A Formalism for Stochastic Adaptive Systems

Benoît Boyer, Axel Legay, and Louis-Marie Traonouez

Inria / IRISA, Campus de Beaulieu, 35042 Rennes CEDEX, France

Abstract. Complex systems such as systems of systems result from the combination of several components that are organized in a hierarchical manner. One of the main characteristics of those systems is their ability to adapt to new situations by modifying their architecture. Those systems have recently been the subject of a series of works in the software engineering community. Most of those works do not consider quantitative features. The objective of this paper is to propose a modeling language for adaptive systems whose behaviors depend on stochastic features. Our language relies on an extension of stochastic transition systems equipped with (1) an adaptive operator that allows to reason about the probability that a system has to adapt its architecture over time, and (2) dynamic interactions between processes. As a second contribution, we propose a contract-based extension of probabilistic linear temporal logic suited to reason about assumptions and guarantees of such systems. Our work has been implemented in the Plasma-Lab tool developed at Inria. This tool allows us to define stochastic adaptive systems with an extension of the Prism language, and properties with patterns. In addition, Plasma-Lab offers a simulation-based model checking procedure to reason about finite executions of the system. First experiments on a large case study coming from an industrial driven European project give encouraging results.

1 Context

Critical systems increasingly rely on dynamically adaptive programs to respond to changes in their physical environments. Reasoning about such systems require to design new verification techniques and formalisms that take this model of reactivity into account [7].

This paper proposes a complete formalism for the rigorous design of stochastic adaptive systems (SAS), whose components' behaviors and environment changes are represented via stochastic information. Adding some stochastic feature to components' models is more realistic, especially regarding the environment aspect, e.g. the probability of hardware failure, the fire frequency in a forest or a growing city population...

We view the evolution of our system as a sequence of views, each of them representing a topology of the system at a given moment of time. In our setting, views are represented by a combination of Markov chains, and stochastic adaptive transitions that describe the environment evolution as transitions between different views of the SAS (e.g. adding or removing components). Each view thus associates the new environment behaviour and a new system configuration (a new

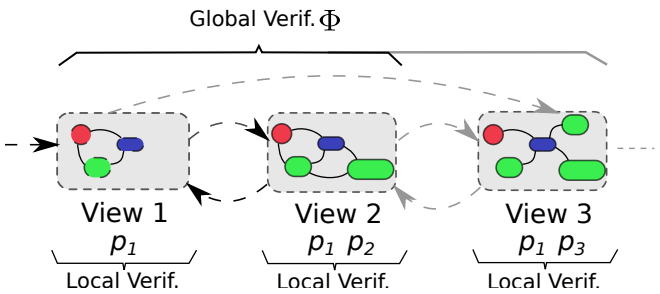

Fig. 1. Illustration of SAS Methodology

topology, addition or suppression of system components...). The incremental design is naturally offered to the system architect who can extend easily an existing model by creating new views.

Properties of views can be specified with Bounded Linear Temporal Logic (BLTL) that allows to reason about finite execution. To reason about sequences of view, we propose Adaptive BLTL (A-BLTL) that is an extension of BLTL with an adaptive operator to reason about the dynamic change of views. We also show that the formalism extend to contracts that permit to reason about both assumptions and guarantees of the system.

Consider the system described in Figure 1. This system is composed by three different views linked by adaptive transitions (represented by dashed arrows). Each view contains some system components in a particular topology denoting a configuration of the SAS. Each local property p_1, p_2, p_3 is attached to one or more views. There is also global property Φ. The SAS is initially designed by View 1 and View 2 and the black dashed arrows. Properties p_1, p_2 are validated for the corresponding views and Φ is validated against this complete initial version of the system. To fit with the upcoming settings of the system, the system architect updates the model by adding View 3 with new adaptive transitions (in grey). This requires to only validate p_1 and p_3 against View 3 and to validate again the global property Φ against the system including all the three views.

We propose a new Statistical Model Checking (SMC) [22,20] algorithm to compute the probability for a SAS to satisfy an A-BLTL property. SMC can be seen as a trade-off between testing and formal verification. The core idea of SMC is to generate a number of *simulations* of the system and verify whether they satisfy a given property expressed in temporal logics, which can be done by using *runtime verification approaches* [12]. The results are then used together with algorithms from the statistical area in order to decide whether the system satisfies the property with some probability. One of the key points to implement an SMC algorithm is the ability to bound a priori the size of the simulation, which is an important issue. Indeed, although the SAS can only spend a finite amount of time in a given view, the time bound is usually unknown and simulation cannot be bounded. To overcome the problem, we expand on the work of Clarke [21]

and consider a combination of SMC and model checking algorithm for untimed systems.

As a second contribution, we propose high-level formalisms to represent both SAS and A-BLTL/contracts. The formalism used to specify SAS relies on an extension of the Reactive Module Language (RML) used by the popular Prism toolset [16]. Properties are represented with an extension of the Goal and Contract Specification Language (GCSL) [2] defined in the DANSE IP project [11]. This language offers English-based pattern to reason about timed properties without having to learn complex mathematics inherent to any logic.

Finally, as a last contribution, we have implemented our work in Plasma-Lab [5] – a new powerful SMC model checker. The implementation has been tested on a realistic case study defined with industry partners of DANSE.

2 Modeling Stochastic Adaptive Systems

In this section, we present the formal model used to encode behaviors of adaptive systems. In Section 2.1, we introduce Markov chains (MC) to represent individual components of a view. Then, in Section 2.2, we show how to describe views as well as relations between views, i.e., adaptive systems.

2.1 Discrete and Continuous Time Markov Chains

Definition 1. *A (labelled)* transition system *is a tuple* $\mathcal{T} = (Q, \overline{q}, \Sigma, \rightarrow, AP, L)$ *where* Q *is a set of states,* $\overline{q} \in Q$ *is the initial state,* Σ *is a finite set of actions,* $\rightarrow : Q \times \Sigma \times Q$ *is the transition relation,* AP *is a set of atomic propositions, and* $L : Q \rightarrow 2^{AP}$ *is a state labelling function that assigns to each state the set of propositions that are valid in the state.*

We denote by $q \xrightarrow{a} q'$ the transition $(q, a, q') \in \rightarrow$. A *trace* is a finite or infinite alternating sequence of states and time stamps $\rho = t_0 q_0 t_1 q_1 t_2 q_2 \ldots$, such that $\forall i. \exists a_i \in \Sigma. q_i \xrightarrow{a_i} q_{i+1}$. Time stamps measure the cumulative time elapsed from a time origin. In discrete time models delays are integer values (i.e., $t_0 = 0$, $t_1 = 1$, $t_2 = 2$) and therefore they can be omitted. In continuous time models they are real values. We denote by $|\rho|$ the length of a trace ρ. If ρ is infinite then $|\rho| = \infty$. A trace is initial if $q_0 = \overline{q}$ and $t_0 = 0$. We denote by $\mathsf{trace}_n(\mathcal{T})$ (resp. $\mathsf{trace}(\mathcal{T})$) the set of all initial traces of length n (resp. infinite traces) in $\mathcal{T}$. Let $0 \leq i \leq |\rho|$, we denote $\rho_{|i} = t_0 q_0 t_1 q_1 \ldots t_{i-1} q_{i-1}$ the finite prefix of ρ of length i, $\rho^{|i} = t_i q_i t_{i+1} q_{i+1} \ldots$ the suffix of ρ that starts at position i, and $\rho[i] = q_i$ the state at position i.

We now extend transition systems with probabilities to represent uncertainty of behaviors or of material on which the system is running. We present two semantics, either with discrete or continuous time, that are both compatible with our setting. A discrete time Markov chain (DTMC) is a state-transition system in which each transition is labelled by a probability $\mathbf{P}(s, s')$ to take the transition from state s to state s'.

Definition 2. *A (labelled) DTMC is a tuple* $\mathcal{D} = (Q, \overline{q}, \Sigma, \rightarrow, AP, L, \mathbf{P})$ *where:*

- $(Q, \overline{q}, \Sigma, \rightarrow, AP, L)$ *is a labelled transition system,*
- $\mathbf{P} : Q \times Q \rightarrow [0, 1]$ *is a* transition probability matrix*, such that* $\sum_{q' \in Q} \mathbf{P}(q, q') = 1$ *for all* $q \in Q$*,*
- $\rightarrow$ *is such that* $q \xrightarrow{a} q'$ *iff* $\mathbf{P}(q, q') > 0$*, and for each state* q *there is at most one action* $a \in \Sigma$ *such that* $q \xrightarrow{a} q'$ *for some* q'*.*

In continuous time Markov chains (CTMCs) transitions are given a rate. The sum of rates of all enabled transitions specifies an exponential distribution that determines a real value for the time spent in the current state. The ratio of the rates then specifies which discrete transition is chosen.

Definition 3. *A (labelled) CTMC is a tuple* $\mathcal{C} = (Q, \overline{q}, \Sigma, \rightarrow, AP, L, \mathbf{R})$ *where:*

- $(Q, \overline{q}, \Sigma, \rightarrow, AP, L)$ *is a labelled transition system,*
- $\mathbf{R} : Q \times Q \rightarrow \mathbb{R}_{\geq 0}$ *is a* transition rate matrix*,*
- $\rightarrow$ *is such that* $q \xrightarrow{a} q'$ *iff* $\mathbf{R}(q, q') > 0$*, and there is a unique* $a \in \Sigma$ *such that* $q \xrightarrow{a} q'$*.*

In our setting, a view of a system is represented by the combination of several components. We can compute the parallel composition $C_1 \parallel C_2$ of two DTMCs (resp. CTMCs) defined over the same alphabet Σ. Let $(Q_1, \overline{q}_1, \Sigma, \rightarrow_1, AP_1, L_1)$ and $(Q_2, \overline{q}_2, \Sigma, \rightarrow_2, AP_2, L_2)$ be the two underlying transition systems. We first compute their parallel composition, which is a labelled transition system $(Q, \overline{q}, \Sigma, \rightarrow, AP, L)$, where $Q = Q_1 \times Q_2$, $\overline{q} = (\overline{q}_1, \overline{q}_2)$, $AP = AP_1 \cup AP_2$, $L(q) = L_1(q_1) \cup L_2(q_2)$ and the transition relation $\rightarrow$ is defined according to the following rule:

$$\frac{q_1 \xrightarrow{a}_1 q_1' \quad q_2 \xrightarrow{a}_2 q_2'}{(q_1, q_2) \xrightarrow{a} (q_1', q_2')} \tag{1}$$

Then, in case of DTMCs, the new transition probability matrix is such that $\mathbf{P}((q_1, q_2), (q_1', q_2')) = \mathbf{P}(q_1, q_1') * \mathbf{P}(q_2, q_2')$, and in case of CTMCs the new transition rate matrix is such that $\mathbf{R}((q_1, q_2), (q_1', q_2')) = \mathbf{R}(q_1, q_1') * \mathbf{R}(q_2, q_2')$. DTMCs with different alphabets can also be composed and they synchronize on common actions. However, if both DTMCs can perform a non synchronized action, a uniform distribution is applied to resolve the non determinism. In case of CTMCs, the two actions are in concurrence, such that if $q_1 \xrightarrow{a}_1 q_1'$ with $a \notin \Sigma_2$, then $(q_1, q_2) \xrightarrow{a} (q_1', q_2)$ and $\mathbf{R}((q_1, q_2), (q_1', q_2)) = \mathbf{R}(q_1, q_1')$. In what follows, we denote by $Sys = C_1 \parallel C_2 \parallel \cdots \parallel C_n$ the DTMC (resp. CTMC) that results from the composition of the components $C_1, C_2, \ldots, C_n$.

2.2 Stochastic Adaptive Systems (SAS)

An adaptive system consists in several successive views. It starts in an initial view that evolves until it reaches a state in which an adaptation is possible. This adaptation consists in a view change that depends on a probability distribution that represents uncertainty of the environment.

Definition 4. *A SAS is a tuple* $(\Delta, \Gamma, S, \overline{sys}, \rightsquigarrow)$ *where:*

- $\Delta = \{C_1, C_2, \ldots, C_n\}$ *is a set of DTMCs (resp. CTMCs) that are the components of the SAS.*
- Γ *is the set of views of the SAS, such that each view is a stochastic system obtained from the parallel composition some components from* Δ*.*
- $\overline{sys} \in \Gamma$ *is the initial view.*
- S *is the set of states of the SAS.* S *is the union of the states of each view in* Γ*, i.e., for each state* $s \in S$ *there exists* $\{C_1, C_2, \ldots, C_k\} \subseteq \Delta$ *such that* $s \in Q_1 \times Q_2 \times \cdots \times Q_k$ *(where* $\forall i, 1 \leq i \leq k$*,* Q_i *is the set of states of* C_i*).*
- $\rightsquigarrow \subseteq S \times [0,1]^S$ *is a set of adaptive transitions.*

Observe that the number of components per state may vary. This is due to the fact that different views may have different components. Observe also that it is easy to add new views to an existing adaptive system without having to re-specify the entire set of views. An element $(s, \mathbf{p}) \in \rightsquigarrow$ consists in a state s from a view $sys \in \Gamma$ and a probability distribution $\mathbf{p}$ over the states in S. When $s \neq s'$, we denote $s \rightsquigarrow s'$ if there exists $\mathbf{p}$ such that $(s, \mathbf{p}) \in \rightsquigarrow$ and $\mathbf{p}(s') > 0$, which means that state s can be adapted into state s' with probability $\mathbf{p}(s')$.

A trace ρ in a SAS is either a finite combination of n traces $\rho = \rho_0 \rho_1 \ldots \rho_n$, such that for all $0 \leq i \leq n-1$, $\rho_i = t_{0i} s_{0i} t_{1i} s_{1i} \ldots t_{li} s_{li}$ is a finite trace of $sys_i \in \Gamma$, and $s_{li} \rightsquigarrow s_{0i+1}$, and $t_{0i+1} = 0$, and ρ_n is a finite or infinite trace $sys_n \in \Gamma$. Otherwise ρ may be an infinite combination of finite traces $\rho = \rho_0 \rho_1 \ldots$ that satisfy for all i the same constraints.

3 A logic for SAS properties

3.1 Probabilistic Adaptive Bounded Linear Temporal Logic

We consider quantitative verification of dynamic properties that are expressed via a quantitative extension of the Adaptive Linear Temporal Logic (A-LTL) proposed in [23]. Our logic, which we call Adaptive Bounded Linear Temporal Logic (A-BLTL), relies on an extension of Bounded Linear Temporal Logic (BLTL) combined with an adaptive operator. Although the logic is not strictly more expressive than BLTL, it is more suitable to describe properties of individual views, as well as global properties of the adaptive system, and it allows to develop specific algorithms for these properties. In the last part of the section, we also show how one can define contracts on such logic, where a contract [17] is a pair of assumptions/guarantees that must be satisfied by the system.

We first introduce BLTL, a logic used to express properties on individual views. In BLTL, formulas are built by using the standard Boolean connectors $\wedge$, $\vee$, $\Longrightarrow$, $\neg$, and the temporal operators G, F, X, U borrowed from Linear Temporal Logic (LTL). The main difference between BLTL and classical LTL is that each temporal modality is indexed by a bound k that defines the length of the run on which the formula must hold. The semantics of a BLTL formula is defined in Table 1 for finite executions of CTMC/DTMC $\rho = t_0 s_0 t_1 s_1 t_2 s_2 \ldots$,

$\rho \models X_{\leq k}\Phi \equiv \exists i,\ i = max\{j \mid t_0 \leq t_j \leq t_0 + k\}$ and $\rho^{\mid i} \models \Phi$	
$\rho \models \Phi_1\ U_{\leq k}\Phi_2 \equiv \exists i,\ t_0 \leq t_i \leq t_0 + k$ and $\rho^{\mid i} \models \Phi_2$ and $\forall j, 0 \leq j \leq i,\ \rho^{\mid j} \models \Phi_1$	
$\rho \models \Phi_1 \wedge \Phi_2 \equiv \rho \models \Phi_1$ and $\rho \models \Phi_2$	$\rho \models \neg\Phi \equiv \rho \not\models \Phi$
$\rho \models P \equiv P \in L(s_0)$	$\rho \models true \quad \rho \not\models false$

Table 1. Semantics of BLTL

with $|\rho| \geq k$. If $|\rho| < k$, it is extended by duplicating the last state enough times. In case formulas are nested, the value of k adapts incrementally.

We now generalize BLTL to adaptive systems. For doing so, we introduce an adaptive operator in the spirit of [24]. The new logic A-BLTL is an extension of BLTL with an adaptive operator $\Phi \overset{\Omega}{\Rightarrow}_{\leq k} \Psi$, where Φ is a BLTL formula, Ψ is an A-BLTL formula, Ω is a predicate over the states of different views of the SAS, and k is a time bound that limits the execution time of the adaptive transition. We will also consider unbounded versions of the adaptive operator.

Definition 5 (A-BLTL semantics). *Let $\Phi \overset{\Omega}{\Rightarrow}_{\leq k} \Psi$ be an A-BLTL formula and $\rho = t_0 s_0 t_1 s_1 t_2 s_2 \ldots$ be an execution of the SAS:*

$$\rho \models \Phi \overset{\Omega}{\Rightarrow}_{\leq k} \Psi \equiv \exists i,\ i = min\{j \mid t_0 \leq t_{j-1} \leq t_0 + k \wedge \rho_{\mid j} \models \Phi \wedge s_{j-1} \rightsquigarrow s_j \wedge \Omega(s_{j-1}, s_j)\} \wedge \rho^{\mid i} \models \Psi \quad (2)$$

The property is unbounded if $k = \infty$, and in that case we write $\Phi \overset{\Omega}{\Rightarrow} \Psi$.

According to Definition 5, an execution ρ satisfies an adaptive formula $\Phi \overset{\Omega}{\Rightarrow}_{\leq k} \Psi$ if and only if there exists a minimal prefix of ρ that satisfies Φ and reaches a state s_{j-1}, such that $\Omega(s_{j-1}, s_j)$ is satisfied, and such that the suffix of ρ from state s_j satisfies Ψ. Therefore to satisfy this formula it is necessary to observe an adaptation compatible with Ω. We relax this constraint by introducing a new operator $\Phi \overset{\Omega}{\longrightarrow}_{\leq k} \Psi$, for which an adaptation is not necessary but triggers a check of Ψ when it happens. It is equivalent to the following formula: $\Phi \overset{\Omega}{\longrightarrow}_{\leq k} \Psi \equiv (\Phi \overset{\Omega}{\Rightarrow}_{\leq k} true) \implies (\Phi \overset{\Omega}{\Rightarrow}_{\leq k} \Psi)$.

We finally introduce stochastic contracts, that are used to reason about both the adaptive system and its environment via assumptions and guarantees.

Definition 6 (Contracts for SAS). *A* contract *is defined as a pair (A, G), where A and G are respectively called the Assumption and the Guarantee. A SAS $\mathcal{M}$ satisfies the contract (A, G) iff $\forall \rho,\ \rho \models A \Rightarrow \rho \models G$, where ρ is a trace of $\mathcal{M}$ and $\rho \models A$ (resp. G) means the trace ρ satisfies the assumption A (resp. the guarantee G). In that case we write $\mathcal{M} \models (A, G)$.*

3.2 Verifying SAS Properties using SMC

SMC [22,20] is an alternative to model checking [3,9] that employs Monte Carlo methods to avoid the state explosion problem. SMC estimates the probability that

a system satisfies a property using a number of statistically independent simulation traces of an executable model. By using results from the statistic area, SMC decides whether the system satisfies the property with some degree of confidence, and therefore it avoids an exhaustive exploration of the state-space of the model that generally does not scale up. It has already been successfully experimented in biology area [10,15,16], software engineering [8] as well as industrial area like aeronautics [4].

The basic algorithm used in SMC is the Monte Carlo algorithm. This algorithm estimates the probability that a system Sys satisfies a BLTL property P by checking P against a set of N random executions of SyS. The estimation $\hat{p}$ is given by $\hat{p} = \frac{\sum_1^N f(\rho_i)}{N}$ where $f(\rho_i) = 1$ if $\rho_i \models P$, 0 otherwise.

Using the formal semantics of BLTL, each execution trace ρ_i is monitored in order to check if P is satisfied or not. The accuracy of the estimation increases with the number of monitored simulations. This accuracy can be controlled thanks to the Chernoff-Hoeffding bound [14]. It relates N to δ and ε, that are respectively the confidence and the error bound of $\hat{p}$:

$$Pr(|p - \hat{p}| < \varepsilon) \geq 1 - \delta \quad \text{if} \quad N \geq \frac{ln(\frac{2}{\delta})}{2\varepsilon^2} \tag{3}$$

According to this relation, the user is able to trade off analysis time in return for accuracy of the result using the parameters δ and ε.

Knowing that there exists techniques to monitor BLTL properties [13], the model checking of A-BLTL is rather evident. Given an A-BLTL property $\Phi_1 \overset{\Omega}{\Rightarrow}_{\leq k} \Phi_2$, the monitor will first check whether the run satisfies Φ_1 using classical runtime verification techniques. If no, then the property is not satisfied. If yes, then one checks whether there is a pair of two successive states between t_0 and $t_0 + k$ that satisfies Ω. The latter is done by parsing the run. If this pair does not exist, then the property is not satisfied. Else we start a new monitor from the suffix of the run starting in the second state of the pair in order to verify Φ_2.

3.3 Verifying unbounded SAS Properties using SMC

We propose a method inspired by [21] to check unbounded A-BLTL properties. The principle is to combine a reachability analysis by model checking the underlying finite-state machine, with a statistical analysis of the stochastic model using the algorithms introduced previously.

We consider an A-BLTL property $\Phi \overset{\Omega}{\Rightarrow} \Psi$, where Φ and Ψ are BLTL formulas. We first consider the reachability problem with objective $G = \{s \mid \exists s'.s \rightsquigarrow s' \wedge \Omega(s, s')\}$. The preliminary to the statistical analysis is to compute the set $Sat(Reach(G))$, that is all the states of the SAS that may eventually reach a state in G. This can be computed using classical model checking algorithms for finite-state machines. Only the underlying automata of the DTMCs or CTMCs of the SAS is used for this analysis. As stochastic quantities are ignored, efficient symbolic techniques exist to speed up this process [6].

```
1: procedure CHECK(Φ ⇒^Ω Ψ, ρ)
2:     if ¬CHECK(Φ, ρ) then return false
3:     end if
4:     i ← 0, prec ← null, curr ← null
5:     while i < |ρ| do
6:         prec ← curr, curr ← ρ[i]
7:         if curr ∉ Sat(Reach(G)) then return false
8:         end if
9:         if Ω(prec, curr) then return CHECK(Ψ, ρ^|i)
10:        end if
11:        i ← i + 1
12:    end while
13: end procedure
```

Fig. 2. Algorithm to monitor unbounded A-BLTL properties

Once this preliminary computation is performed, the CHECK algorithm from Figure 2 is used to monitor the runs of the SAS. The algorithm takes as input the A-BLTL property and a run ρ. The run should be in general infinite as there is no bound on the length of the runs that satisfied an unbounded A-BLTL property. In that case the states would be generated on-the-fly. The algorithm returns true or false, whether the run satisfied the property. We also denote CHECK(Φ, ρ) as the monitoring of the BLTL property Φ. Then the first step on line 2 is to monitor Φ on the run ρ. If the result is false then the property $\Phi \overset{\Omega}{\Rightarrow} \Psi$ is not satisfied. Otherwise the algorithm searches through ρ for two states $prec$ and $curr$ such that $\Omega(prec, curr)$ is true. This is possible if $curr$ belongs to the precomputed set $Sat(Reach(G))$. If Ω is satisfied the last step on line 10 is to monitor Ψ from the current position in ρ.

For homogeneous Markov chains (with constant probability matrices, as it is the assumption in this paper), the algorithm almost surely (with probability 1) terminates, since it either reaches a state where Ω is unreachable, or the probability to reach two states that satisfy Ω is not null. It can be iterated to check sequences of adaptive operators, that is to say properties where Ψ is also an unbounded A-BLTL.

4 A software engineering point of view

In this section, we propose high level formalisms to specify both adaptive systems and their properties. Then, we define semantics of those formalisms by exploiting the definitions introduced in the previous sections. This gives us a free verification technology for them. The situation is illustrated in Figure 3.

4.1 Adaptive RML systems as a high level formalism for SAS

We represent adaptive systems with Adaptive Reactive Module Language (A-RML), an extension of the Reactive Module Languages (RML) used by the

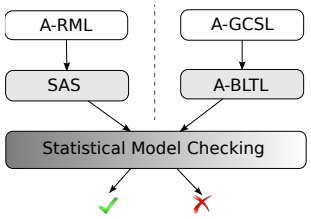

Fig. 3. SAS verification flow

PRISM toolset [16]. Due to space limit, the syntax common to RML and A-RML is only briefly described here [1].

The RML language is based on the synchronisation of a set of modules defined by the user. A module is declared as a DTMC or CTMC, i.e., some local variables with a set of guarded commands. Each command has a set of actions of the form λ_i:$\mathtt{a}_i$ where λ_i is the probability (or the rate) to execute $\mathtt{a}_i$. A-RML extends RML such that each module can have some parameters in order to define its initial state.

```
module MOD_NAME(<Parameters>)
   <local_vars>
      ...
   [chan] g_k -> (λ_0:a_0) + ... + (λ_n:a_n);
      ...
endmodule
```

The optional channel identifiers prefixing commands are used to strongly synchronise the different modules of a RML system. A module is synchronised over the channel `chan` if it has some commands prefixed by `chan`. We say a command is independent if it has no channel identifier.

In A-RML a system is a set of modules and global variables. The modules synchronise on common channels such that the system commands are the independent commands of each module and the synchronised commands. A command synchronised on `chan` forces all the modules that synchronised over `chan` to simultaneously execute one of their enabled commands prefixed by `chan`. If one module is not ready, *i.e.* it has no enabled commands for `chan`, the system has no enabled command over `chan`. Similarly to a module, if the system reaches a state with a non-deterministic choice, it is solved by a stochastic behaviour based on a uniform distribution. This solution allows to execute the A-RML system in accordance with the DTMC/CTMC models.

```
system SYS_NAME(<Parameters>)
   <global_variables>
   ...
   <module_declarations>
   ...
endsystem
```

[1] The full syntax can be found at http://project.inria.fr/plasma-lab/

An adaptive system consists in a set of different views, each represented by an A-RML system, and a list of adaptations represented by adaptive commands. The `adaptive` environment is used to specify which one is the initial view and what adaptations are possible.

```
adaptive
   init at SYS_NAME(<Initial values>)
   ...
   { SYS_NAME | g_k } -> λ_0:{a_0} + ... + λ_n:{a_n};
   ...
endadaptive
```

An adaptive command is similar to module command. It has a guard $\mathtt{g}_k$ that applies to the current view `SYS_NAME`, and a set of actions $\lambda_i{:}\mathtt{a}_i$ where λ_i is the probability (or the rate) to execute action $\mathtt{a}_i = \mathtt{SYS\_NAME'}(\mathtt{e}_0, \ldots, \mathtt{e}_m)$. This action defines the next view `SYS_NAME'` after performing the adaptation. This view is determined according to the states of the previous view by setting the parameters of `SYS_NAME'` with the expressions $\mathtt{e}_0, \ldots, \mathtt{e}_m$ evaluated over `SYS_NAME`. The execution of the adaptive command is done in accordance with the SAS semantics.

Theorem 1. *The semantics of A-RML can be defined in terms of SAS.*

The proof of the above theorem is a direct consequence of the fact that semantics of RML is definable in terms of composition of MC, and that the definition of an adaptive command can also be represented as a MC.

4.2 A contract language for SAS specification

The Goal and Contract Specification Language (GCSL) was first proposed in [2] to formalise properties of adaptive systems in the scope of the DANSE project. It has a strong semantics based on BLTL but it has a syntax close to the hand written English requirements. Dealing with formal temporal logic is often an issue to formalise correctly the initial English requirements. Most of the time the formalisation frequently contains some mistakes, which is due to the nesting of the temporal operators. The difficulty for correctly specifying properties is enough to make the overall methodology useless.

The GCSL syntax combines a subset of the Object Constraint Language (OCL) [18] (used to define state properties, i.e., Boolean relations between the system components) and English behavioural patterns used to express the evolution of these state properties during the execution of the system. The usage of OCL is illustrated in Example 1.

Example 1. We consider a SAS describing the implementation of an emergency system in a city. The city area is divided as a set of districts where each district may have some equipment to fight against the fire, e.g. some fire stations with fire brigades and fire fighting cars. Each district is also characterised by a risk of fire and the considered damages are mainly related to the population size

of each district. The requirement *"Any district cannot have more than 1 fire station, except if all districts have at least 1"* ensures the minimal condition for the equipment distribution in the city. We use syntactic coloration to make the difference between the parts of the language used in the property: the words in red are identifiers from the model, the blue part is from OCL, like collection handling, and the black words are variables:

City.itsDdistricts→exists(district | district.**ownedFireStations > 1) implies**
City.itsDistricts→forAll(district | district.**ownedFireStations ≥ 1)**

GCSL patterns are used to specify temporal properties. In this section we only present a subset of such patterns that is considered to be general enough to specify properties of a large set of industry-examples from the DANSE project. After having read this section, the user shall understand that the set can be easily increased. Each pattern can nest one or more state properties, denoted in the grammar by the non-terminals `<OCL-prop>` and `<arith-rel>`, that respectively denote a state property written in OCL or an arithmetic relation between the identifiers used in the model. The non-terminal `<int>` denotes a finite time interval over which the temporal pattern is applied, and `<N>` is a natural number. The patterns can be used directly or combined with OCL: applying a pattern to a collection of system components defines a behavioural property that is applied to each element of the collection. We present below an excerpt of the complete GCSL grammar available in [2]:

```
<GCSL> ::= <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>
```

Example 2. Consider the following requirement about the model described in Example 1: *"The fire fighting cars hosted by a fire station shall be used all simultaneously at least once in 6 months"*. This requirement uses both GCSL and OCL patterns:

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

We now propose A-GCSL, a syntax extension for GCSL that can be used to describe adaptive requirements of SAS. A-GCSL extends the GCSL grammar by adding a new pattern that allows to express adaptive relations as done with the two adaptive operators defined in Section 3. The first pattern of `<dyna-spec>` is equivalent to the operator $\overset{\Omega}{\Rightarrow}$ and the second one denotes $\overset{\Omega}{\longrightarrow}$. Any adaptive

requirement has three elements (A, Ω, G) that are called assumption, trigger and guarantee, respectively. The assumption and guarantee are specified in GCSL, whereas the trigger is in OCL. The syntax allows to compose the patterns by specifying the guarantee with an adaptive pattern. For instance, a composed requirement of the form `if` Φ_1 `holds and for all rule that satisfies` Ω `then` (`if` Φ_2 `holds and for all rule that satisfies` Ω' `then` Φ_3 `holds`) `holds` is equivalent to the property $\Phi_1 \overset{\Omega}{\Rightarrow} \Phi_2 \overset{\Omega'}{\Longrightarrow} \Phi_3$. The A-GCSL grammar is the following:

```
<dyna-spec> ::= if [<GCSL>] holds and for all rule that satisfies [<prop>]
                then ( <GCSL> | <dyna-spec> ) holds
| if [<GCSL>] holds then there exists a rule satisfying [<prop>]
                and ( <GCSL> | <dyna-spec> ) holds
```

Example 3. Consider again the system in Example 1 and the following A-GCSL requirement:
if [**City.underFire = 0**] **holds and for all rule such that rule satisfies** [**City.underFire ≥ 3**] **then** [**City.itsDistricts→forall(**district | district.**decl = false** $\Longrightarrow$ **whenever** [district.**decl = true**] **occurs,** [district.**fire = 0**] **occurs within [50 hours]** **)**] **holds**

The attribute `underFire` denotes the number of districts in which a fire has been declared. If there are more than three fires in the city, then the fire stations change their usual emergency management into a crisis one. When such management is activated, the firemen have 50 hours to fix the problem. The requirement can be translated in A-GCSL using the following formula:

$$\Phi_6 = \big(underFire = 0\big) \xrightarrow{underFire \geq 3}_{\leq 10000} \bigwedge_{d_i : district} \Big(\neg d_i.decl \implies G_{\leq 10000}\ (d_i.decl \implies F_{\leq 50}\ d_i.fire = 0)\Big)$$

In [2], we have showed that any GCSL pattern can be translated into a BLTL formula. The result extends as follows.

Theorem 2. *Any A-GCSL pattern can be translated into an A-BLTL property.*

This result is an immediate consequence of the definition of the adaptive pattern.

5 Experiments with SAS

Our work has been implemented in a new statistical model checker named Plasma-Lab [5], a platform that includes efficient SMC algorithms, a graphical user interface (GUI) and multiple plugins to support various modelling languages. The tool is written in Java that offers maximum cross-platform compatibility. The GUI allows to create projects and experiments, and it implements a simple and practical mean of distributing simulations using remote clients. Plasma-Lab also provides a library to write new plugins and create custom statistical model checkers based on arbitrary modelling languages. Developers will only need to

implement a few simple methods to build a simulator and a logic checker, and then beneficiate from Plasma-Lab SMC algorithms.

Plasma-Lab can be used as a standalone application or be instantiated within other softwares. It has already been incorporated in various performance-critical softwares and embedded hardware platforms. The current plugins allow to simulate biologic models, models written in RML and A-RML, but it has also the capabilities to drive an external engine to perform the simulations, like MATLAB, Scilab, or DESYRE a simulator for adaptive systems developed by Ales [1].

5.1 CAE model

Together with our industrial partners in the DANSE project, we have developed the Concept Alignment Example (CAE). The CAE is a fictive adaptive system 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 adaptive systems designs.

The CAE describes the organization of the firefighting forces. We consider in our study that the city is initially divided into 4 districts, and that the population might increase by adding 2 more districts. Different and even more complex examples can be built using the components of this design.

A fire station is assigned to the districts, but as the fire might spread within the districts, the system can adapt itself by hiring more firemen. We can therefore design a SAS with three views as described in Figure 4.

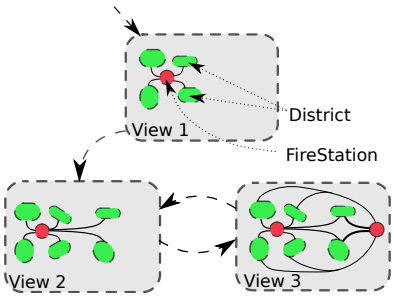

Fig. 4. Components and Views in the CAE model

Adaptive transitions exist between these views to reflect changes in the environment and adaptations of the system. Initially in View 1, the system can switch to View 2 when the population of the city increase. This change models an uncertainty of the environment, and for the purpose of this study we fix its probability to 0.01 Then, if the number of fires becomes greater than 2, the system adapts itself by switching to View 3. If the number of fires eventually reduces and becomes lower than 2, the system might return to View 2. Again, as this change is uncertain, we fix its probability to 0.8.

We design several A-RML models of the system that consist in two types of modules: `District` and `FireStation`, both based on a CTMC semantics. First, we study a model `AbstractCAE` that is an abstract view of the SAS. In this model, the `District` module, presented below, is characterized by a constant parameter `p`, that determines the probability of fire, and by two Boolean variables `decl` and `men`, that respectively defines if a fire has been declared and if the firemen are allocated to the district. The module `fireStation` has one constant parameter $\texttt{distance}_\texttt{i}$ for each module of the system. This parameter determines the probability to react at a fire, such that the greater the distance, the lower the probability. However a fire station can only treat one fire at a time, which is encoded with a Boolean variable `allocated`. The fire stations and the districts synchronize on channels `allocate` and `recover`, that respectively allocate firefighters to the district and bring them back when the fire is treated. The different views are constructed by instantiating and renaming the modules presented above.

```
module District( const int p )
  decl : bool init false;
  men: bool init false;
  [] !decl -> p/1000: (decl'=true);
  [allocate] decl & !men -> (men'=true);
  [recover] decl & men -> 1/p: (decl'=false) & (men'=false);
endmodule
```

We refine this model to better encode the behaviour of the SAS. In this new model `ConcreteCAE` a new variable `fire` of module `District` ranges from 0 to 10 and grades the intensity of the fire. The fire stations can now assign several cars (from 0 to 5) to each districts. Therefore the variables `men` and `allocated` becomes integers.

```
module District( const int p )
  fire : [0..10] init 0;
  decl : bool init false;
  men: [0..5] init 0;
  [] fire=0 -> p/1000: (fire'=1);
  [] fire>0 & fire<10 -> p/((1+men)*100): (fire'=fire+1);
  [] fire>0 & !decl -> (fire*fire)/10: (decl'=true);
  [allocateSt1] decl & fire>0 -> (fire*fire)/10: (men'=men+1);
  [allocateSt2] decl & fire>0 -> (fire*fire)/10: (men'=men+1);
  [] men>0 & fire>0 -> men/10: (fire'=fire-1);
  [recover] decl>0 & fire=0 -> 1000: (men'=0)&(decl'=false);
endmodule
```

From the two models we can consider several subparts composed by one or several views of the SAS. Adaptive commands are used to model the transitions between the different views.

- `AbstractCAE_1` consists in View 1 and 2 from model `AbstractCAE`.
- `AbstractCAE_2` consists in View 2 and 3.

- `AbstractCAE_3` has the same views as `AbstractCAE_2` but is initiated in View 3 instead of View 2.
- `ConcreteCAE_1` only consists in View 1 from model `ConcreteCAE`.
- `ConcreteCAE_2` only consists in View 2.
- `ConcreteCAE_3` only consists in View 3.
- `ConcreteCAE_Full` is the full model of `ConcreteCAE`, with the 3 views and all the adaptive transitions between them.

5.2 Checking requirements

The requirements are expressed in A-GCSL and translated to A-BLTL. We first check the model `AbstractCAE` against A-BLTL properties with adaptive operators. Our goal is to verify that the transitions between the different views of the system occurs and satisfy some properties.

The first property, *if* [`true`] *holds then there exists a rule satisfying* [`underfire` $\leq$ `1`] *and Always* [!`maxfire`], checks that when the system is in View 1, it eventually switches to View 2 when the number of districts that have declared a fire (`underfire`) is still lower than 1, and that as a result the system remains safe for a limited time period, i.e., the number of districts that have declared a fire is not maximum (`maxfire` is false). To check this property we limit the analysis to the model `AbstractCAE_1` with only View 1 and View 2. The A-GCSL property is translated in an A-BLTL formula: $\Phi_1 = \texttt{true} \xRightarrow{\texttt{underfire}\leq 1} G_{\leq 1000}\ \texttt{!maxfire}$, and the results in Table 2 show that the probability to satisfy the property is only 50%. This justify the need to add a second fire station, as in View 3.

The second property, *if* [`true`] *holds then there exists a rule satisfying* [`true`] *and Always* [!`maxfire`], checks that from View 2 a second fire station is quickly added, which switches the system to View 3, and that then the system is safe. The property is checked on the model `AbstractCAE_2` using the A-BLTL formula : $\Phi_2 = \texttt{true} \xRightarrow{\texttt{true}}_{\leq 100} G_{\leq 10000}\ \texttt{!maxfire}$.

Finally, with the property *if* [`true`] *holds then there exists a rule satisfying* [`true`] *and* [`true`], we check that from View 3 the system eventually returns to View 2. Therefore we use the model `AbstractCAE_3` that starts in View 3 and we check the A-BLTL formula $\Phi_3 = \texttt{true} \xRightarrow{\texttt{true}}_{\leq 100} \texttt{true}$.

The `AbstractCAE` models are simple enough to be able to perform reachability analyses and check the unbounded A-BLTL properties presented above using Algorithm 2. In a second step we consider the models `ConcreteCAE` to better evaluate the safety of the system. The state spaces of these models contain several millions of states, and therefore, they can only be analyzed by purely SMC algorithms. We verify the two following properties:

- *Always* !`maxfire`, to check that the maximum of fire intensity of 10 is never reached in any district. This corresponds to $\Phi_4 = G_{\leq 10000}\ \texttt{!maxfire}$.
- *Whenever* [`fire` $>$ `0`] *occurs* [`fire` $=$ `0`] *within* [50 *hours*], to check that a fire in a district is totally extinct within 50 hours. This corresponds to $\Phi_5 = G_{\leq 10000}\big(\texttt{d6.fire} > 0 \implies F_{\leq 50}\ \texttt{d6.fire} = 0\big)$.

These two properties are first checked for each view of the system. The results in Table 2 show that while View 1 and View 3 are surely safe, View 2 is frequently unsafe. But when we check these properties on the complete adaptive model `ConcreteCAE_Full`, with the three views, we can show that the system remains sufficiently safe. It proves that after a change of the environment (the increase of population) the system is able to adapt itself to guaranty its safety.

In the last experiment of Table 2 we check the A-GCSL property presented in Example 3. This bounded adaptive A-GCSL property is checked using the full `ConcreteCAE` model.

We have performed each experiment in Plasma-Lab with a confidence $\delta = 0.01$ and an error bound $\varepsilon = 0.02$. The results in Table 2 give the probabilities estimation and the time needed to perform the computation.

Property	CAE Model	Estimation interval	Consumed Time
Φ_1	`AbstractCAE_1` View 1, View 2	$[0.53, 0.56]$	1351s
Φ_2	`AbstractCAE_2` View 2, View 3	$[0.84, 0.86]$	11s
Φ_3	`AbstractCAE_3` AbstractCAE_2 starting from View 3	$[0.98, 1]$	1363s
Φ_4	`ConcreteCAE_6` 4 dist. 1 sta.	$[0.95, 0.99]$	11s 9s
Φ_4 Φ_5	`ConcreteCAE_2` 6 dist. 1 sta.	$[0.46, 0.5]$ $[0.21, 0.25]$	15s 13s
Φ_4 Φ_5	`ConcreteCAE_3` 6 dist. 2 sta.	$[0.98, 1]$ $[0.98, 1]$	30s 31s
Φ_4 Φ_5	`ConcreteCAE_Full` 4-6 dist. 1-2 sta.	$[0.89, 0.93]$ $[0.82, 0.86]$	25s 42s
Φ_6	`ConcreteCAE_Full` 4-6 dist. 1-2 sta.	$[0.47, 0.51]$	109s

Table 2. Experiments on CAE models

6 Conclusion

This paper presents a new methodology for the rigorous design of stochastic adaptive systems. Our model is general, but the verification procedure can only reason on a finite and known set of views. Our formalism is inspired from [24], where both the stochastic extension and high level formalisms are not considered. In future work, we will extend this approach to purely dynamic systems. Another objective is to extend the work to reason about more complex properties such as energy consumption. Finally, we shall exploit extensions of SMC algorithms such as CUSUM [19] which permits to reason on switches of probability satisfaction. This would allow us to detect emergent behaviors.

References

1. Ales Corp.: Advanced laboratory on embedded systems, http://www.ales.eu.com/
2. Arnold, A., Boyer, B., Legay, A.: Contracts and behavioral patterns for systems of systems: The EU IP DANSE approach. In: AiSoS. EPTCS (2013)
3. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
4. Basu, A., Bensalem, S., Bozga, M., Delahaye, B., Legay, A.: Statistical abstraction and model-checking of large heterogeneous systems. Int. J. Softw. Tools Technol. Transf. 14(1), 53–72 (Feb 2012)
5. Boyer, B., Corre, K., Legay, A., Sedwards, S.: Plasma-lab: A flexible, distributable statistical model checking library. In: QEST. LNCS, vol. 8054, pp. 160–164 (2013)
6. Burch, J.R., Clarke, E., McMillan, K.L., Dill, D., Hwang, L.J.: Symbolic model checking: 1020 states and beyond. In: LICS. pp. 428–439 (1990)
7. Cheng et al.: Software engineering for self-adaptive systems: A research roadmap. In: Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525 (2009)
8. Clarke, E., Donzé, A., Legay, A.: On simulation-based probabilistic model checking of mixed-analog circuits. Form. Methods Syst. Des. 36(2), 97–113 (Jun 2010)
9. Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge, MA, USA (1999)
10. Clarke, E., Faeder, J., Langmead, C., Harris, L., Jha, S., Legay, A.: Statistical model checking in biolab: Applications to the automated analysis of t-cell receptor signaling pathway. In: CMSB, LNCS, vol. 5307, pp. 231–250 (2008)
11. DANSE: Designing for adaptability and evolution in sos engineering (dec 2013), https://www.danse-ip.eu/home/
12. Havelund, K., Rosu, G.: Preface: Volume 70, issue 4. ENTCS pp. 201 – 202 (2002), Runtime Verification
13. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: TACAS, LNCS, vol. 2280, pp. 342–356 (2002)
14. Hoeffding, W.: Probability inequalities for sums of bounded random variables. Journal American Statistical Association 58(301), 13–30 (March 1963)
15. Jha, S.K., Clarke, E.M., Langmead, C.J., Legay, A., Platzer, A., Zuliani, P.: A bayesian approach to model checking biological systems. In: CMSB. pp. 218–234 (2009)
16. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV'11. LNCS, vol. 6806, pp. 585–591. Springer (2011)
17. Meyer, B.: Applying "design by contract". Computer 25(10), 40–51 (1992)
18. OMG: Ocl v2.2 (feb 2010), http://www.omg.org/spec/OCL/2.2/
19. Page, E. S.: Continuous inspection schemes. Biometrika 41(1/2), pp. 100–115 (1954)
20. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: CAV. pp. 266–280 (2005)
21. Younes, H., Clarke, E., Zuliani, P.: Statistical verification of probabilistic properties with unbounded until. In: Formal Methods: Foundations and Applications. LNCS, vol. 6527, pp. 144–160. Springer (2011)
22. Younes, S., Clarke, E.M., Gordon, G.J., Schneider, J.G.: Verification and planning for stochastic processes with asynchronous events. Tech. rep. (2005)
23. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: ICSE. ACM (2006)
24. Zhang, J., Cheng, B.H.: Using temporal logic to specify adaptive program semantics. Journal of Systems and Software 79(10), 1361 – 1369 (2006)