	3.20.2 3.22 3.24.1

The semicolon (;) can be replaced by the comma (,), and is only mandatory when two or more expressions must be separated in sets or operator calls.

type ident is the declaration of the element of type *type* and name *ident*.

type is the type of the expression, to be chosen among Real, Integer, String, Boolean, Event, etc., or user-defined types.

ident is the identifier of the expression which is unique within its scope (or namespace). An identifier is a character string composed of a non-digit optionally followed by digits or non-digits:

```
ident: non-digit [{digit | non-digit}]
non-digit: '_' | characters 'a' to 'z' | characters 'A' to 'Z'
digit: characters '0' to '9'
```

The identifier, and therefore the type, are not explicitly written when the expression is an argument in the call of an operator: the type and the identifier are automatically inferred by the argument binding mechanism from the position of the argument in the function call, and the alias of the argument type if any.

The identifier is omitted for anonymous sets.

There is a special syntax for the declaration of an operator in the natural language syntax, cf. Section 4.13.

The type is not explicitly written when the expression is an argument in the constructor of an object. It is automatically inferred from its identifier and from the object's class definition.

value is the definition of the element. It denotes the value of the expression, which depends on the type of the element. It is expressed by an expression using a constructor or an operator that take expressions as arguments and return a value of the element type. Expressions are thus recursively defined until they involve constructors with no arguments or external expressions:

```
expr: [[type] ident is] [value | external]
value: ([new] constructor [{expr}]) | (operator {expr})
```

The value of the expression is always explicit: it cannot be obtained by solving an equation. The keyword external stands for the *value* when the element is external.

If the name of the constructor is the same as the name of a user-defined class, then the keyword new must be used to create an instance of the class.

There are two kinds of CRML elements: sets and set elements. Sets contain set elements. A set can be a set element of another set.

A set can depend on time, depending on its type, which means that the set elements can be added or removed dynamically. Dynamic sets are of the type Clock or Periods.

A set can be empty.

There are three kinds of sets: typed sets, special sets, and the universal set. In a typed set, all elements are of the same type. In a special set, elements can be of different types. The special sets are of the type type, class, model, library or package. The universal set is the only set that has no expression: it has no identifier and is implicitly defined as being the set that contains all sets except itself.

The scope of an expression is the set where it belongs.

The implicit type of the universal set (the set that contains all sets) is package, because package is the only set that can contain (directly or indirectly) all sets of the language.

An expression refers to another expression outside of its scope by appending the relative path of the outside expression to the identifier of the outside expression (absolute path is not possible because the universal set has no identifier).

4.4. Comments

Comments are identical to C++ (or Modelica).

There are two kinds of comments, single-line comment and multiline comments.

// This is a single-line comment

The characters from // until the end of the line are ignored.

/* This is a

multiline comment */

The characters enclosed between /* and */ are ignored, including return carriages. Multiline comments cannot be nested.

Expressions in comments are not part of the CRML language.

4.5. Type Real

4.5.1. Constructors

Natural language syntax	Mathematical syntax	Semantics	Reference
Real x is *decimal_value*;	Real x = *decimal_value*;	$\mathbb{R}: x = decimal\_value$	

Integer n;
Real x is new Real n;
	Integer n;
	Real x = new Real n;
		$\mathbb{N}: n$
		$\mathbb{R}: x = n$	

A decimal value is defined as

```
decimal_value: [sign] [{‘ ‘}] {digit} [‘.’] [{digit}] [exponent]
sign: ‚+‘ | ‚-‚
exponent: (‚E‘ | ‚e‘) [sign] {digit}
digit: characters ‘0’ to ‘9’
```

where:

- `‘+’ | ‘-‘` denotes the + or the – sign,
- `{‘ ‘}` denotes one or more spaces,
- `{digit}` denotes one or more digits,
- `‘.’` is the decimal point,
- `‘E’` is the decimal exponent: 1En = 10^n where n is a positive or negative integer,
- `‘e’` can be used instead of `‘E’`.

Example 1: correct expressions

Real x1 is –28.775E+3;

Real x2 is 0.28e–7;

Real x3 is +3.E10;

Real x4 is – 25.3;

Real x5 is 25;

Example 2: incorrect expressions

Real x6 is .7; // No digit before the decimal point

Real x7 is –28.775E +3; // Space after the decimal exponent

Real x8 is –28.775 E+3; // Space before the decimal exponent

Real x9 is –28.775E+ 3; // Space after the + sign

Real x10 is E+3; // No digit before the decimal exponent

Real x11 is 1.E+3.14; // Decimal point in the decimal exponent

4.5.2. Operators

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Binary addition	Real x1, x2;
	Real x is x1 + x2;
		Real x1, x2;
		Real x = x1 + x2;
			$\mathbb{R}: x_1, \mathbb{R}: x_2$
			$\mathbb{R}: x = x_1 + x_2$	

Binary subtraction	Real x1, x2;
	Real x is x1 – x2;
		Real x1, x2;
		Real x = x1 – x2;
			$\mathbb{R}: x_1, \mathbb{R}: x_2$
			$\mathbb{R}: x = x_1 - x_2$	

Unary addition	Real x1;
	Real x2 is +x1;
		Real x1;
		Real x2 = +x1;
			$\mathbb{R}: x_1$
			$\mathbb{R}: x_2 = +x_1$	

Unary subtraction	Real x1;
	Real x2 is –x1;
		Real x1;
		Real x2 = –x1;
			$\mathbb{R}: x_1$
			$\mathbb{R}: x_2 = -x_1$	

Multiplication	Real x1, x2;
	Real x is x1 * x2;
		Real x1, x2;
		Real x = x1 * x2;
			$\mathbb{R}: x_1, \mathbb{R}: x_2$
			$\mathbb{R}: x = x_1 \times x_2$	

Division	Real x1, x2;
	Real x is x1 / x2;
		Real x1, x2;
		Real x = x1 / x2;
			$\mathbb{R}: x_1, \mathbb{R}^*: x_2$
			$\mathbb{R}: x = x_1 / x_2$	

Exponentiation	Real x1, x2;
	Real x is x1 ^ x2;
		Real x1, x2;
		Real x = x1 ^ x2;
			$\mathbb{R}: x_1, \mathbb{R}: x_2$
			$\mathbb{R}: x = {x_1}^{x_2}$	

Greater than	Real x1, x2;
	Boolean b is x1 >= x2;
		Real x1, x2;
		Boolean b = x1 >= x2;
			$\mathbb{R}: x_1, \mathbb{R}: x_2$
			$\mathbb{B}_2: b = x_1 \geq x_2$	

Less than	Real x1, x2;
	Boolean b is x1 <= x2;
		Real x1, x2;
		Boolean b = x1 <= x2;
			$\mathbb{R}: x_1, \mathbb{R}: x_2$
			$\mathbb{B}_2: b = x_1 \leq x_2$	

Strictly greater than	Real x1, x2;
	Boolean b is x1 > x2;
		Real x1, x2;
		Boolean b = x1 > x2;
			$\mathbb{R}: x_1, \mathbb{R}: x_2$
			$\mathbb{B}_2: b = x_1 > x_2$	

Strictly less than	Real x1, x2;
	Boolean b is x1 < x2;
		Real x1, x2;
		Boolean b = x1 < x2;
			$\mathbb{R}: x_1, \mathbb{R}: x_2$
			$\mathbb{B}_2: b = x_1 < x_2$	

Sine	Real x1, x2;
	Real x2 is sin x1;
		Real x1, x2;
		Real x2 = sin (x1);
			$\mathbb{R}: x_1$
			$\mathbb{R}: x_2 = \sin x_1$	

Cosine	Real x1, x2;
	Real x2 is cos x1;
		Real x1, x2;
		Real x2 = cos (x1);
			$\mathbb{R}: x_1$
			$\mathbb{R}: x_2 = \cos x_1$	

Inverse sine	Real x1, x2;
	Real x2 is asin x1;
		Real x1, x2;
		Real x2 = asin x1;
			$\mathbb{R}: x_1$
			$\mathbb{R}: x_2 = \operatorname{asin} x_1$	

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Inverse cosine	Real x1, x2;
	Real x2 is acos x1;
		Real x1, x2;
		Real x2 = acos x1;
			$\mathbb{R}: x_1$
			$\mathbb{R}: x_2 = \operatorname{acos} x_1$	

Exponential	Real x1, x2;
	Real x2 is exp x1;
		Real x1, x2;
		Real x2 = exp (x1);
			$\mathbb{R}: x_1$
			$\mathbb{R}: x_2 = e^{x_1}$	

Logarithm	Real x1, x2;
	Real x2 is log x1;
		Real x1, x2;
		Real x2 = log (x1);
			$\mathbb{R}_+^*: x_1$
			$\mathbb{R}: x_2 = \log_e x_1$	

Base 10 logarithm	Real x1, x2;
	Real x2 is log10 x1;
		Real x1, x2;
		Real x2 = log10 (x1);
			$\mathbb{R}_+^*: x_1$
			$\mathbb{R}: x_2 = \log_{10} x_1$	

If then else	Boolean b;
	Real x1, x2;
	Real x is if b then x1 else x2;
		Boolean b;
		Real x1, x2;
		Real x = if b then x1 else x2;
			$\mathbb{B}: b, \mathbb{R}: x_1, \mathbb{R}: x_2$
			$\mathbb{R}: x = \begin{cases} x_1 & \text{if } b = true \\ x_2 & \text{else} \end{cases}$	

At event	Real x1;
	Clock c;
	Real x2 is x1 at c;
		Real x1;
		Clock c;
		Real x2 = x1 at c;
			$\mathbb{R}: x_1, \mathcal{D}: c$
			$\mathbb{R}: x_2 = x_1(@\, t/\Omega)$
				3.7.1

The expression

x2 is x1 at c;

takes the value of x1 at every tick of clock c. The value of x2 between two ticks t1 and t2 is equal to the value of x1 at tick t1.

Example 1: verifying that x1 is greater than x2 when y becomes positive.

```
Real x1;
Real x2;
Real y;
Clock c is new Clock (y > 0);
Boolean b is (x1 > x2) at c;
```

It is also possible to write:

```
Boolean b is (x1 at c) > (x2 at c);
```

4.6. Type Integer

4.6.1. Constructors

Natural language syntax	Mathematical syntax	Semantics	Reference
Integer n is *integer_value*;	Integer n = *integer_value*;	$\mathbb{Z}: n = integer\_value$	

Real x;
Integer n is new Integer x;
	Real x;
	Integer n = new Integer x;
		$\mathbb{R}: x$
		$\mathbb{Z}: n = x$	

An integer value is defined as

```
integer_value: [sign] [{' '}] {digit} [exponent]

sign: '+' | '-'

exponent: ('E' | 'e') {digit}

digit: characters '0' to '9'
```

where:

- `'+' | '-'` denotes the + or the – sign,
- `{' '}` denotes one or more spaces,
- `{digit}` denotes one or more digits,
- `'E'` is the decimal exponent: 1.En = 10^n where n is a positive or negative integer,
- `'e'` can be used instead of `'E'`.

Example 1: correct expressions

Integer x1 is 25;

Integer x2 is + 25;

Integer x3 is –25;

Integer x4 is –28E3;

Integer x5 is new Integer –25.8; // The value of x is –25

Example 2: incorrect expressions

Integer x6 is –25.8; // Decimal point

Integer x7 is –28E 3; // Space after the decimal exponent

Integer x8 is –28 E+3; // Space before the decimal exponent

Integer x9 is –28E+3; // Sign after the decimal exponent

Integer x10 is E3; // No digit before the decimal exponent

Integer x11 is 1E3.; // Decimal point in the decimal exponent

4.6.2. Operators

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Binary addition	Integer n1, n2;
	Integer n is n1 + n2;
		Integer n1, n2;
		Integer n = n1 + n2;
			$\mathbb{Z}: n_1, \mathbb{Z}: n_2$
			$\mathbb{Z}: n = n_1 + n_2$	

Binary subtraction	Integer n1, n2;
	Integer n is n1 – n2;
		Integer n1, n2;
		Integer n = n1 – n2;
			$\mathbb{Z}: n_1, \mathbb{Z}: n_2$
			$\mathbb{Z}: n = n_1 - n_2$	

Unary addition	Integer n1;
	Integer n2 is +n1;
		Integer n1;
		Integer n2 = +n1;
			$\mathbb{Z}: n_1$
			$\mathbb{Z}: n_2 = +n_1$	

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Unary subtraction	Integer n1;
	Integer n2 is –n1;
		Integer n1;
		Integer n2 = –n1;
			$\mathbb{Z}: n_1$
			$\mathbb{Z}: n_2 = -n_1$	

Multiplication	Integer n1, n2;
	Integer n is n1 * n2;
		Integer n1, n2;
		Integer n = n1 * n2;
			$\mathbb{Z}: n_1, \mathbb{Z}: n_2$
			$\mathbb{Z}: n = n_1 \times n_2$	

Integer division	Integer n1, n2;
	Integer n is n1 / n2;
		Integer n1, n2;
		Integer n = n1 / n2;
			$\mathbb{Z}: n_1, \mathbb{Z}^*: n_2$
			$\mathbb{Z}: n = n_1/n_2$	

Exponentiation	Integer n1, n2;
	Integer n is n1 ^ n2;
		Integer n1, n2;
		Integer n = n1 ^ n2;
			$\mathbb{Z}: n_1, \mathbb{N}: n_2$
			$\mathbb{Z}: n = {n_1}^{n_2}$	

Greater than	Integer n1, n2;
	Boolean b is n1 >= n2;
		Integer n1, n2;
		Boolean b = n1 >= n2;
			$\mathbb{Z}: n_1, \mathbb{Z}: n_2$
			$\mathbb{B}_2: b = n_1 \geq n_2$	

Less than	Integer n1, n2;
	Boolean b is n1 <= n2;
		Integer n1, n2;
		Boolean b = n1 <= n2;
			$\mathbb{Z}: n_1, \mathbb{Z}: n_2$
			$\mathbb{B}_2: b = n_1 \leq n_2$	

Strictly greater than	Integer n1, n2;
	Boolean b is n1 > n2;
		Integer n1, n2;
		Boolean b = n1 > n2;
			$\mathbb{Z}: n_1, \mathbb{Z}: n_2$
			$\mathbb{B}_2: b = n_1 > n_2$	

Strictly less than	Integer n1, n2;
	Boolean b is n1 < n2;
		Integer n1, n2;
		Boolean b = n1 < n2;
			$\mathbb{Z}: n_1, \mathbb{Z}: n_2$
			$\mathbb{B}_2: b = n_1 < n_2$	

Equal to	Integer n1, n2;
	Boolean b is n1 == n2;
		Integer n1, n2;
		Boolean b = n1 == n2;
			$\mathbb{Z}: n_1, \mathbb{Z}: n_2$
			$\mathbb{B}_2: b = (n_1 = n_2)$	

Different from	Integer n1, n2;
	Boolean b is n1 <> n2;
		Integer n1, n2;
		Boolean b = n1 <> n2;
			$\mathbb{Z}: n_1, \mathbb{Z}: n_2$
			$\mathbb{B}: b = (n_1 \neq n_2)$	

Modulo	Integer n1, n2;
	Integer n is n1 mod n2;
		Integer n1, n2;
		Integer n = n1 mod n2;
			$\mathbb{Z}: n_1, \mathbb{Z}: n_2$
			$\mathbb{Z}/n_2\mathbb{Z}: n = n_1 \mathrm{mod}\, n_2$	

If then else	Boolean b;
	Integer n1, n2;
	Integer n is if b then n1 else n2;
		Boolean b;
		Integer n1, n2;
		Integer n = if b then n1 else n2;
			$\mathbb{B}: b, \mathbb{Z}: n_1, \mathbb{Z}^*: n_2$
			$\mathbb{Z}: n = \begin{cases} n_1 & \text{if } b = true \\ n_2 & \text{else} \end{cases}$	

At event	Integer n1;
	Clock c;
	Integer n2 is n1 at c;
		Integer n1;
		Clock c;
		Integer n2 = n1 at c;
			$\mathbb{Z}: n_1, \mathcal{D}: c$
			$\mathbb{Z}: n_2 = n_1(@\, t/\Omega)$
				3.7.1

The expression

n2 is n1 at c;

takes the value of n1 at every tick of clock c. The value of n2 between two ticks t1 and t2 is equal to the value of n1 at tick t1.

Example 1: verifying that n1 is greater than n2 when y becomes positive.

Integer n1;

Integer n2;

Real y;

Clock c is new Clock (y > 0);

Boolean b is (n1 > n2) at c;

It is also possible to write:

Boolean b is (n1 at c) > (n2 at c);

4.7. Type String

4.7.1. Constructors

Natural language syntax	Mathematical syntax	Semantics	Reference
String s is *string_value*;	String s = *string_value*;	$\mathbb{S}: x = string\_value$	

Real x;
String s is new String x;
	Real x;
	String s = new String x;
		$\mathbb{R}: x$
		$\mathbb{S}: s = \text{string } x$	

Integer n;
String s is new String n;
	Integer n;
	String s = new String n;
		$\mathbb{N}: n$
		$\mathbb{S}: s = \text{string } n$	

Boolean b;
String s is new String b;
	Boolean b;
	String s = new String b;
		$\mathbb{B}: b$
		$\mathbb{S}: s = \text{string } b$	

A string value is defined as:

```
string_value: '"' [ { digit | non-digit | special_char | escape_char} ] '"'

non-digit: '_' | characters 'a' to 'z' | characters 'A' to 'Z'

digit: characters '0' to '9'

special_char: '!' | '#' | '$' | '%' | '&' | '(' | ')' | '*' |      '+' | ','
              | '-' | '.' | '/' | ':' | ';' | '<' | '>' | '=' | '?' | '@' |
              '[' | ']' | '^' | '{' | '}' | '|' | '~' | ' '
escape_char: '\"' | '\\' | '\''
```

Example 1: correct expressions

String s1 is "This is a string";

String s2 is new String "This is a \"nice\" string"; // The value of s is " This is a "nice" string "

String s3 is ""; // Empty string

String s4 is "$PO1_ is the identifier of the pump";

String s5 is new String -23; // The value of s is "-23"

String s6 is new String 10.5E7; // The value of s is "10.5E7"

String s7 is new String true; // The value of s is "true"

String s8 is new String false; // The value of s is "false"

String s9 is new String undecided; // The value of s is "undecided"

String s10 is new String undefined; // The value of s is "undefined"

Example 2: incorrect expressions

String s11 is This is a string"; // First double quote missing

String s12 is "This is " a string"; // Escape char \ missing before second double quote

4.7.2. Operators

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Concatenation	String s1, s2;
	String s is s1 + s2;
		String s1, s2;
		String s = s1 + s2;
			$\mathbb{S}: s_1, \mathbb{S}: s_2$
			$\mathbb{S}: s = s_1 + s_2$	

Example 1: string concatenation

Integer n is 3;

Real x is 2.5;

String s is "The " + new String n + " ships follow each other from a distance of " + new String x + " kilometers.";

The value of s is "The 3 ships follow each other from a distance of 2.5 kilometers."

Example 2: string concatenation

String s is "The set of 4-valued Booleans is { " + new String undecided + ", " + new String undefined + ", " + new String false + ", " + new String true + " }. "

The value of s is “The set of 4-valued Booleans is { undecided, undefined, false, true }.”

4.8. Type Boolean

4.8.1. Constructors

Natural language syntax	Mathematical syntax	Semantics	Reference
Boolean b is true;	Boolean b = true;	$\mathbb{B}: b = true$	3.2
Boolean b is false;	Boolean b = false;	$\mathbb{B}: b = false$	3.2
Boolean b is undecided;	Boolean b = undecided;	$\mathbb{B}: b = undecided$	3.2
Boolean b is undefined;	Boolean b = undefined;	$\mathbb{B}: b = undefined$	3.2

Clock c;
Boolean b is new Boolean c;
	Clock c;
	Boolean b = new Boolean c;
		$\mathcal{D}: \mathrm{c}$
		$\mathbb{B}: b = b(\mathrm{c})$
			3.6

4.8.2. Operators

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Conjunction	Boolean b1, b2;
	Boolean b is b1 and b2;
		Boolean b1, b2;
		Boolean b = b1 and b2;
			$\mathbb{B}: b_1, \mathbb{B}: b_2$
			$\mathbb{B}: b = b_1 \wedge b_2$
				3.2

Disjunction	Boolean b1, b2;
	Boolean b is b1 or b2;
		Boolean b1, b2;
		Boolean b = b1 or b2;
			$\mathbb{B}: b_1, \mathbb{B}: b_2$
			$\mathbb{B}: b = b_1 \vee b_2$
				3.2

Conjunction with event	Boolean b1;
	Event e;
	Boolean b2 is b1 and e;
		Boolean b1;
		Event e;
		Boolean b2 = b1 and e;
			$\mathbb{B}: b_1, \mathcal{E}: e$
			$\mathbb{B}: b_2 = b_1 \wedge e$
				3.5.11

Disjunction with event	Boolean b1;
	Event e;
	Boolean b2 is b1 or e;
		Boolean b1;
		Event e;
		Boolean b2 = b1 or e;
			$\mathbb{B}: b_1, \mathcal{E}: e$
			$\mathbb{B}: b_2 = b_1 \vee e$
				3.5.11

Negation	Boolean a;
	Boolean b is not a;
		Boolean a;
		Boolean b = not a;
			$\mathbb{B}: a$
			$\mathbb{B}: b = \neg a$
				3.2

Filter	Boolean b1, b2;
	Boolean b is b1 * b2;
		Boolean b1, b2;
		Boolean b = b1 * b2;
			$\mathbb{B}_3: b_1, \mathbb{B}: b_2$
			$\mathbb{B}: b = b_1 \times b_2$
				3.11.2

Accumulation	Boolean b1, b2;
	Boolean b is b1 + b2;
		Boolean b1, b2;
		Boolean b = b1 + b2;
			$\mathbb{B}: b_1, \mathbb{B}: b_2$
			$\mathbb{B}: b = b_1 + b_2$
				3.11.1

Integration	Boolean a;
	Period P;
	Boolean b is integrate a on P;
		Boolean a;
		Period P;
		Boolean b = integrate (a, P);
			$\mathbb{B}: a, \mathcal{P}: P$
			$\mathbb{B}: b = \int_P a$
				3.11.6

Duration	Boolean a;
	Period P;
	Real d is duration a on P;
		Boolean a;
		Period P;
		Real d = duration (a, P);
			$\mathbb{B}: a, \mathcal{P}: P$
			$\mathbb{R}^+: d = \text{duration}(a, P)$
				3.11.3

Equality	Boolean b1, b2;
	Boolean b is b1 == b2;
		Boolean b1, b2;
		Boolean b = (b1 == b2);
			$\mathbb{B}: b_1, \mathbb{B}: b_2$
			$\mathbb{B}: b = (b_1 = b_2)$
				3.2

If then else	Boolean b;
	Boolean b1, b2;
	Boolean b3 is if b then b1 else b2;
		Boolean b;
		Boolean b1, b2;
		Boolean b3 = if b then b1 else b2;
			$\mathbb{B}: b, \mathbb{B}: b_1, \mathbb{B}: b_2$
			$\mathbb{B}: b_3 = \begin{cases} b_1 & \text{if } b = true \\ b_2 & \text{else} \end{cases}$	

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

At event	Boolean b1;
	Clock c;
	Boolean b2 is b1 at c;
		Boolean b1;
		Clock c;
		Boolean b2 = b1 at c;
			$\mathbb{B}: b_1, \mathcal{D}: c$
			$\mathbb{B}: b_2 = b_1(@\, t/\Omega)$
				3.7.1

The expression

b2 is b1 at c;

takes the value of b1 at every tick of clock c. The value of b2 between two ticks t1 and t2 is equal to the value of b1 at tick t1.

Example 1: verifying that b1 is equal to b2 when y becomes positive.

```
Boolean b1;
Boolean b2;
Real y;
Clock c is new Clock (y > 0);
Boolean b is (b1 == b2) at c;
```

It is also possible to write:

```
Boolean b is (b1 at c) == (b2 at c);
```

4.9. Type Event

4.9.1. Constructors

Natural language syntax	Mathematical syntax	Semantics	Reference

Boolean b;
Event e is new Event b;
	Boolean b;
	Event e = new Event b;
		$\mathbb{B}: b$
		$\mathcal{E}: e = b \uparrow$ (first occurrence of $b \uparrow$)
			3.3

4.9.2. Operators

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Projection	Event e1;
	Clock c;
	Event e2 is e1 proj c;
		Event e1;
		Clock c;
		Event e2 = proj (e1, c);
			$\mathcal{E}: e_1, \mathcal{D}: c$
			$\mathcal{E}: e_2 = e_1/c$
				3.7.1

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Bounded projection	Event e1;
	Clock c;
	Real d;
	Event e2 is e1 proj(d) c;
		Event e1;
		Clock c;
		Real d;
		Event e2 = e1 proj (e1, c, d);
			$\mathcal{E}: e_1, \mathcal{D} : c, \mathbb{R}^+: d$
			$\mathcal{E}: e_2 = e_1 /^{d} c$
				3.7.2

Delay	Event e1;
	Real d;
	Event e2 is e1 + d;
		Event e1;
		Real d;
		Event e2 = e1 + d;
			$\mathcal{E}: e_1, \mathbb{R}^+: d$
			$\mathcal{E}: e_2 = e_1 + d$
				3.5.8

Elapsed	Event e1, e2;
	Real d is e2 - e1;
		Event e1, e2;
		Real d = e2 - e1;
			$\mathcal{E}: e_1, \mathcal{E}: e_2,$
			$\mathbb{R}^+: d = e_2 - e_1$
				3.5.7

Before	Event e1, e2; Boolean b is e1 <= e2;	Event e1, e2; Boolean b = e1 <=	$\mathcal{E}: e_1, \mathcal{E}: e_2,$ $\mathbb{R}^+: b = e_1 \leq e_2$	3.5.3

After	Event e1, e2;
	Boolean b is e1 >= e2;
		Event e1, e2;
		Boolean b = e1 >= e2;
			$\mathcal{E}: e_1, \mathcal{E}: e_2,$
			$\mathbb{R}^+: b = e_1 \geq e_2$
				3.5.4

Strictly before	Event e1, e2;
	Boolean b is e1 < e2;
		Event e1, e2;
		Boolean b = e1 < e2;
			$\mathcal{E}: e_1, \mathcal{E}: e_2,$
			$\mathbb{R}^+: b = e_1 < e_2$
				3.5.3

Strictly after	Event e1, e2;
	Boolean b is e1 > e2;
		Event e1, e2;
		Boolean b = e1 > e2;
			$\mathcal{E}: e_1, \mathcal{E}: e_2,$
			$\mathbb{R}^+: b = e_1 > e_2$
				3.5.4

Elapsed physical time	Event e; Real d is time from e;	Event e; Real d is time - e;	$\mathcal{E}: e$ $\mathbb{R}^+: d = \mathfrak{C} \uparrow - e$	3.3 3.5.7

Conjunction	Event e1, e2;
	Clock e is e1 and e2;
		Event e1, e2;
		Clock e = e1 and e2;
			$\mathcal{E}: e_1, \mathcal{E}: e_2,$
			$\mathcal{D}: c = e_1 \wedge e_2$
				3.5.9

Disjunction	Event e1, e2;
	Clock e is e1 or e2;
		Event e1, e2;
		Clock e = e1 or e2;
			$\mathcal{E}: e_1, \mathcal{E}: e_2,$
			$\mathcal{D}: c = e_1 \vee e_2$
				3.5.10

4.10. Type Clock

4.10.1.Constructors

Natural language syntax	Mathematical syntax	Semantics	Reference

Boolean b;
Clock c is new Clock b;
	Boolean b;
	Clock c = new Clock b;
		$\mathbb{B}: b$
		$\mathcal{D}: c = c(b)$
			3.6

4.10.2.Operators

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Delay	Clock c1;
	Integer d;
	Clock c2 is c1 + d;
		Clock c1;
		Integer d;
		Clock c2 = c1 + d;
			$\mathcal{D}: c_1, \mathbb{N}: d$
			$\mathcal{D}: c_2 = c_1 + d$
				3.7.3

Projection	Clock c1, c2;
	Clock c is c1 proj c2;
		Clock c1, c2;
		Clock c = proj (c1, c2);
			$\mathcal{D}: c_1, \mathcal{D}: c_2$
			$\mathcal{D}: c = c_1 / c_2$
				3.7.1

Bounded projection	Clock c1, c2;
	Real d;
	Clock c is c1 proj (d) c2;
		Clock c1, c2;
		Real d;
		Clock c = proj (c1, c2, d);
			$\mathcal{D}: c_1, \mathcal{D}: c_2, \mathbb{R}^+: d$
			$\mathcal{D}: c = c_1 /^{d} c_2$
				3.7.2

Current tick	Clock c;
	Event e is tick c;
		Clock c;
		Event e = tick (c);
			$\mathcal{D}: c$
			$\mathcal{E}: e = c^{\|c\|}$
				3.6

Filter	Clock c1, c2;
	Operator cond [Boolean] Integer i = *value* (i);
	Clock c2 is c1 filter cond tick;
		Clock c1, c2;
		Operator cond [Boolean] Integer i = *value* (i);
		Clock c2 = c1 filter cond tick;
			$\mathcal{D}: c_1, \mathbb{O}(\mathbb{N} \longrightarrow \mathbb{B}): \mathrm{cond}$
			$\mathcal{D}: c_2 = c_1 \left(\mathrm{cond}(\mathrm{rank}(*))\right)$
				3.6
				3.7.7

Conjunction of clocks	Clock c1, c2;
	Clock c is c1 and c2;
		Clock c1, c2;
		Clock c = c1 and c2;
			$\mathcal{D}: c_1, \mathcal{D}: c_2$
			$\mathcal{D}: c = c_1 \wedge c_2$
				3.7.5

Conjunction of event and clock	Event e;
	Clock c1
	Clock c2 is e and c1;
		Event e;
		Clock c1
		Clock c2 = e and c1;
			$\mathcal{E}: e, \mathcal{D}: c_1$
			$\mathcal{D}: c_2 = e \wedge c_1$
				3.5.9

Conjunction of clock and event	Clock c1
	Event e;
	Clock c2 is c1 and e;
		Clock c1
		Event e;
		Clock c2 = c1 and e;
			$\mathcal{D}: c_1, \mathcal{E}: e$
			$\mathcal{D}: c_2 = c_1 \wedge e$
				3.5.9

Disjunction of clocks	Clock c1, c2;
	Clock c is c1 or c2;
		Clock c1, c2;
		Clock c = c1 or c2;
			$\mathcal{D}: c_1, \mathcal{D}: c_2$
			$\mathcal{D}: c = c_1 \vee c_2$
				3.7.6

Disjunction of event and clock	Event e;
	Clock c1
	Clock c2 is e or c1;
		Event e;
		Clock c1
		Clock c2 = e or c1;
			$\mathcal{E}: e, \mathcal{D}: c_1$
			$\mathcal{D}: c_2 = e \vee c_1$
				3.5.10

Disjunction of clock and event	Clock c1
	Event e;
	Clock c2 is c1 or e;
		Clock c1
		Event e;
		Clock c2 = c1 or e;
			$\mathcal{D}: c_1, \mathcal{E}: e$
			$\mathcal{D}: c_2 = c_1 \vee e$
				3.5.10

Number of ticks	Clock c;
	Integer n is card c;
		Clock c;
		Integer n = card (c);
			$\mathcal{D}: c$
			$\mathbb{N}: n = \|c\|$
				3.6

4.11. Type Period

4.11.1.Constructors

Natural language syntax	Mathematical syntax	Semantics	Reference

Event e1, e2;
Period P is [e1, e2];
	Event e1, e2;
	Period P = [e1, e2];
		$\mathcal{E}: e_1, \mathcal{E}: e_2$
		$\mathcal{P}: P = [e_1, e_2]$
			3.8

Event e1, e2;
Period P is] e1, e2];
	Event e1, e2;
	Period P =] e1, e2];
		$\mathcal{E}: e_1, \mathcal{E}: e_2$
		$\mathcal{P}: P =]e_1, e_2]$
			3.8

Event e1, e2; Period P is [e1, e2 [;	Event e1, e2; Period P = [e1, e2 [;	$\mathcal{E}: e_1, \mathcal{E}: e_2$ $\mathcal{P}: P = [e_1, e_2[$	3.8
Event e1, e2; Period P is] e1, e2 [;	Event e1, e2; Period P =] e1, e2 [;	$\mathcal{E}: e_1, \mathcal{E}: e_2$ $\mathcal{P}: P =]e_1, e_2[$	3.8

4.11.2.Operators

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Opening event	Period P;
	Event e is P start;
		Period P;
		Event e = start (P);
			$\mathcal{P}: P$
			$\mathcal{E}: e = P \uparrow$
				3.8

Closing event	Period P;
	Event e is P end;
		Period P;
		Event e = end (P);
			$\mathcal{P}: P$
			$\mathcal{E}: e = P \downarrow$
				3.8

4.12. Type Periods

4.12.1.Constructors

Natural language syntax	Mathematical syntax	Semantics	Reference

Period P1, P2, …, Pn;
Periods P is { P1, P2, …, Pn };
	Period P1, P2, …, Pn;
	Periods P = { P1, P2, …, Pn };
		$\mathcal{P}: P_1, \mathcal{P}: P_2, \dots, \mathcal{P}: P_n$
		$2^{\mathcal{P}}: P = \{P_1, P_2, \dots, P_n\}$
			3.9

Clock c1, c2;
Periods P is [c1, c2];
	Clock c1, c2;
	Periods P = [c1, c2];
		$\mathcal{D}: c_1, \mathcal{D}: c_2$
		$2^{\mathcal{P}}: P = \Pi([c_1, c_2])$
			3.9

Clock c1, c2;
Periods P is] c1, c2];
	Clock c1, c2;
	Periods P =] c1, c2];
		$\mathcal{D}: c_1, \mathcal{D}: c_2$
		$2^{\mathcal{P}}: P = \Pi(]c_1, c_2])$
			3.9

Clock c1, c2; Periods P is [c1, c2 [;	Clock c1, c2; Periods P = [c1, c2 [;	$\mathcal{D}: c_1, \mathcal{D}: c_2$ $2^{\mathcal{P}}: P = \Pi([c_1, c_2[)$	3.9

Clock c1, c2; Periods P is] c1, c2 [;	Clock c1, c2; Periods P is] c1, c2 [;	$\mathcal{D}: c_1, \mathcal{D}: c_2$ $2^{\mathcal{P}}: P = \Pi(]c_1, c_2[)$	3.9

4.12.2.Operators

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

While	Periods P1;
	Periods P2;
	Periods P is P2 while P1;
		Periods P1;
		Periods P2;
		Periods P = while (P2, P1);
			$2^{\mathcal{P}}: P_1, 2^{\mathcal{P}}: P_2$
			$2^{\mathcal{P}}: P = P_1 \supset P_2$
				3.10.2

4.13. Type Operator

4.13.1.Constructors

Natural language syntax	Semantics	Reference
Type T, T1, T2, …, Tn; Operator [T] w1 T1 e1 w2 T2 e2 … wn Tn en = *value*;	$f: T_1 \times T_2 \times \ldots \times T_n \longrightarrow T$ $(e_1, e_2, \ldots, e_n) \longmapsto f(e_1, e_2, \ldots, e_n) = expr$ $\mathcal{C}(f) = \emptyset$	3.17 3.19

Mathematical syntax	Semantics	Reference
Type T, T1, T2, …, Tn; Operator f = new Operator [T] (T1 e1, T2 e2, … , Tn en) = *value*;	$f: T_1 \times T_2 \times \ldots \times T_n \longrightarrow T$ $(e_1, e_2, \ldots, e_n) \longmapsto f(e_1, e_2, \ldots, e_n) = expr$ $\mathcal{C}(f) = \emptyset$	3.17 3.19

Mixed natural language / mathematical syntax	Semantics	Reference
Type T, T1, T2, …, Tn; Operator f is new Operator [T] w1 T1 e1 w2 T2 e2 … wn Tn en = *value*;	$f: T_1 \times T_2 \times \ldots \times T_n \longrightarrow T$ $(e_1, e_2, \ldots, e_n) \longmapsto f(e_1, e_2, \ldots, e_n) = expr$ $\mathcal{C}(f) = \emptyset$	3.17 3.19

Operators always return a value whose type is declared between square brackets. Therefore, in expression Operator [T] w1 T1 e1 w2 T2 e2 … wn Tn en = *value*; the type of *value* is T.

Three syntaxes are possible for the declaration and definition of operators:

1. Natural language syntax: the name of the operator is declared by several words w1, w2, …, wn that can have at most one function argument before the first word, between two words, or after the last word.

   A word of the name of an operator in the natural language syntax is:

```
word: ident | quote-ident

quote-ident: ''' { quote-char } '''

quote-char:  non-digit | digit | '!' |  '&' | '|' | '+' | '-' | '*' |
             '/' | '%' | '<' | '>' | '=' | '^' | '_' | ' '
```

The declaration and definition of an operator is

```
op-expr: "Operator" '['type-ident']' ([type-ident ident] {word [type-
                          ident ident]}) '=' value

type-ident: ident
```

The call of an operator is

```
op-call: [ident] | {word [ident]}
```

ident and *value* are defined in Section 4.3.

The signature of the operator is obtained by concatenating all words and type names of the operator declaration separated by the $ sign. Therefore, the $ sign cannot be used in CRML expressions.

For an operator declaration such as Operator [T] w1 T1 e1, w2 T2 e2, …, wn Tn en, the signature is T$w1$T1$w2$T2…$wn$Tn. The function call that computes the operator is then T y = T$w1$T1$w2$T2…$wn$Tn (e1, e2, …, en).

Operator overloading is possible because operators with declarations made of identical words and different type names have different signatures. For instance, operators Operator [T] w T1 and Operator [T] w T2 have the same operator name (w), but different signatures (T$w$T1 and T$w$T2).

Example 1: function that returns a clock with the ticks of another clock c that are within a given time period P.

Operator [Clock] ticks Clock c inside Period P = c filter (tick >= P start) and (tick <= P end);

Period P is *value*;

Clock c1 is *value*;

Clock c2 is ticks c1 inside P;

The function is call is Clock c2 = Clock$ticks$Clock$inside$Period (c1, P).

Example 2: disjunction of two Booleans.

Operator [Boolean] Boolean b1 or Boolean b2 = not (not b1 and not b2);

Boolean b1 is *value*;

Boolean b2 is *value*;

Boolean b is b1 or b2;

The function call is Boolean b = Boolean$Boolean$or$Boolean (b1, b2).

Example 3: definition of the <= operator in the domain of integers from the operators < and ==

Operator [Boolean] Integer n1 '<=' Integer n2 = (n1 < n2) or (n1 == n2);

Integer n1 is *value*;

Integer n2 is *value*;

Boolean b is n1 '<=' n2;

The function call is Boolean b = Boolean$Integer$'<=' $Integer (n1, n2).

2. Mathematical syntax: the name of the operator is made of one word only and the arguments are placed in a comma separated list enclosed in parenthesis, as it is customary when using mathematical notation. The signature of the operator is obtained by concatenating the name of the operator with the names of the argument types.

   Example 4: function that returns a clock with the ticks of another clock c that are within a given time period P.

   Operator subClock = new Operator [Clock] (Clock c, Period P) = c filter (tick >= P start) and (tick <= P end);

   Period P is *value*;

   Clock c1 is *value*;

   Clock c2 = subClock (c1, P);

   The function call is Clock c2 = Clock$subClock$Clock$Period (c1, P).

3. Mixed natural language and mathematical syntax: the name of the operator is declared using the natural language syntax, and the operator is assigned to a variable that is the operator's name in the mathematical syntax. Then the operator can be called using the natural language or the mathematical syntax. The signature of the operator is the natural language syntax signature.

   Example 5: function that returns a clock with the ticks of another clock c that are within a given time period P.

   Operator f is new Operator [Clock] ticks Clock c inside Period P = c filter (tick >= P start) and (tick <= P end);

   Period P is *value*;

   Clock c1 is *value*;

   Clock c2 is ticks c1 inside P; // Natural language syntax call

   Clock c2 = f (c1, P); // Mathematical syntax call

   The function call is Clock$ticks$Clock$inside$Period (c1, P), thus corresponding to the natural language syntax.

Example 6: count function, cf. Eq. (82). Natural language notation.

Operator [Integer] count [Boolean] b inside [Period] P = card ticks new Clock b inside P;

Example 7: count function, cf. Eq. (82). Mathematical notation.

Operator count = new Operator [Integer] (Boolean] b, [Period] P) = card (subClock (new Clock b, P));

Example 8: count function, cf. Eq. (82). Mixed natural language / mathematical notation.

Operator count = Operator [Integer] count [Boolean] b inside [Period] P = card (subClock (new Clock b, P));

Operator count = Operator [Integer] count [Boolean] b inside [Period] P = card ticks new Clock b inside P;

Operator count = Operator [Integer] count [Boolean] b inside [Period] P = card (ticks new Clock b inside P);

4.14. Type Template

4.14.1.Constructors

Natural language syntax	Semantics	Reference
Template w1 e1 w2 e2 … wn en = *value*;	$f\colon\ \mathbb{B}^n \longrightarrow \mathbb{B}$ $(e_1, e_2, \dots, e_n) \longmapsto f(e_1, e_2, \dots, e_n) = expr$	3.18

Mathematical syntax	Semantics	Reference
Template f = new Template (e1, e2, … , en) = *value*;	$f\colon\ \mathbb{B}^n \longrightarrow \mathbb{B}$ $(e_1, e_2, \dots, e_n) \longmapsto f(e_1, e_2, \dots, e_n) = expr$	3.18

Mixed natural language / mathematical syntax	Semantics	Reference
Template f is new Template w1 e1 w2 e2 … wn en = *value*;	$f\colon\ \mathbb{B}^n \longrightarrow \mathbb{B}$ $(e_1, e_2, \dots, e_n) \longmapsto f(e_1, e_2, \dots, e_n) = expr$	3.18

Templates are operators on Booleans. The syntactic rules for templates are therefore the same as for operators, with the exception that types are not specified.

Example 1: disjunction of two Booleans.

Template b1 or b2 = not (not b1 and not b2);

The name of the function is or and the function is called as follows:

Boolean b1 is *value*;

Boolean b2 is *value*;

Boolean b is b1 or b2;

The signature of the function is Boolean$Boolean$or$Boolean.

4.15. Type Category

4.15.1.Constructors

Natural language syntax	Semantics	Reference

Operator f1, f2, …, fn;
Operator g1, g2, …, gn;
Category w = { (f1, g1), (f2, g2), …, (fn, gn) };
	$\mathcal{C}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2): c$
	$f_i \longmapsto g_i = c(f_i)$
		3.19

Mathematical syntax	Semantics	Reference

Operator f1, f2, …, fn;
Operator g1, g2, …, gn;
Category c = { (f1, g1), (f2, g2), …, (fn, gn) };
	$\mathcal{C}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2): c$
	$f_i \longmapsto g_i = c(f_i)$
		3.19

Mixed natural language / mathematical syntax	Semantics	Reference

Operator f1, f2, …, fn;
Operator g1, g2, …, gn;
Category c is new Category w = { (f1, g1), (f2, g2), …, (fn, gn) };
	$\mathcal{C}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2): c$
	$f_i \longmapsto g_i = c(f_i)$
		3.19

Example 1: Example 1 in Section 3.193.12.1.

Category increasing = { (>, >), (>=, >=), (==, >), (<>, >), (<, >=), (<=, >) };

4.15.2. Operators

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Association	Category c;
	Operator f;
	Category {} C is associate c with f;
		Category c;
		Operator f;
		Category {} C = associate (c, f);
			$\mathcal{C}(\mathbb{D}_1 \longrightarrow \mathbb{D}_2): c$
			$\mathbb{O}(\mathbb{D}_3 \longrightarrow \mathbb{D}_4): f$
			$c \in \mathcal{C}(f)$
			$2^{\mathcal{C}(*\longrightarrow *)}: C = \mathcal{C}(f)$
				3.19

The associate operator inserts category c into the set of categories associated with f and returns the new set of categories associated with f. The notation $2^{\mathcal{C}(*\longrightarrow *)}: C$ means that the set C contains categories of any sources and targets.

Example 1:

Operator f;

Category increasing = { (>, >), (>=, >=), (==, >), (<>, >), (<, >=), (<=, >) };

Category {} C is associate increasing with f;

The value of C is the set of categories { increasing, … } where … denotes the categories previously associated with f.

4.16. Sets

4.16.1. Constructors

Natural language syntax	Mathematical syntax	Semantics	Reference

// Non-empty typed set
Type T;
T e1, e2, …, en;
[T {} S is] { e1, e2, …, en };
	// Non-empty typed set
	Type T;
	T e1, e2, …, en;
	[T {} S =] { e1, e2, …, en };
		$T: e_1, T: e_2, \dots, T: e_n$
		$T: S = \{e_1, e_2, \dots, e_n\}$
			3.20.1

// Empty typed set
Type T;
[T {} S is] {};
	// Empty typed set
	Type T;
	[T {} S =] {};
		$T: S = \emptyset$	3.20.1

// Special set
Type T;
T S is { *expr* };
	// Special set
	Type T;
	T S = { *expr* };
		$\mathcal{C}: S$
		$\mathcal{M}: S$
		$\mathcal{L}: S$
		$\mathcal{T}: S$
			3.23
			3.24
			3.25
			3.26

There is no dedicated keyword to denote a set.

There are two kinds of sets: typed sets and special sets.

A typed set is a set such that all its elements are of the same type, which is the type of the typed set. A typed set is declared as T {} S, where T is the type of its elements.

A special set is a set that can contain elements of different types. Special sets are of the following types only: class, model, library, package. The declaration of special sets does not feature curly braces {} after the type name, e.g. model M is { *expr* } (and not model {} M is { *expr* }).

Other time-dependent sets such as Clock and Periods have special features such that they are not considered as typed sets and are handled separately.

The identifier is omitted for anonymous typed sets and can be omitted for the empty set.

4.16.2. Operators

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Union of two sets	*Type* T;
	T {} S1, S2;
	T {} S is S1 union S2;
		Type T;
		T {} S1, S2;
		T {} S = S1 union S2;
			$2^T: S_1, 2^T: S_2$
			$2^T: S = S_1 \cup S_2$
				3.20.1

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Union of an element with a set	*Type* T;
	T e1;
	T {} S2;
	T {} S is e1 union S2;
		Type T;
		T {} e1;
		T {} S2;
		T {} S = e1 union S2;
			$T\!: e_1, 2^T\!: S_2$
			$2^T\!: S = \{e_1\} \cup S_2$
				3.20.1

Union of two elements	*Type* T;
	T e1, e2;
	T {} S is e1 union e2;
		Type T;
		T e1, e2;
		T {} S is e1 union e2;
			$T\!: e_1, T\!: e_2$
			$2^T\!: S = \{e_1\} \cup \{e_2\}$
				3.20.1

Flattening of a set	*Type* T;
	T {} S1;
	T {} S2 is flatten S1;
		Type T;
		T {} S1;
		T {} S2 = flatten S1;
			$2^T\!: S_1$
			$2^T\!: S_2 = \Cup S_1$
				3.20.1

Number of elements	*Type* T;
	T {} S;
	Integer n is card S;
		Type T;
		T {} S;
		Integer n = card (S);
			$2^T\!: S$
			$\mathbb{N}\!: n = \|S\|$
				3.20.1

Filter	*Type* T;
	T {} S1;
	Operator [Boolean] cond T e = *value*;
	T {} S2 is filter S1 cond element;
		Type T;
		T {} S1;
		Operator [Boolean] cond (T e) = *value*;
		T {} S2 = filter (S1 (cond (element));
			$2^T\!: S_1, \mathbb{O}(T \longrightarrow \mathbb{B})\!: \mathrm{cond}$
			$2^T\!: S_2 = S_1(\mathrm{cond}(*))$
				3.21.3

Left unary operator	*Type* T1, T2;
	T1 {} S1;
	Operator [T2] op T1 e = *value*;
	T2 {} S2 is op S1;
		Type T1, T2;
		T1 {} S1;
		Operator [T2] op T1 e = *value*;
		T2 {} S2 = op S1;
			$2^{T_1}\!: S_1, \mathbb{O}(T_1 \longrightarrow T_2)\!: op$
			$2^{T_2}\!: S_2 = op\ S_1$
				3.21.1

Right unary operator	*Type* T1, T2;
	T1 {} S1;
	Operator [T2] T1 e op = *value*;
	T2 {} S2 is S1 op;
		Type T1, T2;
		T1 {} S1;
		Operator [T2] T1 e op = *value*;
		T2 {} S2 = S1 op;
			$2^{T_1}\!: S_1, \mathbb{O}(T_1 \longrightarrow T_2)\!: op$
			$2^{T_2}\!: S_2 = S_1\ op$
				3.21.1

Binary operator	*Type* T;
	T {} S1, S2;
	Operator [T] T e1 op T e2 = *value*;
	T e is S1 op S2;
		Type T;
		T {} S1, S2;
		Operator [T] T e1 op T e2 = *value*;
		T e = S1 op S2;
			$2^T\!: S_1, 2^T\!: S_2, \mathbb{O}(T \times T \longrightarrow T)\!: op$
			$T\!: e = S_1\ op\ S_2$
				3.21.2

All operators except left, right and binary operators, apply also to special sets.

The union operator applies to all sets, but the result of the union of two sets (the merged set) must be a legal set. For instance, the union of a library with a model should be declared as a model, and the library should be flattened if it contains sub-libraries, because models can contain all elements of a library, except libraries. The merged set provides a common namespace for the elements of both sets, so that elements from one set can be used by elements of the other set without using paths. This is useful when using operators and templates in the natural language syntax. However, one must ensure that all identifiers are unique within the merged set.

Example 1: negation of a set of Booleans.

```
Boolean R1, R2, R3, R4;
Boolean S is not { R1, R2, R3, R4 };
```

The value of S is { not R1, not R2, not R3, not R4}.

Note that { R1, R2, R3, R4 } is an anonymous set.

Example 2: frame period P applied to a set of time periods.

```
Periods P;
Periods P1, P2, P3, P4;
Periods {} S is { P1, P2, P3, P4 } while P;
```

The value of S is { P1 while P, P2 while P, P3 while P, P4 while P }.

Example 3: getting the values of same attribute for a set of objects.

```
class C is {
    Real x is external;
    Real y is external;
};
C {} S is { C O1(x is 1, y is 1), C O2(x is 2, y is 1), C O3(x is 3, y is 1) } ;
Real {} R is S.x;
```

The value of R is { 1, 2, 3 }.

Example 4: conjunction of a set of Booleans.

```
Boolean R1, R2, R3, R4;
Boolean R is and { R1, R2, R3, R4 };
```

The value of R is R1 and R2 and R3 and R4.

Example 5: inference chain of a set of requirements

```
Template b1 imply b2 = not b1 or b2;
Boolean R1, R2, R3, R4;
Boolean R is imply { R1, R2, R3, R4 };
```

The value of R is R1 imply R2 imply R3 imply R4.

Example 6: extracting requirements from a model.

```
model M is { …, R1, R2, …, R3, R4 };
```

where R1, R2, R3, R4 are of the type Requirement, and … stands for elements of other types than

Requirement.

The set of requirements of M is obtained with

Requirement {} R is filter M (type element == Requirement);

The value of R is { R1, R2, R3, R4 }.

Example 7: extracting from a set of pieces of equipment the pieces of equipment that are in operation

```
class Equipment is {
    String id; // Equipment identification
    Boolean inOperation is external; // inOperation is true if the equipment is in operation
};
Equipment {} S is filter { Equipment (id is "E1"), Equipment (id is "E2") } (element.inOperation == true);
```

Example 8: union of elements in a set

```
Requirement {} R1 is { a1, a2, { b1, b2, b3, { c1, c2 } }, { b3, b4 }, a3 };
Requirement {} R2 is union R1;
Then R2 = a1 union a2 union { b1, b2, b3, { c1, c2 } } union { b3, b4 } union a3
          = { a1, a2 } union { b1, b2, b3, { c1, c2 } } union { b3, b4 } union a3
          = { a1, a2, b1, b2, b3, { c1, c2 }, b3, b4,  a3 }
```

Example 9: flattening of a set

```
Requirement {} R1 is { a1, a2, { b1, b2, b3, { c1, c2 } }, { b3, b4 }, a3 };
Requirement {} R2 is flatten R1;
```

The value of R2 is { a1, a2, b1, b2, b3, c1, c2, b4, a3 }.

4.17. Type type

4.17.1.Constructors

Natural language syntax	Mathematical syntax	Semantics	Reference

type T2 is (T1 x is *value* (a1, a2, …, an)) {
Type D1, D2, …, Dn;
D1 a1 [is *value*];
D2 a2 [is *value*];
…
Dn an [is *value*];
};
	type T2 = (T1 x = *value* (a1, a2, …, an)) {
	Type D1, D2, …, Dn;
	D1 a1 [= *value*];
	D2 a2 [= *value*];
	…
	Dn an [= *value*];
	};
		$\mathfrak{T}: \mathbb{T}_2 = \mathbb{T}_1: x(a_1, a_2, \dots a_n) + \{\mathbb{D}_1: a_1, \mathbb{D}_2: a_2, \dots \mathbb{D}_n: a_n\}$	3.14.2
			3.14.3

4.17.2.Operators

Name	Natural language syntax	Mathematical syntax	Semantics	Reference
Alias	type T1 is *value*; String s is *value*; type T2 is T1 alias s;	type T1 = *value*; String s = *value*; type T2 = alias (T1, s);	$\mathfrak{T}: T_1, \mathbb{S}: s$ $\mathfrak{T}: T_2 = T_1(s)$	3.14.5
Partial type	partial type T is *value*;	partial type T = *value*;	partial $\mathfrak{T}: T$	3.14.4
Forbid	type T1 is *value*; type U1 is *value*; … type Un is *value*; Operator f1 is new Operator [U1] w11 T1 e1 … w1p1 T1 ep1 = *value*; … Operator fn is new Operator [Un] wn1 T1 e1 … wnpn T1 epn = *value*; type T2 is T1 forbid { f1, …, fn };	type T1 = *value*; type U1 = *value*; … type Un = *value*; Operator f1 = Operator [U1] w11 T1 e1 … w1p1 T1 ep1 = *value*; … Operator fn = Operator [Un] wn1 T1 e1 … wnpn T1 epn = *value*; type T2 = T1 forbid { f1, …, fn };	$\mathfrak{T}: T_1, \mathfrak{T}: T_2 \subset T_1$ $\mathfrak{T}: U_1, \dots, \mathfrak{T}: U_n$ $f_1: {T_1}^{p_1} \longrightarrow U_1$ … $f_n: {T_1}^{p_n} \longrightarrow U_n$ $f_1 \notin \mathbb{O}(T_2 \longrightarrow *)$ … $f_n \notin \mathbb{O}(T_2 \longrightarrow *)$	3.14.2
Type equality	type T1 is *value*; type T2 is *value*; Boolean b is T1 == T2;	type T1 is *value*; type T2 is *value*; Boolean b = (T1 == T2);	$\mathfrak{T}: T_1, \mathfrak{T}: T_2$ $\mathbb{B}: b = (T_1 = T_2)$	3.14.1

It is possible to combine operators alias, partial and forbid in the same statement by writing partial type: T is *value* alias s forbid { f1, …, fn }.

The alias s must be enclosed between square brackets when using special characters, e.g. s = [m/s2], because `'/'` is a special character.

Example 1: defining the type of physical quantities in SI units by extending the type of real numbers.

```
partial type Quantity is (Real q is rate*u + offset)  {
    String SIUnit;          // SI unit for quantity q
    String userUnit;        // User unit for quantity q
    Real u;                 // Quantity q expressed in user units
    Real rate;              // Conversion rate between user units and SI units
    Real offset;            // offset between user units and SI units
};
```

Partial indicates that the type Quantity must be extended or specialized.

When writing

Quantity x = Quantity (SIUnit = "Pa", userUnit = "bar", rate = 1.e5, offset = 0, u = 3);

the expression (Real q is rate*u + offset) in the definition of Quantity computes the value of x from the value of u: x = rate*u + offset. The value of u can be computed back from the value of x using the automatically inverted formula (x – offset) / rate (thus rate must be different from zero).

It is also possible to write

Real x = Quantity (SIUnit = "Pa", userUnit = "bar", rate = 1.e5, offset = 0, u = 3);

because all elements of type Quantity are elements of type Real as the type Quantity extends the type Real. Then the expression (Real q is rate*u + offset) can be seen as converting a quantity expressed in the specific type Quantity to the general type of Real.

Example 2: defining physical units for pressures by specializing the domain of physical quantities.

partial type Pressure is Quantity (SIUnit = "Pa") ;

Partial indicates that the type Pressure must extended or specialized.

type PressureBar is Pressure (userUnit = "bar", rate = 1.e5, offset = 0) ;

Example 3: defining physical units for pressures with an alias.

type PressurePa is Pressure (userUnit = "Pa", rate = 1, offset = 0) alias Pa;

type PressureBar is Pressure (userUnit = "bar", rate = 1.e5, offset = 0) alias bar;

Example 4: defining physical units for absolute temperatures and forbidding the use of temperature addition and multiplication.

partial type AbsoluteTemperature is Quantity (SIUnit = "K") forbid { +, * };

+ and * are operators on type Real. Therefore, they apply to all subtypes of Real. Thus, they apply to AbsoluteTemperature. Because it is not physically meaningful to add or multiply absolute temperatures, these operators must be forbidden on AbsoluteTemperature.

type AbsoluteTemperatureKelvin is AbsoluteTemperature (userUnit = "K", rate = 1, offset = 0) alias K;

type AbsoluteTemperatureCelsius is AbsoluteTemperature (userUnit = "Celsius", rate = 1, offset = 273.15) alias Celsius;

Example 5: defining the physical unit for acceleration.

type Length is Quantity (SIUnit = "m", userUnit = "m", rate = 1, offset = 0) alias K;

partial type Time is Quantity (SIUnit = "s");

type TimeSecond is Time (userUnit = "s", rate = 1, offset = 0) alias s;

type Hour is Time (userUnit = "h", rate = 3600, offset = 0) alias h;

type Acceleration is Quantity (SIUnit = "m/s2", userUnit = "m/s2", rate = 1, offset = 0) alias [m/s2];

The alias is enclosed in square brackets because the special character `'/'` is used.

Note that the quantity Acceleration can be defined even if the quantities Time and Length are not defined. To relate the three quantities together, the definition of Quantity should be extended in order to accommodate dimensional analysis. For instance, a new attribute SIDimension could be added to express the SIunit in the fundamental units of the considered unit system (SI, CGS, etc.). The extended definition of Quantity would look like

```
partial type Quantity is (Real q is rate*u + offset)  {
    String SIUnit;          // SI unit for quantity q
    String userUnit;        // User unit for quantity q
    String SIDimension; // Unit for quantity q expressed in the fundamental units
    String SIFUnits = "[m][kg][s][K]"; // Fundamental units of the chosen unit system
    Real u;                 // Quantity q expressed in user units
    Real rate;              // Conversion rate between user units and SI units
    Real offset;            // offset between user units and SI units
};
```

For Pressure, SIDimension = "[kg][m-1][s-2]".

Example 6: defining a type for requirements.

Some temporal operators on Booleans have been introduced for the evaluation of requirements only. They should not be used in requirement models. This can be enforced by forbidding these temporals operators on Booleans that represent requirements.

```
type Requirement is Boolean forbid { *, +, integrate };
```

4.18. Elements

4.18.1.Constructors

Natural language syntax	Mathematical syntax	Semantics	Reference
Type T; T x is *value*;	*Type* T; T x = *value*;	$x \in \mathbb{D}$ $T : x = expr$	3.16

Type T;
T x is external;
	Type T;
	T x = external;
		$x \in \mathbb{D}$
		$T : x$
			3.16

There are three kinds of elements: variables, sets and operators.

Type T denotes any domain.

4.18.2.Operators

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

External	*Type* T;
	T x is external;
		Type T;
		T x is external;
			$x \in \mathbb{V}$
			$*.T : x$
				3.22
				3.24.1

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Constant	*Type* T;
	constant T x;
		Type T;
		constant T x;
			$x \in \mathbb{B} \cup \mathbb{Z} \cup \mathbb{R}$
			$T: x = constant$
				3.16

Parameter	*Type* T;
	parameter T x;
		Type T;
		parameter T x;
			$x \in \mathbb{B} \cup \mathbb{Z} \cup \mathbb{R} \cup \mathbb{S}$
			$T: x = constant$
				3.16

Get type	*Type* T;
	T x is *value*;
	Type T is new type x;
		Type T;
		T x is *value*;
		Type T = new type (x);
			$x \in \mathbb{D}$
			$T: \mathrm{domain}(x)$
				3.14.1

Only variables, objects and sets can have external values.

Variables with external values cannot be assigned any value within the requirement model. Their values are provided by the binding mechanism.

To enforce the binding mechanism:

- A variable can have an external value in classes only.
- In models, only objects and sets can have external values.

Only variables that belong to the following types can be declared as constant: Boolean, Integer, Real.

A parameter is a variable constant over one simulation run. A parameter can have different values for different simulation runs. Only variables that belong to the following types can be declared as parameter: Boolean, Integer, Real, String.

Example 1: defining a variable of type Quantity of Example 1 in Section 4.17.

No variable of type Quantity can be defined because Quantity is a partial type.

Example 2: defining a variable of type PressureBar of Example 2 in Section 4.17.

PressureBar P is 3.

The value of P is 3 bar in user units.

The conversion of 3 bar to SI units is made automatically using 3 bar = rate*3 + offset = (1.e5)*3 + 0 = 3.e5 Pa.

Example 3: defining variables of type Pressure of Example 3 in Section 4.17.

Pressure P1 is 10.e5 Pa;

Pressure P2 is 3 bar;

The conversion of 3 bar to SI units is made automatically using 3 bar = rate*3 + offset = (1.e5)*3 + 0 = 3.e5 Pa.

It is possible to write

Pressure P is (P1 + P2) bar;

P1 and P2 are automatically converted to SI units, they are added, and the result is displayed in bar.

It is possible to write

Pressure P is (10.e5 Pa + 3 bar) bar;

The conversion of 3 bar to SI units is made automatically using 3 bar = rate*3 + offset = (1.e5)*3 + 0 = 3.e5 Pa. The value of P is P = 10.e5 + 3.e5 = 13.e5 in SI units (Pa), that is used for internal calculations. The result is displayed in bar.

It is possible to write

Real P is (P1 + 3 bar) Pa;

P1 and 3 bar are automatically converted to SI units, they are added, and the result is displayed in Pa.

It is not possible to write

Pressure P is 3;

thus omitting the unit, because Pressure is a partial type that cannot be instantiated.

It is not possible to write

Pressure P is 3 K;

thus specifying a wrong unit, because K is not a subtype of Pressure.

Assuming that the mass, length, and acceleration units have been defined, it is possible to write:

Real m is 10 g;

Real a is 5 [m/s2];

Real S is 2 cm2;

Real P is (m*a/S) bar;

The conversion of the different factors of m*a/S to SI units, and the conversion of the result to bar are done automatically. However, dimensional analysis is required to check the consistency of the formula unit wise.

Example 4: defining variables of type AbsoluteTemperature of Example 4 in Section 4.17.

AbsoluteTemperature T1 is 390 K;

AbsoluteTemperature T2 is 20 Celsius;

The conversion of 20 Celsius to SI units is made automatically using 20 Celsius = rate*20 + offset = 1*20 + 273.15 = 293.15 K.

It is not possible to write

AbsoluteTemperature T = (T1 + T2) K;

because operator + is forbidden on AbsoluteTemperature.

It is possible to write

AbsoluteTemperature T = (T1 – 10 Celsius) Celsius;

The conversion of 10 Celsius to SI units is made automatically using 10 Celsius = rate*10 + offset K. The result is automatically converted to Celsius using the inverted formula.

Example 5: defining external variables

Real x is external;

Pressure P is external bar;

4.19. Type class

4.19.1.Constructors

Natural language syntax	Mathematical syntax	Semantics	Reference

class C is {
Type T1, T2, …, Tn;
T1 a1 [is [*value* | external]];
T2 a2 [is [*value* | external]];
…
Tn an [is [*value* | external]];
};
	class C = {
	Type T1, T2, …, Tn;
	T1 a1 [= [*value* | external]];
	T2 a2 [= [*value* | external]];
	…
	Tn an [= [*value* | external]];
	};
		$\mathcal{C}: \hat{C} = \{T_1: a_1, T_2: a_2, \dots, T_n: a_n\}$	3.23.1
			3.23.3

4.19.2.Operators

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Class extension	class C1;
	class C2 is { *expr* } extends C1;
		class C1;
		class C2 = { *expr* } extends C1;
			$\mathcal{C}: C_2 = \mathcal{C}: C_1 + \{expr\}$	3.23.2

Multiple class extension	class C1;
	…
	class Cn;
	class C is { *expr* } extends { C1, …, Cn };
		class C1;
		…
		class Cn;
		class C = { *expr* } extends { C1, …, Cn };
			$\mathcal{C}: C_2 = \mathcal{C}: C_1 + \dots + \mathcal{C}: C_n + \{expr\}$	3.23.2

Partial class	partial class C is { *expr* };	partial class C = { *expr* };	$\mathcal{C}: \text{partial } C$	3.23.4

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Attribute re-declaration	class C1 is {
	Type T1, …, Tn;
	T1 a1 [is [*value* | external]];
	…
	Tn an [is [*value* | external]];
	};
	class C2 is {
	Type U1, …, Un;
	[redeclare a1] U1 b1 [is [*value* | external]];
	…
	[redeclare an] Un bn [is [*value* | external]];
	} extends C1;
		class C1 = {
		Type T1, …, Tn;
		T1 a1 [= [*value* | external]];
		…
		Tn an [= [*value* | external]];
		};
		class C2 = {
		Type U1, …, Un;
		[redeclare a1] U1 b1 [= [*value* | external]];
		…
		[redeclare an] Un bn [= [*value* | external]];
		} extends C1;
			$\mathcal{C}: C_1 = \{T_1: a_1, \dots T_n: a_n\}$
			$\mathcal{C}: C_2 = C_1 + \begin{matrix} \{\, U_1: a_1 \to b_1, \\ \dots, \\ U_n: a_n \to b_n \,\} \end{matrix}$
				3.23.2

It is not possible to write in the same model

class C2 is { *expr* };

class C2 extends C1;

because class C2 extends C1 has no value.

Redeclaration of an attribute 'an' of a class C1 can be done in a class C2 that extends C1. The redeclare operator has two arguments: the name of the redeclared attribute 'an', and the new declaration of 'an'. The name and/or the type of 'an' can be changed in the new declaration. Writing 'redeclare an Un bn is external' changes the declaration of 'an' in C1 to 'Un bn is external' in C2.

Example 1: class Equipment

```
partial class Equipment is {
    String id; // Equipment identification
    Boolean inOperation is external; // Indicates whether the equipment is in operation or not
};
```

Example 2: class Pump that extends class Equipment.

```
class Pump is {
    Real c is external; // Pump characteristic
    Real efficiency is external; // Pump efficiency
    Boolean cav is external; // Indicates whether the pump cavitates or not
    /* Requirement that states that the pump must not cavitate while in operation */
    Boolean nocav = during inOperation ensure not cav;
} extends Equipment;
```

Example 3: class CoolingSystem that extends class Equipment and uses class Pump.

```
class CoolingSystem is {
    /* The design of the cooling system features 3 pumps */
    Pump P1;
    Pump P2;
    Pump P3;
} extends Equipment;
```

Example 4: class CentrifugalPump that extends class Pump with one redeclaration.

```
class CentrifugalPump is {
    /* The name of the attribute c is changed to fc that means "full characteristics" */
    redeclare c Real fc is external;
} extends Pump;
```

Example 5: class CoolingSystemRefined that extends class CoolingSystem to use class CentrifugalPump instead of class Pump.

```
class CoolingSystemRefined is {
    /* Pumps are redeclared to be centrifugal pumps without changing their names */
    redeclare P1 CentrifugalPump P1;
    redeclare P2 CentrifugalPump P2;
    redeclare P3 CentrifugalPump P3;
} extends CoolingSystem;
```

Example 6: class CoolingSystemRefined that extends class CoolingSystem to use class CentrifugalPump instead of class Pump.

```
class CoolingSystemRefined is {
    /* Pumps are redeclared to be centrifugal pumps with new names */
    redeclare P1 CentrifugalPump CentrifugalP1;
    redeclare P2 CentrifugalPump CentrifugalP2;
    redeclare P3 CentrifugalPump CentrifugalP3;
} extends CoolingSystem;
```

4.20. Objects

4.20.1.Constructors

Natural language syntax	Mathematical syntax	Semantics	Reference

class C is {
Type T1, T2, …, Tn;
T1 a1 [is [*value* | external]];
T2 a2 [is [*value* | external]];
…
Tn an [is [*value* | external]];
};
C o is new C ([a1 is *value*,] [a2 is *value*,] …[, an is *value*]);
	class C = {
	Type T1, T2, …, Tn;
	T1 a1 [= [*value* | external]];
	T2 a2 [= [*value* | external]];
	…
	Tn an [= [*value* | external]];
	};
	C o = new C ([a1 = *value*,] [a2 = *value*,] …[, an = *value*]);
		$\mathcal{C}_C = \{T_1 : a_1, T_2 : a_2, \ldots, T_n : a_n\}$
		$\mathcal{C}_C : o = (a_1 = expr, a_2 = expr, \ldots, a_n = expr)$
			3.22

An object o is instantiated from a class C by providing values to all attributes which are not declared as external or that are not defined (i.e., have no value) in the class definition. It is also possible to override the value of an attribute defined in the class definition, even if the value is external.

Writing 'C o is new C (…)' seems redundant as class C is written twice. However, following the general rule to separate declaration from definition, C o declares object o as being an instance of class C, and new C (…) creates and defines (i.e., provides a value to) object o. An alternative would be to give the possibility to merge declaration and definition in a single statement such as 'C (…) o' or 'C o (…)' as an exception to the general rule.

Example 1: instantiation of class Equipment in Example 1 of Section 4.17.

Class Equipment cannot be instantiated because it is a partial class.

Example 2: instantiation of class Pump in Example 2 of Section 4.17.

Pump pump1 is new Pump (id = "Pump1");

'id' is the only attribute that does not have a definition in class Pump. However, it is possible to override the value of attribute nocav with a new requirement:

Pump pump2 is new Pump (id = "Pump2", nocav = …);

The other attributes are external and are not required to be given a value at class instantiation.

Example 3: instantiation of class CoolingSystem in Example 3 of Section 4.17.

CoolingSystem coolingSystem is new CoolingSystem (id = "CoolingSystem", P1 = pump1, P2 = pump2, P3 = Pump (id = "Pump3", nocav = …));

The definitions of pump1 and pump2 of Example 2 are used in the definition of coolingSystem.

Example 4: instantiation of class CentrifugalPump in Example 4 of Section 4.17.

CentrifugalPump centrifugaPump1 is new CentrifugalPump (id = “CentrifugalPump1”);

Example 5: instantiation of class CoolingSystemRefined in Example 5 of Section 4.17.

CoolingSystemRefined coolingSystemRefined is new CoolingSystemRefined (id = "CoolingSystem", P1 = centrifugaPump1, P2 = CentrifugalPump (id = "CentrifugalPump2"), P3 = CentrifugalPump (id = "CentrifugalPump3", nocav = …));

The definition of centrifugaPump1 of Example 4 is used in the definition of CoolingSystemRefined.

Example 6: instantiation of class CoolingSystemRefined in Example 6 of Section 4.17.

CoolingSystemRefined coolingSystemRefined is new CoolingSystemRefined (id = "CoolingSystem", CentrifugalP1 = centrifugaPump1, CentrifugalP2 = CentrifugalPump (id = "CentrifugalPump2"), CentrifugalP3 = CentrifugalPump (id = "CentrifugalPump3", nocav = …));

The definition of centrifugaPump1 of Example 4 is used in the definition of CoolingSystemRefined.

4.20.2.Operators

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Get value	class C is {
	Type T1, …, Tn;
	T1 a1 [is [*value* | external]];
	…
	Tn an [is [*value* | external]];
	};
	C o ([a1 is *value*,] … [, an is *value*]);
	Ti vi is o.ai;
		class C = {
		Type T1, …, Tn;
		T1 a1 [= [*value* | external]];
		…
		Tn an [= [*value* | external]];
		};
		C o ([a1 = *value*,] … [, an = *value*]);
		Ti vi = o.ai;
			$\mathcal{C}_C = \{T_1: a_1, \dots, T_n: a_n\}$
			$\mathcal{C}_C: o = (a_1 = v_1,$
			$\dots$
			$a_i = v_i,$
			$\dots$
			$a_n = v_n)$
			$T_i: v_i = o.a_i$
				3.22
				3.20.2

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Set value	class C is {
	Type T1, …, Tn;
	T1 a1 [is [*value* | external]];
	…
	Tn an [is [*value* | external]];
	};
	C o ([a1 is *value*,] … [, an is *value*]);
	Ti vi;
	o.ai is vi;
		class C = {
		Type T1, …, Tn;
		T1 a1 [= [*value* | external]];
		…
		Tn an [= [*value* | external]];
		};
		C o ([a1 = *value*,] … [, an = *value*]);
		Ti vi;
		o.ai = vi;
			$\mathcal{C}_C = \{T_1: a_1, \dots, T_n: a_n\}$
			$\mathcal{C}_C: o = (a_1 = v_1,$
			$\dots$
			$a_i = v_i,$
			$\dots$
			$a_n = v_n)$
			$o.a_i = T_i: v_i$
				3.22
				3.20.2

The value ‘vi’ of attribute ‘ai’ of object ‘o’ is given by ‘vi = o.ai’ where ‘o’ denotes the path of object ‘o’.

The value ‘vi’ is set to attribute ‘ai’ of object ‘o’ by writing ‘o.ai = vi’ where ‘o’ denotes the path of object ‘o’. When using this function, no value should have been set to ‘o.ai’ when instantiating object o from class C.

Therefore, it is possible to write:

C o is new C;

o.ai = vi;

But it is not possible to write

C o is new C (ai = vi);

o.ai = vi;

because o.ai is then defined twice. Neither is it possible to write

C o is new C;

alone, because o.ai is then not defined, unless o.ai is defined in the class definition:

class C is { …; Ti ai is vi; }

4.21. Type model

4.21.1.Constructors

Natural language syntax	Mathematical syntax	Semantics	Reference

model M is {
model M' is *value*;
type T is *value*;
class C is *value*;
Operator [T] w1 T1 e1 w2 T2 e2 … wn Tn en = *value*;
Category c = *value*;
C o is *value*;
C {} S is *value*;
T x [is [*value* | external]];
};
	model M = {
	model M' = *value*;
	type T = *value*;
	class C = *value*;
	Operator [T] w1 T1 e1 w2 T2 e2 … wn Tn en = *value*;
	Category c = *value*;
	C o = *value*;
	C {} S = *value*;
	T x [= [*value* | external]];
	};
		$\mathcal{M}: M = \{T_1: a_1, \dots, T_n: a_n\}$	3.24.1

Although it is possible to define external variables in a model, it is recommended to define external variables in classes only in order to benefit from the automatic binding of external variables via objects.

4.21.2.Operators

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Model extension	model M1;
	model M2 is { *expr* } extends M1;
		model M1;
		model M2 = { *expr* } extends M1;
			$\mathcal{M}: M_2 = \mathcal{M}: M_1 + \{expr\}$	3.24.2

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Attribute re-declaration	model M1 is {
	Type T1, …, Tn;
	T1 a1 [is [*value* | external]];
	…
	Tn an [is [*value* | external]];
	};
	model M2 is {
	Type U1, …, Un;
	[redeclare a1] U1 b1 [is [*value* | external]];
	…
	[redeclare an] Un bn [is [*value* | external]];
	} extends M1;
		model M1 = {
		Type T1, …, Tn;
		T1 a1 [= [*value* | external]];
		…
		Tn an [= [*value* | external]];
		};
		model M2 = {
		Type U1, …, Un;
		[redeclare a1] U1 b1 [= [*value* | external]];
		…
		[redeclare an] Un bn [= [*value* | external]];
		} extends M1;
			$\mathcal{M}: M_1 = \{T_1: a_1, \dots T_n: a_n\}$
			$\mathcal{M}: M_2 = M_1 + \{ U_1: a_1 \to b_1, \dots, U_n: a_n \to b_n \}$
				3.24.2

While	Periods P is *value*; model M is { *expr* } while P;	Periods P is *value*; model M = { *expr* } while P;	$2^{\mathcal{P}}: P, \mathcal{M}: M$ $\sqsupset M = P$	3.24.1

Get value	model M is {
	Type Ti;
	model M' is {
	Ti ai' [is [*value* | external]];
	};
	Ti ai is M'.ai';
	};
		model M = {
		Type Ti;
		model M' is {
		Ti ai' [= [*value* | external]];
		};
		Ti ai = M.ai';
		};
			$\mathcal{M}: M' = \{\dots, T_i: a_i', \dots\}$
			$\mathcal{M}: M = \{ \mathcal{M}: M', \dots, T_i: a_i = M'.a_i', \dots\}$
				3.22
				3.20.2

The extend operator can be combined with the while operator:

model M2 is { *expr* } extends M1 while P;

model M2 is { *expr* } while P extends M1;

The value vi' of variable ai' of model M' is given by vi' = M'.ai' where M' denotes the path of model M'.

Example 1: a simple cooling system. This model expresses requirements on the pumps of the system only.

model CoolingSystem is {

    /* Class Equipment is an abstract class for all pieces of equipment in the cooling system to provide an identifier and the state of the piece of equipment */

```
partial class Equipment is {
    String id; // Unique identifier
    Boolean s is external; // State of the equipment, started or not started
};
```

/* Type Requirement is aimed at providing a dedicated keyword to requirements and forbid the use of temporal operators by the user. */

```
type Requirement is Boolean forbid { *, +, integrate };
```

/* Define type Quantity to handle physical units */

```
partial type Quantity is (Real q is rate*u + offset)  {
    String SIUnit;              // SI unit for quantity q
    String userUnit;            // User unit for quantity q
    String SIDimension;         // Unit for quantity q expressed in the fundamental units
    String SIFUnits = "[m][kg][s][K]"; // Fundamental units of the chosen unit system
    Real u;                     // Quantity q expressed in user units
    Real rate;                  // Conversion rate between user units and SI units
    Real offset;                // Offset between user units and SI units
};
```

/* Define unit RotationalVelocity */

```
constant Real pi is 3.141592;

partial type RotationalVelocity is Quantity (SIUnit = "rad/s", SIDimension = "[s-1]");
type RotationalVelocityRPM is RotationalVelocity (userUnit = "rpm", rate = 2*pi/60, offset = 0) alias
rpm;
```

/* Define operator "during". This operator opens a time period when a Boolean b becomes true and closes the time period when b becomes false. */

```
Operator [ Periods ] during Boolean b = Periods [ new Clock b, new Clock not b ];
```

/* It is assumed that the operator check that tells whether a requirement composed of a condition cond and a time period P is satisfied or not is defined. This operator is fully defined in the ETL library. */

```
// Operator [ Boolean ] Periods check cond over P = value;
```

/* Class pump represents the way pumps are seen from the perspective of requirement modelling for the cooling system. The attributes omega and omega_n are given for illustration purposes only as they are not used in the model. */

```
class Pump is {
    RotationalVelocity omega is external rpm;     // Rotational velocity
```

```
        RotationalVelocity omega_n is 1400 rpm;          // Nominal rotational velocity
        Boolean cav is external;                   // Indicates whether the pump cavitates or not
        /* Requirement that states that the pump should not cavitate while it is in operation */
        Requirement nocav is during s ensure not cav;
    } extends Equipment;

    /* Class CoolingSystem represents a preliminary design of the cooling system that features only
    pumps that should not cavitate. */
    class CoolingSystem is {
        Pump {} pumps; // Set of pumps of the cooling system
        /* Requirement that states that no pump should cavitate within the cooling system */
        Requirement nocav is and pumps.nocav;
    } extends Equipment;

    /* The cooling system is represented by one instance of class CoolingSystem that features 3
    pumps. The instantiation of class CoolingSystem automatically creates 3 instances of class Pump,
    that in turn create one no-cavitation requirement for each pump. */
    CoolingSystem coolingSystem is new
                    CoolingSystem (pumps is { Pump (id is "P1"), Pump (id is "P2"), Pump (id is "P3") });
};
```

Example 2: refinement of the simple cooling system. This model expresses additional requirements on the pumps of the system following design choices regarding the pumps.

```
model RefinedCoolingSystem is {
    /* Define unit day */
    partial type Time is Quantity (SIUnit = "s", SIDimension= "[s]");
    type TimeDay is Time (userUnit = "d", rate = 3600*24, offset = 0) alias d;

    /* Define operator "becomes true". This operator raises an event each time the Boolean b becomes
    true. The set of generated events is a clock. */
    Operator [ Clock ] Boolean b becomes true = new Clock b;

    /* Define operator "count inside". This operator counts the number of ticks of clock C inside a single
    time period P. */
    Operator [ Clock ] ticks Clock c inside Period P = c filter (tick >= P start) and (tick <= P end);
    Operator [ Integer ] count [ Boolean ] b inside [ Period ] P = card ticks new Clock b inside P;

    /* The class RefinedPump extends the class Pump defined in model CoolingSystem to include the
    additional requirement due to pump technology limitations: "The pump must not be started more
    than twice in a sliding time period of one day". */
    class RefinedPump is {
        /* Sliding time period of 1 day. It is a multiple time period that is composed of possibly
        overlapping single time periods that start each time the pump is started, and end 1 day (24
        hours) later. */

        Periods oneDay is [ s becomes true, s becomes true + 1 d [;

        /* No-start requirement to be satisfied */

        Requirement nostart is check (count s inside oneDay) <= 2 over oneDay;

    } extends Pump;


    /* Extend the class CoolingSystem to redeclare the set of pumps as being instances of the new
    class RefinedPump */

    class RefinedCoolingSystem is {

        /* The attribute pumps in CoolingSystem is redeclared to refinedpumps */

        redeclare pumps RefinedPump {} refinedpumps;

    } extends CoolingSystem;


    /* Redeclare the object coolingSystem to be an instance of the new class RefinedCoolingSystem.
    In the new design, only two pumps are considered with different ids. */

    redeclare coolingSystem RefinedCoolingSystem refinedCoolingSystem is (refinedpumps is
    { RefinedPump (id is "PO1"), RefinedPump (id is "PO2") });


    /* Then in model RefinedCoolingSystem, the additional requirements on the pumps are
    automatically generated */

} extends CoolingSystem;
```

Example 3: define a model that provides a global oracle for the cooling system model by taking the conjunction of all requirements defined in the cooling system model.

```
model GlobalCoolingSystem is {

    /* A model being a set, the set of all requirements in the model can be obtained by filtering the
    elements of the model by type Requirement. The set must be flattened to use the 'and' operator,
    because binary operators are not defined on sets when two sets are involved, i.e. S1 and S2. It is
    possible to flatten the set before or after filtering. */

    Requirement global is and (flatten (filter CoolingSystem (type element == Requirement)));

};
```

Example 4: architectural modelling of the cooling system

The cooling system architecture is represented by the class diagram in Fig. 44.

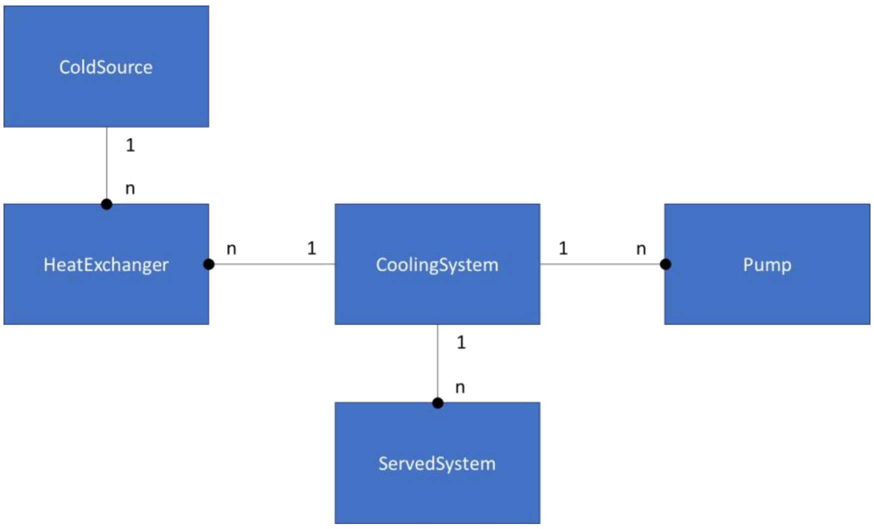

Fig. 44. Class diagram of the cooling system architecture

```
model CoolingSystemArchitecture is {
    type Requirement is Boolean forbid { *, +, integrate };
    type Assumption is Requirement;

    /* Define class Equipment */
    partial class Equipment is {
        String id;                          // Equipment identification
        Boolean inOperation is external;  // State in operation or not of the equipment
    };

    /* Define the abs function */
    Operator [ Real ] abs [ Real ] x = if x >= 0 then x else –x;

    /* Define class CoolingSystem */
    class CoolingSystem is {
        // Cooling system has 3 pumps
        Pump {} pumps is { Pump (id = "PO1"),  Pump (id = "PO2"), Pump (id = "PO3") };
        // Cooling system has 2 heat exchangers
        HeatExchanger  {}  heatExchangers  is  {  HeatExchanger  (id  =  "HE1"),  HeatExchanger  (id  =
        "HE1")};
        // Cooling system serves 2 systems
        ServedSystem {} servedSystems is { ServedSystem (id = "SE1"), ServedSystem (id = "SE2")};
        // Total mass flow rate through the pumps
        MassFlowRate m_flow_pump is + pumps.m_flow;
        // Total mass flow rate through the heat exchangers
        MassFlowRate m_flow_heatExchanger is + heatExchangers.m_flow;
        // Ensure that the heat exchangers are connected to the pumping system
        Requirement r_flow_1 is during inOperation ensure abs (m_flow_heatExchanger –
        m_flow_pump) < eps;
        // Ensure that the mass flow rate though the cooling system is quasi-constant around 800 kg/s
        Requirement r_flow_2 is during inOperation ensure abs (m_flow_pump – 800 [kg/s]) < 20 [kg/s];
    } extends Equipment;

    /* Instantiation of the unique cooling system */
    coolingSystem is new CoolingSystem;

    /* Define class Pump */
    class Pump is {
        /* Cooling system the pump belongs to */
        CoolingSystem coolingSystem is CoolingSystem.coolingSystem;
        Boolean cav is external;            // Indicates whether the pump cavitates or not
        Temperature temp is external;       // Pump temperature
        MassFlowRate m_flow is external;          // Mass flow rate through the pump
        Requirement r_noCav is during inOperation ensure not cav;  // No cavitation requirement
        Requirement r_temp is during inOperation ensure temp >= 5 Celsius and temp <= 40 Celsius;
    } extends Equipment;

    /* Define class ColdSource */
    class ColdSource is {
        /* Cooling system the cold source belongs to */
        CoolingSystem coolingSystem is CoolingSystem.coolingSystem;
        /* Heat exchangers cooled by the cold source */
        HeatExchanger {} heatExchangers is CoolingSystem.heatExchangers;
        Temperature temp is external;             // Cold source temperature
        /* Cold source temperature is assumed to be in the range [5 Celsius, 35 Celsius] while the plant
        is in operation */
        Assumption a_temp is during coolingSystem.inOperation ensure temp >= 5 Celsius and temp
        <= 35 Celsius;
    };
```

```
    /* Define class HeatEchanger */
    class HeatExchanger is {
        /* Cooling system the heat exchanger belongs to */
        CoolingSystem coolingSystem is CoolingSystem.coolingSystem;
        /* Cold source that cools the heat exchanger */
        ColdSource coldSource;
        MassFlowRate m_flow is external;          // Mass flow rate through the pump
    } extends Equipment;

     /* Define class ServedSystem */
    class ServedSystem is {
        /* Cooling system the served system is attached to */
        CoolingSystem coolingSystem is CoolingSystem.coolingSystem;
     };
};
```

4.22. Type library

4.22.1.Constructors

Natural language syntax	Mathematical syntax	Semantics	Reference

library L is {
library L' is *value*;
type T is *value*;
class C is *value*;
Operator [T] w1 T1 e1 w2 T2 e2 … wn Tn en = *value*;
Category c = *value*;
};
	library L = {
	library L' = *value*;
	type T = *value*;
	class C = *value*;
	Operator [T] w1 T1 e1 w2 T2 e2 … wn Tn en = *value*;
	Category c = *value*;
	};
		$\mathcal{L}: L = \{T_1: a_1, \dots, T_n: a_n\}$	3.25

Example 1: library that contains the definitions of types and classes of Example 1 in Section 4.21.

```
library CoolingSystemLib is {
    /* Class Equipment is an abstract class for all equipment in the cooling system to provide an
    identifier and the state of the equipment */
    partial class Equipment is {
        String id; // Unique identifier
        Boolean s is external; // State of the equipment, started or not started
```

```
};

/* Type Requirement is aimed at providing a dedicated keyword to requirements and forbid the use
of temporal operators by the user. */

type Requirement is Boolean forbid { *, +, integrate };

/* Define type Quantity to handle physical units */

partial type Quantity is (Real q is rate*u + offset)  {
    String SIUnit;               // SI unit for quantity q
    String userUnit;             // User unit for quantity q
    String SIDimension;          // Unit for quantity q expressed in the fundamental units
    String SIFUnits = "[m][kg][s][K]"; // Fundamental units of the chosen unit system
    Real u;                      // Quantity q expressed in user units
    Real rate;                   // Conversion rate between user units and SI units
    Real offset;                 // offset between user units and SI units
};

/* Define unit RotationalVelocity */

constant Real pi is 3.141592;

partial type RotationalVelocity is Quantity (SIUnit = "rad/s", SIDimension= "[s-1]");

type RotationalVelocityRPM is RotationalVelocity (userUnit = "rpm", rate = 2*pi/60, offset = 0) alias
rpm;

/* Define operator "during". This operator opens a time period when a Boolean b becomes true,
and closes the time period when b becomes false. */

Operator [ Periods ] during Boolean b = Periods [ new Clock b, new Clock not b ];

/* It is assumed that the operator check that tells whether a requirement composed of a condition
cond and a time period P is satisfied or not is defined. This operator is fully defined in the ETL
library. */

// Operator [ Boolean ] Periods check cond over P = value;

/* Class pump represents the way pumps are seen from the perspective of requirement modelling
for the cooling system. The attributes omega and omega_n are given for illustration purposes only
as they are not used in the model. */

class Pump is {
    RotationalVelocity Real omega is external rpm;        // Rotational velocity
    RotationalVelocity omega_n is 1400 rpm;          // Nominal rotational velocity
    Boolean cav is external;                    // Indicates whether the pump cavitates or not
    /* Requirement that states that the pump should not cavitate while it is in operation */
```

```
        Requirement nocav is during s ensure not cav;
    } extends Equipment;

    /* Class CoolingSystem represents a preliminary design of the cooling system that features only
    pumps that should not cavitate. */
    class CoolingSystem is {
        Pump {} pumps; // Set of pumps of the cooling system
        /* Requirement that states that no pump should cavitate within the cooling system */
        Requirement nocav is and pumps.nocav;
};
```

Then model CoolingSystem can be rewritten using library CoolingSystemLib.

```
model CoolingSystem is {
    /* The cooling system is represented by one instance of class CoolingSystemLib.CoolingSystem
    that features 3 pumps. The instantiation of class CoolingSystemLib.CoolingSystem automatically
    creates 3 instances of class CoolingSystemLib.Pump, that in turn create one no-cavitation
    requirement for each pump that must be satisfied. */
    CoolingSystem coolingSystem is new
                    CoolingSystemLib.CoolingSystem (pumps is { CoolingSystemLib.Pump (id is "P1"),
                    CoolingSystemLib.Pump (id is "P2"), CoolingSystemLib.Pump (id is "P3") });
};
```

In order to avoid using the path CoolingSystemLib. for every CoolingSystemLib definition inside model CoolingSystem, it is possible to insert library CoolingSystemLib into model CoolingSystem so that the elements of CoolingSystemLib are within the namespace of CoolingSystem.

```
model CoolingSystem is CoolingSystemLib union {
    /* The cooling system is represented by one instance of class CoolingSystemLib.CoolingSystem
    that features 3 pumps. The instantiation of class CoolingSystemLib.CoolingSystem automatically
    creates 3 instances of class CoolingSystemLib.Pump, that in turn create one no-cavitation
    requirement for each pump that must be satisfied. */
    CoolingSystem coolingSystem is new
                   CoolingSystem (pumps is { Pump (id is "P1"), Pump (id is "P2"), Pump (id is "P3") });
};
```

Alternatively, it is possible to write

```
model CoolingSystem is {
    /* The cooling system is represented by one instance of class CoolingSystemLib.CoolingSystem
    that features 3 pumps. The instantiation of class CoolingSystemLib.CoolingSystem automatically
    creates 3 instances of class CoolingSystemLib.Pump, that in turn create one no-cavitation
    requirement for each pump that must be satisfied. */
    CoolingSystem coolingSystem is new
                   CoolingSystem (pumps is { Pump (id is "P1"), Pump (id is "P2"), Pump (id is "P3") });
} union CoolingSystemLib;
```

4.23. Type package

4.23.1. Constructors

Natural language syntax	Mathematical syntax	Semantics	Reference

package P is {
package P’ is *value*;
library L is *value*;
model M is *value*;
};
	package P = {
	package P’ = *value*;
	library L = *value*;
	model M = *value*;
	};
		$\mathcal{T}: P = \{T_1: a_1, \dots, T_n: a_n\}$	3.26

Example 1: a package for the cooling system that contains the CoolingSystemLib library and the CoolingSystem model

```
package CoolingSystem is {
    library CoolingSystemLib is {
        /* Class Equipment is an abstract class for all equipment in the cooling system to provide an
        identifier and the state of the equipment */
        partial class Equipment is {
            String id; // Unique identifier
            Boolean s is external; // State of the equipment, started or not started
        };

        /* Type Requirement is aimed at providing a dedicated keyword to requirements and forbid the
        use of temporal operators by the user. */
        type Requirement is Boolean forbid { *, +, integrate };

        /* Define type Quantity to handle physical units */
        partial type Quantity is (Real q is rate*u + offset)  {
            String SIUnit;            // SI unit for quantity q
            String userUnit;          // User unit for quantity q
            String SIDimension;       // Unit for quantity q expressed in the fundamental units
            String SIFUnits = "[m][kg][s][K]"; // Fundamental units of the chosen unit system
            Real u;                   // Quantity q expressed in user units
            Real rate;                // Conversion rate between user units and SI units
            Real offset;              // offset between user units and SI units
        };

        /* Define unit RotationalVelocity */
        constant Real pi is 3.141592;
        partial type RotationalVelocity is Quantity (SIUnit = "rad/s", SIDimension= "[s-1]");

        type RotationalVelocityRPM is RotationalVelocity (userUnit = "rpm", rate = 2*pi/60, offset = 0)
        alias rpm;


        /* Define operator "during". This operator opens a time period when a Boolean b becomes true,
        and closes the time period when b becomes false. */

        Operator [ Periods ] during Boolean b = Periods [ new Clock b, new Clock not b ];


        /* It is assumed that the operator check that tells whether a requirement composed of a
        condition cond and a time period P is satisfied or not is defined. This operator is fully defined in
        the ETL library. */

        // Operator [ Boolean ] Periods check cond over P = value;


    /* Class pump represents the way pumps are seen from the perspective of requirement modelling
    for the cooling system. The attributes omega and omega_n are given for illustration purposes only
    as they are not used in the model. */

        class Pump is {

            RotationalVelocity omega is is external rpm;     // Rotational velocity

            RotationalVelocity omega_n is 1400 rpm;          // Nominal rotational velocity

            is external Boolean cav is external;     // Indicates whether the pump cavitates or not

            /* Requirement that states that the pump should not cavitate while it is in operation */

            Requirement nocav is during s ensure not cav;

        } extends Equipment;


        /* Class CoolingSystem represents a preliminary design of the cooling system that features only
        pumps that should not cavitate. */

        class CoolingSystem is {

            Pump {} pumps; // Set of pumps of the cooling system

            /* Requirement that states that no pump should cavitate within the cooling system */

            Requirement nocav is and pumps.nocav;

        };

    };


    model CoolingSystem is CoolingSystemLib union {

        /* The cooling system is represented by one instance of class CoolingSystemLib.CoolingSystem
        that features 3 pumps. The instantiation of class CoolingSystemLib.CoolingSystem
        automatically creates 3 instances of class CoolingSystemLib.Pump, that in turn create one no-
        cavitation requirement for each pump that must be satisfied. */

        CoolingSystem coolingSystem is new
                    CoolingSystem (pumps is { Pump (id is “P1”), Pump (id is “P2”), Pump (id is “P3”) });

    };

};
```

4.24. Type Probability

4.24.1.Constructors

Natural language syntax	Mathematical syntax	Semantics	Reference

Boolean b;
Probability x is new Probability b;
	Boolean b;
	Probability x = new Probability (b);
		$\mathbb{R}: t, \mathbb{O}(\Omega \longrightarrow \mathbb{B}_2): b$
		$\mathbb{O}(\mathbb{B}_2 \longrightarrow \mathbb{R}): x = \mathbb{P}(b = true \mid t)$
			3.13

Boolean b;
Clock c;
Probability x is new Probability b at c;
	Boolean b;
	Clock c;
	Probability x = new Probability (b, c);
		$\mathcal{D}: c, \mathbb{O}(\Omega \longrightarrow \mathbb{B}_2): b$
		$\mathbb{O}(\mathbb{B}_2 \longrightarrow \mathbb{R}): x = \mathbb{P}(b = true \mid c^i)$
			3.13

new Probability b computes the probability that b is true at all time instants t.

new Probability b at c computes the probability that b is true at all ticks of clock c.

4.24.2.Operators

Name	Natural language syntax	Mathematical syntax	Semantics	Reference

Probability estimator	Probability x;
	Real y is estimator x;
		Probability x;
		Real y = estimator x;
			$\mathbb{O}(\mathbb{B}_2 \longrightarrow \mathbb{R}): x$
			$\mathbb{R}: y = \mathbb{E}[x]$
				3.13

Probability variance	Probability x;
	Real y is estimator variance x;
		Probability x;
		Real y = estimator variance x;
			$\mathbb{O}(\mathbb{B}_2 \longrightarrow \mathbb{R}): x$
			$\mathbb{R}: y = \mathbb{V}\mathrm{ar}[x]$
				3.13

Example 1: While the system is in operation, there should not be more than two pump failures in a sliding time period of one month with a probability greater than 99%.

/* Type Requirement is aimed at providing a dedicated keyword to requirements and forbid the use of temporal operators by the user. */

type Requirement is Boolean forbid { *, +, integrate };

/* Type Events synonymous to Clock */

type Events is Clock;

/* Operator that defines time periods while a Boolean is true, including the opening events and excluding the closing events */

Operator [Periods] during Boolean b excluding closing events = [b becomes true, b becomes false [;

/* Operator that defines a sliding time period deltaT over a given time period P for occurring events e */

operator [Periods] during Time deltaT sliding while Periods P for Events e = [e or P start, (e or P start) + deltaT [while P;

```
/* Class pump that contains the requirements on pumps. The operators used in this example are
  defined in the ETL and FORM-L libraries. */
class Pump is {
    /* External variable that tells whether the system is in operation */
    Boolean inOperation is external;
    /* External variable that tells whether the pump is in a failure state */
    Boolean failure is external;
    /* No-start non-probabilistic requirement. It is not declared as a requirement because it is not the
      final requirement to be satisfied.  */
    Boolean nostart is (during inOperation excluding closing events) ensure ((during 1 month sliding
    while inOperation for Events failure) check count Events failure <= 2);
    /* Probability that requirement no-start is true at end of the system operation */
    Real p is estimator new Probability noStart at inOperation becomes false;
    /* Probabilistic no-start requirement. It is declared as a requirement because it is the final
    requirement to be satisfied. */
    Requirement noStartProb is during inOperation check at end p > 0.99;
};
```

5. ETL library

5.1. Operators on Booleans

Logical disjunction

$$\mathbb{B}: b_1, \mathbb{B}: b_2$$

$$\mathbb{B}: b_1 \lor b_2 = \neg(\neg b_1 \land \neg b_2)$$

Template b1 or b2 = not (not b1 and not b2);

Exclusive logical disjunction

$$\mathbb{B}: b_1, \mathbb{B}: b_2$$

$$\mathbb{B}: b_1 \oplus b_2 = (b_1 \lor b_2) \land \neg(b_1 \land b_2)$$

Template b1 xor b2 = (b1 or b2) and not (b1 and b2);

Logical inference

$$\mathbb{B}: b_1, \mathbb{B}: b_2$$

$$\mathbb{B}: b_1 \Rightarrow b_2 = \neg b_1 \lor b_2$$

Template b1 implies b2 = not b1 or b2;

5.2. Operators on clocks

Filter clock ticks inside a time period

$$\mathcal{D}: C, \mathcal{P}: P$$

$$\mathcal{D}: inside(C, P) = C(\mathrm{rank}(*) \geq P \uparrow \land \mathrm{rank}(*) \leq P \downarrow)$$

Operator [Clock] Clock C inside Period P = C filter (tick >= P start) and (tick <= P end);

Count the occurrences of events inside a time period

$$\mathcal{D}: C, \mathcal{P}: P$$

$$\mathbb{N}: count.inside(C, P) = |inside(C, P)|$$

Operator [Integer] count Clock C inside Period P = card (C inside P);

5.3. Operators on events

Events generated when a Boolean becomes true

$$\mathbb{B}: b, \mathcal{D}: c$$

$$\mathcal{D}: becomes.true\ (b) = c(b)$$

Operator [Clock] Boolean b becomes true = new Clock b;

Events generated when a Boolean becomes false

$$\mathbb{B}: b, \mathcal{D}: c$$

$$\mathcal{D}: becomes.false\ (b) = c(\neg b)$$

Operator [Clock] Boolean b becomes false = new Clock not b;

<u>Events generated when a Boolean becomes true inside a time period</u>

$$\mathbb{B}: b, \mathcal{P}: P$$

$$\mathcal{D}: becomes.true.inside\ (b, P) = inside\ (becomes.true\ (b), P)$$

Operator [Clock] Boolean b becomes true inside Period P = (b becomes true) inside P;

<u>Events generated when a Boolean becomes false inside a time period</u>

$$\mathbb{B}: b, \mathcal{P}: P$$

$$\mathcal{D}: becomes.false.inside\ (b, P) = inside\ (becomes.false\ (b), P)$$

Operator [Clock] Boolean b becomes false inside Period P = (b becomes false) inside P;

5.4. Operators for the evaluation of requirements

<u>Check</u>

$$\mathbb{B}: \varphi, 2^{\mathcal{P}}: P = \{P_i\}_{1 \le i \le |P|}$$

$$\mathbb{B}: check.over(\varphi, P) = \varphi \otimes P\ := \bigwedge_{i=1}^{|P|} \varphi \otimes \{P_i\}$$

Operator [Boolean] check Boolean phi over Periods P = and (evaluate phi over P);

<u>Evaluate</u>

$$\mathcal{P}: P, \mathbb{B}: \varphi$$

$$\mathbb{B}: evaluate.over(\varphi, P) = \varphi \otimes P\ := \int_P a(\varphi, P) \times \varphi$$

Operator [Boolean] evaluate Boolean phi over Period P = integrate ((decide phi over P) * phi) on P;

<u>Decide</u>

$$\mathcal{P}: P, \mathbb{B}: \varphi$$

$$\mathbb{B}: decide.over(\varphi, P) = a(\varphi, P) \coloneqq\ \varphi \vee P\downarrow$$

Operator decide is Operator [Boolean] decide Boolean phi over Period P = phi or (P end));

$$\mathcal{C}(\mathbb{N}^2 \longrightarrow \mathbb{B}): \mathrm{increasing1}, \mathcal{C}(\mathbb{R}^2 \longrightarrow \mathbb{B}): \mathrm{increasing2}, \mathcal{C}(\mathbb{B} \longrightarrow \mathbb{B}): \mathrm{varying1}, \mathcal{C}(\mathbb{B} \longrightarrow \mathbb{B}): \mathrm{varying2}$$

$$\{\ \mathrm{increasing1}, \mathrm{increasing2}, \mathrm{varying1}, \mathrm{varying2}\ \} \subset \mathcal{C}\big(decide \cdot over(\varphi, P)\big)$$

Category c1 is Category increasing1 = { (>, >), (>=, >=), (<, >=), (<=, >), (==, >), (<>, >) };

Category {} C1 is associate increasing1 with decide;

Category c2 is Category increasing2 = { (>, >), (>=, >=), (<, >=), (<=, >) };

Category {} C2 is associate increasing2 with decide;

Operator [Boolean] id Boolean b = b;

```
Operator [ Boolean ]  cte_false  Boolean b = false;
Operator [ Boolean ]  cte_true  Boolean b = true;

Category c3 is Category varying1 = { (id, cte_false) };
Category {} C3 is associate varying1 with decide;

Category c4 is Category varying2 = { (id, cte_true) };
Category {} C4 is associate varying2 with decide;
```

6. FORM-L library

6.1. Time periods

From events occur

$$\mathcal{D}: E$$

$$2^{\mathcal{P}}: from\ (E) = [E, \emptyset]$$

Operator [Periods] from Clock E = Periods [E, new Clock false];

After events occur

$$\mathcal{D}: E$$

$$2^{\mathcal{P}}: after\ (E) =]E, \emptyset]$$

Operator [Periods] after Clock E = Periods] E, new Clock false];

Before events occur

$$\mathcal{D}: E$$

$$2^{\mathcal{P}}: before\ (E) = [\emptyset, E[$$

Operator [Periods] before Clock E = Periods [new Clock false, E [;

Until events occur

$$\mathcal{D}: E$$

$$2^{\mathcal{P}}: until\ (E) = [\emptyset, E]$$

Operator [Periods] until Clock E = Periods [new Clock false, E];

While a Boolean is true

$$\mathbb{B}: b, \mathcal{D}: c$$

$$2^{\mathcal{P}}: during\ (b) = [\mathrm{c(b)}, c(\neg b)]$$

Operator [Periods] during Boolean b = Periods [new Clock b, new Clock not b];

After events occur and before events occur

$$\mathcal{D}: E_1, \mathcal{D}: E_2$$

$$2^{\mathcal{P}}: after.before\ (E_1, E_2) =]E_1, E_2[$$

Operator [Periods] after Clock E1 before Clock E2 = Periods] E1, E2 [;

After events occur and until events occur

$$\mathcal{D}: E_1, \mathcal{D}: E_2$$

$$2^{\mathcal{P}}: after.until\ (E_1, E_2) =]E_1, E_2]$$

Operator [Periods] after Clock E1 until Clock E2 = Periods] E1, E2];

After events occur and for an elapsed time

$$\mathcal{D}: E, \mathbb{R}: d$$

$$2^{\mathcal{P}}: after.for\ (E, d) =]E, E + d]$$

Operator [Periods] after Clock E for Real d = Periods] E, E + d];

After events occur and within an elapsed time

$$\mathcal{D}: E, \mathbb{R}: d$$

$$2^{\mathcal{P}}: after.within\ (E, d) =]E, E + d[$$

Operator [Periods] after Clock E within Real d = Periods] E, E + d [;

From events occur and before events occur

$$\mathcal{D}: E_1, \mathcal{D}: E_2$$

$$2^{\mathcal{P}}: from.before\ (E_1, E_2) = [E_1, E_2[$$

Operator [Periods] from Clock E1 before Clock E2 = Periods [E1, E2 [;

From events occur and until events occur

$$\mathcal{D}: E_1, \mathcal{D}: E_2$$

$$2^{\mathcal{P}}: from.until\ (E_1, E_2) = [E_1, E_2]$$

Operator [Periods] from Clock E1 until Clock E2 = Periods [E1, E2];

From events occur and for an elapsed time

$$\mathcal{D}: E, \mathbb{R}: d$$

$$2^{\mathcal{P}}: from.for\ (E, d) = [E, E + d]$$

Operator [Periods] from Clock E for Real d = Periods [E, E + d];

From events occur and within an elapsed time

$$\mathcal{D}: E, \mathbb{R}: d$$

$$2^{\mathcal{P}}: from.within\ (E, d) = [E, E + d[$$

Operator [Periods] from Clock E within Real d = Periods [E, E + d [;

When events occur

$$\mathcal{D}: E$$

$$2^{\mathcal{P}}: when\ (E) = [E, E]$$

Operator [Periods] when Clock E = Periods [E, E];

6.2. Requirements

Checking that a requirement is satisfied at the end of a time period

$$2^{\mathcal{P}}: P, \mathbb{B}: b, \mathcal{C}(\mathbb{B} \longrightarrow \mathbb{B}): \mathrm{varying1}$$

$$\mathbb{B}: check.at.end\ (P, b) = check.over\ (\mathrm{varying1}\ (b), P)$$

Operator [Boolean] Periods P check at end Boolean b = check varying1 id b over P;

<u>Checking that a requirement is satisfied at any time instant of a time period</u>

$$2^{\mathcal{P}}: P, \mathbb{B}: b, \mathcal{C}(\mathbb{B} \longrightarrow \mathbb{B}): \text{varying2}$$

$$\mathbb{B}: check.anytime\ (P, b) = check.over\ (\text{varying2}\ (b), P)$$

Operator [Boolean] Periods P check anytime Boolean b = check varying2 id b over P;

<u>Ensuring that a requirement is satisfied all along a time period</u>

$$2^{\mathcal{P}}: P, \mathbb{B}: b$$

$$\mathbb{B}: ensure\ (P, b) = check.over\ (count.over(becomes.true\ (b), P) = 0, P) \wedge check.anytime\ (b, P)$$

Operator [Boolean] Periods P ensure Boolean b = (check (count (b becomes true) == 0) over P) and (P check anytime b);

<u>Checking that the number of event occurrences at the end of a time period is lower or higher than a threshold</u>

$$2^{\mathcal{P}}: P, \mathcal{D}: E, \mathbb{N}: n, \mathcal{C}(\mathbb{N}^2 \longrightarrow \mathbb{B}): \text{increasing1}$$

$$\mathbb{B}: check.count\ (P, E, n) = check.over\ (count.inside\ (E, P)\ \text{increasing1}\ (<)\ n, P)$$

Operator [Boolean] Periods P check count Clock E '<' constant Integer n = check ((count E inside P) increasing1 < n) over P;

$$2^{\mathcal{P}}: P, \mathcal{D}: E, \mathbb{N}: n, \mathcal{C}(\mathbb{N}^2 \longrightarrow \mathbb{B}): \text{increasing1}$$

$$\mathbb{B}: check.count\ (P, E, n) = check.over\ (count.inside\ (E, P)\ \text{increasing1}\ (\leq)\ n, P)$$

Operator [Boolean] Periods P check count Clock E '<=' constant Integer n = check ((count E inside P) increasing1 <= n) over P;

$$2^{\mathcal{P}}: P, \mathcal{D}: E, \mathbb{N}: n, \mathcal{C}(\mathbb{N}^2 \longrightarrow \mathbb{B}): \text{increasing1}$$

$$\mathbb{B}: check.count\ (P, E, n) = check.over\ (count.inside\ (E, P)\ \text{increasing1}\ (>)\ n, P)$$

Operator [Boolean] Periods P check count Clock E '>' constant Integer n = check ((count E inside P) increasing1 > n) over P;

$$2^{\mathcal{P}}: P, \mathcal{D}: E, \mathbb{N}: n, \mathcal{C}(\mathbb{N}^2 \longrightarrow \mathbb{B}): \text{increasing1}$$

$$\mathbb{B}: check.count\ (P, E, n) = check.over\ (count.inside\ (E, P)\ \text{increasing1}\ (\geq)\ n, P)$$

Operator [Boolean] Periods P check count Clock E '>=' constant Integer n = check ((count E inside P) increasing1 >= n) over P;

$$2^{\mathcal{P}}: P, \mathcal{D}: E, \mathbb{N}: n, \mathcal{C}(\mathbb{N}^2 \longrightarrow \mathbb{B}): \text{increasing1}$$

$$\mathbb{B}: check.count\ (P, E, n) = check.over\ (count.inside\ (E, P)\ \text{increasing1}\ (=)\ n, P)$$

Operator [Boolean] Periods P check count Clock E '==' constant Integer n = check ((count E inside P) increasing1 == n) over P;

$$2^{\mathcal{P}}: P, \mathcal{D}: E, \mathbb{N}: n, \mathcal{C}(\mathbb{N}^2 \longrightarrow \mathbb{B}): \text{increasing1}$$

$$\mathbb{B}: check.count\ (P, E, n) = check.over\ (count.inside\ (E, P)\ \text{increasing1}\ (\neq)\ n, P)$$

Operator [Boolean] Periods P check count Clock E '<>' constant Integer n = check ((count E inside P) increasing1 <> n) over P;

<u>Checking that the duration of a condition at the end of a time period is lower or higher than a threshold</u>

$$2^{\mathcal{P}}: P, \mathbb{B}: b, \mathbb{R}: d, \mathcal{C}(\mathbb{R}^2 \longrightarrow \mathbb{B}): \text{increasing2}$$

$$\mathbb{B}: check.duration\ (P, b, d) = check.over\ (duration\ (b, P)\ \text{increasing2}\ (<)\ d, P)$$

Operator [Boolean] Periods P check duration Boolean b '<' constant Real d = check ((duration b on P) increasing2 < d) over P;

$$2^{\mathcal{P}}: P, \mathbb{B}: b, \mathbb{R}: d, \mathcal{C}(\mathbb{R}^2 \longrightarrow \mathbb{B}): \text{increasing2}$$

$$\mathbb{B}: check.duration\ (P, b, d) = check.over\ (duration\ (b, P)\ \text{increasing2}\ (\leq)\ d, P)$$

Operator [Boolean] Periods P check duration Boolean b '<=' constant Real d = check ((duration b on P) increasing2 <= d) over P;

$$2^{\mathcal{P}}: P, \mathbb{B}: b, \mathbb{R}: d, \mathcal{C}(\mathbb{R}^2 \longrightarrow \mathbb{B}): \text{increasing2}$$

$$\mathbb{B}: check.duration\ (P, b, d) = check.over\ (duration\ (b, P)\ \text{increasing2}\ (>)\ d, P)$$

Operator [Boolean] Periods P check duration Boolean b '>' constant Real d = check ((duration b on P) increasing2 > d) over P;

$$2^{\mathcal{P}}: P, \mathbb{B}: b, \mathbb{R}: d, \mathcal{C}(\mathbb{R}^2 \longrightarrow \mathbb{B}): \text{increasing2}$$

$$\mathbb{B}: check.duration\ (P, b, d) = check.over\ (duration\ (b, P)\ \text{increasing2}\ (\geq)\ d, P)$$

Operator [Boolean] Periods P check duration Boolean b '>=' constant Real d = check ((duration b on P) increasing2 >= d) over P;

7. Bibliography

88Solutions Corporation et al. (2021). Kernel Modeling Language (KerML). Object Management Group, Inc. (OMG).

88Solutions Corporation et al. (2021). OMG Systems Modeling Language TM. Object Management Group, Inc. (OMG).

Azzouzi, E. (2021). Multi-Faceted Modelling of Multi-Energy Systems : Stakeholders Coordination. *PhD Thesis*. Université Paris-Saclay.

Azzouzi, E., Bouskela, D., Jardin, A., Mhenni, F., & Choley, J.-Y. (2022). A New Contract Formalism for the Coordination of Large Systems of Systems. *J. of Advanced Engineering Informatics*. Elsevier. Under submission.

Azzouzi, E., Jardin, A., Bouskela, D., Mhenni, F., & Choley, J.-Y. (2019). Towards a Rigourous Approach to Coordinate Stakeholders of a Multi-Energy Cyber-physical System. *CPI 2019: Advances in Integrated Design and Production*.

Azzouzi, E., Jardin, J., Bouskela, D., Mhenni, F., & Choley, J.-Y. (2019). A survey on systems engineering methodologies for large multi-energycyber-physical systems. *2019 IEEE International Systems Conference (SysCon), 2019, pp. 1-8.*

Bouissou, M., & Buffoni, L. (2020). Generic method to transform a Modelica simulation model into a dynamic reliability model. *22e Congrès de Maîtrise des Risques et Sûreté de Fonctionnement λµ22*.

Bouquerel, M., Kremers, E., van der Kamp, J., Nguyen, T., & Jardin, A. (2019). Requirements modelling to help decision makers to efficiently renovate energy systems of urban districts. *Proceedings of the 2019 Summer Simulation Conference. Society for Computer Simulation International, 2019, p. 44*.

Bouskela, D., & Jardin, J. (2018). ETL: A New Temporal Language for the Verification of Cyber-Physical Systems. *2018 Annual IEEE International Systems Conference (SysCon), 2018, pp. 1-8*.

Bouskela, D., Nguyen, T., & Jardin, A. (2017). Toward a rigorous approach for verifying cyber-physical systems against requirements. *Canadian Journal of Electrical and Computer Engineering, 2017*.

Chechik, M., Devereux, B., Easterbrook, S., & Gurfinkel, A. (2004). Multi-valued symbolic model-checking. *ACM Transactions on Software Engineering and Methodology*.

Darimont , R., & van Lamsweerde, A. (1996). Formal refinement patterns for goal-driven requirements elaboration. *1996 ACM 0-89791-797-9/96/0010*. SIGSOFT'96 CA, USA.

Fuxman, A., Liu, L., Mylopoulos, J., Pistore, M., Roveri, M., & Traverso, P. (2004). Specifying and Analyzing Early Requirements in Tropos. *Requirements Engineering 9(2):132-150*.

Garro, A., Tundis, A., Bouskela, D., Jardin, A., Nguyen, T., Otter, M., . . . Olsson, H. (2016). On formal cyber physical system properties modeling: A new temporal logic language and a Modelica-based solution. *2016 IEEE International Symposium on Systems Engineering (ISSE)*.

Kanso, B., & Taha, S. (2012). Temporal Constraint Support for OCL. *SLE2012, Sep 2012, Dresden, Germany. pp.83-103. hal-00762150*.

Li, F.-L., Horkoff, J., Borgida, A., Guizzardi, G., Liu, L., & Mylopoulos, J. (2015). From stakeholder requirements to formal specifications through refinement. *REFSQ 2015: Requirements Engineering: Foundation for Software Quality pp 164–180*.

Nguyen, T. (2019). Formal requirements and constraints modelling in FORM-L for the engineering of complex socio-technical systems. *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW), Sep. 2019, pp. 123–132*.

Nguyen, T. (2021). The BASAALT (Behaviour Analysis & Simulation All Along Life Time) systems engineering method, and the FORM-L Language (Formal Requirements Modelling Language).

EDF R&D Technical Report n° 6125-3113-2019-03969-EN.

Nguyen, T., & Bouissou, M. (2020). Early Integration of Dependability Studies in the Design of Cyber-Physical Systems -. *Proceedings of the 30th European Safety and Reliability Conference*. ESREL, Research Publishing, Singapore.

Ponsard, C., & Devroey, X. (2011). Generating High-Level Event-B System Models from KAOS Requirements Models. *INFORSID: Actes du XXIXème Congrès INFORSID. pp. 317-332*.

Santos, C. A., Saleh, A. H., Schrijvers, T., & Nicolai, M. (2019). CONDEnSe: Contract-Based Design Synthesis. *2019 ACM/IIIEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE.

Stachtiari, E., Mavridou, A., Katsaros, P., Bliudze, S., & Sifakis, J. (2018). Early validation of system requirements and design through correctness-by-construction. *Journal of Systems and Software*. Elsevier.

Walden, D., Roedler, G., Forsberg, K., Hamelin, D., & Shortell, T. (s.d.). INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities, 4th Edition. INCOSE, ISBN: 978-1-118-99940-0.

Whittle, J., Bencomo, N., Cheng, B., & Bruel, J.-M. (2010). Language to Address Uncertainty in Self-Adaptive Systems Requirements. *Requirements Engineering*.

Zhang, H., Dufour, F., Dutuit, Y., & Gonzalez, K. (2008). Piecewise deterministic Markov processes. *Proc. IMechE Vol. 222 Part O: J. Risk and Reliability*.