Contracts and Behavioral Patterns for Systems of systems: The EU IP DANSE approach

Alexandre Arnold, Benoît Boyer, Axel Legay

Contracts and Behavioral Patterns for Systems of systems: The EU IP DANSE approach

Alexandre Arnold
Benoît Boyer
Axel Legay

RESEARCH REPORT
N° 7003
January 2012
Project-Teams Triskell

ISSN 0249-6399 ISRN INRIA/RR--7003--FR+ENG

Contracts and Behavioral Patterns for Systems of systems: The EU IP DANSE approach

Alexandre Arnold[*]
Benoît Boyer[†]
Axel Legay[†]

Project-Teams Triskell

Research Report n° 7003 — January 2012 — 21 pages

Abstract: This report presents some of the results of the first year of DANSE, one of the first EU IP projects dedicated to System of Systems. Concretely, we offer a tool chain that allows to specify SoS and SoS requirements at high level, and analyse them using powerful toolsets coming from the formal verification area. At the high level, we use UPDM, the system model provided by the british army as well as a new type of contract based on behavioral patterns. At low level, we rely on a powerful simulation toolset combined with recent advances from the area of statistical model checking. The approach has been applied to a case study developed at EADS Innovation Works.

Key-words: SoS, System of Systems, Contract, specification, OCL, SysML, UPDML, Model Checking, Modeling, Statistical Model Checking, Verification

[*] Innovation Works EADS, `Alexandre.Arnold@eads.net`
[†] INRIA - Rennes, `First.Last@inria.fr`

Contracts and Behavioral Patterns for Systems of systems: The EU IP DANSE approach

Résumé : Ce document présente les résultats de la première année du projet Danse, un des premiers projets IP de recherche portant sur les systèmes de Systèmes (SoS en anglais). Concrètement, une chaîne d'outils a été développée de façon à spécifier à "haut niveau" puis à analyser formellement un SoS avec un ensemble prérequis, c'est à dire un ensemble de propriétés que le SoS doit valider. L'analyse du SoS repose sur l'utilisation d'outils efficaces de vérification formelle. À "haut niveau", le SoS est décrit en UPDML, le langage dédié à la modélisation des SoS que l'armée britannique à développé, alors que les prérequis du SoS sont spécifiés au moyen d'un langage de contrats décrivant les comportements attendus du systèmes. Cette spécification "haut niveau" est compilée en une représentation "bas-niveau" qui est simulée et analysée grâce à des outils émanant des récentes techniques de vérification statistique. L'approche a en particulier été appliquée sur un cas d'étude développé par EADS Innovation Works.

Mots-clés : SoS, System de Systems, Contrat, Spécification formelle, OCL, SysML, UPDML, Vérification, Modélisation, Statistique

1 Introduction

While SysML [17], the Systems Modeling Language derived from UML [16], has been widely adopted for Systems Engineering applications, the specificities of Systems of Systems (SoS) fostered the creation of further customizations. The Unified Profile for DoDAF and MoDAF (UPDM) [18], based on the US and UK military architectural framework, is one of them and is used on a regular basis in SoS Engineering.

Specific extensions of SysML/UPDM are considered in Danse [8], one of the first European project aiming at developing a methodological and technical framework for SoS Engineering with associated tool support. This framework shall support the SoS architect from the modeling activities to the analysis phase (abstraction, simulation, formal verification), especially by providing concrete solutions to address common SoS issues: constant evolution of a large-scale SoS and its stakeholders' needs, unexpected emergent behaviors, limited awareness of the global situation...

In the frame of Danse, we extend the language of SysML/UPDM to add formalized requirements for an SoS. Formalizing the SoS goals makes it possible to verify them automatically (with an adjustable probability) using a statistical model checker such as Plasma-Lab [11, 12] in combination with a simulation platform such as DESYRE. The challenge is to propose a high-level formal language that is directly usable by an SoS architect, while being still automatically translatable to the expressive low-level specification of the model checker, in a similar way to editors like IBM Rhapsody that could make an executable specification (FMI) out of a high-level formalism (SysML/UPDM behavioral diagram).

For our purpose, the low-level specification is the Bounded Linear Temporal Logic (B-LTL), an extension of the Linear Temporal Logic in which each temporal operator is bound by a temporal constant. This logic is expressive enough to cover a large set of properties and to write static as well as behavioral SoS goals. But this logic is defined using the standard temporal operators, which are quite low-level: defining complex properties often requires to interlock several layers of nested operators. Writing or understanding such formulas is difficult, thus error-prone, and does not fit at all with our target of a clear and simple specification language.

So we propose in this paper the very first contract language for SysML/UPDM, defined using a strong B-LTL based semantics, but close to hand written English requirements for SoS on the surface. This language of goal formalisation, which is developed in the scope of the Danse project, is called the Goal and Contract Specification Language (GCSL).

GCSL makes use of the Object Constraint Language (OCL), a formal language by the Object Management Group (OMG) [14] used to describe static properties on UML models, thus also on SysML/UPDM ones. OCL can be used for a number of different purposes, but especially as a model-based query language and for writing expressions, which perfectly suits our needs here. GCSL also reuses the Contract Specification Language (CSL) [23], developed in the previous SPEEDS European project [24], which comes with convenient temporal patterns. The three key elements required for the formalization of behavioral goals and the way we address them in our approach are (1) being able to refer to model elements: use of the same names as in the SysML/UPDM model, (2) being able to write static properties about them: use of OCL and (3) being able to integrate these expressions inside behavioral patterns: use of CSL patterns.

After a short description of the SoS modeling in Section 2, this paper presents in Section 3 the GCSL based on the semantics of UPML modeling, thus we show how to translate the properties into B-LTL formulas (Section 4) into order to check them using the statistical model checking framework for SoS (Section 5). Finally, Section 6 illustrates the approach applied to the case study of Danse.

2 System of Systems Modeling

Overview of SysML/UPDM

SysML [17] is a general-purpose modeling language defined as an extension of a subset of the Unified Modeling Language (UML) [16] using UML's profile mechanism. SysML is used for Systems Engineering applications, whereas UML is more targeted towards object-oriented Software Engineering. A large set of diagrams is provided with SysML to model a system's requirements, structure (e.g. block definition diagram, internal block diagram), behavior (e.g. state machine, activity diagram), etc.

Using the same UML's profile mechanism, another language built on top of UML/SysML has de facto become a standard for SoS architects: UPDM. This profile is the result of the unification effort of the US Department of Defense and the UK Ministry of Defense architecture frameworks and associated meta-models. It adds a layer of new meta-objects that are typically (but not exclusively) used in the context of military SoS, as well as a significant amount of predefined views (e.g. system views, operational views, capability views) which help splitting the whole modeling activity in smaller tasks.

The executable part of a UPDM modeling can be compiled into a program based on the Functional MockUp Interface (FMI) [21] that defines a standardized interface used in simulations of complex systems. In Danse, the SoS is compiled into FMI program and executed by the simulation engine DESYRE [7]. This whole FMI program can be considered as a state transition system, e.g. the formal semantics on which we will use to propose our language. The states denote the global states of the SoS, e.g. the result of collecting the internal states of each constituent in the SoS. The transitions denote the actions and events that occur in the SoS and eventually modify the internal state of some system constituents and thus, the global state of the system.

Definition 1 (State Transition Systems)**.** *Let X be a set of variables that are mapped to the values of $\mathbb{D}$, the set of all possible values. We define S a set of states. Each state s is characterized by a mapping $\mu_s : X \to \mathbb{D}$ such that the valuation of any variable $y \in X$ in the state s is $\mu_s(y)$. We thus define a transition system as a 4-tuple $< S, s_0, R, \{\mu_s \mid s \in S\} >$ such that*

- *$s_0 \in S$ contains the initial states of the system*
- *$R \subseteq S \times S$ is the transition relation. We use the more convenient notation $s \to s'$ to denote $(s, s') \in R$.*

All valid execution (or run) of a transition system is a sequence of states led by the R from any initial state. A run of length n will denoted as $\pi = s_0; s_1; s_2; \ldots; s_n$ where $s_0 \in I$ and $s_i \to s_{i+1}$ holds for $0 \leq i < n$. Each transition system has a global clock, which is denoted by the variable t. We note $t_i = \mu_{s_i}(t)$, the observed time value of t when the executed system reaches the state s_i. For any execution path the system is in state s_i when $t_i \leq t < t_{i+1}$ and the evolution of the time is monotonically increasing, e.g. $t_i < t_{i+1}$.

In the first year of the Danse project we limit ourselves to systems of systems who environment's behaviors are fully known in advance (hence representable via state transition systems), like it is the case for most of adaptive systems studied in the litterature [26, 10, 3, 9]. The reason is that this corresponds to the current possibilities of the UPDM. In future work, we will study more complex aspects such as unknown environment, hence more complex dynamicity features. For this, we will first have to consider extension of the UPDM model.

Stochastic aspects of the model

Stochastic modeling is a way to describe behaviors that are not deterministic by nature, or to abstract a behavior that is simply too complex to be modeled explicitly (as white box). So it is typically very useful in a SoS context. Behavioural modeling in SoS examples such as an Emergency Response to a city fire typically shows numerous attributes/parameters that would not be deterministic, such as the time between two fires or the duration of an action performed by a human.

A first proposal of how to put stochastic data in the SysML/UPDM model has been integrated into the DANSE project. It is based on a set of attribute stereotypes that can be applied to any block attribute. This idea is close to the suggestion of the non-normative distribution extensions made in appendix of the SysML 1.3 specification, but adds the possibility to regenerate a distribution-based random value whenever needed (and not only at initialization). This addition is important because even the same person does never perform the same task in the exact same amount of time, so that the duration of the task shall be recalculated every time.

Adding stochastic data to the SoS model implies of course that each simulation is likely to generate a different trace than the previous ones, and as a consequence that one run will not be enough to verify whether the SoS meets its requirements or not. This is why being able to automate this verification process in a mathematical way (provided the requirements are formalised) is a great support for the SoS architect when assessing a candidate architecture.

Since the SoS we consider exhibit some stochastic behaviors, each run has an associated probability of being executed. This probability is given by an unkonwn distribution due to the high complexity of the model: a system is designed by the paralell composition of components that may have a stochastic behavior.

3 A Contract Language for UPDM/SysML Requirements

Before defining the new contract language, we introduce the notion of contract for the SoS formalized as a stochastic state transition systems.

Definition 2 (Contracts for State Transition Systems)**.** *A contract is defined as a pair (A, P) where A and P are respectively called the Assumption and the Promise. Considering a state transition system, A and P are properties about the execution of the system. Thus, a contract for the system specifies what the system shall ensure the promise when the system shall satisfy the assumption. The notation $Sys \models (A, P)$ means that the contract (A, P) is satisfied by the system Sys. Relying on the state transition system semantics, the satisfaction of a contract is*

$$Sys \models (A, P) \quad \textit{iff} \quad \forall \pi,\ \pi \models A \Rightarrow \pi \models P$$

where π is a valid run of Sys and $\pi \models A$ (or P) means the run π satisfies the assumption A (the promise P resp.).

For stochastic systems, it is generally more meaningful to quantify how a system satisfies a contract: this valuation is given by the probability that the system satisfies the contract. Intuitively, if the distribution to execute each run of a given stochastic system is known, the probability that this system satisfies the contract is the sum of the probabilities of all the runs that satisfy the contract (see Section 5).

Definition 3 (Contracts for Stochactic State Transition Systems)**.** *Let be a stochastic system Sys, a contract (A, G) and a threshold value $k \in [0..1]$. For the system Sys, we now consider the contract $P_{\sim k}(A, G)$, where $\sim \in \{<, \leq, =, \geq, >\}$ and $0 \leq k \leq 1$. The contract is satisfied if and only if the relation holds, e.g. if the probability p of $Sys \models (A, G)$ satisfies the relation $p \sim k$.*

In this work, for efficiency reasons, we decided to estimate the probability p using statistical model checking rather than computing it with a numerical approach such as Prism [20]. Another reason to use SMC is that it relies on monitoring traces, hence it allows to verify properties that cannot be expressed in classical logics. In this paper, this aspect will not be explored, but it is a main topic of Danse. SMC consists in verifying the property (here contract) against several simulations of the system. Then, an algorithm from the statistic area is used to estimate the probability to satisfy the property. The contract to monitor is translated into a B-LTL formula (see Section 4) that characterizes a set of simulation traces. Thus, the simulation monitoring consists of observing each simulation to decide if the B-LTL formulas holds or not.

We now introduce the language to express the assumptions and promises dedicated to the System of Systems. The GCSL syntax for patterns is a combination of the Object Constraint Language (OCL) and the contract patterns of the CSL Ã la "SPEEDS" [24]. The SPEEDS contract specification patterns are introduced in the SPEEDS Deliverable D.2.5.4 "Contract Specification Language (CSL)" [23] and used to give a high-level specification of real-time components. They have been introduced to enable the user to reason about event triggering that are equivalently replaced in Danse by property satisfaction. The properties handled by these patterns are about the state of a SoS. We use OCL to specify these state properties. This language allows to build some behavioral properties to express some temporal relations about facts or events of the system denoted by the state properties. It is sufficiently powerful to describe precisely a state of a SoS. Here, we will only consider a subset of the OCL language, but it is not unrealistic to consider a larger subset of OCL to describe the requirements. We restrict the language here to express some properties that can be verified using the SMC techniques applied to SoS's.

We briefly recall the notion of Collection that we will use in the rest of the paper.

Collections in OCL: in OCL, it is the usual way to define some properties about set of elements in a system. Considering a SoS as a state transition system, the root identifier `SoS` denotes σ the state currently reached by the SoS. The collections allow to handle some set of instances of components in the current state σ. A collection built over the state σ can be viewed as a projection of σ: it is defined by selecting some component instances or attribute values in the state σ.

For example, the expression `SoS.itsFireStations` denotes collection of all the instances of type `FireStation` at state σ. OCL defines some operators that can be applied to any collection: `SoS. itsFireStations` $\rightarrow$ `size`() counts the number of instances of type `FireStation`. The most important feature of the collection is the predicates we can define using quantification:

- `SoS.coll` $\rightarrow$ `forAll`(`x`|ϕ(`x`)) denotes that for all element `x`, which belongs to the collection `SoS.coll`, the property $\phi(x)$ holds.
- `SoS.coll` $\rightarrow$ `exists`(`x`|ϕ(`x`)) denotes for that there exists one element `x`, which belongs to the collection `SoS.coll`, the property $\phi(x)$ holds.

State properties in OCL

Originaly introduced to supplement UML, the Object Constraint Language (OCL) [15] is particularly adapted to describe the internal state of a component. The Object Constraint Language is a rather simple-to-write, yet formal text language that provides constraint and object query expressions based on any meta-model, so for instance the SysML/UPDM ones. It has a concise notation for accessing, collecting, filtering and evaluating model elements. More generally, it allows to write invariants on a model, that we use in our approach to write the static properties that we insert in the behavioral contracts. As we will see in the following paragraphs, we also pushed the concept further by sometimes embedding a CSL pattern inside an OCL-like expression, when we want to state that the pattern shall hold for some or all elements in a set. We recall some OCL notations used in the rest of the paper, but the reader can find the whole specification in [15]. Components store internal values into attributes that are denoted by the standard dot-syntax. For example, the number of people in the district 1 at σ, the state reached by the SoS, is `district₁.population`. More particular to OCL, it is also possible to define a collection of attributes using the same syntax: the expression `SoS.itsDistricts.population` $\rightarrow$ `sum()` denotes the number of total people. For the sake of clarity in the rest of the paper, we only focus on the `Collection` type without considering all its refinements (`Set`, `Ordered Sets`, ...), and the subset of Boolean and arithmetic expressions over the attributes of the SoS' component instances.

The behavioral patterns

The semantics of the patterns is based on the satisfiability of any predicate on the whole set of execution paths that defines the pattern, which the definition of the following patterns are based upon. Consider the state property Ψ and a time value sequence $t_0, t_1, \ldots, t_n$ that defines the state sequence $\sigma_0, \sigma_1, \ldots, \sigma_n$ such that t_i is the time value where the system reaches σ_i. In other words, the system is in state σ_i when $t_i \leq time < t_{i+1}$.

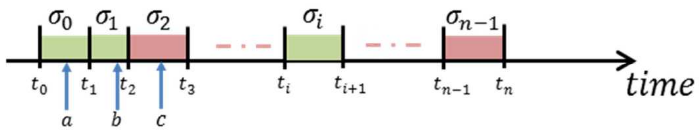
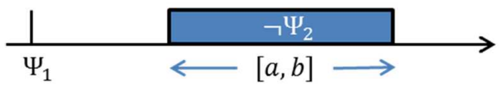


Figure 1: Satisfaction of Ψ during an execution path.

Figure 1 illustrates the satisfaction of a state property Ψ, e.g. the green state σ_0, σ_1 and σ_i are the only states of the sequence that satisfy Ψ. It means that Ψ holds when $time \in [t_0, t_2) \cup [t_i, t_{i+1})$. We observe that Ψ holds continuously for σ_0, σ_1, hence the number of occurrences where Ψ holds is 2 during the time interval $[t_0, t_n)$. If we finally consider any the time ticks a, b and c, Ψ holds during $[a, b]$ but does not during $[a, b]$ nor $[b, c]$ and the occurrence number of Ψ is 1 in $[a, b]$, $[a, c]$ or $[b, c]$.

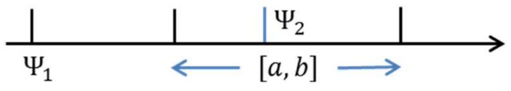
We define some selected patterns, but the more exhaustive list can be found in Appendix B. These patterns proved very useful for SoS applications. We assume that Ψ and Ψ_i are state properties and a, b, c are time constants such that the time intervals defined in the patterns are valid.

whenever Ψ_1 occurs Ψ_2 does not occur during following $[a, b]$



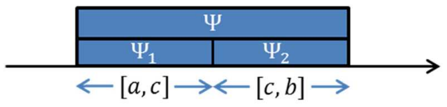
This pattern specifies that Ψ_2 is never satisfied during the relative interval $[a, b]$ after Ψ_1, i.e. $\neg\Psi_2$ holds during $[a, b]$. By relative we means that when Ψ occurs at t, the relative interval corresponds to $[a + t, b + t]$.

Whenever Ψ_1 occurs Ψ_2 occurs within $[a, b]$



The constraint Ψ_2 must be satisfied at least once during $[a,b]$ after Ψ_1.

Ψ **during** $[a,b]$ **implies** Ψ_1 **during** $[a,c]$ **then** Ψ_2 **during** $[c,b]$



Whenever Ψ holds during $[a,b]$ there exists a split at c of $[a,b]$ such that Ψ_1 holds during $[a,c]$ then Ψ_2 holds during $[c,b]$.

The CSL patterns are originally designed to specify the behavior of any 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. Let us consider a SoS with a set of components `District` and two `District` properties Psi_1 and Ψ_2 in OCL. The patterns allow to express the behavioral property for some explicit component, e.g. **Whenever** $[\Psi_1(\texttt{district}_1)]$ **occurs** $[\Psi_2(\texttt{district}_1)]$ **occurs within** $[a,b]$, it is not possible to generalize the behavioral property to any `District` of the system, e.g. a property like "For all `district`, **Whenever** $[\Psi_1(\texttt{district})]$ **occurs** $[\Psi_2(\texttt{district})]$ **occurs within** $[a,b]$".

To overcome this important limitation, we extend the proposed grammar (see Appendix B) by overlapping the patterns with the OCL collection predicates, e.g. `forAll(x|...)` and `exists(y|...)`. Then, the generalized behavioral property presented below is now:

$$\texttt{SoS.itsDistricts} \rightarrow \texttt{forAll}(\texttt{district} \mid \\ \textbf{Whenever } [\Psi_1(\texttt{district})] \textbf{ occurs } [\Psi_2(\texttt{district})] \textbf{ occurs within } [a,b])$$

The root collection `SoS.itsDistricts` is defined on the initial state σ_0 of the SoS. In SoS, the initial state is describe by the Internal Block Diagram that is defining the initial state of each component. Using these OCL predicates for quantify the patterns keep the language not so different in comparison with the original OCL, except we restrict the nesting capability. The OCL syntax allows to nest the quantification without any limit. If there is no theoretical reasons to have limit, we impose a limit of 2 nested quantifications in our language. From the verification point of view, a behavioral formula with more nested quantifications is not practically check-able. Moreover, we never need more to express the requirements of CEA incubator in Danse. So we assume in the next, that the patterns have are of the form $\texttt{SoS.coll}_1 \rightarrow \texttt{forAll}(\texttt{x}|\texttt{SoS.coll}_2 \rightarrow \texttt{forAll}(\texttt{y}|\dots\textbf{Pattern}(\texttt{x},\texttt{y})\dots)$, where **Pattern** is any behavioral pattern.

Another important limitation of this combination OCL + patterns is the inability of express property about cumulative values during an execution path: to solve this problem we introduce the path operators `mean`(), `sum`(), `prod`() to denote the value of a numerical expression: for example, $\texttt{mean}(\texttt{district}_1\texttt{. population})$ denotes the average value of the attribute $\texttt{district}_1\texttt{.population}$) computed with the values obtained of the different state of the path.

Examples of Requirements

Table 1 illustrates the kind of properties that we will express with our language. We use syntactic coloring to distinguish the different parts 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. These requirements show the capabilities of our language using different requirements of this use case. Whereas the requirement 1 is purely structural, the requirements 2 and 3 are relative to the execution of the SoS: the first one is written using strictly OCL, the second one shows the cumulative operators we introduced and the third one is defined with a behavioral pattern. The presented requirements are contracts without assumption or, more precisely, they are contracts with an assumption that is implicitly "true".

"Any district cannot have more than 1 fire station, except if all districts have at least 1" **SoS.itsDistricts→exists(**district | district.**containedFireStations→size()** $>$ **1) implies** **SoS.itsDistricts→forAll(**district | district.**containedFireStations→size()** $\geq$ **1)**
"The mean city area under fire shall be less than 0.01%" **mean(SoS.itsDistricts.fireArea→sum())** ≤ 0.0001
"The fire fighting cars hosted by a fire station shall be used all simultaneously at least once in 6 months" **SoS.itsFireStations→forAll(**fireStation | **Whenever** [fireStation.**hostedFireFightingCars→exists(**ffCar | ffCar.**isAtFireStation)**] **occurs,** [fireStation.**hostedFireFightingCars→forall(**ffCar | ffCar.**isAtFireStation** = **false)**] **occurs within [6 months])**

Table 1: Examples of Requirements formulated in the CAE incubator

The proposed language ,composed by 11 SPEEDS patterns, is sufficient expressive to formalize the behavior from 15 requirements identified in CAE incubator. This list of patterns can be easily extended for the future needs, but the experiments conducted in SPEEDS and DANSE show it covered all the requirements to be expressed.

4 Translating Contracts into Bounded-LTL Formulas

Bounded Linear Temporal Logic

As said previously, the Bounded Linear Temporal Logic (B-LTL) is an extension of the Linear Temporal Logic (LTL) [5] in which each temporal operator is bound by a temporal constant. This Logic is such expressive that it covers precisely a large set of properties. It is particularly adapted to Statistical Model Checking (SMC) [25, 22]. The SMC principle is to monitor some simulations in order to check a B-LTL property and use the results from the statistics area (sequential hypothesis testing or Monte Carlo simulation) in order to decide whether the system satisfies the B-LTL property or not with some degree of confidence. Since the conducted simulations are finite, the infinite path semantics of LTL has no sense, whereas checking B-LTL formulas does.

The formulas are built using the standard logic connectors $\wedge$, $\vee$, $\implies$, $\neg$ and the common temporal modalities G, F, X, U over some atomic propositions. Each temporal modality is indiced by a bound defining the length of the run on which the formula must hold. The validation of a B-LTL formula against an execution path has a meaning only if the length of this path is enough to reach all bounds constituting the formula.

The atomic propositions used in the B-LTL formulas are build using some state predicates or run predicates. These predicates only require to be decidable for a given input, e.g. a state or a run section, and we assume this decision to be performed by an external procedure. Considering $\pi = s_0 s_1 \ldots s_n$ a finite run of a transition system and Φ a B-LTL property, $\pi \models \Phi$ means that the run π satisfies the property Φ. The suffix $s_i s_{i+1} \ldots s_n$ of π is noted π^i. Assuming $k > 0$, a run $\pi = s_0 s_1 \ldots s_n$, a state predicate P and a run predicate Q, the satisfiability of the B-LTL formulas Φ, Φ_1 and Φ_2 is defined in Table 2.

$\pi \models F_{\leq k}\Phi$	$\equiv$	$\exists i,\ t_0 \leq t_i \leq t_0 + k$ and $\pi^i \models \Phi$	
$\pi \models G_{\leq k}\Phi$	$\equiv$	$\forall i,\ t_0 \leq t_i \leq t_0 + k$ and $\pi^i \models \Phi$	
$\pi \models X_{\leq k}\Phi$	$\equiv$	$\forall i,\ i = max\{j \mid t_0 \leq t_j \leq t_0 + k\}$ and $\pi^i \models \Phi$	
$\pi \models \Phi_1\ U_{\leq k}\Phi_2$	$\equiv$	$\exists i,\ t_0 \leq t_i \leq t_0 + k$ and $\pi^i \models \Phi_2$ and $\forall j, 0 \leq j \leq j,\ \pi^j \models \Phi_1$	
$\pi \models \Phi_1\ W_{\leq k}\Phi_2$	$\equiv$	$\pi \models (\Phi_1\ U_{\leq k}\Phi_2) \vee G_{\leq k}\Phi_2$	
$\pi \models \Phi_1 \implies \Phi_2$	$\equiv$	$\pi \models \neg\Phi_1 \vee \Phi_2$	
$\pi \models \Phi_1 \vee \Phi_2$	$\equiv$	$\pi \models \Phi_1$ or $\pi \models \Phi_2$	
$\pi \models \Phi_1 \wedge \Phi_2$	$\equiv$	$\pi \models \Phi_1$ and $\pi \models \Phi_2$	
$\pi \models \neg\Phi$	$\equiv$	$\pi \not\models \Phi$	
$\pi \models P$	$\equiv$	$P(s)$ holds	checked by an external procedure
$\pi \models Q$	$\equiv$	$Q(\pi)$ holds	
$\pi \models true$			
$\pi \not\models false$			

Table 2: Semantics of B-LTL

Example 1 (Example of B-LTL formula)**.** *Let us consider the formula $G_{\leq 5}(A \implies X_{\leq 1}F_{\leq 2}B)$ where A and B are state propositions and an execution path π such that A and B hold as illustrated below:*

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 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 [19].

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' and P' such that A' (or P) and A (P' resp.) are equivalent for any k-bounded simulations, then the B-LTL formula $A' \implies P'$ is equivalent to the contract (A, P) for any k-bounded simulations.*

The proof is a trivial consequence of Definition 2 written using B-LTL. Moreover, extending the translation to a stochastic contract is natural. The pair (A, P) of any stochastic contract is similarly treated.

OCL quantification translation

OCL expressions occur at two levels within a pattern: as atomic propositions to define a state condition and as quantifications. The first case will be directly treated by an external 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, adding quantifications to the logic increases significantly the complexity of the satisfiablity decision.

Moreover, the instances of each component type are statically specified in the Internal Block Diagram (IBD) by the SoS architect. In the CAE incubator, the IDB is named `idbFireEmergency` and gives the list of all system constituents instantiated in the SoS: 10 districts, 1 fire headquarter, 3 fire stations and 7 fire fighting cars shared by the fire stations, etc. Since the number of constituents is known and finite in the SoS, any universal quantification (or an existential quantification) over a collection can be interpreted as a conjunction (a disjunction resp.). Using the CAE incubator and assuming a valid property ϕ for the fire stations, the property $\texttt{SoS.itsFireStations} \rightarrow \texttt{forAll}(\texttt{x}|\ \phi(\texttt{x}))$ is equivalent to $\phi(\texttt{fireStation}_1) \wedge \phi(\texttt{fireStation}_2) \wedge \phi(\texttt{fireStation}_3)$.

In cases where ϕ contains also a quantification, ϕ must also be unfold. The generalization of the process is recursively defined as:

$$\begin{cases} unfold\big(\texttt{coll} \rightarrow \texttt{forAll}(\texttt{x}|\phi(\texttt{x})\big) & = \bigwedge_{\texttt{x}\in\texttt{coll}} unfold\big(\phi(\texttt{x})\big) \\ unfold\big(\texttt{coll} \rightarrow \texttt{exists}(\texttt{x}|\phi(\texttt{x})\big) & = \bigvee_{\texttt{x}\in\texttt{coll}} unfold\big(\phi(\texttt{x})\big) \\ unfold\big(\texttt{expr}) & = \texttt{expr}, \quad \text{otherwise} \end{cases}$$

where `coll` is an OCL collection and `expr` any other valid expression of the contract language.

Pattern translation

For the purpose of translation to B-LTL, we assume that the constant k, is an additional parameter given by the user: it corresponds to the execution time for which we expect the property hold. Whenever an unbound pattern to write a property like "Always ... " is meaning full for the specification, statistical model checking still checks the B-LTL property against a simulation that is finite: k is used to replace the implicit unbound value of the property. 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 simulation (and so the constant k) is reached.

Proposition 2. *Assuming Ψ, Ψ_1 and Ψ_2 denote some state propositions nested in a pattern P, e.g. OCL propositions, and a constant $k > 0$ then, for any run of length k, there exists a B-LTL formula equivalent to P. Table 3 summarizes the valid B-LTL translations.*

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 path that are necessarily finite, for practical reasons (termination). Thus, the infinite bound is replaced by the user constant k provided for the verification.

	Pattern	B-LTL translation	Consistency Condition
	BASIC B-LTL PATTERNS WITH ABSOLUTE INTERVALS		
1	**always** Ψ	$G_{\leq k}\Psi$	-
2	**whenever** Ψ_1 **occurs** Ψ_2 **holds**	$G_{\leq k}(\Psi_1 \implies \Psi_2)$	-
3	Ψ_1 **implies** Ψ_2 **during following** $[a, b]$	$X_{\leq a}G_{\leq a-b}(\Psi_1 \implies \Psi_2)$	$a \leq b$
4	Ψ_1 **during** $[a, b]$ **raises** Ψ_2	$(X_{\leq a}G_{\leq a-b}\Psi_1) \implies X_{\leq b}\Psi_2$	$a \leq b \leq k$
5	Ψ **during** $[a, b]$ **implies** Ψ_1 **during** $[a, c]$ **then** Ψ_2 **during** $[c, b]$	$X_{\leq a}G_{\leq b-a}\big(X_{\leq a}G_{\leq c-a}(\Psi_1) \wedge X_{\leq c}G_{\leq b-c}(\Psi_2)\big)$	$a \leq c \leq b$
	EXTENDED B-LTL PATTERNS WITH ABSOLUTE INTERVALS		
6	Ψ_1 **occurs** n **times during** $[a, b]$ **raises** Ψ_2	$occ(\Psi_1, a, b) \geq n \implies X_{\leq b}F_{\leq k-b}\Psi_2$	$a \leq b \leq k$
7	Ψ **occurs at most** n **times during** $[a, b]$	$occ(\Psi, a, b) \leq n$	$a \leq b$
	BASIC B-LTL PATTERNS WITH SLIDING INTERVALS		
8	**whenever** Ψ_1 **occurs** Ψ_2 **holds during following** $[a, b]$	$G_{\leq k-b}(\Psi_1 \implies X_{\leq a}G_{\leq b-a}\Psi_2)$	$a \leq b \leq k$
9	**whenever** Ψ **occurs** Ψ_1 **implies** Ψ_2 **during following** $[a, b]$	$G_{\leq k-b}(\Psi \implies X_{\leq a}G_{\leq b-a}(\Psi_1 \implies \Psi_2))$	$a \leq b \leq k$
10	**whenever** Ψ_1 **occurs** Ψ_2 **does not occur during following** $[a, b]$	$G_{\leq k-b}(\Psi_1 \implies X_{\leq a}G_{\leq b-a}\neg\Psi_2)$	$a \leq b \leq k$
11	**whenever** Ψ_1 **occurs** Ψ_2 **occurs within** $[a, b]$	$G_{\leq k-b}(\Psi_1 \implies X_{\leq a}F_{\leq b-a}\Psi_2)$	$a \leq b \leq k$

Table 3: Pattern mapping

Extended B-LTL patterns The patterns 6 and 7 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 $occ(\Psi, a, b)$ that counts the number of times where Ψ is satisfied and compare it to the value n. We use similarly some external treatment to evaluate the operators $sum()$, $mean()$, ...that compute a accumulated value of an expression during a time interval.

Sliding intervals the interval $[a, b]$ to consider is located after time t at which the first part of the pattern "**Whenever** Ψ **occurs**" is satisfied: this sliding interval in pattern 8 is encoded as the property Ψ_2 holds during the duration $b - a$ after a units of time after we observe Ψ_1 is true.

Illustration of the full translation

We illustrate the translation for the third requirement in Table 1.

SoS.itsFireStations→forAll(fireStation |
Whenever [fireStation.**hostedFireFightingCars→exists(isAtFireStation)**] **occurs,**
[fireStation.**hostedFireFightingCars→forall(isAtFireStation** = **false)**]
occurs within [6 months])

Assuming that we have the time bound $k \geq 6months$ the pattern is translated to the B-LTL formula following the rule 12 in Table 3:

$$\phi = \begin{cases} G_{\leq k-6months}(\Psi_1(\texttt{fireStation}_1) \implies X_{\leq 0}F_{\leq 6months}\Psi_2(\texttt{fireStation}_1)) \\ \qquad\qquad\qquad\qquad\qquad \bigwedge \\ G_{\leq k-6months}(\Psi_1(\texttt{fireStation}_2) \implies X_{\leq 0}F_{\leq 6months}\Psi_2(\texttt{fireStation}_2)) \\ \qquad\qquad\qquad\qquad\qquad \bigwedge \\ G_{\leq k-6months}(\Psi_1(\texttt{fireStation}_3) \implies X_{\leq 0}F_{\leq 6months}\Psi_2(\texttt{fireStation}_3)) \end{cases}$$

where $\Psi_1(\texttt{fireStation}_\texttt{i})$ and $\Psi_2(\texttt{fireStation}_\texttt{i})$ correspond to the OCL expressions in brackets, e.g. `fireStation.hostedFireFightingCars` $\rightarrow$ `exists(isAtFireStation)` and `fireStation .hostedFireFightingCars` $\rightarrow$ `forall(isAtFireStation = false)`. We notice that the modality $X_{\leq 0}$ could be cleaned in ϕ, but we leave it for the sake of clarity.

As for the OCL quantification at the root of the requirement, we unfold the OCL quantifications that occur in Ψ_1 and Ψ_2. The next table gives the result of this unfolding in $\Psi_1(\texttt{fireStation}_\texttt{i})$ and $\Psi_2(\texttt{fireStation}_\texttt{i})$ for each `fireStation`. Finally, replacing all occurences of $\Psi_1(\texttt{fireStation}_\texttt{i})$ and $\Psi_2(\texttt{fireStation}_\texttt{i})$ in ϕ gives the complete translation in B-LTL.

`Component`	Ψ_1	Ψ_2
`fireStation1`	`fireFightingCar1.isAtFireStation` $\vee$ `fireFightingCar2.isAtFireStation` $\vee$ `fireFightingCar3.isAtFireStation`	$\neg$ `fireFightingCar1.isAtFireStation` $\wedge$ $\neg$ `fireFightingCar2.isAtFireStation` $\wedge$ $\neg$ `fireFightingCar3.isAtFireStation`
`fireStation2`	`fireFightingCar4.isAtFireStation` $\vee$ `fireFightingCar5.isAtFireStation`	$\neg$ `fireFightingCar4.isAtFireStation` $\wedge$ $\neg$ `fireFightingCar5.isAtFireStation`
`fireStation3`	`fireFightingCar6.isAtFireStation` $\vee$ `fireFightingCar7.isAtFireStation`	$\neg$ `fireFightingCar6.isAtFireStation` $\wedge$ $\neg$ `fireFightingCar7.isAtFireStation`

5 Statistical Model Checking of SoS Contracts

The interest of SMC [25, 22] is to propose an alternative to the approach of the classical model checking [1, 5]. By using results from the statistic area (including sequential hypothesis testing or Monte Carlo simulation) in order to decide whether the system satisfies the property or not with some degree of confidence, SMC avoids an exhaustive exploration of the state-space of the model that generally does not scale up. It has already successfully experimented in biology area [6, 13, 20], software engineering [4] as well as industrial area[2] More recently, in DANSE [8], we adapt the SMC techniques to treat large heterogeneous systems like Systems of Systems. Among them, one finds systems integrating multiple heterogeneous distributed applications communicating over a shared network. We proposed to extend UPDM specification - the SoS specification - with some requirements that the SoS must satisfy. These requirements, are specified with the contract language we specially designed for the SoS's. These goals are viewed as behavioral objectives that support the SoS architect in assessing different strategies and finding the best ones. As shown in Figure 2, these contracts are compiled into B-LTL formulas that are verified against the SoS (whose constituent systems are compiled into FMI executables) using the Statistical Model Checker Plasma-Lab [11] combined with the efficient simulation engine DESYRE developed by Ales [7]. The SMC tool-chain gives an estimation of the satisfiability of the contract by the SoS. The different results help the SoS architect to make good decisions about how to optimize the SoS strategies.

The main algorithm we used in DANSE is the Monte Carlo algorithm. This algorithm estimates the probability that a system *Sys* satisfies a B-LTL property *P* by checking *P* against a

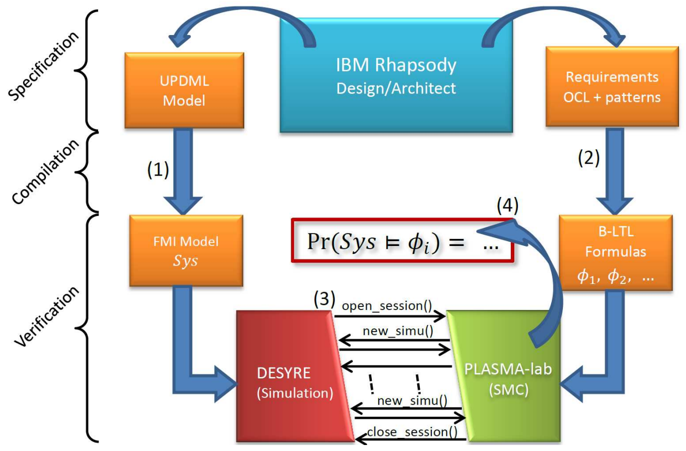


Figure 2: The SMC process in DANSE

set of N random executions of SyS. The estimation $\hat{p}$ is given by

$$\hat{p} = \frac{\sum_1^N f(ex_i)}{N} \quad \text{where } f(ex_i) = 1 \text{ if } ex_i \models P,\ 0 \text{ otherwise}$$

Using the formal semantics of B-LTL, each execution trace is monitored in order to check if P is satisfied or not. The accuracy of the estimation increases with a bigger number of monitored simulations.

Plasma-Lab[11] implements a set of tools from the statistical area to perform the SMC. It provides some engines for simulating biologic models, models written in the Prism language [20], but it has also the capabilities to drive an external engine to perform the simulations like MathLab, SciLab, or DESYRE.

6 Illustration using the CAE incubator

In the frame of the DANSE project, the Concept Alignment Example (CAE) is a fictive SoS example inspired by real-world Emergency Response data to a city fire. It has been built as a playground to demonstrate new methods and models for the analysis and visualization of SoS designs. All structural modeling has been performed using UPDM views, and behaviors have been added on a subset of the constituents that we called "CAE incubator", using simple SysML constructs (modeled in state machines) extended by a few stereotypes (e.g. for storing stochastic information).

Behavioral modeling in the CAE incubator is focused on following constituent systems: Fire HQ, Fire Station, Fire Fighting Car and District. The city districts have been added as constituent systems because they play an important role in the SoS: their behavior describes how the fires arise, expand and spread to neighbor districts. In the frame of the CAE, all behaviors are abstracted in state machines using IBM Rhapsody, but it would be possible to use any other language and tool as long as it is compliant with the FMI export format.

The following figure shows the overall architecture of the CAE incubator as well as the behavior of one of the constituent types: a fire fighting car.

We attached to the CAE incubator the following requirement, written accordingly to our proposed formalism:

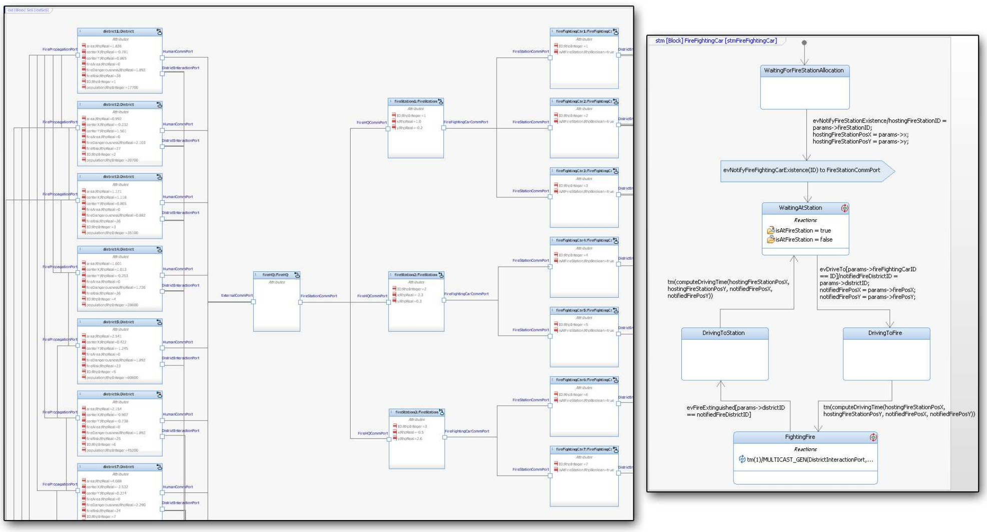

stm [Block] FireFightingCar [stmFireFightingCar]
WaitingForFireStationAllocation
evNotifyFireStationExistence/hostingFireStationID = params->fireStationID; hostingFireStationPosX = params->x; hostingFireStationPosY = params->y;
evNotifyFireFightingCarExistence(ID) to FireStationCommPort
WaitingAtStation
Reactions
isAtFireStation = true
isAtFireStation = false
tm(computeDrivingTime(hostingFireStationPosX, hostingFireStationPosY, notifiedFirePosX, notifiedFirePosY))
evDriveTo[params->fireFightingCarID == ID]/notifiedFireDistrictID = params->districtID; notifiedFirePosX = params->firePosX; notifiedFirePosY = params->firePosY;
DrivingToStation
DrivingToFire
evFireExtinguished[params->districtID == notifiedFireDistrictID]
tm(computeDrivingTime(hostingFireStationPosX, hostingFireStationPosY, notifiedFirePosX, notifiedFirePosY))
FightingFire
Reactions
tm(1)/MULTICAST_GEN(DistrictInteractionPort,...

Figure 3: CAE incubator - architecture example and behavior of a fireman

"The mean city area under fire shall be less than 0.01%"
$\text{mean}(\textbf{SoS.itsDistricts.fireArea}\rightarrow\textbf{sum}()) \leq 0.01\ \%$

As described in the paper, we were able to translate this requirement to the low-level B-LTL specification for the statistical model checker Plasma-Lab and use it in conjunction with the simulation platform DESYRE to assess the probability that this goal is met in the specified time range (the simulation time for each run was 4 months). By choosing the Monte Carlo option, Plasma-Lab was able to give us the following estimation as a result on a given number of runs:

$$\boxed{Prob(\text{mean city area under fire} \leq 0.01\%) \approx 92.3\%}$$

In addition to the computation of the estimated probability that this goal is met on a given number of runs, Plasma-Lab can also compute how many runs are necessary to prove that a given probability threshold is passed by choosing the Chernov option.

Conclusion

This papers presents the results of the very first contract-based language for UPDM/SysML model of SoS we developed in the DANSE project. The SoS model used in the project remains rather simple, but powerful enough to capture behaviors and requirements of a CAE case study developed in collaboration with EADS. Also, we are the first to study the relation between a modeling language used in industry (UPDM) and a verification approach developed by academic.

As a future work, we plan to offer more dynamicity, which we will do by exploiting and extending the work done on adaptative systems [26, 10, 3, 9]. This will also requires to adapt the UPDM framework.

Another interesting future work will be to add more quantitative information directly in the patterns assumption and guarantee. This will permit us to reason on complex problematic such as energy consumption.

All these future extensions will be discussed and designed jointly with the business units as the DANSE partners.

A Grammar

⟨*contract*⟩ ::= ⟨*viewpoint-id*⟩+ ‘contract’ ⟨*identifier*⟩ {‘Assumption:’ ⟨*property*⟩} ? ‘Goal:’
⟨*property*⟩ ‘Confidence:’ ⟨*threshold*⟩

⟨*viewpoint-id*⟩ ::= ‘dynamicity’ | ‘behavior’ | ‘structure’ | ‘safety’ | ‘liveness’ | ...

⟨*threshold*⟩ ::= *Float*‘%’ | ⟨*probability*⟩

⟨*probability*⟩ ::= $x,\ x \in (0;1]$

⟨*property*⟩ ::= ⟨*OCL-coll*⟩ ‘->forAll(’⟨*variable*⟩ ‘|’ ⟨*pattern*⟩‘)’
| ⟨*OCL-coll*⟩ ‘->exists(’⟨*variable*⟩ ‘|’ ⟨*pattern*⟩‘)’
| ⟨*OCL-prop*⟩
| ⟨*pattern*⟩

⟨*pattern*⟩ ::= ‘whenever’ ‘[’⟨*prop*⟩‘]’ ‘occurs’ ‘[’⟨*prop*⟩‘]’ ‘holds’ ‘during’ ‘following’ ‘[’⟨*int*⟩‘]’
| ‘whenever’ ‘[’⟨*prop*⟩‘]’ ‘occurs’ ‘[’⟨*prop*⟩‘]’ ‘implies’ ‘[’⟨*prop*⟩‘]’ ‘during’ ‘following’ ‘[’⟨*int*⟩‘]’
| ‘whenever’ ‘[’⟨*prop*⟩‘]’ ‘occurs’ ‘[’⟨*prop*⟩‘]’ ‘does’ ‘not’ ‘occur’ ‘during’ ‘following’ ‘[’⟨*int*⟩‘]’
| ‘whenever’ ‘[’⟨*prop*⟩‘]’ ‘occurs’ ‘[’⟨*prop*⟩‘]’ ‘occurs’ ‘within’ ‘[’⟨*int*⟩‘]’
| ‘[’⟨*prop*⟩‘]’ ‘during’ ‘[’⟨*int*⟩‘]’ raises ‘[’⟨*prop*⟩‘]’
| ‘[’⟨*prop*⟩‘]’ ‘occurs’ ‘[’ℕ‘]’ times during ‘[’⟨*int*⟩‘]’ ‘raises’ ‘[’⟨*prop*⟩‘]’
| ‘[’⟨*prop*⟩‘]’ ‘occurs’ ‘at’ ‘most’ ‘[’ℕ‘]’ ‘times’ ‘during’ ‘[’⟨*int*⟩‘]’
| ‘[’⟨*prop*⟩‘]’ ‘during’ ‘[’⟨*int*⟩‘]’ ‘implies’ ‘[’⟨*prop*⟩‘]’ ‘during’ ‘[’⟨*int*⟩‘]’ ‘then’ ‘[’⟨*prop*⟩‘]’ ‘during’ ‘[’⟨*int*⟩‘]’

⟨*prop*⟩ ::= ⟨*OCL-prop*⟩ | ⟨*arith-rel*⟩

⟨*arith-rel*⟩ ::= ⟨*expr*⟩ (‘<’ | ‘<=’ | ‘=’ | ‘>=’ | ‘>’) ⟨*expr*⟩

⟨*arith-expr*⟩ ::= ⟨*expr*⟩ ⟨*operator*⟩ ⟨*expr*⟩ | ‘(’⟨*expr*⟩‘)’
| ⟨*OCL-expr*⟩
| ‘mean(’ ⟨*OCL-expr*⟩ ‘)’ | ‘sum(’ ⟨*OCL-expr*⟩ ‘)’
| ‘prod(’ ⟨*OCL-expr*⟩ ‘)’ | ‘at(’ ⟨*OCL-expr*⟩‘,’ ⟨*time*⟩ ‘)’

⟨*operator*⟩ ::= ‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’

⟨*int*⟩ ::= {‘`[`’ | ‘`(`’} ⟨*time*⟩ {‘`-`’ ⟨*time*⟩}? {‘`]`’ | ‘`)`’}

⟨*time*⟩ ::= $\mathbb{N}$ ⟨*time-unit*⟩ | $+\infty$

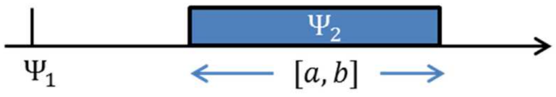
⟨*time-unit*⟩ ::= ‘`ms`’ | ‘`s`’ | ‘`min`’ | ‘`hour`’ | ‘`day`’ | . . .

The non-terminal ⟨*time-unit*⟩ can be any multiple of the application basic time unit (i.e. day, hour, min, sec, ms, ...). The latest revision of the OCL specification can be found at [15] and more particularly the grammar of the language. We just give an overview of the relevant subset used in this language: ⟨*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 (without treatments, e.g. the functions `map(...), iter(...)`), numerical values, model-related values, . . . Some relevant details about OCL collections are in the chapters 7.7 (Collection operations) and 11.6 (Collection-related types) of the OCL specification [15].

B Patterns

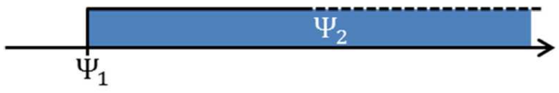
We give the list of all the SPPEDS patterns we reuse and we give give their semantics based on the statifiability given in Section 3.

a. **whenever Ψ_1 occurs Ψ_2 holds during following** $[a, b]$



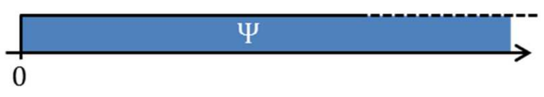
The interval $[a, b]$ is located relatively after the satisfaction of Ψ_1 . The interval, in which Ψ_2 must be satisfied, starts a units of time after the observed occurrence of Ψ_1.

b. Ψ_1 **implies** Ψ_2 **holds forever**



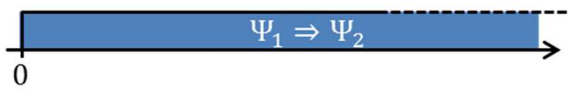
From the very moment when Ψ_1 is satisfied Ψ_2 must hold during all the rest of the execution path.

c. **always** Ψ

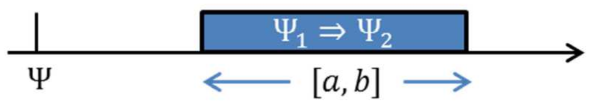


Ψ must hold during all the execution path.

d. **whenever Ψ_1 occurs Ψ_2 holds**

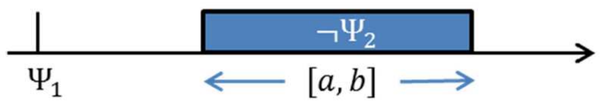


e. **whenever Ψ occurs Ψ_1 implies Ψ_2 during following** $[a, b]$



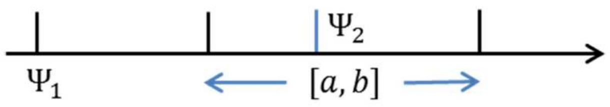
As for the previous pattern, the interval $[a, b]$ is relative. At each time value between a and b. where Ψ_1 holds, Ψ_2 must also hold. Replacing Ψ is replaced by *true* allows to create a new simpler pattern:
Ψ_1 **implies** Ψ_2 **during following** $[a, b]$

f. **whenever Ψ_1 occurs Ψ_2 does not occur during following** $[a, b]$



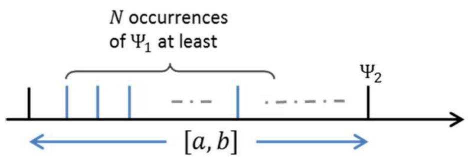
This pattern specifies that Ψ_2 is never satisfied during the relative interval $[a, b]$, i.e. $\neg\Psi_2$ holds during $[a, b]$.

g. **whenever Ψ_1 occurs Ψ_2 occurs within** $[a, b]$



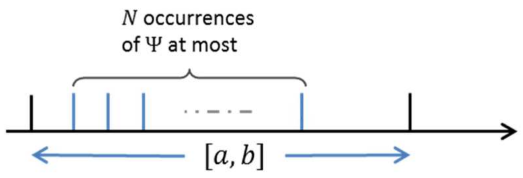
The constraint Ψ_2 must be satisfied at less one time during $[a, b]$ after Ψ_1.

h. Ψ_1 **occurs** n **times during** $[a, b]$ **raises** Ψ_2



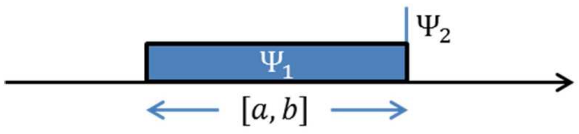
When Ψ_2 is satisfied at less n times during $[a, b]$, Ψ_2 starts to hold at b.

i. Ψ **occurs at most** n **times during** $[a, b]$



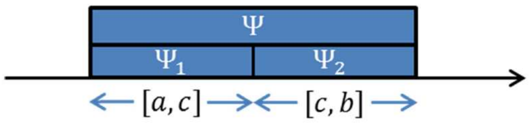
As previously mentioned, an occurrence of Ψ is counted when Ψ becomes satisfied. If Ψ holds for a state in $[a, b]$, to observe Ψ holds for the following one (also in $[a, b]$) does not increase the occurrence number of Ψ.

j. Ψ_1 **during** $[a, b]$ raises Ψ_2



If Ψ_1 holds during $[a, b]$ then Ψ_2 must hold at b.

k. Ψ **during** $[a, b]$ **implies** Ψ_1 **during** $[a, c]$ **then** Ψ_2 **during** $[c, b]$



Whenever Ψ holds during $[a, b]$ there exists a split at c of $[a, b]$ such that Ψ_1 holds during $[a, c]$ then Ψ_2 holds during $[c, b]$.

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[2] Ananda Basu, Saddek Bensalem, Marius Bozga, Beno&#x00ee;t Delahaye, and Axel Legay. Statistical abstraction and model-checking of large heterogeneous systems. *Int. J. Softw. Tools Technol. Transf.*, 14(1):53–72, February 2012.

[3] Cheng and all. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, 2009.

[4] Edmund Clarke, Alexandre Donzé, and Axel Legay. On simulation-based probabilistic model checking of mixed-analog circuits. *Form. Methods Syst. Des.*, 36(2):97–113, June 2010.

[5] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[6] EdmundM. Clarke, JamesR. Faeder, ChristopherJ. Langmead, LeonardA. Harris, SumitKumar Jha, and Axel Legay. Statistical model checking in biolab: Applications to the automated analysis of t-cell receptor signaling pathway. In Monika Heiner and AdelindeM. Uhrmacher, editors, *Computational Methods in Systems Biology*, volume 5307 of *Lecture Notes in Computer Science*, pages 231–250. Springer Berlin Heidelberg, 2008.

[7] Ales Corp. Adevanced laboratory on embedded systems, June.

[8] Danse. Designing for adaptability and evolution in sos engineering, dec 2013.

[9] Jasmin Fisher, Thomas A. Henzinger, Dejan Nickovic, Nir Piterman, Anmol V. Singh, and Moshe Y. Vardi. Dynamic reactive modules. In *CONCUR*, volume 6901 of *LNCS*, 2011.

[10] Carlo Ghezzi. Engineering evolving and self-adaptive systems: An overview. In *Software and Systems Safety - Specification and Verification*, volume 30 of *NATO*. IOS Press, 2011.

[11] INRIA. Plasma-lab: a statistical model checker, December 2012.

[12] Cyrille Jégourel, Axel Legay, and Sean Sedwards. A platform for high performance statistical model checking - plasma. In *TACAS*, pages 498–503, 2012.

[13] Sumit K. Jha, Edmund M. Clarke, Christopher J. Langmead, Axel Legay, André Platzer, and Paolo Zuliani. A bayesian approach to model checking biological systems. In *Proceedings of the 7th International Conference on Computational Methods in Systems Biology*, CMSB '09, pages 218–234, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] OMG. Object managment group, feb.

[15] OMG. Ocl v2.2, feb 2010.

[16] OMG. Uml v2.1.2, November 2011.

[17] OMG. Sysml v1.3, June 2012.

[18] OMG. Updm v2.0, January 2012.

[19] Atos Origin. Mdt ocl/ocl checker, December 2011.

[20] Kwiatkowska M. Parker D., Norman G. The probabilistic model checker prism, jan 2012.

[21] Modelica Association Project. Fmi v2.0 beta 4, aug 2012.

[22] Koushik Sen, Mahesh Viswanathan, and Gul Agha. On statistical model checking of stochastic systems. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, pages 266–280, 2005.

[23] SPEEDS. D 2.5.4: Contract specification language, apr 2008.

[24] SPEEDS. Speculative and exploratory design in systems engineering, apr 2010.

[25] Samir Younes, Edmund M. Clarke, Geoffrey J. Gordon, and Jeff G. Schneider. Verification and planning for stochastic processes with asynchronous events. Technical report, 2005.

[26] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE*. ACM, 2006.

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399