Specification of the goal contracts specification language

D_6.3.2

<table>
<thead>
<tr>
<th>Deliverable Information</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Nature</strong></td>
</tr>
<tr>
<td><strong>Project</strong></td>
</tr>
<tr>
<td><strong>Deliverable ID</strong></td>
</tr>
<tr>
<td><strong>Status</strong></td>
</tr>
<tr>
<td><strong>Contact Person</strong></td>
</tr>
<tr>
<td><strong>Phone</strong></td>
</tr>
</tbody>
</table>
## AUTHORS TABLE

<table>
<thead>
<tr>
<th>Name</th>
<th>Company</th>
<th>E-Mail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leonardo Mangeruca</td>
<td>ALES</td>
<td><a href="mailto:leonardo.mangeruca@ales.eu.com">leonardo.mangeruca@ales.eu.com</a></td>
</tr>
<tr>
<td>Sibylle Froeschle</td>
<td>OFFIS</td>
<td><a href="mailto:sibylle.froeschle@offis.de">sibylle.froeschle@offis.de</a></td>
</tr>
<tr>
<td>Christoph Etzien</td>
<td>OFFIS</td>
<td><a href="mailto:etzien@offis.de">etzien@offis.de</a></td>
</tr>
<tr>
<td>Tayfun Gezgin</td>
<td>OFFIS</td>
<td><a href="mailto:gezgin@offis.de">gezgin@offis.de</a></td>
</tr>
<tr>
<td>Axel Legay</td>
<td>INRIA</td>
<td><a href="mailto:axel.legay@inria.fr">axel.legay@inria.fr</a></td>
</tr>
<tr>
<td>Benoît Boyer</td>
<td>INRIA</td>
<td><a href="mailto:benoit.boyer@inria.fr">benoit.boyer@inria.fr</a></td>
</tr>
<tr>
<td>Alexandre Arnold</td>
<td>EADS</td>
<td><a href="mailto:alexandre.arnold@eads.net">alexandre.arnold@eads.net</a></td>
</tr>
<tr>
<td>Michael Masin</td>
<td>IBM</td>
<td><a href="mailto:michaelm@il.ibm.com">michaelm@il.ibm.com</a></td>
</tr>
<tr>
<td>Evgeny Shindin</td>
<td>IBM</td>
<td><a href="mailto:evgensh@il.ibm.com">evgensh@il.ibm.com</a></td>
</tr>
</tbody>
</table>

## CHANGE HISTORY

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Reason for Change</th>
<th>Pages Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CONTENT

1 INTRODUCTION .......................................................................................................................... 5
  1.1 OVERVIEW ............................................................................................................................. 5
  1.2 DOCUMENT STRUCTURE ......................................................................................................... 5

2 DESCRIPTION OF SEMANTICS ................................................................................................. 6
  2.1 DETAILED BASIC UNDERLYING SEMANTICS .................................................................... 6
  2.2 B-LTL SYNTAX AND SEMANTICS ......................................................................................... 7
  2.3 CONTRACT IN FO-LTL ........................................................................................................... 9
    2.3.1 Adding Stochastic Information in a Transition System .................................................... 9
    2.3.2 Contracts for Stochastic Transition Systems .................................................................... 9

3 GCSL LANGUAGE DESCRIPTION ............................................................................................ 10
  3.1 EXAMPLE OF PATTERNS ....................................................................................................... 10
  3.2 MAPPING OF PATTERNS TO B-LTL .................................................................................... 16
    3.2.1 Speeds Pattern translation ............................................................................................... 17
    3.2.2 OCL expression ................................................................................................................ 18
    3.2.3 Illustration of the full translation of a GCSL formula ....................................................... 20
  3.3 SYNTAX BY DESCRIPTION OF PATTERNS ......................................................................... 21
  3.4 SYNTAX OF A GCSL CONTRACT ......................................................................................... 22

4 GCSL EXTENSION FOR OPTIMIZATION ................................................................................. 24
  4.1 MOTIVATION .......................................................................................................................... 24
  4.2 LANGUAGE EXTENSION ........................................................................................................ 26
  4.3 EXAMPLES ............................................................................................................................. 28

5 CONCLUSIONS AND NEXT STEPS ....................................................................................... 29

6 ABBREVIATIONS AND DEFINITIONS ..................................................................................... 30

7 BIBLIOGRAPHY ........................................................................................................................ 31
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Language structure</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Satisfaction $\Psi$ during an execution path</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>Classification by property framework extension for analysis and optimization</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>Example of cost function specification in GCSL</td>
<td>28</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Overview

The semantics of the GCSL language and its relationship with the structure of the GCSL language syntax is shown in Figure 1: Language structure. The semantics is structured in two levels. The underlying basic semantics rests on the concept of timed first order transition system, which mathematically defines the basic notions of states, timed state transitions and system runs. The B-LTL (Kamide, 2012) semantics is a simpler and more abstract semantics that introduces basic temporal operators to reason on invariants over bounded runs. B-LTL is derived from the classical LTL language (Vardi, 1996). The OCL unfolding is a procedure to convert expressions involving OCL collections into expressions over the TFOSTS semantics.

The GCSL user language is structured into four layers. The user can refer to the state of the SoS through OCL basic expressions over component attributes and OCL expressions involving collections. Expressions over OCL collections allow expressing conditions over the structure of the SoS. The two upper layers of the language specify the behavioral patterns, which include the temporal operators, and the contracts, defined as pairs comprised of an assumption and a promise.

1.2 Document structure

This document is organized as follows. The semantics of the language is defined in Section 2. Here the basic semantics and the B-LTL abstraction are presented in separate sections. In Section 3 the language layers are presented driven by the behavioral patterns, which are inherited and adapted from the SPEEDS IP European Project (http://www.speeds.eu.com).

The presented patterns are suitable for SysML-based contract and goal specifications. In Appendix 0 some speculative patterns are proposed to consider for wider scope UPDM-based contract and goal specifications, where different types of architectural views may be considered.
2 Description of Semantics

2.1 Detailed Basic Underlying Semantics

Let $\tau: \mathbb{N} \rightarrow \mathbb{R}$ be a monotonically increasing time sequence, i.e. for all $i \geq 0$: $\tau_i \leq \tau_{i+1}$ and let $\Sigma$ be an alphabet. A (infinite) sequence $\sigma_0 \sigma_1 \sigma_2 ...$ of symbols of $\Sigma$ is called a trace. A timed trace $(\sigma, \tau) = (\sigma_0, \tau_0, \sigma_1, \tau_1, \sigma_2, \tau_2)$ where $\sigma$ is a trace and $\tau$ is a time sequence is called a timed trace.

A signature defines the relevant objects and relations of the considered problem domain, e.g. a set of natural numbers and their relations between those numbers. Formally, let $\text{SIG} = (S, F, P)$ be a signature with $S$ a set of sorts or types, $F$ a set of function symbols, and $P$ a set of predicate symbols, both typed over $S$. Let $L_{\text{FO}L(\text{SIG})}$ denote the standard first-order language over $\text{SIG}$. Further, a temporal signature is given by $T\text{SIG} = (\text{SIG}, R, A)$, where $R$ is a set of so called flexible predicates, and $A$ a set of action predicates, both typed over $S$.

An interpretation of the symbols of a given signature is given by a signature. Formally, let $\mathcal{S}$ be a structure for $\text{SIG}$ which defines the domain for each $s \in S$, written by $|S|_s$. Let further be $|S| = \bigcup_{s \in S} |S|_s$ called the universe of $\mathcal{S}$. The structure further interprets the predicates and functions of the signature, i.e. $f^\mathcal{S}: |S|^n \rightarrow |S|$ for $f \in F$ and $p^\mathcal{S} \subseteq |S|^n$.

A timed first-order (labeled) state transition system ($T\text{FO}LTS^1$) $\tau = (S, W, w_0, T, X, L)$ over temporal signature $T\text{SIG} = (\text{SIG}, R, A)$ is given by a structure $\mathcal{S}$ over $T\text{SIG}$, a set $W$ of system states, an initial state $w_0 \in W$, a relation $T \subseteq W \times A \times \phi(X) \times 2^X \times W$, called (labeled) transition relation, where $A$ is an action predicated, $\phi(X)$ the set of clock constraints defined over the set of clock variables $X$, $2^X$ is the set of clocks which are reset. Note, that a clock constraint $\phi$ is defined by the grammar $\phi ::= x_i \sim c \mid x_i \sim x_j \sim c \mid \phi_1 \land \phi_2$ where $\sim \in \{\leq, <, =, >\}$, $c \in \mathbb{N}$, $x_i, x_j \in X$. The function $L$ associates with each state $w_i \in W$ and every flexible predicate $q_{(x_1,...,x_n)} \in R$ a mapping $q^\mathcal{S}(w_i) : |S|_{x_1} \times ... \times |S|_{x_n} \rightarrow \{f, tt\}$.

A labeled execution sequence (run) $r$ of $\tau$ over a timed trace $(\sigma, \tau)$, where $\sigma_i \in A$ is an infinite sequence of states, action predicates, and time instances $r = (w_0, t_0) \xrightarrow{a_1, t_1} (w_1, t_1) \xrightarrow{a_2, t_2} ...$ with $w_i \in W$, $t_i: X \rightarrow \mathbb{R}$, $t_i(x) = 0$ for all $x \in X$. For this sequence it holds that for all $i$ there exists an edge $(w_{i-1}, a_i, \phi_i, y_i, w_i) \in T_\tau$ such that $t_{i-1} + t_i - t_{i-1}$ satisfies $\phi_i$ and $t_i(x)$ equals $u_{i-1}(x) + t_i - t_{i-1}$ for clocks $x$ not in $y_i$, else equals to zero. Note that $i + c$ is a short version for $i(x) + c$ for all $x \in X$.

The set of traces for which there exists an execution sequence of $\tau$ is called the language of $\tau$.

Let $F_{\text{non}}$ be the set of non-flexible atomic formulas (i.e. formulas over $F, P$) and $F_{\text{flex}}$ be the set of flexible atomic formulas (i.e. formulas over $R, A$).

A variable valuation $\xi: Y \rightarrow |S|$ induces the mapping $S^\xi: T \rightarrow |S|$ where $T$ is the set of terms, and $S^\xi: F_{\text{non}} \rightarrow \{tt, ff\}$:

- $S^\xi(\tau) = \xi(\tau)$ for $\tau \in Y$
- $S^\xi(f(t_1, ..., t_n)) = f^S(S^\xi(t_1), ..., S^\xi(t_n))$ for $f \in F$
- $S^\xi(p(t_1, ..., t_n)) = p^S(S^\xi(t_1), ..., S^\xi(t_n))$ for $p \in F$
- $S^\xi(t_1 = t_2) = tt$ iff $S^\xi(t_1) = S^\xi(t_2)$.

Let $\tau$ be a $T\text{FO}LTS$ and $r = (w_0, t_0) \xrightarrow{a_1, t_1} (w_1, t_1) \xrightarrow{a_2, t_2} ...$ be a run of $\tau$ over a timed trace $k = (a_1, t_1)(a_2, t_2) ...$. Then $S^\xi(w_0): F_{\text{flex}} \rightarrow \{tt, ff\}$ is defined as follows:

---

The timed trace $k$ satisfies formula $F$, in short $k \models F$ if $k, \xi, 0 \models F$ for every variable valuation $\xi$. The $\text{TFOSTS} \tau$ satisfies $F$, if $k \models F$ for every $k$ in $\tau$.

An atomic formula in FOLTL is one of the following:

- An atomic formula of $L_{\text{POL(SIG)}}$ (as defined above),
- $q(t_1, \ldots, t_n)$, where $q(s_1, \ldots, s_n) \in R$ is a flexible predicate symbol and each $t_i$ is a term of sort $s_i$ for $1 \leq i \leq n$, or
- $\text{exec } a(t_1, \ldots, t_n)$, where $q(s_1, \ldots, s_n) \in A$ is an action predicate symbol.

The combination of standard LTL operators and timing annotations allows specifying that a timed trace shall satisfy the following:

- **Textual requirement:** “If “incident” for “district d” holds and “fire department fd” is in relation “responsible to” with “district d” then “fire department fd” is in relation “handle district” with “district d” within “1.5 minutes”.
- **Formal requirement:**

\[
\text{incident}(d) \land \text{responsible}(fd, d) \Rightarrow \phi \text{ executes handleDistrict(fd, d)}
\]

with $\text{incident} \in M_{\text{district}}, \text{responsible} \in M_{\text{firedepartment,district}} \in R, \text{handleDistrict} \in M_{\text{firedepartment,district}} \in A$ and $\text{firedepartment, district} \in \text{Sorts}$. The “handle district” action summarizes the activities required to send a fire brigade to the incident location.

### 2.2 B-LTL Syntax and Semantics

The formulas are defined using B-LTL (Bounded LTL), the input language of PLASMA-DESYRE where atomic propositions are state predicates or path predicates. A state predicate is a proposition parameterized by a state. Its interpretation depends on its argument: thus a state predicate $\phi$ can be seen as a function $P : S \rightarrow \{\text{true}, \text{false}\}$ such that $P$ holds for $s$ is equivalent to $P(s) = \text{true}$. Moreover, we only consider all state predicate $P$ which is provided with a procedure $\text{decide}_{P}$ to decide for all state $s$ if $P(s)$ holds or not. Similarly, we state the same assumptions for the run predicates that we want use in the formulas. The B-LTL syntax is very close to the FO-LTL one except that each temporal operator is bounded with a time interval:

\[
\phi, \; \phi_1, \; \phi_2 := \quad F_j \phi
\]
The time bounds give the length of the run on which the nested formula must hold. For the atomic propositions any decidable property for the states or the runs can be used in the formulas. It only requires that each property be provided with means (external procedure) to decide whether it holds or not for a given input (state or run). More generally, considering a run of a transition system and a B-LTL property, \( \pi \models \phi \) means that the run \( \pi \) satisfies the property \( \phi \). The semantics of temporal modalities is the semantics of the corresponding LTL modality restricted to a time interval.

### 2.2.1 B-LTL with probabilistic operator:

We also extend the logic to consider some probabilistic quantification over the paths in a B-LTL formula. We add to the logic the operator \( Pr_{2k} \phi \) where \( \phi \) is a B-LTL formula and \( k \in [0,1] \). This operator has to be considered as a special state_property that means:

\[
Pr_{2k} \phi \equiv \frac{\int_{0}^{k} f(\pi)}{\int_{0}^{1} f(\pi)} \geq k \text{ where } f(\pi) \text{ returns } 1 \text{ if } \pi \models \phi \text{ and } 0 \text{ otherwise, and all } \pi \text{ starts form }
\]

### 2.2.2 B-LTL For Adaptive systems:

An adaptive system is a system that can be reconfigured in a finite number of reconfigurations. Each configuration is defined as a particular TFOSTS. The reconfiguration of the adaptive system, i.e., when the adaptive system switches from a configuration to another one is by \( \forall a \) set of transitions of the form \( \tau_i \xrightarrow{A,G} \tau_{i+1} \) where \( G \) is the guard and \( A \) is the action. \( A \) and \( G \) are both defined in the TFOSTS constraint style. Such a transition can be fired from a state \( w \) of \( \tau_i \) to the state \( w' \) of \( \tau_{i+1} \) if \( G(w) \) and \( A(w,w') \) hold. Thus, a run of an adaptive system is a sequence of the form:

\[
\Pi = (\tau_0, w_0^0) \rightarrow (\tau_0, w_0^1) \rightarrow (\tau_0, w_0^2) \rightarrow (\tau_1, w_1^0) \rightarrow (\tau_1, w_1^1) \rightarrow (\tau_2, w_2^0) \rightarrow \ldots
\]

Where the sequence \( (\tau_i, w_i^0) \rightarrow (\tau_i, w_i^1) \) denotes the execution of the adaptive system in the configuration \( \tau_i \) and the reconfiguration are done at \( w_i^0 \) and at \( w_i^1 \).

We introduce a new operator to specify properties over adaptive systems. We note the ternary operator \( \dashv \). An adaptive property is defined as a formula \( \phi_1 \dashv \Omega \phi_2 \) where \( \phi_1 \) are B-LTL formulas and \( \Omega \) is a binary state relation. This formula holds for the adaptive path \( \Pi \) if and only if when the two relations \( w_0^0, \ldots, w_0^1 \) hold, then \( w_1^0, \ldots, w_1^1 \) hold too. Informally, the meaning of \( \phi_1 \dashv \Omega \phi_2 \) when the adaptive system satisfies \( \phi_1 \) and it is reconfigured in a way satisfying \( \Omega \) then the system must also satisfy \( \phi_2 \). Moreover the operator can be combined linearly in order to describe properties about
more complex properties for the sequences of adaptive systems: \( \phi_1 \overline{\Omega} \phi_2 \overline{\Omega}^t \phi_3 \). This formula is satisfied by \( \Pi \) iff
\[
\begin{align*}
w_0^0, \ldots, w_n^0 &\models \phi_1 \text{ and } \Omega(w_n^0, w_0^2) \bigg\} \text{ then } w_0^2, \ldots, w_p^2 \models \phi_3 \\
w_0^1, \ldots, w_n^1 &\models \phi_2 \text{ and } \Omega'(w_n^1, w_0^2) \bigg\}
\end{align*}
\]

2.3 Contract in FO-LTL

A contract is defined as a pair \( (A, G) \) where \( A \) and \( G \) are respectively called the assumption and the promise. Intuitively, considering a transition system and a contract, the contract specifies that the system must ensure the promise when the system satisfies the assumption. The notation \( \text{Sys} \models (A, G) \) means that the contract \( (A, G) \) is satisfied by the system \( \text{Sys} \). Relying on the semantics we previously introduced, the satisfaction of a contract by a \( \text{TFOSTS} \) is
\[
\mathcal{T} \models (A, G) \quad \text{iff} \quad \forall k \in \mathcal{T}, \, k \models A \Rightarrow k \models G
\]

2.3.1 Adding Stochastic Information in a Transition System

A stochastic system is a transition system where each transition has a probability to be fired. For each transition \( s \rightarrow s' \), the function \( P : X \times X \rightarrow [0; 1] \) denotes the firing probability, i.e. the value \( P(s; s') \). The function \( P \) must ensure that the sum of the probabilities of the transitions that can be fired from a same state be equal to 1:
\[
\forall s, \sum_{s \rightarrow s'} P(s, s') = 1.
\]

2.3.2 Contracts for Stochastic Transition Systems

For the stochastic systems, it is generally more meaningful to quantify how the system satisfies this property: this valuation is given by the probability that the system satisfies the property. Intuitively, if the distribution to execute each run of a given stochastic system was known, the probability that this system satisfies the contract is the sum of the probabilities of all the runs that satisfy the property. Let \( \text{Sys} \) be a stochastic system, \( (A, G) \) a contract and \( p \) the probability that \( \text{Sys} \models (A, G) \). We will now consider, for the same system \( \text{Sys} \), the contract \( C = P \sim k(A, G) \) where \( \sim \in \{\leq; <; =; >\} \), and \( 0 \leq k \leq 1 \). The contract \( C \) is satisfied if and only if the relation \( p \sim k \) holds. The challenge is to estimate the probability \( p \), since it is not possible under general settings to exactly compute this probability.
3 GCSL Language Description

3.1 Example of Patterns

The GCSL syntax for patterns is a combination of the Object Constraint Language (OCL) and the contract pattern à la “SPEEDS” (http://www.speeds.eu.com). The SPEEDS contract specification patterns are introduced in the SPEEDS Deliverable D.2.5.4 “Contract Specification Language (CSL)” and used to give a high-level specification of real-timed components. They have been introduced to enable to user to reason about event triggering that are equivalently replaced in this document by constraint satisfaction. The constraints handled by these patterns are about the state of a SoS. We use OCL to specify these state constraints. The Object Constraint Language is a well-established standard in software design. The language is sufficiently powerful to describe precisely a state of a SoS. In this document, we will only consider a subset of the OCL language. In particular, we mainly focus on the Collection type without considering all its refinements (Set, Ordered Sets, …) and the most common predicates about Collections (forall(x|…), exists(y| …), select(…), …). The relevant part of the language considered is the expression subset of the language, i.e. Boolean, Collection or arithmetic expressions.

The patterns are designed to specify the behavior of each component instance by totally abstracting its environment without quantification. It is not possible to specify a contract about the interaction between two anonymous components. By anonymous, we mean that no particular instance is explicitly referenced by the component identifier. It is an important limitation for DANSE: to overcome it, we introduce a way to express quantifications that overlap a contract pattern. We use the OCL predicates forall() and exists() over OCL Collections to introduce the capability to quantify over the instances of any component type of an SoS.

The following table illustrates the kind of properties that we will express with our language. We use syntactic coloring to differentiate the different part of the language used in the property: the words in red are identifiers from the model, the blue part is from OCL and bold black keywords are temporal operators.

<table>
<thead>
<tr>
<th>“All FireStation hosts at least one Fire Fighting Car”</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoS.itsFireStations-&gt;forall(fstation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>“Any district cannot have more than 1 fire station, except if all districts have at least 1”</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoS.itsDistricts-&gt;exists(district</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>“The fire fighting cars hosted by a fire station shall be used all simultaneously at least once in 6 months”</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoS.itsFireStations-&gt;forall(fireStation</td>
</tr>
<tr>
<td>[fireStation.hostedFireFightingCars-&gt;forall(isAtFireStation = false)] occurs within [6 months])</td>
</tr>
</tbody>
</table>

Table 1-1: Some requirements extracted from the CAE example v1.5
We summarize the most interesting SPEEDS patterns. For each one, we briefly describe their semantics, that will be formally expressed by \( \text{B-LTL} \).

We first precise the semantics of the validity of a predicate during the execution path, which are the definition of the following patterns are based upon. Consider the state constraint \( \Psi \) and a time value sequence \( t_0, t_1, ..., t_n \) that defines the state sequence \( \sigma_0, \sigma_1, ..., \sigma_n \) such that \( t_i \) is the time value where the system reaches \( \sigma_i \). In other words, the system is in state \( \sigma_i \) when \( t_i \leq \text{time} < t_{i+1} \).

Figure 2: Satisfaction \( \Psi \) during an execution path.

Figure 2 illustrates the satisfaction of \( \Psi \), e.g. the green state \( \sigma_0, \sigma_1, \) and \( \sigma_i \) are the only states of the sequence that satisfy \( \Psi \). It means that \( \Psi \) holds when \( \text{time} \in [t_0, t_2) \cup [t_i, t_{i+2}) \). We observe that \( \Psi \) holds continuously for \( \sigma_0, \sigma_1 \) whereas the number of occurrences where \( \Psi \) holds is 2 during the time interval \( [t_0, t_n) \). Finally if we consider any time interval, \( \Psi \) holds during \( [a, b] \) but not during \( [a, c] \) nor \( [b, c] \) and the occurrence number of \( \Psi \) is 1 in \( [a, b] \), \( [a, c] \) or \( [b, c] \).

All the SPEEDS patterns introduced in the next are related to this definition of the constraint satisfaction.

<table>
<thead>
<tr>
<th>a)</th>
<th>whenever ( \Psi_1 ) occurs ( \Psi_2 ) holds during following ( [a, b] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Psi_1 )</td>
<td>( \Psi_2 )</td>
</tr>
<tr>
<td>( a )</td>
<td>( b )</td>
</tr>
<tr>
<td>The interval ( [a, b] ) is located relatively after the satisfaction of ( \Psi_1 ). The interval, in which ( \Psi_2 ) must be satisfied, starts ( a ) units of time after the observed occurrence of ( \Psi_1 ).</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b)</th>
<th>( \Psi_1 ) implies ( \Psi_2 ) holds forever</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Psi_1 )</td>
<td>( \Psi_2 )</td>
</tr>
<tr>
<td>From the very moment when ( \Psi_1 ) is satisfied ( \Psi_2 ) must hold during all the rest of the execution path.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c)</th>
<th>always ( \Psi )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Psi )</td>
<td></td>
</tr>
<tr>
<td>( 0 )</td>
<td>( \Psi ) must hold during all the execution path.</td>
</tr>
</tbody>
</table>
c) \textbf{whenever }\Psi_1 \textbf{ occurs }\Psi_2 \textbf{ holds}

\[ \Psi_1 \Rightarrow \Psi_2 \]

As for the previous pattern, the interval \([a, b]\) is relative. At each time value between \(a\) and \(b\) where \(\Psi_1\) holds, \(\Psi_2\) must also hold.

A derived pattern with \(\Psi\) is \textit{true}

\ \Psi_1 \textbf{ implies }\Psi_2 \textbf{ during following }[a,b]

d) \textbf{whenever }\Psi \textbf{ occurs }\Psi_1 \textbf{ implies }\Psi_2 \textbf{ during following }[a,b]

\[ \Psi \leftrightarrow [a, b] \]

As for the previous pattern, the interval \([a, b]\) is relative. At each time value between \(a\) and \(b\) where \(\Psi_1\) holds, \(\Psi_2\) must also hold.

This pattern specifies that \(\Psi_2\) is never satisfied during the relative interval \([a, b]\), \textit{i.e.} \(\neg\Psi_2\) holds during \([a, b]\).

e) \textbf{whenever }\Psi_1 \textbf{ occurs }\Psi_2 \textbf{ does not occur during following }[a,b]

\[ \Psi_1 \leftrightarrow [a, b] \]

f) \textbf{whenever }\Psi_1 \textbf{ occurs }\Psi_2 \textbf{ occurs within }[a,b]

\[ \Psi_1 \leftrightarrow [a, b] \]

The constraint \(\Psi_2\) must be satisfied at least one time during \([a,b]\), after \(\Psi_1\).
g) $\Psi_1$ occurs $n$ times during $[a, b]$ raises $\Psi_2$

$N$ occurrences of $\Psi_1$ at least

When $\Psi_2$ is satisfied at less $N$ times during $[a, b]$, $\Psi$ must hold at $b$.

h) $\Psi$ occurs at most $n$ times during $[a, b]$

$N$ occurrences of $\Psi$ at most

As previously mentioned, an occurrence of $\Psi$ is counted when $\Psi$ becomes satisfied. If $\Psi$ holds for a state in $[a, b]$, to observe $\Psi$ holds for the following one (also in $[a, b]$) does not increase the occurrence number of $\Psi$.

i) $\Psi_1$ during $[a, b]$ raises $\Psi_2$

If $\Psi_1$ holds during $[a, b]$ then $\Psi_2$ must hold at $b$.

j) $\Psi$ during $[a, b]$ implies $\Psi_1$ during $[a, c]$ then $\Psi_2$ during $[c, b]$

Whenever $\Psi$ holds during $[a, b]$ there exists a split at $c$ of $[a, b]$ such that $\Psi_1$ holds during $[a, c]$ then $\Psi_2$ holds during $[c, b]$. 
### 3.2 Extending patterns with probabilities

We propose to enrich these patterns in order to quantify probabilistically over the occurrences of properties or events: these nested probabilities allow to have more flexibility for the occurrence of properties in intervals by example. These extensions precise the confidence level we want to observe. We propose the following patterns:

k) **whenever \( \Psi_1 \) occurs \( \Psi_2 \) holds during following \([a, b]\) with a probability \(p\)**

\[
\Pr(\Psi_2) \geq p
\]

The interval \([a, b]\) is located relatively after the satisfaction of \(\Psi_1\). The interval, in which the probability that \(\Psi_2\) is satisfied must be greater than \(p\), starts \(a\) units of time after the observed occurrence of \(\Psi_1\).

l) **whenever \( \Psi \) occurs \( \Psi_1 \) implies \( \Psi_2 \) during following \([a, b]\) with a probability \(p\)**

\[
\Pr(\Psi_1 \Rightarrow \Psi_2) \geq p
\]

As for the previous pattern, the interval \([a, b]\) is relative. At each time value between \(a\) and \(b\) where \(\Psi_1\) holds, \(\Psi_2\) must also hold.

A derived pattern with \( \Psi \) is true

\[
\Psi_1 \text{ implies } \Psi_2 \text{ during following } [a, b] \text{ with a probability } p
\]

m) **whenever \( \Psi_1 \) occurs \( \Psi_2 \) does not occur during following \([a, b]\) with a probability \(p\)**

\[
\Pr(\neg\Psi_2) \geq p
\]

This pattern specifies that \(\Psi_2\) is never satisfied during the relative interval \([a, b]\), i.e. \(\neg\Psi_2\) holds during \([a, b]\).

n) **whenever \( \Psi_1 \) occurs \( \Psi_2 \) occurs within \([a, b]\) with a probability \(p\)**

\[
\Pr(\Psi_2) \geq p
\]

The constraint \(\Psi_2\) must be satisfied at least one time during \([a, b]\). after \(\Psi_1\).
o) $\Psi_1$ occurs $n$ times during $[a, b]$ raises $\Psi_2$ with a probability $p$

\[
\text{Proba} \geq p
\]

$N$ occurrences of $\Psi_1$ at least

$\Psi_2$

When $\Psi_2$ is satisfied at least $N$ times during $[a, b]$, $\Psi$ must hold at $b$.

p) $\Psi_1$ during $[a, b]$ raises $\Psi_2$ with a probability $p$

\[
\text{Proba} \geq p
\]

$\Psi_1$

If $\Psi_1$ holds during $[a, b]$ then $\Psi_2$ must hold at $b$.

q) $\Psi$ during $[a, b]$ implies $\Psi_1$ during $[a, c]$ then $\Psi_2$ during $[c, b]$ with a probability $p$

\[
\text{Proba} \geq p
\]

Whenever $\Psi$ holds during $[a, b]$ there exists a split at $c$ of $[a, b]$ such that $\Psi_1$ holds during $[a, c]$ then $\Psi_2$ holds during $[c, b]$.
3.3 The GCSL Contracts

The current version of GCSL allows to define two main types of contract. Some contracts to specific requirements about the structure (purely defined in OCL) or the behavior of a SoS. These contracts are defined as contracts we defined for B-LTL it is a triple \((A, P, k)\) where the properties \(A\) and \(P\) are expressed with the GCSL language, i.e. using mixed patterns and OCL subset, and \(k\) is probability threshold to reach.

\[
SoS \models (A, P, k) \quad \text{iff} \quad Pr(\exists R, SoS \models A \Rightarrow SoS \models P) \geq k
\]

When \(A\) and \(P\) are purely expressed using the OCL, \(k\) should be equal to 1. In this settings, \(A\) and \(P\) specify a structural property that must hold at the initial state: in this case, the contract holds or not, the only meaningful threshold to reach is 1.

The second kind of contracts are used to specify some dynamic properties, i.e. when the SoS configuration is changed (e.g. the population grown up to a critical threshold that needs to create new fire stations). This reconfiguration implies some structural modifications and the SoS architect may need to ensure these transformations preserve some properties. Such a contract is defined as quadruple \((A, \Omega, P, k)\) that is satisfied by a SoS is reconfigured into the system \(R(\text{SoS})\) using the transformation rule \(R\), the probability that when SoS satisfies \(A\) and \(R\) holds for \(\Omega\), then \(R(\text{SoS})\) satisfies \(P\) is greater than \(k\). The properties \(A\) and \(P\) are defined using GCSL restricted to the behavioral patterns without the nested properties. The reason is practical and dynamicity contracts with nested probabilities are not supported by the verification tools.

\[
SoS \models (A, \Omega, P, k) \quad \text{iff} \quad Pr(\exists R, \text{SoS} \models A \land R \models \Omega \Rightarrow R(\text{SoS}) \models P) \geq k
\]

We also extend this definition dynamic contract to specify property for sequence of rule application: we propose such contract for any sequence of \(n\) transformation rules and starting from \(\text{SoS}_0\) with \(R_i\) reconfigures \(\text{SoS}_{i+1}\) into the system \(R_i(\text{SoS}_{i+1})\). Notice that the rule sequence has the form \(R_m, R_{m-1}, R_{m-2}, \ldots, R_0\). A contract for any sequence \(R_m, R_{m-1}, R_{m-2}, \ldots, R_0\) is the pair \((C_m, k)\) where \(C_m\) is inductively defined as

\[
\begin{align*}
C_0 &= (P_1, \Omega_1, P_0) \\
C_{i+1} &= (P_{i+2}, \Omega_{i+2}, C_i)
\end{align*}
\]

The contract satisfaction is given by

\[
SoS \models (C_n, k) \quad \text{iff} \quad Pr(\exists R, \text{SoS} \models P_{n+1} \land R \models \Omega_{n+1} \Rightarrow R(\text{SoS}) \models C_{n-1}) \geq k
\]

The concrete syntax for the contracts is given in Section Error! Reference source not found.. with some examples to illustrate it.

3.4 Mapping of Patterns to B-LTL

The B-LTL language is a very expressive language that is able to cover a large set of properties in both timed and space domains. This language is used by Plasma – the SMC-tool developed by INRIA– as input for the properties to check. The B-LTL properties are designed to reason over State Transition Systems. It is FO-LTL where temporal modalities are bounded over a time interval: the satisfaction of the formula handled by bounded modality is only relevant during the time interval.

3.4.1 Overview of the translation procedure

As illustrated in the third example of requirements of Table 1 the language is layered as some behavioral properties defined using the patterns combined with some state properties written in OCL. These behavioral properties can themselves be wrapped into an OCL collection expression to quantify the behavioral properties.
properties over some constituents of the SoS. The translation of a contract will be made by translating from its assumption and its promise only the OCL quantification and the pattern layers. The translated property will be checked against some simulations. The state properties expressed in OCL have to be checked against some states and for them, no treatment is done during the translation. The state properties are kept in the translated formula and there will be dynamically checked. We assume that the satisfiability of the state properties is solved by an external procedure based on an existing OCL-checker.

**Proposition 1.** Let us consider a contract \((A; P)\) of a given SoS and assume any simulation is bounded by \(k\) a maximum time of execution. If there exist two \(B\)-LTL formulas \(A_0\) and \(P_0\) such that \(A_0\) (or \(P\)) and \(A\) (\(P_0\) resp.) are equivalent for any \(k\)-bounded simulations, then the \(B\)-LTL formula \(A_0 \Rightarrow P_0\) is equivalent to the contract \((A; P)\) for any \(k\)-bounded simulations.

Moreover, extending the translation to a stochastic contract is natural. The pair \((A; P)\) of any stochastic contract is similarly treated. Similarly we are able to translate the contract translation to the contract about dynamicity. It is sufficient to replace the sub-formula \((\forall R, SoS \models P_{n+1} \land R \models \Omega_{n+1} = R(SoS) \models C_{n-1})\) in the definition of the GCSL contract by the equivalent adaptive property \(P_{n+1} \Omega_{n+1} C_{n-1}\) and the triple \(C_0 = (P_1, \Omega_1, P_0)\) by \(P_1 \Omega_1 P_0\).

### 3.4.2 Speeds Pattern translation

For the purpose of translation to \(B\)-LTL, we assume that the constant \(k\), used in all pattern, is an additional SMC parameter given by the user: it corresponds to the length (time duration) of the paths for which the property must hold. Moreover, to be successfully translated, the pattern must be consistent, in particular the intervals must have correct bounds and intervals must be all visited before the end of the path (specified with constant \(k\)) is reached.

Here, we assume that all nested OCL propositions, e.g. the state or path propositions, whose are here abstracted by \(\Psi, \Psi_1\) and \(\Psi_2\), are semantically preserved and will be translated independently by an OCL dedicated tool. The following table summarizes the immediate pattern translation into \(B\)-LTL:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>(B)-LTL translation</th>
<th>Consistency condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (\Psi_1) implies (\Psi_2) holds forever</td>
<td>(G_{tsk}(\Psi_1 \Rightarrow G_{sk-1}\Psi_2))</td>
<td></td>
</tr>
<tr>
<td>2 always (\Psi)</td>
<td>(G_{sk}(\Psi))</td>
<td></td>
</tr>
<tr>
<td>3 whenever (\Psi_1) occurs (\Psi_2) holds</td>
<td>(G_{sk}(\Psi_1 \Rightarrow \Psi_2))</td>
<td></td>
</tr>
<tr>
<td>4 (\Psi_1) implies (\Psi_2) during following ([a, b])</td>
<td>(G_{[a,b]}(\Psi_1 \Rightarrow \Psi_2))</td>
<td>(a \leq b)</td>
</tr>
<tr>
<td>5 (\Psi_1) during ([a, b]) raises (\Psi_2)</td>
<td>(G_{[a,b]}(\Psi_1 \Rightarrow F_{[b,k]}\Psi_2))</td>
<td>(a \leq b, b \leq k)</td>
</tr>
<tr>
<td>6 (\Psi) during ([a, b]) implies (\Psi_1) during ([a, c]) then (\Psi_2) during ([c, b])</td>
<td>(G_{[a,b]}\Psi \Rightarrow [G_{[a,c]}\Psi_1 \land G_{[c,b]}\Psi_2])</td>
<td>(a \leq c \leq b)</td>
</tr>
</tbody>
</table>

**Extended \(B\)-LTL patterns with absolute intervals**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>(B)-LTL translation</th>
<th>Consistency condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 (\Psi_1) occurs (n) times during ([a, b]) raises (\Psi_2)</td>
<td>(occ(\Psi_1, a, b) \geq n \Rightarrow F_{[b,k]}\Psi_2)</td>
<td>(a \leq b, b \leq k)</td>
</tr>
<tr>
<td>8 (\Psi) occurs at most (n) times during ([a, b])</td>
<td>(occ(\Psi, a, b) \leq n)</td>
<td>(a \leq b)</td>
</tr>
</tbody>
</table>

**Basic \(B\)-LTL patterns with sliding intervals**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>(B)-LTL translation</th>
<th>Consistency condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 whenever (\Psi_1) occurs (\Psi_2) holds during following ([a, b])</td>
<td>(G_{e\in[0,k-b]}(\Psi_1 \Rightarrow G_{[t+a,t+b]}\Psi_2))</td>
<td>(a \leq b, b \leq k)</td>
</tr>
</tbody>
</table>
10 whenever $\Psi_1$ occurs $\Psi_1$ implies $\Psi_2$ during following $[a,b]$ 
$G_{t\epsilon[0,k-b]}(\Psi_1 \Rightarrow G_{[t+a,t+b]}(\Psi_1 \Rightarrow \Psi_2))$ 
$a \leq b, b \leq k$

11 whenever $\Psi_1$ occurs $\Psi_2$ does not occur during following $[a,b]$ 
$G_{t\epsilon[0,k-b]}(\Psi_1 \Rightarrow G_{[t+a,t+b]} \neg\Psi_2)$ 
$a \leq b, b \leq k$

12 whenever $\Psi_1$ occurs $\Psi_2$ occurs within $[a,b]$ 
$G_{t\epsilon[0,k-b]}(\Psi_1 \Rightarrow (F_{[t+a,t+b]} \Psi_2)$ 
$a \leq b, b \leq k$

B-LTL patterns with nested probabilities

13 whenever $\Psi_1$ occurs $\Psi_2$ holds during following $[a,b]$ with a probability $p$ 
$G_{t\epsilon[0,k-b]}(\Psi_1 \Rightarrow Pr_{zp}(G_{ax-\Psi_2}))$ 
$a \leq b, b \leq k$

14 whenever $\Psi_1$ occurs $\Psi_2$ implies $\Psi_2$ during following $[a,b]$ with a probability $p$ 
$G_{t\epsilon[0,k-b]}(\Psi_1 \Rightarrow Pr_{zp}(G_{t+a,t+b}\Psi_2))$ 
$a \leq b, b \leq k$

15 whenever $\Psi_1$ occurs $\Psi_2$ does not occur during following $[a,b]$ with a probability $p$ 
$G_{t\epsilon[0,k-b]}(\Psi_1 \Rightarrow Pr_{zp}(G_{t+a,t+b}\neg\Psi_2))$ 
$a \leq b, b \leq k$

16 whenever $\Psi_1$ occurs $\Psi_2$ occurs within $[a,b]$ with a probability $p$ 
$G_{t\epsilon[0,k-b]}(\Psi_1 \Rightarrow Pr_{zp}(F_{[t+a,t+b]}\Psi_2))$ 
$a \leq b, b \leq k$

17 $\Psi_1$ occurs $n$ times during $[a,b]$ raises $\Psi_2$ with a probability $p$ 
$occ(\Psi_1,a,b) \geq n$ 
$\Rightarrow Pr_{zp}(F_{[t+b]}\Psi_2)$ 
$a \leq b, b \leq k$

18 $\Psi_1$ during $[a,b]$ raises $\Psi_2$ with a probability $p$ 
$G_{[a,b]}(\Psi_1 \Rightarrow Pr_{zp}(F_{[t+b]}\Psi_2))$ 
$a \leq b, b \leq k$

19 $\Psi$ during $[a,b]$ implies $\Psi_1$ during $[a,c]$ then $\Psi_2$ during $[c,b]$ with a probability $p$ 
$G_{[a,b]}(\Psi \Rightarrow Pr_{zp}(G_{[a,c]}\Psi_1 \land G_{[c,b]}\Psi_2))$ 
$a \leq c \leq b$

Table 1: Pattern mapping

**Unbound time patterns:** The patterns 1 and 2 require that the expressed properties must hold while the system is running, e.g., they have a meaning for infinite execution paths too. But, the verification will be done against simulation paths that are necessarily finite, for practical reasons (termination). Thus, the infinite bound is replaced by the user constant $k$ provided for the verification.

**Extended B-LTL patterns:** The patterns 7 and 8 require to count the number of occurrences in $[a, b]$. Counting is not possible by strictly using B-LTL. We assume that there exist a dedicated procedure that counts the number of times where $\Psi$ becomes satisfied.

**Sliding intervals:** as previously mentioned, it means that the interval $[a, b]$ is located after time $t$, at which the first part of the pattern “Whenever $\Psi$ occurs” is satisfied: it means that the effective interval to consider is $[t + a, t + b]$.

### 3.4.3 OCL expression

OCL expressions occur at two levels within a pattern: as atomic propositions to define state condition and as quantifications. The first case will be directly treated by a simple OCL-checker against a state of the SoS or translated into a more generic semantics provided by the SMC-checker. But some atomic propositions can also contain some quantification about component collections and in this case they can also be processed as explained below.

The second case is the most interesting case. The B-LTL logic has no quantification support; it could be extended but this needs to rewrite the B-LTL checker. Moreover, the instances of each component type are statically specified in the Internal Block Diagram (IBD) by the SoS architect. If we consider the CAE example, the idbFireEmergency diagram gives the list of all system constituents instantiated in the SoS: 10 Districts, 1 Fire Headquarter, 3 Fire stations and 7 fire Fighting Cars, etc. The semantics of quantification is given by unfolding the quantification as a conjunction (or a disjunction) formula.

Any universal quantification over a collection is translated to a logical formula:

$$\text{SoS.itsFireStations} \rightarrow \forall (x) \phi(x)$$

<table>
<thead>
<tr>
<th>Version</th>
<th>Status</th>
<th>Date</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.00</td>
<td>Final</td>
<td>2013-05-21</td>
<td>18 of 31</td>
</tr>
</tbody>
</table>
\[
\bigwedge_{x:FireStation} \phi(x) \\
\implies \\
\phi(\text{fireStation1}) \land \phi(\text{fireStation2}) \land \phi(\text{fireStation3})
\]

Any existential quantification is dually converted as:
\[
\text{SoS.itsFireStations}\rightarrow\exists x.\phi(x) \\
\implies \\
\bigvee_{x:FireStation} \phi(x) \\
\implies \\
\phi(\text{fireStation1}) \lor \phi(\text{fireStation2}) \lor \phi(\text{fireStation3})
\]

The processing of the nested quantifications is similar: it is sufficient to recursively unfold the quantifications in a bottom-up order. This translation can also be easily extended to some collection obtained using operators like \text{select(...), excludes(...), includes(...), etc.}
3.4.4 Illustration of the full translation of a GCSL formula

We illustrate the translation with the formula from the CAE example:

\[
\text{SoS.itsFireStations} \rightarrow \text{forAll} (\text{fireStation} | \quad \text{Whenever} [\text{fireStation.hostedFireFightingCars} \rightarrow \text{exists} (\text{isAtFireStation})] \quad \text{occurs,} \quad [\text{fireStation.hostedFireFightingCars} \rightarrow \text{forall} (\text{isAtFireStation} = \text{false})] \quad \text{occurs within} \quad [6 \text{ months}])
\]

Assuming that \( k \geq 6 \text{ months} \), the translation is given by the rule 12 in Table 1:

\[
\phi = \begin{cases} 
G_{t \in [0,k]} \left( \Psi_1(\text{fireStation1}) \Rightarrow (F_{[t,t+6\text{months}]} \Psi_2(\text{fireStation1})) \wedge \\
G_{t \in [0,k]} \left( \Psi_1(\text{fireStation2}) \Rightarrow (F_{[t,t+6\text{months}]} \Psi_2(\text{fireStation2})) \wedge \\
G_{t \in [0,k]} \left( \Psi_1(\text{fireStation3}) \Rightarrow (F_{[t,t+6\text{months}]} \Psi_2(\text{fireStation3})) \right) \right)
\end{cases}
\]

Where \( \Psi_1(\ldots) \) and \( \Psi_2(\ldots) \) corresponds to the OCL expressions \( \text{fireStation.hostedFireFightingCars} \rightarrow \text{exists} (\text{isAtFireStation}) \) and \( \text{fireStation.hostedFireFightingCars} \rightarrow \text{forall} (\text{isAtFireStation} = \text{false}) \) respectively. The next table summarizes the translation of \( \Psi_1(\ldots) \) and \( \Psi_2(\ldots) \) for each fireStation. Finally replacing all instances of \( \Psi_1(\ldots) \) and \( \Psi_2(\ldots) \) in \( \phi \) gives us the all translated formula.

<table>
<thead>
<tr>
<th>FireStation</th>
<th>( \Psi_1 )</th>
<th>( \Psi_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>fireStation1</td>
<td>( \text{fireFightingCar1.isAtFireStation} \lor )</td>
<td>( \neg \text{fireFightingCar1.isAtFireStation} \land )</td>
</tr>
<tr>
<td></td>
<td>( \text{fireFightingCar2.isAtFireStation} \lor )</td>
<td>( \neg \text{fireFightingCar2.isAtFireStation} \land )</td>
</tr>
<tr>
<td></td>
<td>( \text{fireFightingCar3.isAtFireStation} \lor )</td>
<td>( \neg \text{fireFightingCar3.isAtFireStation} \lor )</td>
</tr>
<tr>
<td>fireStation2</td>
<td>( \text{fireFightingCar4.isAtFireStation} \lor )</td>
<td>( \neg \text{fireFightingCar4.isAtFireStation} \land )</td>
</tr>
<tr>
<td></td>
<td>( \text{fireFightingCar5.isAtFireStation} \lor )</td>
<td>( \neg \text{fireFightingCar5.isAtFireStation} \lor )</td>
</tr>
<tr>
<td>fireStation3</td>
<td>( \text{fireFightingCar6.isAtFireStation} \lor )</td>
<td>( \neg \text{fireFightingCar6.isAtFireStation} \land )</td>
</tr>
<tr>
<td></td>
<td>( \text{fireFightingCar7.isAtFireStation} \lor )</td>
<td>( \neg \text{fireFightingCar7.isAtFireStation} \lor )</td>
</tr>
</tbody>
</table>
### 3.5 Syntax by Description of Patterns

As introduced above, the GCSL syntax is a combination of OCL and the contract pattern à la “SPEEDS”. We give here the pattern syntax and the specification of how to use OCL in it. We define the GCSL syntax as a GSCL grammar in BNF notation. Neither all constructs described above nor all OCL features are intended for implementation in the first version. In the BNF specification, we use the BNF conventions: `expr`? to denote 0 or 1 occurrence of expr and `{expr}`+ to denote at least one occurrence of expr.

```plaintext
<pattern-specification> ::=  
  | <OCL-expr> "forall" "(" <variable> ")" <behavioral-pattern> 
  | <OCL-expr> "exists" "(" <variable> ")" <behavioral-pattern> 
  | <OCL-proposition>  
  | <behavioral-pattern>

<arith-proposition> ::= <expr> ( "<" | "=" | "<=" | ">=" | ">" ) <expr>

<arith-expr> ::= <expr> <operator> <expr>  | "(" <expr> ")" <OCL-expr>  
  | MEAN "(" <OCL-expr >")" | SUM "(" <OCL-expr >")" | PROD "(" <OCL-expr >")" 
  | AT "(" <OCL-expr>, <T> ")"

<operator> ::= "+" | "+" | "*" | "/*" | "" | ...

<behavioral-pattern> ::=  
  | whenever "(" <OCL> ")" occurs "(" <OCL> ")" holds during following "(" <I> ")" <proba-pattern>?  
  | whenever "(" <OCL> ")" occurs "(" <OCL> ")" implies "(" <Prop> ")" during following "(" <I> ")" <proba-pattern>? 
  | whenever "(" <OCL> ")" occurs "(" <OCL> ")" does not occur during following "(" <I> ")" <proba-pattern>? 
  | whenever "(" <OCL> ")" occurs "(" <Prop> ")" occurs within "(" <I> ")" <proba-pattern>?  
  | "(" <OCL> ")" during "(" <I> ")" raises "(" <OCL> ")" <proba-pattern>? 
  | "(" <OCL> ")" occurs "(" <N> ")" times during "(" <I> ")" raises "(" <OCL> ")" <proba-pattern>? 
  | "(" <OCL> ")" occurs at most "(" <N> ")" times during "(" <I> ")" <proba-pattern>? 
  | whenever "(" <OCL> ")" during "(" <I> ")" <proba-pattern>? implies "(" <OCL> ")" during "(" <I> ")" <proba-pattern>?  
  | whenever "(" <OCL> ")" during "(" <I> ")" <proba-pattern>? then "(" <OCL> ")" during "(" <I> ")" <proba-pattern>? 

<proba-pattern> ::= "with a probability" <P>

<Prop> ::= <OCL-proposition> | <arith-proposition>

<OCL> ::= <OCL-proposition>

<N> ::= <natural-number>

<P> ::= <rational-number>

<I> ::= <interval>

<T> ::= n <time-unit> | \( +\infty \)

<interval> ::= ( "[" | "(" ) <T> ( "]" | ")" )
```

<time-unit> should be any multiple of the application basic time unit (i.e. day, hour, min, sec, ms, …).

The OCL specification can be found at [http://www.omg.org/spec/OCL/2.2/](http://www.omg.org/spec/OCL/2.2/) and together with the grammar of the language. We only provide an overview of the relevant subset used in GCSL:

- `<OCL-proposition>` stands for the simple Boolean expressions over collections or primitive types (int, real, boolean, …) of OCL. We also identified `<OCL-expr>`, the OCL subset of non-Boolean expression, e.g. Component Collections (e.g., the functions map(…), iterate(…), etc.), numerical values, model-related values, etc.
## 3.6 Syntax of a GCSL Contract

A GCSL contract will be defined using the following syntax:

```
<contract> ::= <contract-spec>
| <dyna-spec>

<contract-spec> ::= {<viewpoint-id>}+ contract <contract-id>
   {Assumption: <pattern-specification>}? 
   {Promise: <pattern-specification>}
   {Confidence: <threshold>}?

<dyna-spec> ::= dynamicity contract <contract-id>
   if “[“ <pattern-specification> “]” holds and
   for all rule such that rule satisfies “[“ <OCL-spec> “]”
   then ( <pattern-specification> | <contract-id> ) holds
   {Confidence: <threshold>}?
```

```
<viewpoint-id> ::= behavior | structure | safety | liveness | ...
<threshold> ::= <N>"%" | <rational-value>
<rational-value> is any rational number in (0; 1]
<N> ::= <natural-number>
```

The contract is defined with an assumption, a promise and a confidence value, that corresponds to the probability that the contract be satisfied. If the assumption is omitted then it is implicitly equivalent to declare a contract with an assumption that is true under all conditions. The confidence value has the following meaning: the contract is satisfied by the SoS, if the probability to satisfy the contract (assumption, promise) is greater or equal to this value. The viewpoint-id is only for the user documentation of the contract, it has no effective meaning with respect to the semantics of the contract. Notice that the confidence threshold can be omitted. When it is, that the default value of 100% is atomically assumed.

We illustrate the complete GCSL syntax for some simple promises extracted from the CAE example:

```
“All FireStation shall host at least one Fire Fighting Car”

Structure Contract NonEmptyFireStation
   Promise: SoS.itsFireStations->forAll(fstation | fstation.hostedFireFightingCars->size() >= 1)
   Confidence: 100%

“Any district cannot have more than 1 fire station, except if all districts have at least 1”

Structure Contract FireStationSharing
   Promise: SoS.itsDistricts->exists(district | district.containedFireStations->size() > 1) implies 
        SoS.itsDistricts->forAll(containedFireStations->size() >= 1)
   Confidence: 100%

“The fire fighting cars hosted by a fire station shall be used all simultaneously at least once in 6 months with a probability of 90%”

Liveness Contract FireStationSharing
   Promise: 
        SoS.itsFireStations->forAll(fireStation |
Whenever \([\text{fireStation}.\text{hostedFireFightingCars} \rightarrow \text{exists(isAtFireStation)})]\) occurs, \\
\([\text{fireStation}.\text{hostedFireFightingCars} \rightarrow \text{forall(isAtFireStation} = \text{false})]\) \\
occurs within [6 onths]) \\
Confidence: 90%
4 GSCL extension for optimization

4.1 Motivation

One of the main approaches of GSCL extension for optimization is Classification by Property (Parson and Wand 2000). The core basis for this framework is first suggested as a data management framework which emancipates the persistence of information about individual elements and their characteristics in any given domain from being entangled with their possible classifications. The essence of this approach is to define things that possess properties. The sets of all instances and properties are called instance and property base, respectively. The set of properties may include intrinsic properties that are being considered inherent to things along with mutual properties that correspond to two or more things. All things live without any a priori classification. Furthermore, the things can evolve, where new properties can be added while other are removed. Independent from the specification of all things, classes are defined by set of properties. Consequently, things could belong to many classes if they have appropriate sets of properties for all of them.

Masin et al. (2013) extends the Classification by Property framework for analysis and optimization usage. To illustrate our concepts throughout the paper we use a common engineering example shown in Figure 3, purposely simplified for clarity. The design flow includes modeling and custom optimization where the model is gradually changing. The Requirements Layer consists of three system functions: Sensing, Controlling, and Actuating. The layer also consists of two input/output links: Sensing as input for Controlling, and Controlling as input for Actuating. The Architecture Layer consists of five component types: Sensors, Controllers, Actuators, and a data bus consisting of analog to digital adapters – Remote Data Concentrators (RDCs), and Switches. The architectural viewpoint also imposes constraints on the physical structure. Component types have different sets of attributes but all include Cost and Weight. Component libraries for each type are stored in external database tables called Catalogs. The objective is minimizing the total system cost while the total system weight is less than a given threshold. In the design solution, system functions are implemented by architectural components they are mapped to. Similarly, functional input/output links are mapped to architectural routes – from a component implementing input function to a component implementing output function. All this information is usually stored in engineering models, e.g., using SysML, and databases.
Let us consider the following extract from an optimization model:

\[
\text{Minimize } \text{totalCost} \\
\text{Subject to } \\
\text{totalCost} = \sum_{j \in \text{SensorTypes}} \sum_{i \in \text{Sensors}} \text{SensorType}[j].\text{Cost} \cdot \text{sensor}[i][j] + \cdots \\
+ \sum_{j \in \text{SwitchTypes}} \sum_{i \in \text{Switches}} \text{SwitchType}[j].\text{Cost} \cdot \text{switch}[i][j].
\]

The above algebraic expression employs objective, parameters, sets, variables, and constraints where SensorTypes, Sensors, SwitchTypes and Switches are sets of various viewpoint elements, SensorType and SwitchType are known catalog parameters, totalCost is a continuous decision variable, and sensor[i][j] and switch[i][j] are Boolean decision variables. One of the mapping constraints to ensure mapping of each sensing function to some sensor could be as follows:

\[
\sum_{j \in \text{SensorTypes}} \sum_{i \in \text{Sensors}} \text{sensing2sensor}[i][j] = 1 \ \forall l \in \text{SensingFunctions},
\]

where sensing2sensor[i][j] is the mapping Boolean decision variable. In addition, the population of elements in all sets, and the optimization model in general should reflect the actual data defined in all engineering models and stored databases.

As the design gradually evolves, various modifications may be introduced to the models. Our main challenge in this paper is to make our optimization models resilient to such modifications. In this section we first demonstrate how some of such modifications may force undesired model modification. In the next section we introduce a solution to this problem. One possible change could be a requirement for using several types of sensors, e.g., thermal and volume sensors, each having its own attributes and a catalog of available types. This minor change requires changing the above algebraic expression as follows.

\[
\text{totalCost} = \sum_{j \in \text{TermalsSensorTypes}} \sum_{i \in \text{TermalsSensors}} \text{ThermalSensorType}[j].\text{Cost} \cdot \text{termalsSensor}[i][j] \\
+ \sum_{j \in \text{VolumeSensorTypes}} \sum_{i \in \text{TermalsSensors}} \text{VolumeSensorType}[j].\text{Cost} \cdot \text{volumeSensor}[i][j] + \cdots \\
+ \sum_{j \in \text{SwitchTypes}} \sum_{i \in \text{Switches}} \text{SwitchType}[j].\text{Cost} \cdot \text{switch}[i][j].
\]

Another possible modification is adding a new Cable component to connect between other architectural components. It requires engineering modeling of the new type – Cable – with corresponding types catalog data. In the Optimization model, totalCost calculation should be revised to include cables’ costs and cables’ weights, respectively. As the second consequent, the system modeler may recognize that cables’ costs are also dependent on the geometrical layout of the network which is specified in a new Geometrical Viewpoint comprising two types of elements: Compartments and Links (between compartments). Each compartment is characterized by location coordinates and each Link by a length attribute. For each cable component, its cost could be calculated according to its length. The optimization model may reflect this change with additional variables that take into account the geometrical viewpoint realized by an additional index for chosen location in variables used above. Hence, this change implies updating most of existing equations since the new index should be specified in either some of \textit{forall} statements or in some of the aggregating operators, e.g., summation:

\[
\text{totalCost} = \sum_{j \in \text{SensorTypes}} \sum_{i \in \text{Sensors}} \sum_{k \in \text{Compartments}} \text{SensorType}[j].\text{Cost} \cdot \text{sensor}[i][j][k] + \cdots \\
+ \sum_{j \in \text{CableTypes}} \sum_{i \in \text{Cables}} \sum_{k \in \text{Paths}} \text{CableType}[j].\text{Cost} \cdot \text{cable}[i][j][k].
\]

Finally, let us consider changes in the optimization model when safety requirements are added. To satisfy the new requirements, functions and functional links could be implemented by several redundancy channels and the mapping variables and equations shown above should be adjusted accordingly, e.g., as follows:
where RedundancyChannels is the set of possible redundancies and redundancyChannel\[i][r] is the Boolean decision variable for finding the best redundancy option. Note that the new index is defined in forall operator.

The classical “shotgun surgery” term [14] is probably the most suitable description of our experience with the custom optimization modeling: “every time you make a kind of change you have to make a lot of little changes in a lot of classes”. Although the changes in the model could be relatively minor, each one requires re-examining most of the constraints previously specified in the optimization model undermining its reuse sustainability.

The first problem identified in the aforementioned use case is the one of set heterogeneity: elements of different sets belong to different classes and cannot be united into one set. For example, one cannot relate to all physical components, regardless of their concrete type, whose weight is above a given threshold. Another observation is that the classes and corresponding sets defined by them are too specific – any change in a class definition should be reflected in the optimization model, including changes to attributes used as parameters in constraints. In addition, one component type (e.g., a sensor) may have an attribute 'value', whereas another component type (e.g., a switch) may refer to the same attribute as 'cost' – both reflecting the monetary value of a component. Such inconsistency precludes using both types uniformly in the same aggregation function.

To summarize, most weaknesses associated with conventionally formed algebraic expressions are the result of having the algebraic style being directly dependent on element sets specified in one of the system's viewpoints, most commonly being the functional and physical models. This dependency results in having designers devoting significant portion of their time and effort to constantly having to revise the optimization model ensuring its consistency with system's engineering model.

### 4.2 Language extension

The proposed GSCL language extension includes following elements for property based classification.

Property classification and special properties:

- **Intrinsic property** – property of some simple data type (integer, float, boolean, etc.). For example Cost property has type float.
- **Mutual property** – property that links the contained instance to a set of other instances. A mutual property defines a relation between two or more instances, e.g., isContainedIn defines inclusion relation between two instances.

Intrinsic properties can be further customized by value types:

- **Free property** – defines a decision variable for the optimization process. Its value is defined during the optimization process. For example, isSelected is a special Boolean property defined for optimization purposes. All instances are augmented with this property. If it is a free attribute for some instance, then the instance is optional. Overall, the population of instances does not change during optimization, but optional instances could become realized if its isSelected attribute becomes true.
- **Ground property** - defines a variable having an explicitly defined value, i.e., they are treated as parameter for optimization purpose.

Operators:

- **Scope** operator for selecting instances with intrinsic properties.
  Syntax: Set<Instance> Scope{<string PropertyName, [string PropertyType], [value]>}.
  Parameters:
  
  A set of triplets:
  - **PropertyName** – mandatory, specifies name of the property.
  - **PropertyType** – optional, specifies the type of the property.
  - **value** – optional, specified the value of a (specific) property.
Scope returns a set of instances having a set of properties with name PropertyName of type PropertyType (or of any type if PropertyType = null).

Examples:
Scope(<Cost, float>) return all instances having an intrinsic property Cost of type float.

- **LinkedScope** operator for mutual properties.
  Syntax: Map(<Instance>, Set<Instance>) LinkedScope(Instance BaselineInstance, string PropertyName, Instance LinkedInstance).
  Parameters:
  - BaselineInstance – optional, specifies an instance containing a mutual property.
  - PropertyName – mandatory, specifies the name of the mutual property.
  - LinkedInstance – optional, specifies the instance for which the mutual property pointed.

  Returned value: Map where the keys are the BaselineInstances and the values are the appropriate set of LinkedInstances. Each key in the map is an instance of a block with the property PropertyValue and the corresponding map value is the set of instances pointed to by the property PropertyName of that key instance. At least one between BaselineInstance and LinkedInstance must be null. If BaselineInstance is not null, the returned map contains a single key, namely BaselineInstance. If LinkedInstance is not null, the returned map has a key for each instance whose property PropertyName points to LinkedInstance. If both BaselineInstance and LinkedInstance are null, each instance of a block that has the property PropertyName appears as a key in the returned map.

  Examples:
  LinkedScope(null, couldbeLinkedTo, null): for each instance (key) having the mutual property couldbeLinkedTo, the operator returns all instances (value) for which this instance could be linked.
  LinkedScope(SoS, contains, null): returns a map with one key (“SoS”) and its value is the set of all instances contained in the SoS instance.
  LinkedScope(null, couldbeLinkedTo, Antenna1): returns a map, where the key set is the set of all instances which could be linked to the instance Antenna1 and the value is the set with one element “Antenna1”.
  LinkedScope(SoS, couldbeLinkedTo, System1): if the linked property exists, returns Map<SoS, {System1}>, otherwise, returns an empty map.

- **Property definition operator** to define analysis properties/attributes that should be added to some instances.
  DefineProperty defines a new property for a set of elements.
  Syntax: DefineProperty(Set<Instance> InstanceSet, string PropertyName, string PropertyType).
  Parameters:
  - InstanceSet – mandatory. Instance or set of instances for which the new property is defined.
  - PropertyName – mandatory, the name of the new property.
  - PropertyType – mandatory, the type of the new property.

  Example:
  DefineProperty(Scope(costPerM, float), Cost, float) defines property cost of type float for each instance having costPerM float property.

  Note that the InstanceSet will be usually defined using Scope or LinkedScope operators.

- **Set operators.**
  - Set operators includes: union, intersection, difference and symmetric difference.

- **Algebraic syntax.**
  - Algebraic operators, equations and constraints use the syntax of the OPL language (Hentenryck 1999).

  Special values:
  - **Null** – indicates that value of some property or parameter is empty.
  - **Self** – refers to the instance in the context, e.g., in the case where the constraint is attached to some instance in the model.
4.3 Examples

Using the proposed language structure we prevent from changing the optimization model each time the corresponding system model changes. For example, considering the totalCost statement previously illustrated, using the GCSL language extension described above, the total cost equation becomes as follows:

$$\text{totalCost}(i) = \sum_{j \in \text{Scope}(\text{Cost}) \text{ inter } \text{LinkedScope}(\text{Null, isContainedIn}.i).\text{keySet()} \text{ isSelected}(j) \cdot \text{Cost}(j) \ \forall \ i \in \text{Scope(\text{totalCost})}. $$

This new statement is now independent of any changes in any particular Modeling or Analysis Viewpoint and very robust to the design evolution. Mapping Modeling Viewpoint adds new mutual properties connecting Requirement and Architecture elements such as:

- `couldBeMapped[requirementsElement][architecturalElement]` (potential architectural elements for mapping a requirement element)
- `isMapped[requirementsElement][architecturalElement]` (boolean decision variables)

New mapping equations use these properties to define Mapping Viewpoint robust to design evolution:

$$\sum_{j \in \text{LinkedScope}(i, couldBeMapped, \text{Null, get}(i))} \text{isMapped}(i)(j) = \text{isSelected}(i) \ \forall \ i \in \text{Scope(\text{isRequirementsElement})}. $$

The above equations could be implemented in the CAE as shown in Figure 4.

Figure 4: Example of cost function specification in GCSL
5 Conclusions and Next Steps

In the specification and formalization of the requirements for systems of systems, the SoS structure plays an important role, since it is part of the dynamics of the SoS, changing over time. We have shown that combining temporal logic reasoning with the OCL language makes it feasible to specify requirements not only on the computational behavior of the SoS, but also on its possibly dynamically changing structure. The language, being pattern-based, represents a powerful tool for the user, which is able to express requirements over complex dynamic behavior at the UPDM level.

Additional direction is to check applicability of GCSL to define different analysis viewpoints used in DANSE project. Another step to be taken to achieve the objectives set forth in DANSE for the rigorous specification of SoS requirements is the definition of contracts over more abstract UPDM concepts: at the current status of the language, GCSL contract requirements can be successfully expressed over the structure of the SoS and its functional behavior as expressible via SysML. In the next year of the project, we aim at introducing patterns to reason over more abstract UPDM concepts, such as location, mode change, resource, capability, operational activity, plan, strategy, etc. Additional points and user needs may arise as more examples are drawn and experience about using the GCSL language is gathered.
6 Abbreviations and Definitions

B-LTL  Bounded Linear Temporal Logic
CS(s)   Constituent System(s)
DANSE  Designing for Adaptability and evolution in System of systems Engineering
GCSL   Goal and Contract Specification Language
SoS    System of Systems
7 Bibliography