EAI Endorsed Transactions
on Industrial Networks and Intelligent Systems

Research Article ICST.ORG

An Application of SMC to continuous validation of heterogeneous systems

Alexandre Arnold[1], Massimo Baleani[2], Alberto Ferrari[2], Marco Marazza[2], Valerio Senni[2,*], Axel Legay[3], Jean Quilbeuf[3], Christoph Etzien[4]

[1]name.surname@airbus.com, AIRBUS, Toulouse, France
[2]name.surname@utrc.utc.com, ALES - UTRC, Roma, Italy
[3]name.surname@inria.fr, Inria, Rennes, France
[4]name.surname@offis.de, OFFIS, Oldenburg, Germany

Abstract

This paper considers the rigorous design of Systems of Systems (SoS), i.e. systems composed of a set of heterogeneous components whose number evolves with time. Such components cooperate to accomplish functions that they could not achieve in isolation. Examples of SoS include smart cities or airport management system. The dynamical evolution of SoS behavior and architecture makes it impossible to design an appropriate solution beforehand. Consequently, existing approaches build on an iterative process that takes SoS evolution into account. A key challenge in this process is the ability to reason about and analyze a given view of the SoS (on a fixed number of SoS constituents) with respect to a set of goals, and use the results to eventually predict the evolution of the SoS. To address this challenge, we rely on a scalable formal verification technique known as Statistical Model Checking (SMC). SMC quantifies how close the current view is from achieving a given mission. We integrate SMC with existing industrial practice, by addressing both methodological and technological issues. Our contribution is: (1) a methodology for validation of SoS formal requirements; (2) a formal specification language able to express complex SoS requirements; (3) the adoption of current industry standards for simulation and heterogeneous systems integration ; (4) 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 a case study.

Received on 22 November 2016; accepted on 14 January 2017; published on 01 February 2017

Keywords: Systems of systems, statistical model checking, FMI, tool-chain, simulation.

doi:10.4108/eai.1-2-2017.152154

1. Introduction

Context and challenges A System of Systems (SoS) is a large-scale, geographically distributed set of independently managed, heterogeneous Constituent Systems (CS). The heterogeneity of CSs arise from their variations in nature, time scale, and model of computation. Constituent systems are collaborating as a whole to accomplish functions and goals that could not be achieved otherwise by any of them, if considered alone [1]. Those goals can either be global to the system or local to some of its constituents. Constituent Systems are loosely coupled and pursue their own objectives, while collaborating for achieving SoS-level objectives.

★Research supported by the European Community's Seventh Framework Programme [FP7] under grant agreement no 287716 (DANSE).

★★This paper is an extension of the paper "An Application of SMC to continuous validation of heterogeneous systems." published in the proceedings of the SIMUTOOLS 2016 conference.

*Corresponding author. Email: valerio.senni@utrc.utc.com

An SoS adapts itself 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 aircrafts as well as ground-based vehicles and controllers. Such an SoS may evolve in order to accommodate a larger number of passengers, difficult climatic conditions or a modification in the laws regarding security.

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. The manager of a SoS should be able to run these analyses in order to take decisions regarding the evolution of the SoS.

The main characteristics of an SoS have been long debated since [1]. One of the key characteristics of an SoS is *dynamical evolution*, that is, the fact that Constituent Systems may evolve, leave, fail, or be replaced. In fact, dynamical evolution of SoS makes it impossible to design an appropriate solution beforehand. Consequently, the design flow of a SoS is reiterated multiple times during the operation of the SoS, in order to adapt to its evolutions.

As a summary, SoS rigorous design methodologies introduce two major challenges. The first one is to check whether a given view of the system (i.e. a fixed number of components) is able to achieve a mission. The second challenge is to exploit the solution to the first challenge in an engineering methodology that permits dynamical evolution of the system in case new missions are proposed, in case the environment has changed, or in case the current view cannot achieve the mission. This design approach has been followed by the DANSE project which developed a new SoS design and operation methodology [2].

Contributions This paper focuses on the first challenge described above, that is to reason on a given view of the system. A corner stone to solve the challenge is to develop a usable language for expressing formal requirements independent from the number and identity of CSs and thus from the architectural choices.

In this paper, we propose Goals and Contracts Requirement Language (GCSL), which is an expressive pattern-based language designed to specify requirements on an SoS and its CSs. The language uses an extension of OCL to express constraints independently of the view such as: "the total fire area (over a set of districts) is smaller than 1 percent of the total area of the districts". By combining those constraints with temporal patterns, we express timing requirements on the behaviors of the view. For instance, the GCSL formula "always [SoS.itsDdistricts.fireArea→sum()< 0.01∗SoS.itsDistricts.area→sum()]" checks that the above constraint is verified at any point of the simulation. A first version of GCSL was introduced in [3]. In this paper we propose additional patterns for expressing properties about the amount of time during which a given predicate remains satisfied. We show that GCSL is powerful enough to capture most of SoS goals and emergent behaviors proposed by our industry partners. The reader shall observe that the language can methodologically be extended to capture more requirements on demand.

The second main difficulty is to detect emergent behaviors and verify the absence of undesired ones. This requires a suitable verification technique. A first solution would be to use formal techniques such as model checking. Unfortunately, both the complexity and the heterogeneous nature of the constituent systems prevent this solution. To solve this problem, we rely on Statistical Model Checking (SMC) [4]. SMC works by monitoring executions of the system, and rely on statistics to assess the overall correctness. Contrarily to classical Validation techniques, SMC quantifies how close the view is from achieving a given mission. This information shall later be exploited in the reconfiguration process.

The objective of this paper is not to improve existing statistical algorithms, but to integrate the SMC verifier PLASMA [5] into a full tool chain for verifying heterogeneous systems. PLASMA proposes several statistical algorithms such as Monte Carlo or hypothesis testing. In order to implement an SMC algorithm for SoS, one has to propose a simulation approach for heterogeneous components as well as a monitoring approach for GCSL requirements. While a GCSL monitor is easily obtained by translating GCSL to Bounded Temporal Logic (BLTL) as in [6], simulating SoS requires that one defines a formal representation for SoS constituents. In this work, we exploit the Functional Mock-up Interface (FMI) [7] standard as a unified representation for heterogeneous systems. In recent years, the development of DESYRE [8–10], a simulation engine based on the System-C standard, provided us joint simulation for FMI/FMU. One of the main contributions of the paper is a full integrated tool-chain between IBM Rhapsody, the statistical model checker PLASMA [5] and DESYRE. This tool-chain is, to the best of our knowledge, the first one offering a full SMC-based approach for the verification of complex heterogeneous systems.

The third main difficulty of our work is to make sure that the technology will be accepted by practitioners, both in terms of usability and seamless integration with existing industrial practice and tools. Our

first contribution provides a solution for expressing requirements. However, we need a language to specify the system itself. In this paper, we propose to support wide-spread industry standards for SoS. This is done by exploiting UPDM [11] for SoS architecture design and the FMI standard for constituent systems integration. We build on IBM Rhapsody which is a tool used to describe UPDM views, that we enrich with probability distributions. Using probability distributions allows the designer to abstract away complex behaviors as well as model inputs from the environment, failure of components, and in general phenomena for which a probabilistic description is known. Finally, once the model is fully designed, an integrated tool-chain with a single GUI enables the simulation and verification of the model.

Our approach is illustrated on a fire emergency response SoS, at the scale of a city. The constituent systems are the districts of the city, firefighting cars, firemen, fire stations and the central command center. We show how our tools are used to model the architecture and the constituents systems of the SoS. The architecture indicates how the Constituent Systems cooperate to provide an emergent behavior, namely the extinction of the fires. Using our tool-chain, we evaluate the probability of this emergent behavior.

Structure of the paper The paper is organized as follows. In Section 2, we present the modeling of SoSs. We then explain in Section 3 how to produce a simulation trace from the models. In Section 4 we provide a summary of the Statistical Model Checking approach adopted in our tool-chain. Following this, in Section 5 we introduce the GCSL formal language for specifying formal requirements on SoS and CSs behavior. Finally, we discuss in Section 6 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 7, where we show its application to a Fire Emergency Response system designed in DANSE [12], modeling a complex SoS that manages fire emergencies in a large city.

2. SoS Modelling

A SoS model consists of a model of the SoS architecture and a set of models formalizing each Constituent System's behavior. We chose the UPDM language to describe both the SoS architecture and Constituent System's behavior, even though the latter can be described through other modelling languages, as explained in the following.

UPDM, or Unified Profile for DoDAF[1] and MoDAF[2] [11], is a modeling language standardized by the Object Management Group (OMG) in 2012. UPDM 2.0 defines a UML 2 and optional SysML profile to model architectures, support analysis, specification, design and verification for a broad range of complex systems, including SoS architectures and service oriented architectures. Through the *profile* extension mechanism inherited from UML, UPDM allows adding, extending and customizing standard profiles by creating libraries of new components and *stereotypes* to further refine domain-specific concepts. A stereotype defines how an existing UML, SysML or UPDM metaclass can be extended. It can be applied to a configurable set of model elements i.e. for replacing existing features or for enabling new ones. A stereotype can have properties, referred to as *tags* definitions [13]. When a stereotype is applied to a model element, the value of the tags can be set for that model element. The remaining of this Section describes how we extended UPDM to accomplish architectural and behavioural representations of SoSs supporting statistical model checking analysis.

2.1. Modelling Constituent Systems' Behaviour

In order to model an SoS, we need to specify the behavior of each of the constituent systems. In order to perform SMC, the model needs to exhibit a stochastic behaviour. Typical examples when stochastic behaviours are relevant in a model include formalizing variability of external inputs and waiting time before events' occurrence, unpredictability of components' failure, and so on.

In our approach we leave total freedom to the CS designer to use any modelling language, provided that the modelling tool implementing that language is able to export the CS model according to the FMI 1.0 standard [7]. To introduce a stochastic behavior, the designer can use functions provided by the specific tool used for modelling.

UPDM is one of the tool that can be used to model constituent systems. In that context the designer can rely on a DANSE extension to the UPDM profile developed specifically for statistical model checking. We have defined the «Stochastic» profile with a set of ad-hoc stereotypes to support stochastic variables in SV-10b viewpoint (state-charts), the UPDM view we use to model CS behaviours. Our stochastic profile defines Uniform, Normal and CustomDistribution distributions stereotypes on the Integer and Real domains; custom distribution variables are based on user-defined probability distribution functions (further

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

details can be found in [3]); each of the two "Uniform" stereotypes (for Integer and Real values respectively) has two tags to specify the *min* and *max* bounds of the range interval, whereas each of the two Normal stereotypes has two tags to specify the desired *mean* and *standard deviation* parameters.

Again, UPDM is not the only language we support to model Constituent Systems' behaviour. Some languages are tailored for modeling specific physical phenomena, while some others are tailored to model control systems. The designer is free to use the language that best fits the system to describe, as long as the language can export to FMI.

2.2. Modeling the architecture

The architecture of the SoS specifies how instances of constituent systems are interconnected. An SoS does not have a fixed number of CS instances over its execution because CSs are independently managed and may decide to leave or join a given SoS on their own. Therefore, rather than defining exactly how many CS are present in the SoS, we first specify an *abstract architecture* that defines the kind of CSs that compose the system and how they can interconnect with other CSs. For simulation, we fix a *concrete architecture* that is a possible instance of the abstract architecture. Our need for describing the architecture is satisfied by the UPDM 2.0 System Viewpoint 1 (SV-1).

The SoS Abstract Architecture. The abstract architecture is specified using the SV-1 Block Definition Diagram (BDD), a representation of the SoS hierarchy defining:

1. the System of Systems and its parameters,
2. the type of each Constituent System,
3. the interface in terms of ports and their type of each Constituent System,
4. the CS parameters, along with their types and default value, and
5. the composition relations between the SoS and its CSs.

Each type of Constituent System that can participate in the SoS should be materialized in the abstract architecture by a «System» block. This block specifies the expected parameters and state variables for a constituent system of that type. It also defines the default parameters for CSs of that type. Of course, the model of the CS given to the simulator has to conform to the description given by the «System» block in the abstract architecture.

The SoS Concrete Architecture. In order to define the concrete architecture, we rely on the SV-1 Internal Block Diagram (IBD), an integration view of the SoS defining:

1. a set of instances of CS types specified in the SoS abstract architecture,
2. the role of each CS instance in the SoS, including instance-specific parameter assignments, and
3. the connections between ports of CS instances, according to the abstract architecture.

An instance of a «System» block specified in the SoS abstract architecture (UPDM SV-1 BDD) is materialized by a «ResourceRole» block in the SoS concrete architecture (UPDM SV-1 IBD). The *Value* tag of UPDM *Attributes* of each instance can be defined to the default parameters set in the abstract architecture (BDD) or overridden for each instance of the concrete architecture (IBD).

Besides the «Stochastic» profile described in Section 2.1, we defined two other profiles to provide information for joint simulation and to express requirements. With these three profiles we can specify all the information needed to simulate and verify SoSs through statistical model checking.

Providing Information for Joint Simulation. To provide information for joint simulation our «*Simulation*» profile defines the following stereotypes: «*FMI*», «*Traceable*» and «*FMIIgnore*».

In order to simulate an SoS model, the simulation engine needs to load the model of each constituent system. In our case, the model of the system is an executable file (Functional Mockup Unit or FMU). The «*FMI*» stereotype allows the designer to link a «System» block from the abstract architecture to a FMU, by specifying an URI for that FMU. The link «*FMI*» stereotype can also be applied to a particular instance («ResourceRole» block) in the concrete architecture to link that instance to a specific FMU.

SMC analysis requires the observation of the SoS model variables that appear in GCSL contracts. In particular, the SMC checker will expect a trace that contains the evolution of these variables. To allow the simulation tool to properly generate observers to track values of the needed variables, we define the «*Traceable*» stereotype. This stereotype can be applied to any attribute of a «ResourceRole» or a «System» element to declare that the value of this attribute should appear in the output of the simulator.

Sometimes, an SoS model contains information that is not needed for a particular analysis. Filtering out such extra information to leave only the one relevant for that analysis is enabled by our «*FMIIgnore*» stereotype, applicable to model elements' ports and

attributes. When this stereotype is applied to the both ports involved in a connection, the simulator ignores that connection. Connections between ports with «*FMIIgnore*» applied and ports without this stereotype are not allowed. Consistently, the corresponding CS behavioural model shall not generate any data flowing through ports with the «*FMIIgnore*» stereotype.

Expressing Requirements. To pursue our SMC analysis goals on SoS we have defined a number of stereotypes collected by the «*GoalsAndContracts*» profile. One of the stereotypes of the «*GoalsAndContracts*» profile associates GCSL contracts to elements of the abstract architecture, possibly including the root element. To express contracts on model elements throughout our SoS UPDM SV-1 diagrams we rely on the UPDM "Constraint" model element inherited from UML [11]. To link a contract to CS or SoS «System» elements in a UPDM BDD, or to specific CS instances, stereotyped as «ResourceRole» in a UPDM IBD, we use the "constrainedElement" association provided by the "Constraint" model element. We have further extended the features of "Constraint" by defining the «gcsl» stereotype. The «gcsl» stereotype defines tags to specify (1) the GCSL assumption in textual form, (2) the GCSL guarantee in textual form, (3) the probability threshold with which the GCSL requirement has to be satisfied, and (4) the GCSL requirement identifier as an alphanumeric string.

3. Performing Joint Simulation in DESYRE

3.1. Integrating Heterogeneous CSs' Behaviours

Constituent Systems are systems that evolve according to their own nature and mission, independently from the SoS. The nature of the CS implies that its behaviour be modelled using the most appropriate modelling tool for the specific application, which typically leads to a set of heterogeneous format of model representation and interfaces. Formalizing different Constituent Systems' behavior requires supporting a variety of models of computation, including continuous-time dynamics, discrete-time dynamics with variable or fixed integration step intervals and event-based dynamics. Very often an SoS model is composed of CS models whose integration time-step varies of several orders of magnitude. For instance, an SoS can compose systems which act every second with systems which act every month. For these reasons we refer to a SoS model as being a composition of *heterogeneous* Constituent Systems' models. To guarantee correct integration and simulation of heterogeneous CSs models to support statistical model checking we need to translate all these differences to a common format. Our approach requires CS behavioural models to comply with the Functional Mock-up Interface (FMI) [7].

A number of approaches are possible for integrating Heterogeneous CSs' Behaviours. Co-simulation coordinates distinct simulation tools to provide a global simulation of the overall system. The integration can be performed using simulators' API, a common proprietary interface, such as the Simulink S-function, or a standard interface, for instance FMI for co-simulation. The approach proposed in this paper is based on "joint-simulation": a single simulator integrates the models of system components (i.e. CS models in the context of this work), each consisting of differential, algebraic and discrete equations describing components' continuous and discrete dynamics. To maximize tool interoperability the choice for a common interface to components' equations has fallen on the one defined by the Functional Mock-up Interface (FMI) standard [7].

FMI is a tool-independent standard whose development was initiated within the ITEA2 project MODELISAR with the objective of easing the exchange of simulation models between different entities despite the usage of a variety of different tools. FMI standard was developed as a collaborative effort involving simulation tool vendors and research institutes. FMI standard maintenance and development (FMI 2.0 has been published in July 2014) is today performed by the Modelica Associations and it is currently supported by tens of academic and industrial tools.

FMI defines the interface of an executable, called Functional Mock-up Unit (FMU) and supports both model exchange and co-simulation of dynamic models. An FMU can either be self-integrating (i.e. can include its own numerical solver), or require an external simulator to perform its numerical integration. The former is the case of "FMI for co-simulation", the latter is called "FMI for model exchange". FMUs used for joint-simulation have to comply to the "FMI for model exchange" interface, version 1.0. This version of the interface allows handling "hybrid ODEs" [14] i.e. ordinary differential equations in state space form with events. Systems described by hybrid ODEs combine a piecewise continuous state (discontinuities occur at event instants) with a time-discrete state that changes only in correspondence of event instants.

Several modeling and simulation tools like Modelica, JModelica, Dymola, Rhapsody and Simulink/StateFlow support exporting models to FMI 1.0. Such tools can be seamlessly integrated with this core tool-chain thanks to our choice to adopt that standard. Joint simulation capabilities are provided by DESYRE [3, 8–10], 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.

The implementation of a Master Algorithm (MA) is not a trivial task, since the algorithm has to guarantee:

1. the correctness of the composition according to the model(s) of computation (MoC) of both the host environment and the constituent FMUs,
2. the termination of the integration step and
3. the determinism of the composition.

Challenges related to the implementation of Master Algorithms for model composition, have been extensively addressed in the literature. In [15] 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 [16].

3.2. DESYRE Master Algorithm

Within the context of the DANSE project a specific FMI Master Algorithm (MA) has been developed in DESYRE to address the unique needs of Systems of Systems simulation and SMC. The focus is on simulation efficiency, needed because of the SoS model complexity and large observation (i.e. simulation) time span (up to several years) as well as the large number of runs (tens or hundreds of thousands) required by the 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 the fixed point computation at each step. The solver used for the continuous state of continuous-time FMUs is the simple forward Euler algorithm rather than higher order Runge-Kutta (RK) methods. While this method can be numerically unstable and its global error is linear with the integration step, it does not require the computation of state (and inputs) at intermediate steps, which makes the integration method compositional and does not require any "roll-back" mechanism due to the validity of the selected step size. In the current implementation the step size $maxIntStep$ is specified by the user for the entire model but can be easily extended to support a different step size for each FMU. Besides affecting the rounding error and the stability of the algorithm the step size determines also the capability of detecting state events (i.e. zero-crossing) occurring during an integration step as well as their location on the time axis. In fact, state events are checked for only at the end of each integration cycle and, if detected, are given a time stamp equal to current simulation time with an error that in the worst case equals the discretization step $maxIntStep$. All these limitations were proven acceptable in all the use cases exercised within the DANSE project.

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
```

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 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 [7].

3.3. Joint-Simulation Traces

The execution of a SoS model results in a trace, that is exploited by our SMC analysis. The formal definition of trace is provided by Definition 1.

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 $\forall i \;\; t_i \in \mathbb{R}^+ \wedge t_i < t_{i+1}$.

In our setting, a trace contains an element for each time step of the simulation. At each step, the value of all

observed variables are recorded. The observed variables V are the exported variable to which the «traceable» stereotype has been applied.

4. Background on Statistical Model Checking

Analyzing Systems of Systems requires a careful choice of the verification technique to use. A first solution would be to use model checking. However, this formal approach often requires an input written in a dedicated language, which conflicts with industry acceptance. Even if a complete model were made in a suitable language, analysis would not be feasible because of the very large size of SoSs and its heterogeneous nature. Therefore, we rely on Statistical Model Checking (SMC), which is a trade off between testing and model checking. SMC works by simulating the system and verifying properties on the simulations. An algorithm from the statistic area exploits those results to estimate the probability for the system to satisfy a given requirement.

The quantitative results provided by SMC is richer than a Boolean one. Indeed, if the system does not satisfy the requirement, a Boolean tool returns "not satisfied" without any evaluation of the probability of a correct behavior.

In order to apply SMC, one has to assume that the behavior of the system is governed by a stochastic semantic, that is the choice of the next state in an execution depends on a probability distribution. This hypothesis shall not be seen as a drawback. Indeed, most of SoS do make stochastic assumptions on their external environment or on their hardware. In case no distribution is known, one relies on the uniform distribution which has the maximal entropy.

In the rest of this section, we first present the BLTL logic used to express properties on system's executions. Then, we present some statistical algorithms used by the SMC engine.

4.1. BLTL Linear Temporal Logic

In this paper, we focus on requirements of SoS that can be verified on bounded executions. This assumption is used to guarantee that the SMC algorithm will terminate. The bounded hypothesis shall not be viewed as a problem. Indeed, like it is the case in the testing world, it is sufficient to consider that the system has a finite live time.

We present here BLTL, a variant of LTL [17] where each temporal operator is bounded. Properties of SoS will be expressed in GCSL, that instantiates to BLTL (see Section 5). 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:

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

Here, AP is a set of atomic predicates defined below. 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 $\mathsf{F}_t\varphi = \texttt{true}\,\mathsf{U}_{\leq t}\,\varphi$ and $\mathsf{G}_t\varphi = \neg\mathsf{F}_t\neg\varphi$, 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 $p(\tau, i)$ (see below);
$\tau, i \models \varphi_1 \wedge \varphi_2$ if and only if $\tau, i \models \varphi_1$ and $\tau, i \models \varphi_2$;
$\tau, i \models \neg\,\varphi$ if and only if $\tau, i \not\models \varphi$;
$\tau, i \models \varphi_1 \mathsf{U}_t \varphi_2$ 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 $i \leq k < j$;
$\tau, i \models \mathsf{X}_{\leq t}\varphi$ 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 $t'_i = t_{i+k}$;

Typically, a monitor, such as in [18], is used to decide whether a given trace satisfies a given property.

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 depends on the past (i.e. on states preceding the current one). In particular, we are interested in measuring the amount of time during which a given atomic predicate has been true. The syntax for our predicates is as follows:

$$\begin{aligned} \text{AP} &::= \texttt{true} \mid \texttt{false} \mid AP \circ AP \mid Nexp \bowtie Nexp \\ \text{Nexp} &::= \#\texttt{Time} \mid \texttt{Id} \mid \texttt{Constant} \mid dur(AP) \mid occ(AP, a, b) \\ &\quad \mid Nexp \pm Nexp \end{aligned}$$

Here, $\circ$ contains the usual boolean connectors , $\pm$ the usual arithmetic operators and $\bowtie$ 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 have their usual semantics;

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

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

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, $\mathsf{G}_{\leq t}(dur(UP) > 0.9 \cdot \#Time)$ 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.

In order to support the new $\#Time$ and dur constructs, we extend the atomic predicates available in BLTL. Concerning the $\#Time$ variable, we first recall that each state in a trace contains a timestamp, according to Definition 1. Indeed, this value is necessary for verifying patterns involving time bounds. At a given state, each predicate is evaluated by replacing $\#Time$ by the timestamp of that state. To evaluate a predicate involving $dur(\phi)$, we accumulate the amount t_ϕ of time during which the predicate ϕ evaluated to true. More precisely, in the initial state, t_ϕ is set to 0. Then, each time the simulation enters a new state i, we check whether the predicate ϕ was true at state $i-1$. In that case, we increase t_ϕ by the amount of time elapsed between the state $i-1$ and the state i. We then evaluate the predicate by taking the value of t_ϕ for $dur(\phi)$ at the current state.

4.2. Statistical Model Checking

Given a stochastic system $\mathcal{M}$ and a property φ, SMC is a simulation-based analysis technique [4, 19] 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 [19, 20] 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 hypotheses: $H : p \geq \vartheta$ 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 [21].

Quantitative Approach In [22, 23] 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* [24], 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 an idea on how well the model behaves. On the contrary, the qualitative approach determines whether the probability is above a given threshold, with a high confidence and in a minimal number of simulations.

5. Timed OCL Constraints for SoS Requirements

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. The SoS setting introduces further challenges for the need of specifying properties of the state of complex systems and architectures. For scalability reasons, monolithic verification approaches should be avoided in favor of compositional ones, such as those based on the *theory of contracts* [25].

In this section we present the requirements language, called GCSL (Goals and Contracts Requirement Language). The full GCSL language specification, as well as the details of translation of the GCSL *patterns* into BLTL, can be retrieved in [26]. 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.

5.1. A Survey of GCSL

A GCSL contract is a pair of Assume/Guarantee assertions:

$$GCSL\_Contract := \langle Assume, Guarantee \rangle$$

denoting requirements on SoS and CSs inputs and outputs, respectively. Contracts allow us to decompose requirements and perform local or global verification on need [25].

Assertions are built upon GCSL natural-language patterns: we summarize in Figure 1 the principal patterns used in the DANSE project. GCSL patterns are inspired by and extend the Contract Specification Language (CSL) patterns [27], 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) [28], 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. OCL allows us also to define algebraic constraints over Constituent Systems attributes. In summary, GCSL Patterns are able to express real-time constraints over complex portions of the entire SoS state.

To show the expressiveness of GCSL, let us consider the following simple example (based on Pattern 9 from Fig. 1):

```
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 and identity of the mentioned CSs. It can be used as a requirement for any SoS having an architecture that integrates the mentioned CSs types. Moreover, this requirement is still valid even if the SoS changes its architecture, either for intrinsic reconfiguration needs or for external re-design or re-configuration activity. Therefore, GSCL requirements are valid for the entire lifetime of the system and need not be reformulated at any architectural change.

The idea of mixing OCL and temporal logic originates from the need of specifying static and dynamic properties of object-based systems. In [29, 30] OCL has been extended with CTL and (finite) LTL, respectively, without support for real-time properties. The work in [31] 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. On the contrary, BLTL does not include unbounded operators because SMC requires to decide a property on a given finite trace.

As mentioned previously, the introduction and use of patterns is motivated by infeasibility of direct use of BLTL, which is a very low level language. Indeed, commonly used properties may require the nesting of several BLTL temporal operators, with the effect of increasing the risk of errors in requirements formulation and reducing their readability. Patterns provide a convenient way to represent frequently-occurring and well-identified property templates, while avoiding errors due to the complexity of the underlying logic. The methodology developed during the DANSE project prescribes to have a library of patterns [26] that captures the most relevant temporal constraints for the considered domain. So a pattern-library shall be considered as an important asset that is tailored to specific design needs.

Description of the patterns. In the rest of this section we are going to discuss a number of these pre-defined patterns. Clearly, as already discussed, the choice of the set of patterns is inevitably subjective and depends on the field of application as well as on the test case at hand. The expressiveness of BLTL, jointly with OCL, makes the patterns library easily extensible to cover future, domain-specific needs.

Figure 1 shows the most relevant GCSL patterns used in the DANSE project [26]. For every pattern, there is the corresponding BLTL translation that illustrates how temporally-bounded BLTL operators are used and how it can be complex to reason directly in terms of those operators. In all the following patterns, Ψ denotes an argument where an OCL property can be used to describe a state of the SoS. The keywords 'occurs' and 'holds' are used to indicate, respectively, that there is at least one real-time instant where the given property shall be satisfied or that in all real-time instants the given property shall be satisfied. Constants a, b and c specify bounds of time intervals and are real-time values. The constant n is an integer value, while e is a value from 0 to 100, denoting a percentage.

Pattern 1 can be used to specify a causal relation between a condition Ψ_1 (e.g. system activated) and a consequential condition Ψ_2 that is activated by and stable since the originating condition (e.g. active indication light is on).

Pattern 2 can be used to specify a typical invariant property, which express a bound on the set of states the system shall traverse under any condition (e.g. the value of control signal is always within a specified interval). In particular, the OCL property Ψ should be true at every real-time instant within simulation end time k.

Pattern 3 can be used to specify a causal relation between Ψ_1 and Ψ_2 that happens only within an absolute time-frame ($[a, b]$). It is interesting to see how the next operator $X_{\leq a}$ is used to push to the very beginning of the time interval, where it is possible to specify the limited time-span of the condition Ψ_2 over the remaining $b - a$ time. In particular, a and b are offsets from the initial simulation time. This interpretation of bounds is not always valid, since

for some patterns (as we shall see in the following) the bounds of time intervals are offsets referred to a previously occurred condition Ψ.

Pattern 4 can be used to specify that a condition Ψ_1 that is held continuously in the absolute time-frame ($[a, b]$) causes the condition Ψ_2 to occur at the end of the time-frame. For example, to specify the need for a minimum duration of a certain condition before being able to activate the consequent. Clearly, a longer duration will also cause activation.

Pattern 5 can be used to specify that if a condition Ψ_1 shall be held continuously in the absolute time-frame ($[a, b]$) then the two other conditions Ψ_2 and Ψ_3 shall happen contiguously in the same time-frame. This pattern supports the specification of the decomposition of an activity (expressed by Ψ_1) into sub-activities (expressed by Ψ_2 and Ψ_3). Further decomposition can be specified by conjunction with other properties using the same pattern over the contained time-frame ($[a, c]$) or ($[c, b]$).

Pattern 6 can be used to constrain the number of times a condition Ψ can be active within an absolute time-frame. 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 simulation trace.

Pattern 7 can be used to specify a condition Ψ_2 that is triggered by the number of times a condition Ψ is active within an absolute time-frame.

Patterns 8 **and** 9 are not referred to an absolute time-frame. They are used to specify a *liveness* property which is triggered by an initial condition Ψ_1 occurring at time t and discharged by a following condition Ψ_2 that occurs/holds within the interval $[t + a, t + b]$ that is *relative* to the time t of occurrence of Ψ_1. They are useful to express performance specifications such as, for example, the switching of an engine power-on button at time t triggers the burning chamber temperature sensors to detect a temperature increase within $t + 10sec$ (minimal duration) and $t + 15sec$ (maximal duration). Note that its translation into BLTL underpins a number of important assumptions. 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** 10**,** 11**, and** 12 are typically used to express reliability constraints, such as the availability of a system. The BLTL translation of these patterns relies on the $\#Time$ and *dur* operators that we shall discuss in more details in the following. The first pattern requires that in every real-time instant t within the absolute time-frame ($[a, b]$) the amount of time a condition Ψ has been true is at least e% of t. 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. A typical example use of this property is the specification of the system up-time. This can be done by specifying a condition Ψ that captures the states of the system where there is either no fault or a number of faults against which the system has adequate countermeasures. The second pattern is slightly more permissive, allowing to violate the constraint at intermediate real-time instants, but requiring that, at the end of the absolute time-frame ($[a, b]$), the constraint is satisfied. Note that, in both cases, the lower bound a of the absolute time-frame ($[a, b]$) is used to ensure that there are no constraints in the initial system lifetime, where early failures may be expected and not critical. Finally, the last pattern is used when the constraint is applied from the beginning of simulation.

OCL constraints on architecture. We now turn our attention to the OCL part of GCSL and to how it helps us improve the expressiveness of the patterns and the adequacy to the SoS domain. 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*), which is essentially their path from the top of the SoS down through the containment hierarchy collecting Constituent Systems names (*csName*) and terminating with the attribute name (*attribute*), such as SoS.Sensor03.isOn.

The peculiarity of OCL propositions is the ability to predicate properties of sets of *yet unknown* (at requirements-definition time) Constituent Systems. These sets are left undetermined because the requirements may apply to several variant architectures of the same SoS and the SoS architecture may evolve during the SoS life-cycle. Indeed, in [1] 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

ID	Pattern	(below, k is the simulation time and $a < c < b \leq k$)
1	[Ψ_1] occurs implies that [Ψ_2] holds forever	$G_{\leq -1}(\Psi_1 \rightarrow G_{\leq -1}(\Psi_2))$
2	[Ψ] holds	$G_{\leq -1}(\Psi)$
3	[Ψ_1] implies [Ψ_2] holds during [a,b]	$X_{\leq a}G_{\leq b-a}(\Psi_1 \rightarrow \Psi_2)$
4	[Ψ_1] holds during [a,b] raises [Ψ_2]	$(X_{\leq a}G_{\leq b-a}(\Psi_1)) \rightarrow X_{\leq b}(\Psi_2)$
5	[Ψ_1] holds during [a,b] implies [Ψ_2] holds during [a,c] then [Ψ_3] holds during [c,b]	$(X_{\leq a}G_{\leq b-a}(\Psi_1)) \rightarrow (X_{\leq a}G_{\leq c-a}(\Psi_2) \wedge X_{\leq c}G_{\leq b-c}(\Psi_3))$
6	[Ψ_1] occurs at least (at most) n times during [a,b]	$occ(\Psi_1, a, b) \geq (\leq) n$
7	[Ψ_1] occurs at least (at most) n times during [a,b] raises [Ψ_2]	$occ(\Psi_1, a, b) \geq (\leq) n \rightarrow X_{\leq b}F_{\leq -1}(\Psi_2)$
8	whenever [Ψ_1] occurs [Ψ_2] holds during following [a,b]	$G_{\leq -1}(\Psi_1 \rightarrow X_{\leq a}G_{\leq b-a}(\Psi_2))$
9	whenever [Ψ_1] occurs [Ψ_2] occurs within [a,b]	$G_{\leq -1}(\Psi_1 \rightarrow X_{\leq a}F_{\leq b-a}\Psi_2)$
10	always during [a,b], [Ψ] has been true at least [e] % of time	$G_{\leq b}(\#Time < a \vee dur(\Psi) \geq (\frac{e}{100} * \#Time))$
11	at the end of [a,b] [Ψ] has been true at least [e] % of time	$X_{\leq a}F_{\leq b}(dur(\Psi) \geq \frac{e}{100} * (b - a))$
12	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 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* ∘ *OCL-prop* | *OCL-expr* ⋈ *OCL-expr* | *OCL-coll* → **forAll(** *var* | *OCL-prop* [*var*] **)** | *OCL-coll* → **exists(** *var* | *OCL-prop* [*var*] **)** | *OCL-coll* → **empty()** | *OCL-coll* → **notempty()**
OCL-expr ::= *FQN* | *OCL-coll* → **sum()** | *OCL-coll* → **size()** | ...
OCL-coll ::= *attribute* | *csName* | **its(***type***)** | **connected(***type***)** | *OCL-coll* . *OCL-coll*

Figure 2. Simplified OCL fragment for GCSL Atomic Properties, with ∘ ∈ {∧, ∨, ⟶, ⟷} and ⋈ ∈ {>, ≥, =, <, ≤}.

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.

It is interesting to discuss how collections (*OCL-coll*) are specified, as it is a peculiarity of GCSL extending the standard OCL language. Standard OCL allows concatenating 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 by filtering on CS-*type* and (2) an operator (**connected**) that allows to navigate the *neighborhood* of a CS, again in terms of *types*. For example, the expression SoS.its(CriticalComponent) denotes the set of CSs of type CriticalComponent and are contained in the system named SoS. The expression SoS.cc01.connected(CoolingFan) denotes the set of CSs of type CoolingFan that are connected to the critical component cc01, which is itself contained in the system named SoS. Quantifiers occur also in expressions (*OCL-expr*) and allow aggregating the values of (equally-typed) attributes. E.g. the simple expression (SoS.its(Sensor).temp → sum( ))/(SoS.its(Sensor) → 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.

Architecture-dependent translation of OCL. Before concluding this section, we should discuss how the translation of GCSL into BLTL is handled for OCL. Since BLTL does not interpret the OCL syntax (except for *FQN*), the translation from GCSL to BLTL requires the preliminary elimination of all the OCL operators (forAll, exists, empty, notempty, sum, size, its, connected). This elimination step can be performed only once the SoS architecture is determined and it is valid for a fixed architecture state.

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 applying the following algorithm:

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 the algorithm is trivially proved by considering that (i) the number of nested operators is always finite and strictly decreasing at every rewriting step, and (ii) that all the OCL collections contain finitely many elements that are defined by the current state of the architecture. Note that this translation needs be performed every time there is architecture state change. Therefore, the tool-chain dynamically evaluates the formula as the architecture changes. One needs to consider that translation may potentially introduce a combinatorial blow-up of the formula, which suggests the need of adopting an on-the fly evaluation algorithm.

To illustrate the elimination of OCL operators on a concrete example, assume that the requirement considered previously is checked on the architecture in Figure 3. In that Figure, instances whose name start by "cc" are of type CriticalComponent, instances whose name start by "cf" are of type CoolingFan, and finally instances whose name start by "ts" are of type TempSensor. In this case, the collection SoS.its(CriticalComponent) would be resolved into the set {SoS.cc01,SoS.cc02}. Then, the new outermost collections would be SoS.cc01.its(TempSensor), SoS.cc01.connected(CoolingFan), SoS.cc02.its(TempSensor), and SoS.cc02.connected(CoolingFan). These are further resolved into the following sets: {SoS.cc01.ts01, SoS.cc01.ts02}, {SoS.cc01.cf01}, {SoS.cc02.ts03}, and {SoS.cc02.cf01,SoS.cc02.cf02}, respectively.

As a consequence of the translation against the considered architecture, the formula would be:

$$
\begin{aligned}
&\text{whenever} \\
&\quad [\ \text{SoS.cc01.ts01.temp} > \text{SoS.cc01.threshold} \vee \\
&\quad\ \ \text{SoS.cc01.ts02.temp} > \text{SoS.cc01.threshold}\ ] \\
&\text{occurs} \\
&\quad [\ \text{SoS.cf01.on}\ ] \\
&\text{occurs within}\ [\ 1\ \text{min}, 5\ \text{min}\ ] \\
\wedge\ & \\
&\text{whenever} \\
&\quad [\ \text{SoS.cc02.ts03.temp} > \text{SoS.cc02.threshold}\ ] \\
&\text{occurs} \\
&\quad [\ \text{SoS.cf01.on} \vee \text{SoS.cf02.on}\ ] \\
&\text{occurs within}\ [\ 1\ \text{min}, 5\ \text{min}\ ]
\end{aligned}
$$

This elimination-based approach is justified by the fact that SoS development with architecture reconfiguration has, typically, a larger time-scale than the operational level of the CSs. So architecture snapshots can be individually analysed, while architectures alternatives are generated from generic architectural pattern.

After having illustrated the main features of GCSL and how they support the SoS specification and dynamic evolution challenges we discuss, in the following section, how to obtain a unique behavioural model of the SoS by integration of single and heterogeneous CSs behavioural models.

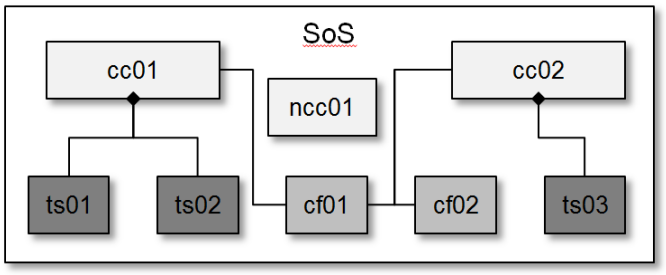

Figure 3. Architecture with connection (e.g. cc01 – cf01) and containment (e.g. cc01 – ts01) relations.

6. Performing Statistical Model Checking

This Section describes the work-flow and its supporting tool-chain (see Figure 4) we have set-up to accomplish statistical model checking analysis on Systems of Systems models.

Our tool-chain has been adopted by the industry partners of the DANSE Project: CARMEQ, IBM, EADS, and Thalès. The core of our SMC tool-chain is composed of three main tools: IBM Rhapsody is the tool implementing the UPDM language to model SoS architectures, DESYRE [8–10] is the tool providing the joint simulation engine for SMC and PLASMA [5] is the tool providing the SMC analysis engine. Other modelling tools like Modelica and Simulink/Stateflow surround this core tool-chain to provide behavioural models in the form of FMUs.

6.1. SMC Analysis Workflow

Figure 4 shows our SMC analysis work-flow. The work-flow starts with two parallel branches. The topmost block of the left branch represents the parallel development of Constituent Systems' models, each with its best-fitting modelling tool as described in Section 2.1. The topmost block of the right branch represents

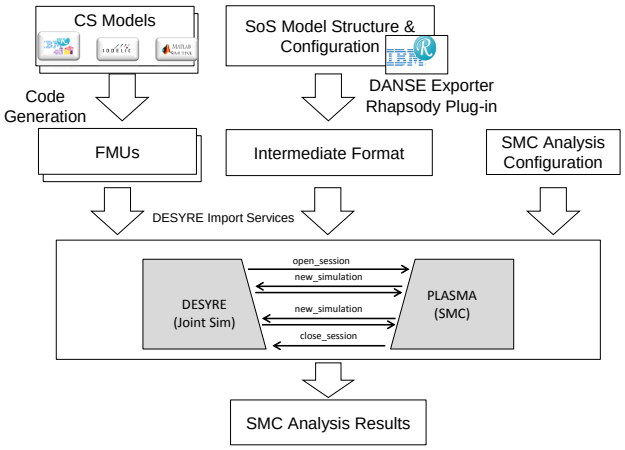

Figure 4. SMC Analysis Workflow.

the development of the SoS architecture in UPDM as described in Section 2.2. Once each CS model gets ready for joint-simulation and SMC analysis, it is exported from its modelling tool into the FMI-compliant format. This step is represented by blocks named "FMUs" in the left branch of Figure 4. On the right branch, when the UPDM blocks in the SoS architecture are 1) in synch with their corresponding CSs models (e.g. same interfaces) and 2) correctly interconnected, our Rhapsody exporter plug-in can be invoked to translate the UPDM model into a form that DESYRE can use to run joint-simulation. At this point of the work-flow all the information needed to run the statistical model checking analysis is available and to launch the SMC analysis the user has to provide a proper SMC analysis configuration.

6.2. The SMC Workflow from the User Perspective

IBM Rhapsody. IBM Rational Rhapsody [32] is a model-based system engineering environment implementing industry-standard languages such as UML, SysML and UPDM. Referring to Figure 4, IBM Rhapsody represents the starting point of the SMC analysis work-flow. The

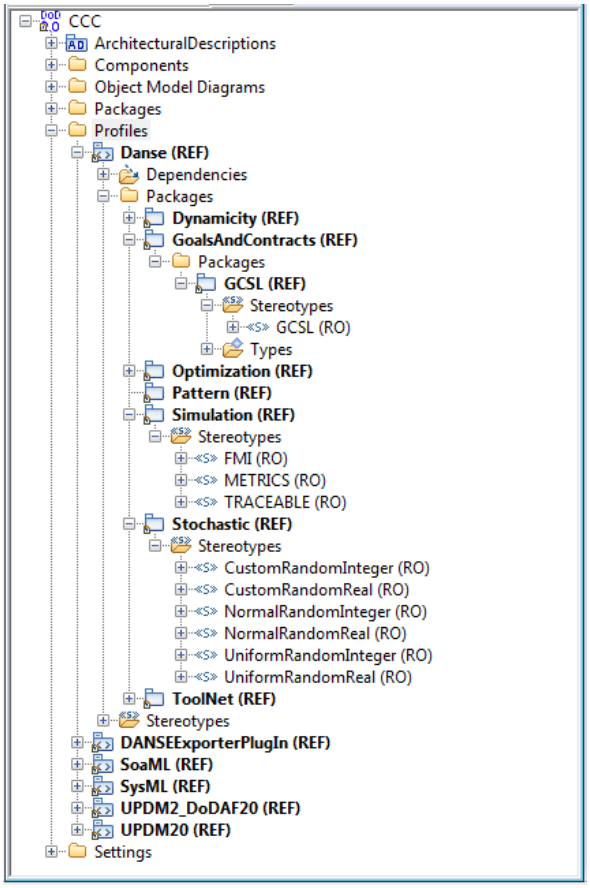

Figure 5. DANSE Profiles and Stereotypes implemented in IBM Rhapsody.

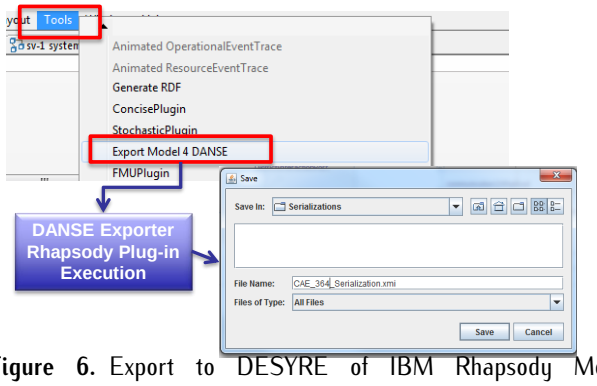

Figure 6. Export to DESYRE of IBM Rhapsody Model Representing the SoS Architecture.

SoS architecture is specified within this tool by using the UPDM industry-standard language, conveniently extended to meet SMC analysis specification needs. Figure 5 shows how the UPDM extensions described in Section 2 have been implemented in IBM Rhapsody. At this point, the user will rely on the profiles to model the architecture of the SoS. This includes attaching GCSL contracts to the Constituent Systems and/or to the whole SoS.

During DANSE Project IBM has extended the Rhapsody tool to generate FMI 1.0 compliant FMUs from most of UPDM behavioural diagrams, thus making it join the pool of those candidate modelling and simulation tools able to export CS models for our SMC analysis purposes. In that context, the user can rely on our profiles for describing stochastic behaviors.

Exporting information for SMC. IBM Rational Rhapsody provides a Java API set for integration with external tools. We developed a Java Exporter Plug-in to translate information from the UPDM SoS architecture model, along with all the information related to SMC analysis i.e. GCSL requirements, FMU names and URIs, and traceability settings to a format intelligible by DESYRE, the joint simulation engine. Our Exporter Plug-in integrates into Rhapsody as an UPDM extension profile («*DANSEExporterPlugIn*» profile in Figure 5). Once invoked from the IBM Rhapsody "Tools" menu our Exporter recursively parses both the UPDM Block Definition Diagram and the Internal Block Diagram to translate all the information needed to perform SMC analysis into the DESYRE internal representation format (XMI). Figure 6 shows how the user exports the Rhapsody UPDM SoS model in the intermediate format compliant with the format expected by DESYRE.

Simulation and SMC analysis. Once the architecture has been exported and all the FMU are compiled, one can consider simulating the SoS model and thus analyzing it through SMC. Both simulation and analysis can be launched from a single GUI. First, the user has to launch the DESYRE GUI and load a SoS model as well as the

corresponding FMUs. At this point the user can run a simulation and graphically plot the evolution of some selected variables, as shown in 11.

To run a DANSE statistical model checking analysis the user creates a statistical model checking configuration in DESYRE. Figure 7 shows DESYRE's SMC Analysis Configuration Editor. From the editor the user selects one SMC *Analysis Method* out of the ones provided by PLASMA. Figure 7 shows that the Chernoff method has been selected. According to the selected analysis method the editor prompts the associated *Analysis Configuration* frame to configure the parameters of the chosen algorithm. The user also specifies the simulation time of each SMC simulation run. GCSL contracts specified in the IBM Rhapsody UPDM model are available in the DESYRE intermediate representation and can be imported into the SMC configuration editor. In this case the user might not be interested in validating all the imported GCSL contracts through the SMC analysis she/he is configuring. To address this need we decided to make the imported GCSL contracts selectable by means of check-boxes. Such check-boxes are visible in Figure 8. The user can also directly write GCSL contracts from the GUI, or edit the ones imported from the UPDM description. To this end, the user can specify the new GCSL contract in its three components: *Assumption*, *Promise* and *Owner*, the latter being the model element to which the GCSL contract predicates on. When the configuration is completed, the user can launch the SMC analysis campaign. After the SMC analysis has terminated the user selects the *SMC Analysis Results* view, that appears as shown in Figure 8. SMC analysis results are displayed per GCSL requirement as shown in the bottom-right corner of Figure 8. On the figure, only one formula was evaluated, and its probability to hold was evaluated to approximately 87%.

PLASMA. PLASMA is a tool for performing SMC analysis. Contrary to existing tools, PLASMA offers a modular architecture which allows plugging new simulators and new input languages on demand. This architecture has been exploited to verify systems and requirements from various languages/specific domains such as systems biology [33], a train station [34], or an autonomous robot [35].

The core of PLASMA is thus a set of SMC algorithms, which includes those presented in Subsection 4.2 as well as more complex ones [5]. This core is completed by two types of plug-ins, that are controlled by the SMC algorithm. First, the simulator plug-ins which implements an interface between PLASMA and a dedicated simulator to produce traces from a dedicated input language on demand. Second, there is a checker plug-ins that verify whether a finite trace satisfy a property.

For the purpose of this paper we extended the facilities of PLASMA as follows. First, we built a new plug-in between PLASMA and DESYRE in order to produce traces from FMI-FMU model. Second, we used the BLTL checker plug-in that we enrich with the two new primitives *dur* and $\#Time$ in order to monitor those traces and report the answer to the SMC algorithm. As a last implementation effort, we also implemented a compiler from GCSL to BLTL.

Interaction between PLASMA and DESYRE. Once the user launches the analysis, DESYRE translates the GCSL properties to BLTL, as described in Section 5. It then invokes PLASMA and transmits the list of properties to check, as well as the algorithm and parameters to use. PLASMA then requests DESYRE to start the first execution. The execution is controlled step-by-step by PLASMA. At each step DESYRE provides a new snapshot of the system state, by giving the current value of every observed port in the architecture. A detailed description of the interaction between PLASMA and DESYRE is provided in [3].

7. A Case Study

This section illustrates the application of our technology to a concept alignment example that was defined for the DANSE Project. This case study has the particularity to embed all the difficulties of the case studies proposed by EADS, THALES, and CARMEQ that, for confidentiality reasons, cannot be described in this paper.

7.1. Modeling

We modeled an emergency response SoS for a city fire scenario in UPDM. The city is partitioned into 10 districts, and we focus on a few fire-fighting constituent systems (CS). We consider the following CSs: the Fire Head Quarter (FireHQ), the Fire Stations, the Fire Fighting Cars and the Fire Men. The behavior of the CSs has been modeled in several FMI-compliant authoring tools. For example, the FireMan has been modeled in OpenModelica (as shown in Figure 9) which is an open-source multi-domain modeling tool based on the Modelica language. Other CSs have been modeled using Rhapsody state-charts. The CSs rely on the new UPDM profile to include probabilistic behavior. For instance, each District models occurrences of fires by randomly choosing the time before the next fire according to an exponential distribution. The time before a fire is reported to the head quarter is also randomly chosen. We remind the reader that the objective of this work is not to learn the probability distribution itself (this has to be done via observations), but rather to show that it is conceptually possible to incorporate such information within the model.

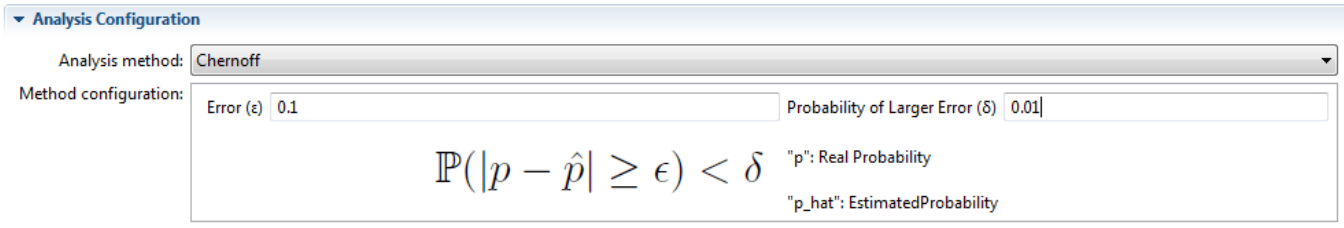

Figure 7. DESYRE's SMC Analysis Configuration Editor.

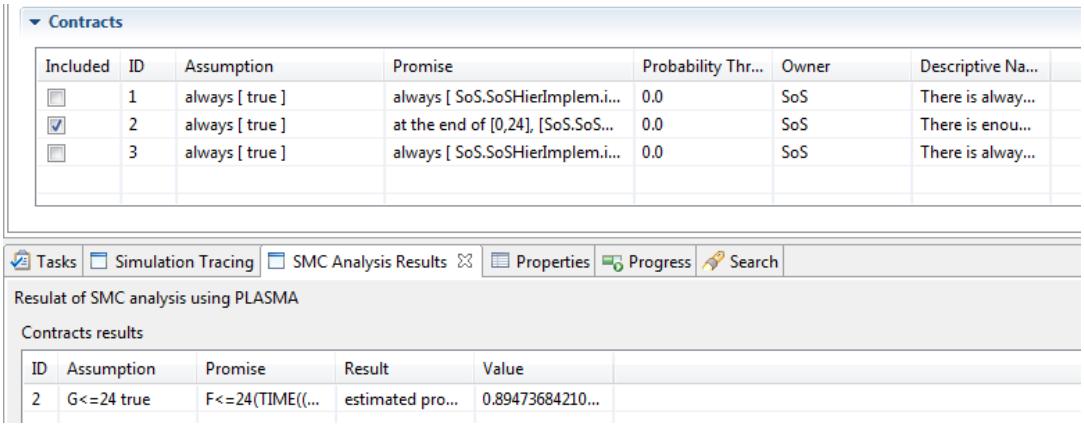

Figure 8. Adding or importing GCSL contracts and interpreting SMC Analysis Results.

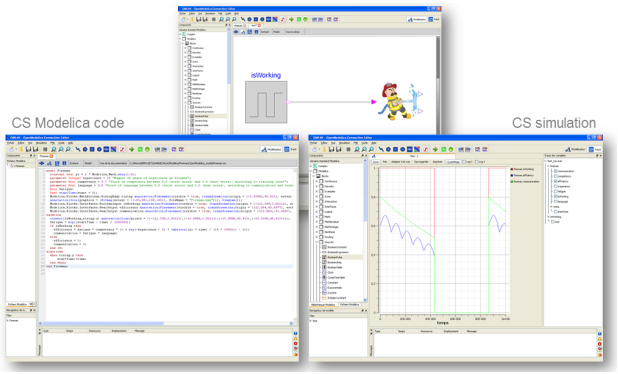

Figure 9. CS behaviour modelled in other tools (e.g. OpenModelica)

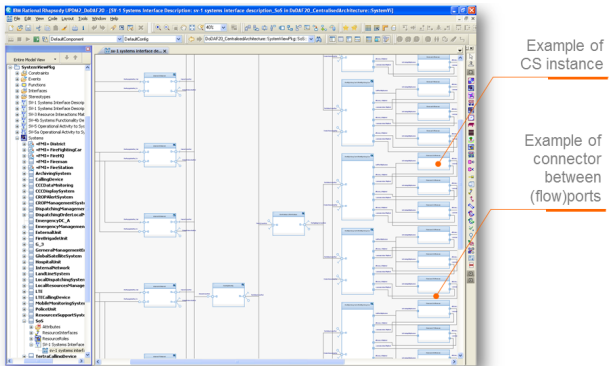

Figure 10. SoS architecture in Rhapsody

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 10. The SoS architecture is exported to DESYRE using a DANSE-specific exporter plug-in. Each CS behavioral 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 to plot some selected variables – see Figure 11 for an illustration.

The simulation is parameterized by its duration, expressed in the time of the model. For our experiments, we choose to simulate 10 000s of execution. Since our model of computation is event based, the computation time needed for running this simulation depends on the number of events occurring during the simulation. Whenever there are few events, such as in the top of Figure 11, the simulation takes a few seconds. In that case, there is no event between two pikes, corresponding to two fires that are very quickly extinguished. Simulations involving more events, such as the one at the bottom of the Figure, require a few dozen of seconds to complete. In that case, fires are not extinguished and the time between two events is kept small to describe the evolution of the fire.

7.2. Expressing Goals of the SoS

Our main objective is to check that the fire area remains small enough. In order to define "small

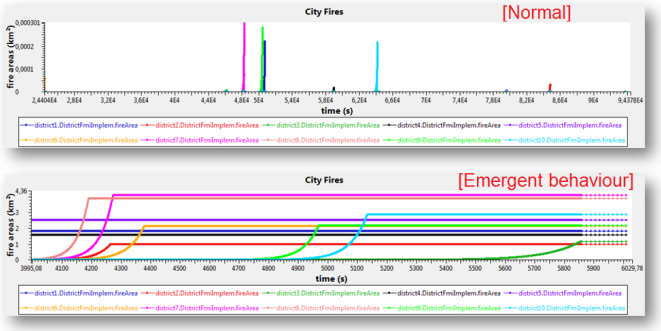

Figure 11. Simulation results in DESYRE

enough" independently of the number of components, we require that the fire is less than a given percentage of the total area.

In our model, each district has two variables of interest, its area and the fire area.

Our first formulation states that the fire area is *always* 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.itsDistricts.fireArea->sum()`. We define Pattern 1 as follows:

always [SoS.itsDistricts.fireArea → sum()
< (X/100)*SoS.itsDistricts.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.itsDistricts.fireArea → sum()
> (X/100)*SoS.itsDistricts.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. As we want the probability that the system behaves correctly, we have to compute the probability of the complementary event. This is done by subtracting the probability that Pattern 2 holds from 1.

7.3. Unwanted Emergent Behaviors Detection and Evaluation

One of the challenges in SoS design is the detection and analysis of unwanted emergent behaviour. In our case, simulation allowed us to detect an (undesired) emergent behaviour which is depicted in the lower part of Figure 11. 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.itsDistricts → forAll(district |
district.fireArea < district.area)]

7.4. Analysis and Discussions

In this section, we use SMC to compute an estimate of the probability for Pattern 1, 2 and 3 to hold. We use the *PESTIM* method which 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 requires 265 simulations traces. These traces are obtained by running stochastic simulations of the model. The length of a simulation is set to 10000s. We present the analysis results and time for Pattern 1 in Table 1. The analysis result is an estimation of the probability that Pattern 1 holds, based on the 256 traces.

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

As expected, the probability that the fire remains smaller than X% of the 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. Indeed, the difference between the two values is less than the error. On the contrary, 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 kilometers. 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 subtract from 1 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). Again, since for each value of X we ran a different set of simulations, it is 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

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

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 unwanted emergent behavior depicted in Figure 11, that is the probability that Pattern 3 holds. The returned result is 0.9622, which means that the 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 behavior 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 behavior 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.

Our Patterns are independent on the actual number of components. Indeed, adding constituent systems such as districts or cars, even if they have a new behavior, do not require specifying new patterns. The analysis is still possible on the modified SoS model.

In the framework 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 [36]. 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.

8. Conclusion, Future work and related work

This paper proposes a full tool-chain for the rigorous design of Systems of Systems via formal reasoning and Statistical Model Checking.

Recent work promotes simulation techniques as the principal way to perform SoS analysis. In [37] 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 modeling and simulation (M&S) of SoS by allowing the user to specify constituent systems' behavior in terms of a so-called Constrained Natural Language. The tool is implemented in Eclipse and employs Xtext, Eclipse Modeling Framework and the Graphical Modeling Project.

Recent work in [38] provides an overview of the underlying theory, methods, and solutions in M&S of systems of systems, to better understand how modeling 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 behaviors. 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 [39] 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 behavior while keeping the analysis cost low is based on Statistical Model Checking (SMC) [4]. SMC is a simulation-based formal analysis providing an estimate of the likelihood of requirement satisfaction and a tunable level of confidence in the accuracy of analysis results.

Some frameworks, such as BIP [40] or Reo [41] allow the user to describe architecture of systems using alternative composition operators. Both these frameworks theoretically permit composition of heterogeneous system, by including existing code into the components. However, they provide no standard such as FMI/FMU allowing to compile a particular component independently of the architecture, which make them unsuitable for an industrial environment. They both include a stochastic extension [42, 43], which is not part of the core framework.

Contracts for reasoning about heterogeneous systems are presented in [44]. However, these contracts are

considered at a very abstract level, and thus are not as useful as FMI/FMU from an industrial point of view.

Future Work Our objective is to improve our solution by exploiting rare-event techniques [45] that would allow us to detect rare emergent behaviors with a minimal amount of simulations. Our second future work is to automatize the relationship between the outcome of SMC and the reconfiguration process, in order to automatically find an architecture satisfying sufficiently the GCSL contracts.

References

[1] Maier, M. (1998) Architecting principles for systems-of-systems. *Systems Engineering* **1**(4): 267–284. doi:10.1002/(SICI)1520-6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D, URL http://dx.doi.org/10.1002/(SICI)1520-6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D.

[2] et al., L.M. (2013) *D4.3 - DANSE Methodology V2*. Tech. rep., DANSE deliverable.

[3] Mignogna, A., Mangeruca, L., Boyer, B., Legay, A. and Arnold, A. (2013) Sos contract verification using statistical model checking. In Larsen, K., Legay, A. and Nyman, U. [eds.] *Proc. 1st Workshop on Advances in Systems of Systems (AiSoS) 2013, EPTCS* **133**: 67–83. doi:10.4204/EPTCS.133.7, URL http://dx.doi.org/10.4204/EPTCS.133.7.

[4] Younes, H. and Simmons, R. (2006) Statistical probabilistic model checking with a focus on time-bounded properties. *Inf. Comput.* **204**(9): 1368–1409. doi:10.1016/j.ic.2006.05.002, URL http://dx.doi.org/10.1016/j.ic.2006.05.002.

[5] Boyer, B., Corre, K., Legay, A. and Sedwards, S. (2013) Plasma lab: A flexible, distributable statistical model checking library. In *QEST'13*: 160–164.

[6] Arnold, A., Boyer, B. and Legay, A. (2013) Contracts and behavioral patterns for sos: The EU IP DANSE approach. In Larsen, K., Legay, A. and Nyman, U. [eds.] *Proc. 1st Workshop on Advances in Systems of Systems (AiSoS) 2013, EPTCS* **133**: 47–66. doi:10.4204/EPTCS.133.6, URL http://dx.doi.org/10.4204/EPTCS.133.6.

[7] Vv.Aa. (2012) *FMI Standard Specification*. Modelica association, https://www.fmi-standard.org/.

[8] Mignogna, A., Ferrante, O., Carloni, M. and Ferrari, A. (2011) A fully configurable RTOS model for large scale distributed embedded systems simulations based on systemC. In *ASM 2011 - Applied Simulation and Modelling*.

[9] Ferrari, A., Carloni, M., Mignogna, A., Menichelli, F., Ginsberg, D., Scholte, E. and Nguyen, D. (2012) Scalable virtual prototyping of distributed embedded control in a modern elevator system. In *7th IEEE International Symposium on Industrial Embedded Systems, SIES 2012, Karlsruhe, Germany, June 20-22, 2012* (IEEE): 267–270. doi:10.1109/SIES.2012.6356593, URL http://dx.doi.org/10.1109/SIES.2012.6356593.

[10] Ferrari, A., Carloni, M., Mignogna, A., Menichelli, F., Ginsberg, D., Scholte, E. and Nguyen, D. (2012) Scalable virtual prototyping of distributed embedded control in a modern elevator system. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*: 267–270. doi:10.1109/SIES.2012.6356593.

[11] Vv.Aa. (2012) *UPDM 2.0 Formal Specification*. OMG, http://www.omg.org/spec/UPDM/2.0/.

[12] Aa., V. (2013) *D3.3.2 - Concept Alignment Example description*. Tech. rep., DANSE deliverable.

[13] Vv.Aa. (2015) *SysML Open Source Specification Project*. OMG, http://www.sysml.org/sysml-specifications/.

[14] Blochwitz, T., Otter, M., Arnold, M., Bausch, C., Clauss, C., Elmqvist, H., Junghanns, A. *et al.* (2011) The functional mockup interface for tool independent exchange of simulation models. In *In Proceedings of the 8th International Modelica Conference*.

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

[16] Broman, D., Brooks, C., Greenberg, L., Lee, E., Masin, M., Tripakis, S. and Wetter, M. (2013) Determinate composition of fmus for co-simulation. In *Proc. of the 11th ACM Int. Conference on Embedded Software*, EMSOFT '13 (IEEE Press): 2:1–2:12. URL http://dl.acm.org/citation.cfm?id=2555754.2555756.

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

[18] Havelund, K. and Rosu, G. (2002) Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*: 342–356. doi:10.1007/3-540-46002-0_24, URL http://dx.doi.org/10.1007/3-540-46002-0_24.

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

[20] Younes, H. (2005) *Verification and Planning for Stochastic Processes with Asynchronous Events*. Ph.D. thesis, Carnegie Mellon.

[21] Jha, S., Clarke, E., Langmead, C., Legay, A., Platzer, A. and Zuliani, P. (2009) A bayesian approach to model checking biological systems. In Degano, P. and Gorrieri, R. [eds.] *Computational Methods in Systems Biology* (Springer Berlin Heidelberg), *Lecture Notes in Computer Science* **5688**, 218–234. doi:10.1007/978-3-642-03845-7_15, URL http://dx.doi.org/10.1007/978-3-642-03845-7_15.

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

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

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

[25] Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A. *et al.* (2012) *Contracts for System Design*. Research Report RR-8147. URL https://hal.inria.fr/hal-00757488.

[26] Aa., V. (2015) *D6.3.3 - GCSL Syntax, Semantics and Metamodel*. Public deliverable, DANSE deliverable.

[27] (2008) *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.

[28] Vv.Aa. (2014) *OCL Language Specification*. OMG, http://www.omg.org/spec/OCL/.

[29] Distefano, D., Katoen, J. and Rensink, A. (2000) On a temporal logic for object-based systems. In Smith, S. and Talcott, C. [eds.] *4th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS) 2000* (Kluwer), *IFIP Conference Proc.* **177**: 305–325. doi:10.1007/978-0-387-35520-7_16, URL http://dx.doi.org/10.1007/978-0-387-35520-7_16.

[30] Ziemann, P. and Gogolla, M. (2003) OCL extended with temporal logic. In Broy, M. and Zamulin, A. [eds.] *5th Conference on Perspectives of Systems Informatics (PSI) 2003* (Springer), *LNCS* **2890**: 351–357. doi:10.1007/978-3-540-39866-0_35, URL http://dx.doi.org/10.1007/978-3-540-39866-0_35.

[31] Flake, S. and Müller, W. (2004) Past- and future-oriented time-bounded temporal properties with OCL. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004)* (IEEE Computer Society): 154–163. doi:10.1109/SEFM.2004.29, URL http://doi.ieeecomputersociety.org/10.1109/SEFM.2004.29.

[32] IBM, Rhapsody. URL http://www-03.ibm.com/software/products/en/ratirhapdesiforsystengi.

[33] David, A., Larsen, K.G., Legay, A., Mikucionis, M., Poulsen, D.B. and Sedwards, S. (2015) Statistical model checking for biological systems. *STTT* **17**(3): 351–367. doi:10.1007/s10009-014-0323-4, URL http://dx.doi.org/10.1007/s10009-014-0323-4.

[34] Cappart, Q., Limbrée, C., Schaus, P., Quilbeuf, J., Traonouez, L.M. and Legay, A. Verification of interlocking systems using statistical model checking. Submitted to IFM2016.

[35] Colombo, A., Fontanelli, D., Legay, A., Palopoli, L. and Sedwards, S. (2015) Efficient customisable dynamic motion planning for assistive robots in complex human environments. *JAISE* **7**(5): 617–634. doi:10.3233/AIS-150338, URL http://dx.doi.org/10.3233/AIS-150338.

[36] Aa., V. (2015) *DANSE Final Report*. Tech. rep., DANSE deliverable.

[37] Zeigler, B. (2013) *Guide to Modeling and Simulation of Systems of Systems - User's Reference*, Springer Briefs in Computer Science (Springer). doi:10.1007/978-1-4471-4570-7, URL http://dx.doi.org/10.1007/978-1-4471-4570-7.

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

[39] Boulanger, J. (2012) *Industrial Use of Formal Methods* (John Wiley and Sons). doi:10.1002/9781118561829.

[40] Basu, A., Bozga, M. and Sifakis, J. (2006) Modeling heterogeneous real-time components in BIP. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 11-15 September 2006, Pune, India*: 3–12. doi:10.1109/SEFM.2006.27, URL http://dx.doi.org/10.1109/SEFM.2006.27.

[41] ARBAB, F. (2004) Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science* **14**: 329–366. doi:10.1017/S0960129504004153, URL http://journals.cambridge.org/article_S0960129504004153.

[42] Nouri, A., Bensalem, S., Bozga, M., Delahaye, B., Jégourel, C. and Legay, A. (2015) Statistical model checking qos properties of systems with SBIP. *STTT* **17**(2): 171–185. doi:10.1007/s10009-014-0313-6, URL http://dx.doi.org/10.1007/s10009-014-0313-6.

[43] Moon, Y.J., Silva, A., Krause, C. and Arbab, F. (2014) A compositional model to reason about end-to-end qos in stochastic reo connectors. *Science of Computer Programming* **80, Part A**: 3 – 24. doi:http://dx.doi.org/10.1016/j.scico.2011.11.007, URL http://www.sciencedirect.com/science/article/pii/S0167642311002073. Special section on foundations of coordination languages and software architectures (selected papers from FOCLASA'10), Special section - Brazilian Symposium on Programming Languages (SBLP 2010) and Special section on formal methods for industrial critical systems (Selected papers from FMICS'11).

[44] Le, T., Passerone, R., Fahrenberg, U. and Legay, A. (2013) A tag contract framework for heterogeneous systems. In Canal, C. and Villari, M. [eds.] *Advances in Service-Oriented and Cloud Computing* (Springer Berlin Heidelberg), *Communications in Computer and Information Science* **393**, 204–217. doi:10.1007/978-3-642-45364-9_17, URL http://dx.doi.org/10.1007/978-3-642-45364-9_17.

[45] Jégourel, C., Legay, A. and Sedwards, S. (2013) Importance splitting for statistical model checking rare properties. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*: 576–591. doi:10.1007/978-3-642-39799-8_38, URL http://dx.doi.org/10.1007/978-3-642-39799-8_38.