Statistical Model Checking of Systems of Systems: An Industrial Approach

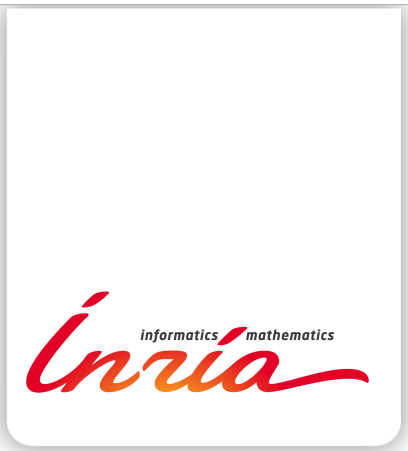
Alexandre Arnold, Massimo Baleani, Alberto Ferrari, Marco Marazza, Valerio Senni, Axel Legay, Jean Quilbeuf, Christoph Etzien

Statistical Model Checking of Systems of Systems An Industrial Approach

Alexandre Arnold, Massimo Baleani, Alberto Ferrari, Marco Marazza, Valerio Senni, Axel Legay, Jean Quilbeuf, Christoph Etzien

RESEARCH REPORT
N° 8828
December 2015
Project-Team ESTASYS

ISSN 0249-6399 ISRN INRIA/RR--8828--FR+ENG



Statistical Model Checking of Systems of Systems
An Industrial Approach

Alexandre Arnold*, Massimo Baleani†, Alberto Ferrari‡, Marco Marazza§, Valerio Senni¶, Axel Legay‖, Jean Quilbeuf**, Christoph Etzien††

Project-Team ESTASYS

Research Report n° 8828 — December 2015 — 16 pages

Abstract: Ensuring a correct behaviour of Systems of Systems (SoS) has a significant social impact. Their complexity and inherent dynamicity pose a serious challenge to traditional design methodologies. We propose a methodology and a tool-chain supporting continuous validation of SoS behaviour against formal requirements, based on a scalable formal verification technique known as Statistical Model Checking (SMC). We integrate SMC with existing industrial practice, by addressing both methodological and technological issues. Our contribution is summarized as follows: (1) a methodology for continuous and scalable validation of SoS formal requirements; (2) a natural-language based formal specification language able to express complex SoS requirements; (3) adoption of widely used industry standards for simulation and heterogeneous systems integration (FMI and UPDM); (4) development of a robust SMC tool-chain integrated with system design tools used in practice. We illustrate the application of our SMC tool-chain and the obtained results on an industrial case study from the DANSE project.

Key-words: systems of systems, FMI, statistical model checking

* AIRBUS
† ALES - UTSCE
‡ ALES - UTSCE
§ ALES - UTSCE
¶ ALES - UTSCE
‖ Inria
** Inria
†† OFFIS

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Résumé : S'assurer qu'un Système de Systèmes (SdS) se comporte correctement est essentiel pour notre société. La complexité et le côté dynamique inhérent à ce type de systèmes constitue un défi pour les méthodes de conception traditionnelles. Nous proposons une méthodologie et une chaîne d'outils permettant de valider la conformité du comportement d'un SdS vis-à-vis d'un ensemble de propriétés formelles. Cette méthodologie est fondée sur la vérification statistique de modèles (SMC). Nous intégrons SMC dans les pratiques industrielles existantes, en résolvant des problèmes méthodologiques et technologiques. Notre contribution comporte: (1) une méthodologie passant à l'échelle pour valider la conformité d'un SdS avec des propriétés formelles ; (2) un langage proche du langage naturel pour exprimer les propriétés formelles complexes attendues du SdS ; (3) l'adoption de standards pour la simulation et l'intégration de modèles hétérogènes largement utilisés dans l'industrie (FMI et UPDM); (4) le développement d'une chaîne d'outils robuste intégrant les outils de conception de systèmes utilisés par les praticiens. Nous illustrons l'utilisation de notre chaîne d'outils et les résultat obtenus avec un cas d'étude industriel provenant du projet DANSE.

Mots-clés : systèmes de systèmes, FMI, vérification statistique de modèles

1 Introduction

A System of Systems (SoS) is a large-scale, geographically distributed set of independently managed, heterogeneous Constituent Systems (CS) collaborating as a whole to accomplish functions that could not be achieved otherwise by any of them, if considered alone [18]. Constituent Systems are loosely coupled and pursue their own objectives, while collaborating for achieving SoS-level objectives. An SoS performs a continuous adaptation to its environment through (1) an evolution of the functions provided by its Constituent Systems and (2) an evolution of its architecture. A typical example of an SoS is the Air Traffic Management System of an airport, which coordinates incoming and outgoing aircraft as well as ground-based vehicles and controllers. Misbehaviours of the functions provided by an SoS can be dangerous and costly. Therefore, it is important to identify a set of analyses and tools to verify that the SoS implementation meets functional and non functional requirements and correctly adapts to changes of the environment during any phase of the design and operation life-cycle. These analyses should support continuous and rapid assessment of SoS behaviour to support run-time decisions.

Recent work promotes simulation techniques as the principal way to perform SoS analysis. In [28] the authors use discrete event specification (DEVS) concepts and tools to support virtual build and test of systems of systems. Their MS4-Me environment enables modelling and simulation (M&S) of SoS by allowing the user to specify constituent systems' behaviour in terms of a so-called Constrained Natural Language. Recent work in [21] provides an overview of the underlying theory, methods, and solutions in M&S of systems of systems, to better understand how modelling and simulation can support the Systems of Systems engineering process. However, simulation is an incomplete analysis and it is not able to assess the likelihood of the simulated behaviours. This is not acceptable from the point of view of SoS analysis, as it does not provide to the designer sufficient confidence of correctness. Other approaches to verification of complex systems are based on exhaustive formal analysis, such as model checking, or simulation-based formal analysis, such as run-time monitoring. Industrial model checking techniques [7] are not adequate to the complexity and dynamicity of SoS. Run-time monitoring does not seem to be adequate to the context of SoS, where failures detection should provide a likelihood estimate of the failure and sufficient time for devising failure-avoidance corrections. In this perspective, a very promising approach to provide sufficient coverage of the SoS behaviour while keeping the analysis cost low is based on Statistical Model Checking (SMC) [27]. SMC is a simulation-based formal analysis providing an estimate of the likelihood of requirement satisfaction and a tunable level of confidence on the goodness of analysis results.

In this paper we describe an industry-oriented tool-chain for SMC-based SoS verification. Our methodology follows the approach to SoS verification developed in the DANSE EU Project, where SoS analysis is a central activity in the continuous SoS development flow. We define a formal specification language which is specific for the SoS context, usable and sufficiently expressive for capturing requirements at the CSs level and at the SoS level. Finally, we focus on ensuring integration of the SMC analysis with existing industrial design tools and standards. In particular, since CSs typically have different models of computation and are developed with distinct authoring tools, we provide an integration framework to handle the heterogeneity of the CSs behavioural models. The contribution of our work is summarized in the following steps: (1) we adopt and extend a very expressive and usable OCL-based formal language (GCSL) to specify requirements on SoS and its CSs; (2) we support wide-spread industry standards, including UPDM [24] for SoS architecture design and the Functional Mock-up Interface (FMI) [23] standard for Constituent Systems behavioural models integration, and (3) we provide a Statistical Model Checking tool-chain, integrated with system design tools used in industrial practice.

The paper is organized as follows. First, in Section 2 we investigate in more details what are

the challenges raised in the analysis of SoS and identify five of them. Then, in Section 3 we provide a summary of the Statistical Model Checking approach adopted in our tool-chain. Following this, in Section 4 we introduce the GCSL formal language for specifying formal requirements on SoS and CSs behaviour. Finally, we discuss in Section 5 the SMC tool-chain and its integration with existing industrial tools for System design. This tool-chain is demonstrated on an industrial case study in Section 6, where we show its application to a Fire Emergency Response system designed in DANSE [2], modelling a complex SoS that manages fire emergencies in a large city.

2 SoS Analysis Challenges

The main characteristics of a System of Systems have been long debated [18]. First, they are dynamic (Constituent Systems may evolve, leave, fail, or be replaced). Second, they are made of heterogeneous and autonomous components (raising a need for their correct integration). Third, control is not centralized but is mostly based on cooperation. Fourth, SoS integration can give rise to unexpected emergent behaviours caused by conflicts in local goals.

Designing appropriate solutions beforehand is beyond reach, thus SoS operation and design is a continuous activity covering its entire life-cycle. This lead in the DANSE EU Project to the development of a new SoS design and operation methodology [10]. A significant part of the methodology is supported by the availability of a formal language for expressing requirements on expected SoS behaviours and of a technology supporting the verification of these requirements. We have identified five challenges that should be addressed by an SoS analysis technology for its adoption in the continuous SoS (re)design and operation.

First Challenge (expressing formal SoS requirements): develop a usable language for expressing formal requirements such that dynamicity is a first-class concept, allowing to abstract from the number and identity of CSs.

Second Challenge (checking achievement of requirements): ensure correct integration and cooperation of CSs, through achievement of both local (CS) and global (SoS) requirements.

Third Challenge (enabling reactivity to change): provide continuous and affordable SoS evaluation to ensure failure prediction and changes evaluation.

Fourth Challenge (detecting emergent behaviours): enable the detection of emergent behaviours and verify the absence of the undesired ones. the DANSE EU Project provided a precise classification of emergent behaviours [10], and SMC contributes in identifying the likelihood of emergent behaviours caused by unpredicted interactions of CSs influencing the achievement of SoS objectives.

Fifth Challenge (ensuring industry acceptance): provide a technology accepted by practitioners, both in terms of usability and in terms of seamless integration with existing industrial practice and tools.

In the rest of the paper we refer to these challenges and illustrate our choices in developing a SoS Statistical Model Checking tool-chain supporting them.

3 Principles of Statistical Model Checking

Analyzing Systems of Systems according to the challenges developed above requires a careful choice of the verification technique to use. The second challenge advocates for a formal verification technique such as model checking, that often requires models to be written in dedicated languages This is in conflicts with industry acceptance (fifth challenge) and even if a complete model was available in a suitable language, analysis would not be feasible because of its very large size. For this reason we rely on Statistical Model Checking (SMC) which we describe in

this section.

In order to perform SMC, the system to analyze needs to be a *stochastic process*. More precisely, each new state is chosen probabilistically based on the current state and possibly on the execution history. SMC scales better than traditional model checking because it relies only on simulations to evaluate the probability that a given property holds. A simulation trace is a finite sequence of timed system states. Throughout this section, we assume the following abstract notion of trace to discuss our formal background.

Definition 1 *Given a set of variables V and their domain $\mathcal{D}$, a state σ is a valuation of the variables, that is $\sigma \in \mathcal{D}^V$. A* trace τ *is a sequence of states and timestamps $(\sigma_0, t_0), \cdots, (\sigma_k, t_k)$, where $\forall i \quad t_i \in \mathbb{R}^+ \wedge t_i < t_{i+1}$.*

3.1 BLTL Linear Temporal Logic

The properties to verify are expressed in BLTL, which is a variant of LTL [20] where each temporal operator is bounded. Bounds make it possible to decide whether the formula holds on a finite (and sufficiently long) trace. The core of BLTL is defined by the following grammar, where the time subscript t is interpreted as an offset from the time instant where the sub-formula is evaluated:

$\varphi \ ::= \ \texttt{true} \mid \texttt{false} \mid p \in AP \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \varphi_1 \, \texttt{U}_{\leq t} \, \varphi_2 \mid \texttt{X}_{\leq t}\varphi$

Here, AP is a set of atomic predicates defined in Section 3.2. In our case, an atomic predicate depends on the past states. The temporal modalities `F` (the "eventually") and `G` (the "always") can be derived from the "until" `U` as $\texttt{F}_t\varphi = \texttt{true}\,\texttt{U}_{\leq t}\,\varphi$ and $\texttt{G}_t\varphi = \neg\texttt{F}_t\neg\varphi$, respectively. The semantics of BLTL is defined with respect to finite traces τ. We denote by $\tau, i \models \varphi$ the fact that a trace $\tau = (\sigma_0, t_0), \cdots, (\sigma_\ell, t_\ell)$ satisfies the BLTL formula φ at point i of execution. The meaning of $\tau, i \models \varphi$ is defined recursively:

$\tau, i \models \texttt{true}$ and $\tau, i \not\models \texttt{false}$;
$\tau, i \models p$ if and only if $p(\tau, i)$ (cf. next Subsection);
$\tau, i \models \varphi_1 \wedge \varphi_2$ if and only if $\tau, i \models \varphi_1$ and $\tau, i \models \varphi_2$;
$\tau, i \models \neg\, \varphi$ if and only if $\tau, i \not\models \varphi$;
$\tau, i \models \varphi_1 \texttt{U}_\texttt{t} \varphi_2$ if and only if there exists an integer $j \geq i$ such that (i) $t_j \leq t_i + t$, (ii) $\tau, j \models \varphi_2$, and (iii) $\tau, k \models \varphi_1$, for each $i \leq k < j$;
$\tau, i \models \texttt{X}_{\leq t}\varphi$ if and only if $\tau^k, 0 \models \varphi$ where $k = \min\{j > i \mid t_j > t_i + t\}$ and $\tau^k = ((s'_0, t'_0), \cdots, (s'_{\ell-k}, t'_{\ell-k}))$ with $s'_i = s_{i+k}$ and $t'_i = t_{i+k}$;

3.2 Atomic Predicates

Usually atomic predicates describe properties of system states, e.g. by comparing a variable with a constant. We propose here an extension where atomic predicates also depend on the past (i.e. from states before the current one). In particular, we are interested in measuring the amount of time during which a given atomic predicate has been true, or the number of steps in which a predicate is true. The syntax for our predicates is as follows:

$AP ::= \texttt{true} \mid \texttt{false} \mid AP \ \texttt{Bop} \ AP \mid \mathit{Nexp} \ \texttt{Cmp} \ \mathit{Nexp}$
$\mathit{Nexp} \ ::= \ \#\texttt{Time} \mid \texttt{Id} \mid \texttt{Constant} \mid \mathit{dur}(AP) \mid \mathit{occ}(AP, a, b) \mid \mathit{Nexp} \ \texttt{Nop} \ \mathit{Nexp}$

Here, `Bop` contains the usual boolean connectors , `Nop` the usual arithmetic operators and `Cmp` the usual comparison operators. Given a trace $\tau = (\sigma_0, t_0), \cdots, (\sigma_k, t_k)$ and a step i, our predicates are interpreted as follows:

$[[\texttt{true}]](\tau, i) = \texttt{true}$ and $[[\texttt{false}]](\tau, i) = \texttt{false}$;
$[[\#Time]](\tau, i) = t_i$ is the simulation time at step i;

$[[\texttt{id}]](\tau, i) = \sigma_i(\texttt{id})$ is the value of var `id` at step i;

Operators and comparisons are interpreted in the standard way;

$[[dur(p)]](\tau, i) = 0, \quad \text{if } i = 1,$
$[[dur(p)]](\tau, i) = dur(p)(\tau, i-1), \quad \text{if } i > 1 \wedge \neg[[p]](\tau, i-1),$
$[[dur(p)]](\tau, i) = dur(p)(\tau, i-1) + t_i - t_{i-1}, \quad \text{otherwise.}$

$[[occ(p, a, b)]](\tau, i) = \sum_{a \leq t_j \leq b} \mathbb{1}_{\{\texttt{true}\}}([[p]](\tau, j))$

The *dur* function computes the amount of time during which the predicate p has been true since the beginning of the trace. The $\#Time$ notation returns the simulation time at the current point. The *occ* function computes the number of steps in which a predicate holds within the given time bound. For instance, $\texttt{G}_{\leq t}(dur(UP) > 0.9 \cdot \#Time)$ is true iff for every step between 0 and t, the amount of time during which UP holds is at least 90% of the elapsed time.

3.3 Statistical Model Checking

Given a stochastic system $\mathcal{M}$ and a property φ, SMC is a simulation-based analysis technique [27, 22] that answers two questions: (1) **Qualitative :** whether the probability p for $\mathcal{M}$ to satisfy φ is greater or equal to a certain threshold ϑ or not; (2) **Quantitative :** what is the probability p for $\mathcal{M}$ to satisfy φ. In both cases, producing a trace τ and checking whether it satisfies φ is modeled as a Bernoulli random variable B_i of parameter p. Such a variable is 0 ($\tau \not\models \varphi$) or 1 ($\tau \models \varphi$), with $Pr[B_i = 1] = p$ and $Pr[B_i = 0] = 1 - p$. We want to evaluate p.

Qualitative Approach The main approaches [26, 22] proposed to answer the qualitative question are based on *Hypothesis Testing*. In order to determine whether $p \geq \vartheta$, we follow a test-based approach, which does not guarantee a correct result but controls the probability of an error. We consider two hypothesis: $H : p \geq \vartheta$ and $K : p < \vartheta$. The test is parameterized by two bounds, α and β. The probability of accepting K (resp. H) when H (resp. K) holds is bounded by α (resp. β). Such algorithms sequentially execute simulations until either H or K can be returned with a confidence α or β, which is dynamically detected. Other sequential hypothesis testing approaches exists, which are based on Bayesian approach [15].

Quantitative Approach In [12, 16] Peyronnet et al. propose an estimation procedure to compute the probability p for $\mathcal{M}$ to satisfy φ. Given a *precision* ϵ, Peyronnet's procedure, which we call *PESTIM*, computes an estimate p' of p with *confidence* $1 - \delta$, for which we have: $Pr(|p' - p| \leq \epsilon) \geq 1 - \delta$. This procedure is based on the *Chernoff-Hoeffding bound* [13], which provides the minimum number of simulations required to ensure the desired confidence level.

The quantitative approach is used when there is no known approximation of the probability to evaluate, i.e. to obtain a first approximation. This method is useful when the goal of the analysis is to have a idea on how well the model behaves. The qualitative approach determines whether the probability is above a given threshold, with a high confidence and in a minimal number of simulations.

4 Timed OCL Constraints for SoS

The challenge in promoting the use of formal specification languages in an industrial setting is essentially to provide a good balance between expressiveness and usability. In this section we present the requirements language, called GCSL (Goal and Contract Requirement Language). The full GCSL language specification, as well as the details of translation of the GCSL *patterns* into BLTL, can be retrieved in [4]. In this section we focus on providing a brief recap of the

language and illustrate why it is appropriate to define SoS and CSs requirements. We also discuss an explicit contribution of this paper to GCSL, which consists of two new patterns and the extension of the supported OCL fragment.

A GCSL contract is a pair of Assume/Guarantee assertions denoting requirements on SoS and CSs inputs and outputs, respectively. Contracts allow us to decompose requirements and perform local or global verification on need. Assertions are built upon GCSL natural-language patterns, some of which are shown in Figure 1. GCSL patterns are inspired by and extend the Contract Specification Language (CSL) patterns [1], developed in the SPEEDS European project. These natural-language based requirements have their formal semantics defined by translation into corresponding BLTL formulas, enabling the application of SMC. To simplify the specification of properties of complex systems and architectures, GCSL integrates the Object Constraint Language (OCL) [25], a formal language used to describe static properties of UML models. OCL is an important means to improve the expressiveness and usability of GCSL patterns. Using OCL we can describe properties about types of CS in the SoS architecture, while being independent of their actual number of instances and, thus, defining requirements that are adaptable to the natural evolution of the SoS, without the need of rewriting them. Consider the following example (based on Pattern 12 from Fig. 1), showing the expressiveness of using OCL within GCSL:

SoS.its(CriticalComponent) → forAll(cc |
  whenever [cc.its(TempSensor) → exists(ts | ts.temp > cc.threshold)] occurs
    [cc.connected(CoolingFan) → exists(f | f.on)] occurs within [1 min, 5 min])

This architecture-abstract requirement says that if any CS of type CriticalComponent has one of *its* TempSensor measuring at time t a temperature that is higher than the threshold set by the *specific* CriticalComponent (the threshold may be different for distinct components) then one of the CoolingFan connected to *that component* should be switched on within the [t+1 min,t+5 min] time frame. This property does not depend on a concrete architecture or on the number of the mentioned CSs but it can be used as a requirement for any SoS having an architecture that integrates the mentioned CSs types.

The idea of mixing OCL and temporal logic originates from the need of specifying static and dynamic properties of object-based systems. In [9, 29] OCL has been extended with CTL and (finite) LTL, respectively, without support for real-time properties. The work in [11] is more similar to ours and it is based on ClockedLTL, a real-time extension of LTL. ClockedLTL is slightly more expressive than BLTL, because it allows unbounded temporal operators, whereas BLTL is decidable on a finite trace. Patterns provide a convenient way to represent frequently occurring and well-identified schemes, while avoiding errors due to the complexity of the underlying logic. The methodology developed during the DANSE EU Project prescribes to have a library of patterns [4] that captures the most relevant temporal constraints for the considered domain. In the rest of this section we are going to discuss a number of these pre-defined patterns. The expressiveness of BLTL, jointly with OCL, makes the patterns library easily extensible to cover future, domain-specific needs.

Figure 1 shows some of the behavioural patterns of GCSL (all the patterns can be found in [4]). Patterns 2 and 3 are typical *safety* properties, where Ψ denotes an argument where an OCL property can be used to describe a state of the SoS. The translation of pattern 2 into BLTL states that the atomic property Ψ should be true at every real-time instant within simulation end time k. Pattern 3 is translated similarly. Pattern 8 shows the joint use of events counting (the n indicates a number of occurrences triggering the pattern) within a real-time interval, such as [*3.2 seconds*, *25.7 minutes*]. Its translation relies on the BLTL operator *occ* and, even if it does not involve any temporal operator, it cannot be decided on a single state but it needs to be checked across the entire trace. In this case we have no explicit mentioning of the simulation

ID	Pattern	(below, k is the simulation time and $a < b \leq k$)
2	always [Ψ]	$G_{\leq k}(\Psi)$
3	whenever [Ψ_1] occurs [Ψ_2] holds	$G_{\leq k}(\Psi_1 \rightarrow \Psi_2)$
	...	...
8	[Ψ_1] occurs at most n times during [a,b]	$occ(\Psi_1, a, b) \leq n$
	...	...
12	whenever [Ψ_1] occurs [Ψ_2] occurs within [a,b]	$G_{\leq k-b}(\Psi_1 \rightarrow X_{\leq a}F_{\leq b-a}\Psi_2)$
13	always during [a,b], [Ψ] has been true at least [e] % of time	$G_{\leq b}(\#Time < a \vee dur(\Psi) \geq (\frac{e}{100} * \#Time))$
14	at [b], [Ψ] has been true at least [e] % of time	$F_{\leq b}(dur(\Psi) \geq \frac{e}{100} * b)$

Figure 1: GCSL Patterns extract and their BLTL translations, with $\Psi, \Psi_1, \Psi_2 \in$ *OCL-prop*, $a, b \in \mathbb{R}$, $e \in$ *OCL-expr*.

OCL-prop ::= **true** | **false** | *FQN* | **not** *OCL-prop* | *OCL-prop* $\circ$ *OCL-prop* |
OCL-expr $\bowtie$ *OCL-expr* | *OCL-coll* $\rightarrow$ **forAll(** *var* | *OCL-prop* [*var*] **)** |
OCL-coll $\rightarrow$ **exists(** *var* | *OCL-prop* [*var*] **)** |
OCL-coll $\rightarrow$ **empty()** | *OCL-coll* $\rightarrow$ **notempty()**
OCL-expr ::= *FQN* | *OCL-coll* $\rightarrow$ **sum()** | *OCL-coll* $\rightarrow$ **size()** | ...
OCL-coll ::= *attribute* | *csName* | **its(***type***)** | **connected(***type***)** | *OCL-coll* . *OCL-coll*

Figure 2: Simplified OCL fragment for GCSL Atomic Properties, with $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ and $\bowtie \in \{>, \geq, =, <, \leq\}$.

time k but we have the overall constraint on all patterns requiring a, b to have appropriate values ($a \leq b \leq k$). Pattern 12 is used to express a *liveness* property which is triggered by an initial condition Ψ_1 occurring at time t and discharged by a following condition Ψ_2 occurring within the interval $[t+a, t+b]$ that is *relative* w.r.t. the time t of occurrence of Ψ_1. Its translation into BLTL underlines a number of important aspects of this pattern. First, the pattern is verified on the entire simulation up to time $k-b$. This is needed because if a condition Ψ_1 occurs at $k-b$, we need to have enough remaining time (actually b) to verify whether either it is discharged by a condition Ψ_2 (making the pattern true) or not. As a consequence, a condition Ψ_1 occurring after time $k-b$ would not require any following discharging condition. Second, on the occurrence of a condition Ψ_1 the pattern requires a shift to time a (as close as possible, depending on the actual states produced by simulation) which is indicated by the next operator $X_{\leq a}$. From that point onwards, we can check for the occurrence of the condition Ψ_2 in the remaining interval using $F_{\leq b-a}$. Finally, patterns 13 and 14 are new w.r.t. [3, 6] and are very suitable for expressing safety and reliability constraints, such as the availability of a SoS, as we are going to demonstrate in Section 6. The BLTL translation of these patterns relies on the novel $\#Time$ and *dur* constructs introduced in Section 3, that were not present in [3, 6]. Pattern 13 checks that, at each simulation point during [a,b], the amount of time during which Ψ has been seen to be true so-far is at least e% of the currently elapsed simulation time. As defined in Section 3, if in (σ_i, t_i) the property Ψ is true, it is also true for the time $t_{i+1} - t_i$, until the next state (σ_{i+1}, t_{i+1}). We use the operator *dur* to accumulate the overall time during which Ψ is true, and we compare the accumulated value with the required portion of the current simulation time, extracted with the operator $\#Time$, at each simulation point. Pattern 14 checks that, within [b] the amount of time during which Ψ has been seen to be true is at least the required portion of b.

Figure 2 shows a (simplified) portion of the grammar defining the OCL integration within the GCSL atomic properties (indicated by *OCL-prop*). Properties are constructed using usual boolean operators ($\circ$) and basic arithmetic comparisons ($\bowtie$) between expressions. Attributes of Constituent Systems can occur in properties or expressions and can be accessed by using their

Fully Qualified Name (*FQN*), such as SoS.Sensor03.isOn.

OCL propositions can describe properties about sets of Constituent Systems that are unknown at requirements-definition time. These sets are left undetermined because (1) the requirements may apply to several variant architectures of the same SoS and (2) the SoS architecture may evolve during the SoS life-cycle. Indeed in [18] one of the five SoS-specific properties of a complex system is that the number and type of systems participating to a SoS may change over time. Ideally, the specifications of the system should remain independent of the number and type of CSs, which is exactly what OCL provides as a feature within GCSL. In order to support properties that are *parametrized* by the SoS architecture, GCSL provides the quantifiers forAll and exists that allow us to instantiate properties over *finite* collections (*OCL-coll*) of Constituent Systems (the *var* ranges over these collections and occurs in the *OCL-prop* which is the scope of the quantifier). A corner case of quantification is provided by the set operators empty and notempty that simply return a truth value after testing the emptiness of the collection. Standard OCL allows to concatenate object names (here indicated as *csName*) by the "."-containment relation, until reaching an *attribute*. In GCSL we add (1) another (weak) containment operator (**its**) that allows to navigate the systems hierarchy in terms of CSs *types* and (2) an operator (**connected**) that allows to navigate the *neighborhood* of a CS, again in terms of *types*. Quantifiers occur also in expressions (*OCL-expr*) and allow to aggregate the values of (equally-typed) attributes. E.g. the simple expression

$$(\mathsf{SoS.its(Sensor).temp} \rightarrow \mathsf{sum}(\,))/(\mathsf{SoS.its(Sensor)} \rightarrow \mathsf{size}(\,))$$

can be used to compute the average temperature in a SoS where the number of CS of type Sensor is unknown or time-dependent.

Since BLTL does not cover the OCL syntax (except for *FQN*), the translation from GCSL to BLTL is done once the architecture is fully know and fixed. OCL operators can be nested and their elimination is performed by induction over the structure of the formula in an outside-in fashion. Conceptually, this can be done by repeated application of three steps: (1) resolution of the outermost OCL collections (that is, replacing a collection with a *finite* set of *FQN*), which eliminates its and connected operators, (2) elimination of the universal (existential) quantifiers, replaced by the corresponding conjunctions (disjunctions) of the instantiated scopes, (3) elimination of the remaining operators by replacement with corresponding Boolean or arithmetic expressions, and (4) recursion on new outermost OCL collections. Termination of this procedure is trivially proved. The tool-chain dynamically evaluates OCL formulas as the architecture changes.

5 SMC Tool-chain

This Section describes the tool-chain we have set-up in the DANSE EU Project to address all the challenges listed in Section 2. The core of our SMC tool-chain is composed of three main tools: IBM Rhapsody is the tool implementing the UPDM language, DESYRE is the tool providing the joint simulation engine for SMC and PLASMA is the tool providing the SMC analysis engine. A SoS model consists of a model of the architecture and a model of each Constituent System. Our choice is to use UPDM to describe the architecture, containing instances of Constituent Systems and their interconnection. In order to encompass heterogeneity of the CSs model, our only requirement is that they comply with the Functional Mock-up Interface (FMI) [23]. Additional modelling and simulation tools like for example Modelica, JModelica, Dymola, Rhapsody and Simulink/StateFlow or any tool exporting models to FMI 1.0 can be seamlessly integrated with this core tool-chain thanks to our choice to adopt that standard. The remaining of this Section describes our core SMC tool-chain.

5.1 Describing SoS Architecture in IBM Rhapsody

UPDM, or Unified Profile for DoDAF[1] and MoDAF[2] [24], is a modelling language based on UML 2 standardized by the Object Management Group (OMG) in 2012. IBM Rational Rhapsody [14] is a model-based system engineering environment implementing industry-standard languages such as UML, SysML and UPDM. The SoS architecture is specified in this tool by using UPDM, conveniently extended with a new DANSE-specific profile enabling (among other features) pseudo-random number generation according to uniform, normal or custom probability distributions to meet SMC analysis specification needs. Rhapsody provides a Java API set for integration with external tools. We developed a Java Exporter Plug-in to translate informations from the UPDM SoS architecture model to a format intelligible by DESYRE, the joint simulation engine.

5.2 Performing Joint Simulation in DESYRE

Joint simulation capabilities are provided by DESYRE [19], a simulation framework based on the SystemC standard and its discrete-event simulation kernel. Inputs to DESYRE are the SoS architecture exported from Rhapsody and the Functional Mock-up Units (FMUs) associated to CS types. Joint simulation of several FMUs, that are units complying with the FMI standard, is implemented by a Master Algorithm (MA), with two alternatives. In *co-simulation*, each FMU embeds its own ODE solver and computes autonomously the evolution of its continuous-time variables. In *model exchange*, the MA is in charge of computing evolution of continuous-time variable. In general the implementation of a master algorithm (MA) is not a trivial task having to guarantee: (1) correctness of the composition according to the model(s) of computation (MoC) of both the host environment and the constituent FMUs, (2) termination of the integration step and (3) determinism of the composition. Challenges related to the implementation of master algorithms for model composition, have been extensively addressed in the literature. In [17] the authors define the operational and denotational semantics of the (hierarchical) composition of Synchronous Reactive (SR), Discrete Event (DE), and Continuous Time (CT) models. Termination and determinacy properties of MA for co-simulation are addressed in [8].

5.2.1 DESYRE Master Algorithm

Within the context of the DANSE EU Project a specific FMI master algorithm has been developed in DESYRE to address the unique needs of Systems of Systems simulation and SMC. The main focus is on simulation efficiency due to SoS model complexity and large observation (i.e. simulation) time span (up to several years) and to support large number of runs (tens or hundreds of thousands) as required by SMC analysis. The MA builds on a set of assumptions that are typically satisfied by the CS models used within the DANSE context. The choice of a MA for model exchange rather than co-simulation provides us with full control of the overall integration algorithm. The MA assumes that none of the FMUs contains *direct feed-through* i.e. FMU output does not depend on the value of its inputs at the current simulation time, removing the need for a causality analysis during fixed-point computation.

Lines 1 to 11 in Algorithm 1 represent the initialization phase, while lines 12 to 25 describe the SoS system simulation loop. The algorithm determines the time synchronization instants for the different FMUs composing the SoS model. Time synchronization points represent those time instants in which (1) the different FMUs are executed, (2) the generated outputs are propagated

[1] Department of Defence Architecture Framework
[2] Ministry of Defence Architecture Framework

Algorithm 1 DESYRE FMI MA for DANSE.

Input: $simStartTime, simEndTime, maxIntStep$;

```
1:  simTime = simStartTime;
2:  isCT = determineIfCT();
3:  for all cs ∈ csList do
4:      csEvt = cs.initialize(simTime);
5:      if (csEvt ≠ ∅); then
6:          evtQueue.addEvt(cs.getID(), csEvt);
7:      end if
8:      if (((evtQueue.getClosestEvtTime()−simTime) > maxIntStep) and isCT) then
9:          evtQueue.addEvt(cs.getID(), maxIntStep);
10:     end if
11: end for
12: while (simTime ≤ simEndTime and not(simStopEvt)) do
13:     simTime = getSimTime();
14:     while (not(isSoSFixPtReached())) do
15:         for all cs ∈ csList do
16:             cs.updateDiscrState(simTime);
17:         end for
18:     end while
19:     for all cs ∈ csList do
20:         cs.updateContState(simTime);
21:     end for
22:     evtQueue.updateEvts();
23:     simTime = evtQueue.getClosestEvtTime();
24:     waitNextActivationEvt();
25: end while
```

among their interfaces (line 16) and (3) FMU continuous state is updated (line 20). Synchronization points are calculated based on time events, state events and step events notified by the different FMUs [23].

5.3 SMC Analysis in PLASMA

PLASMA is a tool for performing SMC analysis. The core of PLASMA is a set of SMC algorithms, including those presented in Subsection 3.3. This core is completed by two types of plug-ins, that are controlled by the SMC algorithm. Simulator plug-ins implement or interface PLASMA with a simulator, in order to produce a trace. Such plug-ins enable verification for RML, SystemC or Matlab/Simulink models. Specification languages are added by providing checker plug-ins that verify whether a finite trace satisfy a property. For instance, there is a checker plug-in for handling BLTL. For the work presented in this paper, we built a new simulator plug-in to integrate PLASMA with DESYRE. The interaction between them is described in details in [19]. We also extended the BLTL plugin to provide new primitives, namely dur, occ and $\#Time$.

6 Industrial Case Study

6.1 Modeling

We modelled an emergency response SoS for a city fire scenario in UPDM. The city is partitioned into 10 districts and the example is focused on a few fire-fighting constituent systems (CS) for which the behaviour was modelled: the Fire Head Quarter (FireHQ), the Fire Stations, the Fire Fighting Cars, the Fire Men. The SoS integrated architecture was built by instantiating the CSs and by specifying how to connect them through an Internal Block Diagram, shown in Figure 3. The behaviour of the CSs has been modelled in several FMI-compliant authoring tools. For example, the FireMan has been modelled in OpenModelica (as shown in Figure 4)

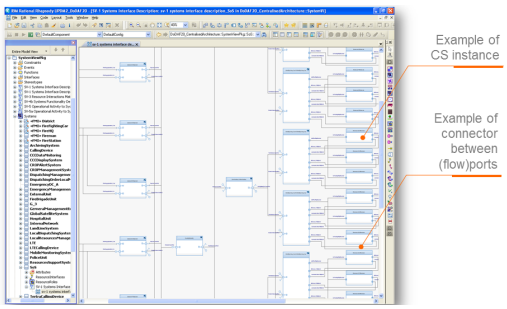


Figure 3: SoS architecture in Rhapsody

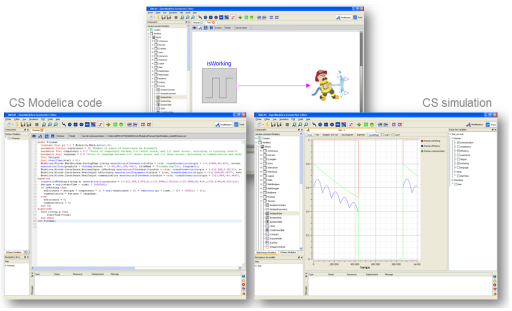


Figure 4: CS behaviour modelled in other tools (e.g. OpenModelica)

which is an open-source multi-domain modelling tool based on the Modelica language. Other CSs have been modelled using Rhapsody state-charts. The SoS architecture is exported to DESYRE using a DANSE-specific exporter plug-in. Each CS behavioural model is exported from the corresponding authoring tool into FMUs, according to the FMI standard. This enables the DESYRE platform to simulate the whole SoS model and plot some selected variables, as shown in Figure 5.

6.2 Expressing Goals of the SoS

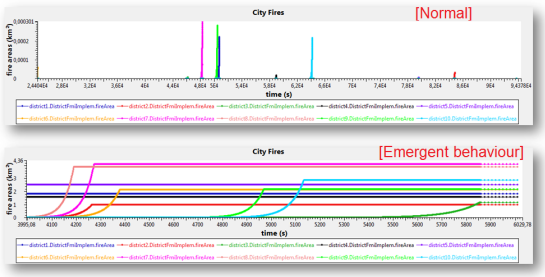


Figure 5: Simulation results in DESYRE

We now show how our tool chain answers Challenge 1 from Section 2. In order to validate our model, we want to check that during simulations, the fire area remains small enough. In order to define "small enough" independently of the number of components, we require that the fire is less than a given percentage of the total area. Each district has two variables of interest, its area and the fire area.

Our first formulation states that the fire area is less than X percent of the total area. The total fire area is the sum of the fire area in each district, which can be expressed in GCSL by `SoS.its(Districts).fireArea->sum()`. We define Pattern 1 as follows:

always [SoS.its(Districts).fireArea → sum() > $(X/100)$*SoS.its(Districts).area → sum()]

As Pattern 1 might be too strong, we propose an alternative formulation. More precisely, we allow the fire area to exceed X% of the total area, but no more than 10% of the time. For technical reasons, we define Pattern 2 as the negation of the above property, namely:

at [10000], [SoS.its(Districts).fireArea → sum() >
$(X/100)$*SoS.its(Districts).area → sum()] has been true at least [10] % of time

Pattern 2 is true whenever the fire area is above the threshold for more than 10% of the time, that is when the SoS behaves incorrectly.

6.3 Emergent Behaviours Detection and Evaluation

The fourth challenge described in Section 2 is about the detection and analysis of emergent behaviour. In our case, simulation allowed us to detect an (undesired) emergent behaviour

which is depicted in the lower part of Figure 5. Our analysis of this emergent behaviour is the evaluation of the probability of its occurrence. The first step is to define a GSCL pattern that characterizes the absence of the emergent behaviour. One key characteristic of this behaviour is that fires spread over entire districts. We assume that the emergent behaviour does not occur if there is no area where the fire has taken the whole district. This is specified in Pattern 3: always [SoS.its(Districts) → forAll(district |district.fireArea < district.area)]

6.4 Analysis and Discussions

We show here how the second challenge from Section 2 is handled by our methodology. More precisely, we evaluate the probability that Pattern 1, 2 and 3 hold. As explained in Subsection 3.3, SMC provides an approximation of such a probability. As we don't know which value to expect, we use the *PESTIM* method. This analysis is parameterized by an allowed error ϵ, and a confidence $1 - \delta$. We chose an error of 0.1 and a confidence of 99% ($\delta = 0.01$), which is attained with 265 simulations. The length of a simulation is set to 10000s. We present the results for Pattern 1 in Table 1. As expected, the probability that the fire remains smaller than X% of the

Table 1: Probability that fire is always smaller than X percent of the total area during 10000 seconds.

X	**Probability**	**Analysis Time**
1	0.98490566	0:34:23
0.1	0.954716981	0:39:54
0.01	0.966037736	0:31:03
0.001	0.939622642	0:36:14
0.0001	0.603773585	0:28:49
0.00001	0.350943396	0:25:23

Table 2: Probability that fire area is smaller than X percent of the total area at least 90% of the time.

X	**Probability**	**Analysis Time**
1	0.954716981	00:40:05
0.1	0.981132075	00:34:25
0.01	0.966037736	00:43:22
0.001	0.977358491	00:42:37
0.0001	0,973584906	00:42:59
0.00001	0,996226415	00:37:25

total area increases when X increases. Indeed, "the fire area remains smaller than X% of the total area" implies that "the fire area remains smaller than Y% of the total area" for any $Y \geq X$. However, the probability returned is an approximation, with an error up to 0.1 with a confidence of 99%. Therefore, the fact that the probability decreases from 0.96 to 0.95 when X increases from 0.01 to 0.1 is not significant as the difference between the two values is less than the error. However, the difference between the probabilities obtained for $X = 0.0001$ and $X = 0.001$ are significant since they are more than twice the error. In our model, the total area is about 23 square kilometres. Therefore the two last lines of the table correspond to respectively an area of 23 and 2.3 square meters.

In order to obtain the probability presented in Table 2, we had to complement the probability that Pattern 2 holds. We obtain the probability that the fire is smaller than X percent of the total area for at least 90% of the time (over 10000s). As discussed previously, since for each value of X we ran a different set of simulation, it's not clear that the probability that the pattern holds increases when X increases. With this more permissive definition, we see that even small fires have a low probability to stay on for more that 10% of the simulation time. By comparing with Table 1, we can conclude that frequently occurring fires (i.e. very small ones) are quickly extinguished, because the probability of the last two lines are significantly higher in Table 2.

Finally, we evaluate the probability to obtain the emergent behaviour depicted in Figure 5, that is the probability that Pattern 3 holds. The returned result is 0.9622, which means that the real probability that the contract holds is between 0.8622 and 1 with a confidence of 99%.

We showed here how our tool chain is used to evaluate whether a given pattern holds. By

evaluating the probability of Pattern 1, 2 and 3, we were able to discover that small fire occur often (last two lines of Table 1) but are not likely to last long (last two lines of Table 2). Finally, the emergent behaviour occurs with a probability between 0 and 0.14, which explains why Pattern 1 and Pattern 2 do not occur with a probability of 1. This problem could be resolved by studying the causes of the emergent behaviour and evolving the SoS to avoid it, for instance by adding more fire fighting cars.

At this level of analysis, a precision of 0.1 is sufficient to obtain a good general idea about the probability that each of the patterns occur. In general, using SMC requires to find the appropriate trade-off between the required precision and the time available for the analysis and subsequent re-engineering.

We address the third challenge, evolution, by having Patterns that are independent on the actual number of components. Indeed, adding constituent systems such as districts or cars, even if they have a new behaviour, do not require specifying new patterns. The analysis is still possible on the modified SoS model.

In the frame of the DANSE project, Industrial Partners built models of their SoS under analysis. SMC and other methods provided them a higher confidence in their models [5]. More precisely, one Partner verified Mean Time Between Failures (safety) requirements in an Air Traffic Control case study. Another Partner verified sufficient water availability (robustness to failures) in a water distribution system of national scale.

Acknowledgements

The research presented in this paper was funded by the FP7 EU project DANSE under grant agreement no. 287716. We acknowledge the contribution by Alessandro Mignogna and Alessandro Ulisse (ALES-UTSCE) for the realization of the DESYRE simulation tool. In particular, the DESYRE Master Algorithm has been developed by the former, while the support for the FMI 1.0 standard has been developed by both of them.

References

[1] Speeds (2008): D2.5.4: Contract Specification Language. Public deliverable, Available at `http://speeds.eu.com/downloads/D_2_5_4_RE_Contract_Specification_Language.pdf`, 2008.

[2] V. Aa. D3.3.2 - Concept Alignment Example description. Technical report, DANSE deliverable, 2013.

[3] V. Aa. D6.3.2 - Goal and Contract Specification Language. Technical report, DANSE deliverable, 2013.

[4] V. Aa. D6.3.3 - GCSL Syntax, Semantics and Meta-model. Public deliverable, DANSE deliverable, 2015.

[5] V. Aa. DANSE Final Report. Technical report, DANSE deliverable, 2015.

[6] A. Arnold, B. Boyer, and A. Legay. Contracts and behavioral patterns for sos: The EU IP DANSE approach. In K. Larsen, A. Legay, and U. Nyman, editors, *Proc. 1st Workshop on Advances in Systems of Systems (AiSoS) 2013*, volume 133 of *EPTCS*, pages 47–66, 2013.

[7] J. Boulanger. *Industrial Use of Formal Methods.* John Wiley and Sons, 2012.

[8] D. Broman, C. Brooks, L. Greenberg, E. Lee, M. Masin, S. Tripakis, and M. Wetter. Determinate composition of fmus for co-simulation. In *Proc. of the 11th ACM Int.Conf. on Embedded Software*, EMSOFT '13, pages 2:1–2:12. IEEE Press, 2013.

[9] D. Distefano, J. Katoen, and A. Rensink. On a temporal logic for object-based systems. In S. Smith and C. Talcott, editors, *4th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS) 2000*, volume 177 of *IFIP Conference Proc.*, pages 305–325. Kluwer, 2000.

[10] L. M. et al. D4.3 - DANSE Methodology V2. Technical report, DANSE deliverable, 2013.

[11] S. Flake and W. Müller. Past- and future-oriented time-bounded temporal properties with OCL. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004)*, pages 154–163. IEEE Computer Society, 2004.

[12] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate Probabilistic Model Checking. In *VMCAI*, pages 73–84, 2004.

[13] W. Hoeffding. Probability inequalities. *J. of the American Statistical Association*, 58:13–30, 1963.

[14] IBM. Rhapsody.

[15] S. Jha, E. Clarke, C. Langmead, A. Legay, A. Platzer, and P. Zuliani. A bayesian approach to model checking biological systems. In P. Degano and R. Gorrieri, editors, *Computational Methods in Systems Biology*, volume 5688 of *Lecture Notes in Computer Science*, pages 218–234. Springer Berlin Heidelberg, 2009.

[16] S. Laplante, R. Lassaigne, F. Magniez, S. Peyronnet, and M. de Rougemont. Probabilistic abstraction for model checking: An approach based on property testing. *ACM TCS*, 8(4), 2007.

[17] E. A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 114–123. ACM, 2007.

[18] M. Maier. Architecting principles for systems-of-systems. *Systems Engineering*, 1(4):267–284, 1998.

[19] A. Mignogna, L. Mangeruca, B. Boyer, A. Legay, and A. Arnold. Sos contract verification using statistical model checking. In K. Larsen, A. Legay, and U. Nyman, editors, *Proc. 1st Workshop on Advances in Systems of Systems (AiSoS) 2013*, volume 133 of *EPTCS*, pages 67–83, 2013.

[20] A. Pnueli. The temporal logic of programs. In *Proc. of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[21] L. Rainey and A. Tolk. *Modeling and simulation support for system of systems engineering applications*. Wiley, Jan 2015.

[22] K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In *CAV*, LNCS 3114, pages 202–215. Springer, 2004.

[23] Vv.Aa. FMI standard specification. Modelica association, `https://www.fmi-standard.org/`, 2012.

[24] Vv.Aa. UPDM 2.0 formal specification. OMG, `http://www.omg.org/spec/UPDM/2.0/`, 2012.

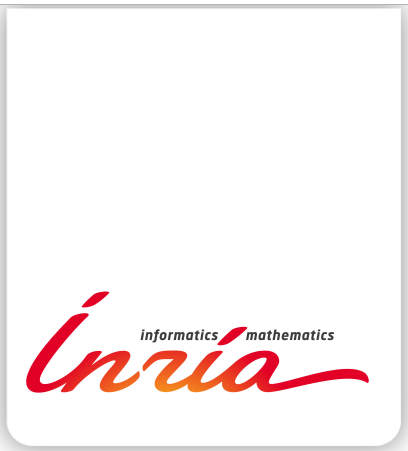
[25] Vv.Aa. OCL language specification. OMG, `http://www.omg.org/spec/OCL/`, 2014.

[26] H. Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Carnegie Mellon, 2005.

[27] H. Younes and R. Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. *Inf. Comput.*, 204(9):1368–1409, 2006.

[28] B. Zeigler. *Guide to Modeling and Simulation of Systems of Systems - User's Reference*. Springer Briefs in Computer Science. Springer, 2013.

[29] P. Ziemann and M. Gogolla. OCL extended with temporal logic. In M. Broy and A. Zamulin, editors, *5th Conference on Perspectives of Systems Informatics (PSI) 2003*, volume 2890 of *LNCS*, pages 351–357. Springer, 2003.



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