Statistical Model Checking of Dynamic Software Architectures

Everton Cavalcante, Jean Quilbeuf, Louis-Marie Traonouez, Flavio Oquendo, Thais Batista, Axel Legay

▶ **To cite this version:**

Everton Cavalcante, Jean Quilbeuf, Louis-Marie Traonouez, Flavio Oquendo, Thais Batista, et al.. Statistical Model Checking of Dynamic Software Architectures. ECSA 2016 - 10th European Conference on Software Architecture, Nov 2016, Copenhague, Denmark. hal-01390707

HAL Id: hal-01390707
https://inria.hal.science/hal-01390707v1

Submitted on 7 Nov 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Statistical Model Checking of Dynamic Software Architectures

Everton Cavalcante[1,2], Jean Quilbeuf[2,3], Louis-Marie Traonouez[3], Flavio Oquendo[2], Thais Batista[1], Axel Legay[3]

[1]DIMAp, Federal University of Rio Grande do Norte, Natal, Brazil
[2]IRISA-UMR CNRS/Université Bretagne Sud, Vannes, France
[3]INRIA Rennes Bretagne Atlantique, Rennes, France

evertonrsc@ppgsc.ufrn.br, jean.quilbeuf.irisa.fr, louis-marie.traonouez@inria.fr, flavio.oquendo@irisa.fr, thais@ufrnet.br, axel.legay@inria.fr

Abstract. The critical nature of many complex software-intensive systems calls for formal, rigorous architecture descriptions as means of supporting automated verification and enforcement of architectural properties and constraints. Model checking has been one of the most used techniques to automatically analyze software architectures with respect to the satisfaction of architectural properties. However, such a technique leads to an exhaustive exploration of all possible states of the system under verification, a problem that becomes more severe when verifying dynamic software systems due to their typical non-deterministic runtime behavior and unpredictable operation conditions. To tackle these issues, we propose using statistical model checking (SMC) to support the analysis of dynamic software architectures while aiming at reducing effort, computational resources, and time required for this task. In this paper, we introduce a novel notation to formally express architectural properties as well as an SMC-based toolchain for verifying dynamic software architectures described in π-ADL, a formal architecture description language. We use a flood monitoring system to show how to express relevant properties to be verified, as well as we report the results of some computational experiments performed to assess the efficiency of our approach.

1 Introduction

One of the major challenges in software engineering is to ensure correctness of software-intensive systems, especially as they have become increasingly complex and used in many critical domains. Ensuring these concerns becomes more important mainly when evolving these systems since such a verification needs to be performed before, during, and after evolution. In this context, software architectures play an essential role since they represent an early blueprint for the system construction, deployment, execution, and evolution, thereby fostering an early analysis of a system and contributing to reduce the cost of software maintenance.

The critical nature of many complex software systems calls for rigorous architectural models (such as formal architecture descriptions) as means of supporting

the automated verification and enforcement of architectural properties. However, architecture descriptions should not cover only structure and behavior of a software architecture, but also the required and desired architectural properties, in particular the ones related to consistency and correctness [16]. For instance, after describing a software architecture, a software architect might want to verify if it is complete, consistent, and correct with respect to architectural properties.

In order to foster the automated verification of architectural properties based on architecture descriptions, they need to be formally specified. Despite the inherent difficulty of pursuing formal methods, the advantage of a formal verification is to precisely determine if a software system can satisfy properties related to user requirements. Additionally, automated verification provides an efficient method to check the correctness of architectural design. As reported by Zhang et al. [20], one of the most popular formal methods for analyzing software architectures is *model checking*, an exhaustive, automatic verification technique whose general goal is to verify if an architectural specification satisfies architectural properties [8]. It takes as inputs a representation of the system (e.g., an architecture description) and a set of property specifications expressed in some notation. The model checker returns true, if the properties are satisfied, or false with the case in which a given property is violated.

Despite its wide and successful use, model checking faces a critical challenge with respect to scalability. Holzmann [10] remarks that no currently available traditional model checking approach is exempted from the *state space explosion problem*, that is, the exponential growth of the state space. This problem is exacerbated in the contemporary dynamic software systems for two main reasons, namely (i) the non-determinism of their behavior caused by concurrency and (ii) the unpredictable environmental conditions in which they operate. In spite of the existence of a number of techniques aimed at reducing the state space, such a problem remains intractable for some software systems, thereby making the use of traditional model checking techniques a prohibitive choice in terms of execution time and computational resources. As a consequence, software architects have to trade-off the risks of possibly undiscovered problems related to the violation of architectural properties against the practical limitations of applying a model checking technique on a very large architectural model.

Verification techniques such as model checking require not only significant execution time and computational resources, but also an unneglectable effort from architects. This is one of the major reasons that often hinders the adoption of formal-based techniques in software industry, as revealed in a recent survey in this context [15]. Therefore, providing affordable, efficient approaches for rigorously verifying properties in dynamic software architectures is a major challenge.

In order to tackle the aforementioned issues, this paper proposes the use of *statistical model checking* (SMC) for supporting the formal analysis of dynamic software architectures while striving to reduce effort, computational resources, and time for performing this task. SMC is a probabilistic, simulation-based technique intended to verify, at a given confidence level, if a certain property is satisfied during the execution of a system [13]. Unlike conventional formal verification

techniques such as model checking, SMC does not analyze the internal logic of the target system, thereby not suffering from the state space explosion problem [12]. Therefore, an architect wishing to assess the correctness of an architecture has to build an executable model of the system. In our opinion, this is much easier than building a model of the system that is abstract enough to be used in a model checker, yet detailed enough to detect meaningful errors. Furthermore, an SMC-based approach promotes better scalability and less consumption of computational resources, important factors to be considered when analyzing software architectures for complex critical systems. SMC requires (i) a model whose execution is probabilistic and (ii) a language for expressing properties to be verified and a monitor for deciding them on finite traces.

Our main contribution is a toolchain for verifying dynamic software architectures described in π-ADL, a formal architecture description language [5, 17]. The π-ADL language does not natively allow a probabilistic execution, but rather provides a non-deterministic specification of a dynamic architecture. Therefore, we obtain a probabilistic model by resolving non-determinism by probabilities, which is enforced by a stochastic scheduler. We used DynBLTL [19], a new logic to express properties about dynamic systems. Furthermore, we use a real-world flood monitoring system to show how to express relevant properties to be verified, as well as we report the results of some computational experiments performed to assess the efficiency of our approach.

The remainder of this paper is organized as follows. Section 2 briefly presents the SMC technique. Section 3 details how to stochastically execute π-ADL architecture descriptions. Section 4 introduces our notation to formally express properties of dynamic software architectures. Section 5 presents the developed SMC-based toolchain to verify dynamic software architectures. In Section 6, a flood monitoring system is used as case study to show how to express properties with DynBLTL and the results of experiments on the computational effort to verify these properties. Finally, Section 7 contains some concluding remarks.

2 Statistical Model Checking

SMC is a probabilistic, simulation-based approach that consists of building a statistical model of finite executions of the system under verification and deducing the probability of satisfying a given property within confidence bounds. This technique provides a number of advantages in comparison to traditional model checking techniques. First (and perhaps the most important one), this technique does not suffer from the state space explosion problem since it does not analyze the internal logic of the system under verification, neither requires the entire representation of the state space, thus making it a promising approach for verifying complex large-scale and critical software systems [12]. Second, SMC requires only the system be able to be simulated, so that it can be applied to larger classes of systems, including black-box and infinite-state systems. Third, the proliferation of parallel computer architectures makes the production of multiple independent simulation runs relatively easier. Fourth, despite SMC can provide approximate

results (as opposed to exact results provided by traditional model checking), it is compensated by a better scalability and less consumption of computational resources. In some cases, knowing the result with less than 100% of confidence is quite acceptable or even the unique available option. Therefore, SMC allows trading-off between verification accuracy and computational time by selecting appropriate precision parameter values. For example, if the project time is limited, it may be more valuable obtaining less precise verification in short time than more precise verification results in much longer time.

Figure 1 illustrates a general schema on how the SMC technique works. A statistical model checker basically consists of a simulator for running the system under verification, a model checker for verifying properties, and a statistical analyzer responsible for calculating probabilities and performing statistical tests. It receives three inputs: (i) an *executable stochastic model* of the target system M; (ii) a formula φ expressing a *bounded property* to be verified, i.e., a property that can be decided over a finite execution of M; and (iii) user-defined *precision parameters* determining the accuracy of the probability estimation. The model M is stochastic in the sense that the next state is probabilistically chosen among the states that are reachable from the current one. As a consequence, some executions of M satisfy φ and others do not satisfy it, depending on the probabilistic choices made during these executions. The simulator executes M and generates an *execution trace* σ_i, composed of a sequence of *states*. Next, the model checker determines if σ_i satisfies φ and sends the result (either success or failure) to the statistical analyzer, which in turn estimates the probability p for M to satisfy φ. The simulator repeatedly generates other execution traces σ_{i+1} until the analyzer determines that enough traces have been analyzed to produce an estimation of p satisfying the precision parameters. It is important to highlight that a higher accuracy of the answer provided by the model checker requires generating more execution traces through simulations.

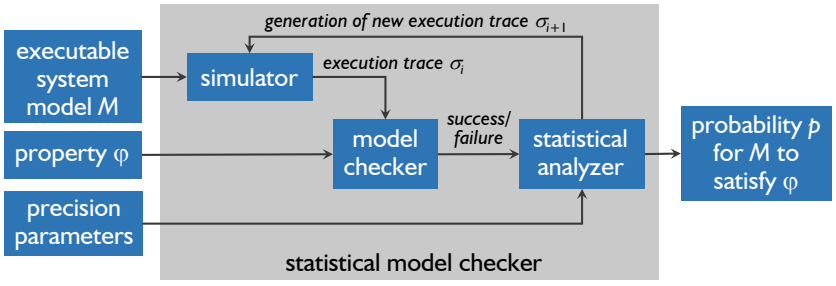


Fig. 1. Working schema of the SMC technique.

3 Stochastic Execution of π-ADL models

In this section, we briefly recall how the π-ADL language allows describing dynamic software architectures. As SMC is a stochastic technique, the executable

model representing the system needs to be stochastic, a feature that π-ADL does not possess. For this reason, we have provided a way of producing a stochastic executable model from π-ADL architecture descriptions, thus allowing for property verification using SMC. Finally, we show how to extract execution traces from a stochastic execution.

3.1 Modeling Dynamic Architectures in π-ADL

π-ADL [17] is a formal, well-founded theoretically language intended to describe software architectures under both structural and behavioral viewpoints. In order to cope with dynamicity concerns, π-ADL is be endowed with architectural-level primitives for specifying programmed reconfiguration operations, i.e., foreseen, pre-planned changes described at design time and triggered at runtime by the system itself under a given condition or event. Additionally, code source in the Go programming language [1] is automatically generated from π-ADL architecture descriptions, thereby allowing for their execution [6].

From the structural viewpoint, a software architecture is described in π-ADL in terms of *components*, *connectors*, and their composition to form the system, i.e., an *architecture* as a configuration of components and connectors. From the behavioral viewpoint, both components and connectors comprise a *behavior*, which expresses the interaction of an architectural element and its internal computation and uses *connections* to send and receive values between architectural elements. The attachment of a component to a connector (and vice-versa) is made by *unifying* their connections. Therefore, the transmission of a value from an architectural element to another is possible only if (i) the output connection of the sender is unified to the input connection of the receiver, (ii) the sender is ready to send a value through that output connection, and (iii) the receiver is ready to receive a value on that input connection.

In π-ADL, dynamic reconfiguration is obtained by *decomposing* architectures [5]. The decomposition action removes all unifications defined in the original architecture, but it does not terminate its elements. The decomposition of a given architecture A is typically called from another coexisting architecture B, which results from a reconfiguration applied over A. After calling the decomposition of A, B can access and modify the elements originally instantiated in A.

3.2 Resolving Non-Determinism in π-ADL

In π-ADL, non-determinism occurs in two different ways. First, whenever several actions are possible, any one of them can be executed as the next action, i.e., the choice of the next action to execute is non-deterministic. Second, some functions can declared as *unobservable*, thus meaning that its internal operations are concealed at the architectural level. In this case, the value returned by the function is also non-deterministic because it is not defined in the model. As a stochastic process is required for performing SMC, we resolve the non-determinism of π-ADL models by using probabilities. In the following, we describe how to proceed in the aforementioned cases.

Resolving non-determinism in the choice of the next action. The Go code from a π-ADL architecture description encodes architectural element (component or connector) as a concurrent goroutine, a lightweight process similar to a thread. The communication between architectural elements takes place via a channel, another Go construct. If several communications are possible, the Go runtime chooses one of them to execute according to a FIFO (first-in, first out) policy. Such a policy is not suitable for SMC since it is necessary to specify how the next action is chosen.

To support the stochastic scheduling of actions, we have implemented a scheduler as a goroutine that controls all non-local actions, i.e., composition, decomposition or communication. Whenever an architectural element needs to perform a non-local action, it informs the scheduler and blocks until the scheduler responds. The scheduler responds with the action executed (if the component submitted a choice between several actions) and a return value, corresponding either to the reception side of a communication or a decomposed architecture.

Fig. 2 depicts the behavior of the scheduler. The scheduler waits until all components and connectors have indicated their possible actions. At this step, the scheduler builds a list of possible rendezvous by checking which declared unifications have both sender and receiver ready to communicate. For this purpose, the scheduler maintains a list of the active architectures and the corresponding unifications. The possible communications are added to the list of possible actions and the scheduler chooses one of them according to a probabilistic choice function. The scheduler then executes the action and outputs its effect to the statistical model checker. Finally, the scheduler notifies the components and connectors involved in the action.

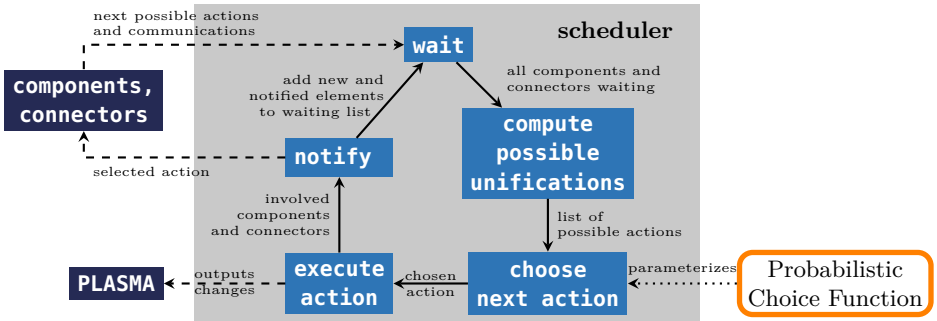


Fig. 2. Scheduler to support the stochastic simulation of a π-ADL model

Resolving non-determinism in unobservable functions. The functions declared unobservable require an implementation to allow simulating the model. In practice, this implementation is provided in form of a Go function whose return value can be determined by a probability distribution. Such an implementation relies on the Go libraries that implement usual probability distribu-

tions. In particular, such functions can model inputs of the systems that have a known probabilistic value, i.e., input to a component, time to the next failure of a component, etc.

3.3 Trace of a Stochastic Execution

In order to perform SMC for verifying dynamic software architectures, we abstract away the inner structure of the components and represent a state of the system as a directed graph $g = (V, E)$ in which V is a finite set of nodes and E is a finite set of edges. Each node $v \in V$ represents an architectural element (component or connector) whereas each direct edge $e \in E$ represents a communication channel between two architectural elements.

The SMC technique relies on checking multiple execution traces resulted from simulations of the system under verification against the specified properties. Therefore, as a simulation ω results in a trace σ composed of a finite sequence of states, σ can be defined as a sequence of state graphs g_i ($i \in \mathbb{N}$), i.e., $\sigma = \{g_0, g_1, \ldots, g_n\}$. Aiming at obtaining an execution trace from an architecture description in π-ADL, the simulation emits explicit messages recording a set of actions on the state graph.

4 A Novel Notation for Expressing Properties in Dynamic Software Architectures

Most architectural properties to be verified by using model checking techniques are temporal [20], i.e., they are qualified and can be reasoned upon a sequence of system states along the time. In the literature, linear temporal logic (LTL) [18] has been often used as underlying formalism for specifying temporal architectural properties and verifying them through model checking. LTL extends classical Boolean logic with *temporal operators* that allow reasoning on the temporal dimension of the execution of the system. In this perspective, LTL can be used to encode formulas about the future of execution paths (sequences of states), e.g., a condition that will be eventually true, a condition that will be true until another fact becomes true, etc.

Besides using standard propositional logic operators, LTL defines four temporal operators, namely: (i) *next*, which means that a formula φ will be true in the next step; (ii) *finally* or *eventually*, which indicates that a formula φ will be true at least once in the time interval; (iii) *globally* or *always*, which means that a formula φ will be true at all times in the time interval; and (iv) *until*, which indicates that either a formula φ is initially true or another previous formula ψ is true until φ become true at the current or a future time. SMC techniques verify *bounded properties*, i.e., where temporal operators are parameterized by a time bound. While LTL-based formulas aim at specifying the infinite behavior of the system, a time-bounded form of LTL called BLTL considers properties that can be decided on finite sequences of execution states.

Traditional versions of temporal logics such as LTL and BLTL are expressed over atomic predicates that evaluate properties to a Boolean value at every point of execution. However, a key characteristic of dynamic software systems is the impossibility of foreseeing the exact set of architectural elements deployed at a given point of execution. Such traditional formalisms do not allow reasoning about elements that may appear, disappear, be connected or be disconnected during the execution of the system for two main reasons. First, specifying a predicate for each property of each element is not possible as the set of architectural elements may be unknown a priori. Second, there is no canonical way of assigning a truth value to a property about an element that does not exist at the considered point of execution. In addition, existing approaches to tackle such issues typically focus on behavioral properties, but they do not address architectural properties [7]. On the other hand, some approaches assume that the architectures are static [3]. These limitations have led us to propose DynBLTL, an extension of BLTL aimed at formally expressing properties in dynamic software architectures.

The novel notation was designed to handle the absence of an architectural element in a given formula expressing a property (c.f. [19]). In practice, this means that a Boolean expression can take three values, namely *true*, *false* or *undefined*. The undefined value refers to the fact that an expression may not be evaluated depending on the current runtime configuration of the system. This is necessary for situations in which it is not possible to evaluate an expression in the considered point of execution, e.g., a statement about an architectural element that does not exist at that moment. Some operators interpret the undefined value as true or false, depending on the context. Such operators have to be used at the root of the formula to ensure that it returns a boolean.

DynBLTL allows expressing properties using (i) arithmetic, logical, and comparison operations on values, (ii) existential and universal quantifications, and traditionally used in predicate logic, and (iii) some predefined functions that can be used to explore the architectural configuration. Furthermore, there are four temporal operators, namely *in*, *eventually before*, *always during*, and *until*, which are similar to the ones defined in both LTL and BLTL. Some examples of DynBLTL properties are presented in Section 6.2.

5 A Toolchain to Simulate and Verify Dynamic Software Architectures

SMC techniques rely on the simulation of an executable model of the system under verification against a set of formulas expressing bounded properties to be verified (see Section 2). These elements are provided as inputs to a statistical model checker, which basically consists of (i) a simulator for running the executable model of the system under verification, (ii) a model checker for verifying properties, and (iii) a statistical analyzer responsible for calculating probabilities and performing statistical tests.

Among the SMC tools available in the literature, PLASMA [2] is a compact, flexible platform that enables users to create custom SMC plug-ins atop it. For instance, users who have developed their own model description language can use it with PLASMA by providing a simulator plug-in. Similarly, users can add custom languages for specifying properties and use the available SMC algorithms through a checker plug-in. Besides its efficiency and good performance results [4, 11, 14], such a flexibility was one of the main reasons motivating the choice of PLASMA to serve as basis to develop the toolchain for specifying and verifying properties of dynamic software architectures.

Fig. 3 provides an overview of our SMC-based toolchain for verifying properties of dynamic software architectures. The inputs for the process are (i) an architecture description in π-ADL and (ii) a set of properties specified in DynBLTL. By following the process proposed in our previous work [5, 6], the architecture description in π-ADL is translated towards generating source code in Go. As π-ADL architectural models do not have a stochastic execution, they are linked to a stochastic scheduler parameterized by a probability distribution for drawing the next action, as described in Section 3. Furthermore, we use existing probability distribution Go libraries to model inputs of system models as user functions. The program resulting from the compilation of the generated Go source code emits messages referring to transitions from a given state to another in case of addition, attachment, detachment, and value exchanges of architectural elements.

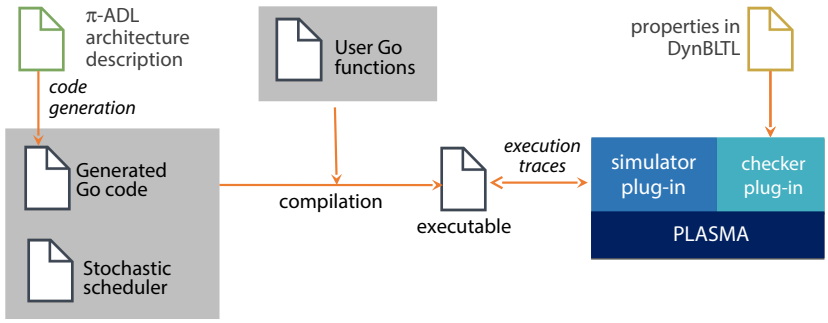


Fig. 3. Overview of the proposed SMC-based toolchain for verifying properties of dynamic software architectures.

We have developed two plug-ins[1] atop the PLASMA platform, namely (i) a simulator plug-in that interprets execution traces produced by the generated Go program and (ii) a checker plug-in that implements DynBLTL. With this toolchain, a software architect is able to evaluate the probability of a π-ADL architectural model to satisfy a given property specified in DynBLTL.

[1] The developed tools are available at `http://plasma4pi-adl.gforge.inria.fr`.

6 Case Study

In this section, we apply our approach to a real-world flood monitoring system used as a case study. Section 6.1 presents an overview of the system and Section 6.2 describes some relevant properties to be verified in the context of this system. At last, Section 6.3 reports some computational experiments performed to assess the efficiency of our approach with the developed toolchain.

6.1 Description

A flood monitoring system can support monitoring urban rivers and create alert messages to notify authorities and citizens about the risks of an imminent flood, thereby fostering effective predictions and improving warning times. This system is typically based on a wireless sensor network composed of sensors that measures the water level in flood-prone areas near the river. In addition, a gateway station analyzes data measured by motes, makes such data available, and can trigger alerts when a flood condition is detected. The communication among these elements takes place by using wireless network connections, such as WiFi, ZigBee, GPRS, Bluetooth, etc.

Fig. 4 shows the main architecture of the system. Sensor components communicate with each other through ZigBee connectors and a gateway component receives all measurements to evaluate the current risk. Each measure taken by a sensor is propagated its neighbors via ZigBee connectors until reaching the gateway.

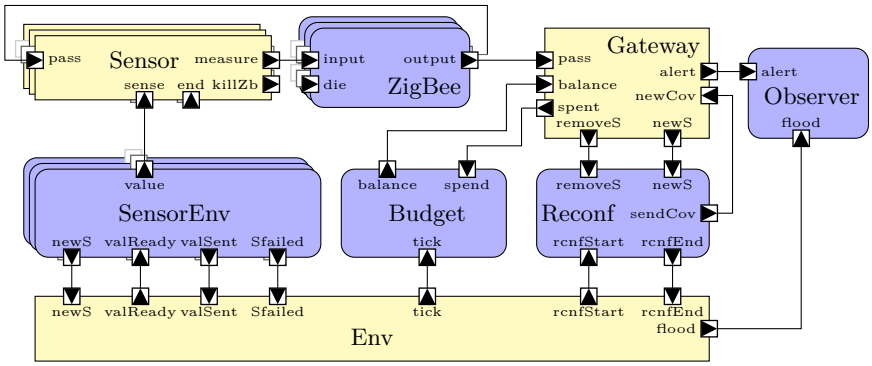


Fig. 4. Overview of the main architecture of the flood monitoring system. Components are yellow-colored whereas connectors are blue-colored.

Fig. 5 shows an excerpt of the π-ADL description for the sensor component. The behavior of this components comprises choosing between two alternatives, either obtaining a new measure (i) from the environment via the *sense* input connection or (ii) from a neighbor sensor via the *pass* input connection. After

receiving the gathered value, it is transmitted through the `measure` output connection. Reading a negative value indicates a failure of the sensor, so that it becomes a *FailingSensor*, which simply ignores all incoming messages.

```
component Sensor is abstraction() {
  type CmH20 is tuple[Behavior,Real]
  type MV is Real
  connection sense is in(MV)
  connection measure is out(CmH20)
  connection pass is in(CmH20)
  connection end is in(Integer)
  connection killZb is out(Boolean)
  behavior is {
    choose {
        via sense receive m : MV
        via measure send CmH20(tuple[self, m])
        if m < 0.0 then {
          become(FailingSensor())
        }

      or
        via pass receive other_measure : CmH20
        via measure send other_measure
    }
  }
}
```

Fig. 5. Partial π-ADL description of the sensor component

The environment is modeled through the *Env* component as well as the *SensorEnv* and *Budget* connectors. *Env* is responsible for synchronizing the model by defining cycles corresponding to the frequency at which measures are taken by sensors. A cycle consists in: (i) signaling *Budget* that a new cycle has started; (ii) updating the river status; (iii) registering newly deployed sensors; (iv) signaling each *SensorEnv* connector to deliver a new measure, and (v) waiting for each *SensorEnv* connector to confirm that a new measure has been delivered.

The *Sensor*, *SensorEnv*, and *ZigBee* elements can dynamically added and removed during the execution of the system, through reconfigurations triggered by the gateway component. We have modeled two reconfigurations, namely adding and removing a sensor, as depicted in Fig. 6.

The gateway component decides to add a sensor if the coverage of the river is not optimal and the budget is sufficient to deploy a new sensor. This operation is triggered by sending a message to *Reconf* via the *newS* connection, with the desired location for the new sensor. The new sensor is connected to other sensors in range via a ZigBee connector, as shown in Fig. 6(a). During this operation, *Reconf* decomposes the main architecture to include the new elements and unifications before recomposing it. The reconfiguration uses the position of each sensor to determine which links have to be created. After triggering the

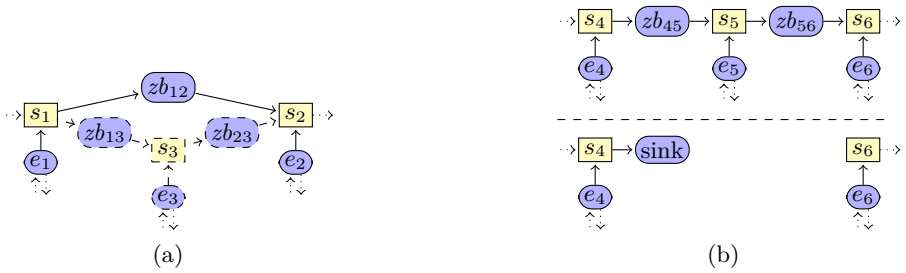


Fig. 6. Reconfigurations in the flood monitoring system: (a) adding sensor s_3, which requires connecting it to existing sensors s_1 and s_2 through new ZigBee connectors; (b) removal of sensor s_5.

reconfiguration, the gateway indicates to the *Budget* connector that it spent the price of a sensor.

The gateway removes a sensor when it receives a message indicating that it is in failure. This operation is triggered by sending a message to *Reconf* via the *removeS* connection, with the name of the sensor to remove. Removing a sensor may isolate other sensors that are further away from the gateway as it in shown in Fig. 6(b). In this case, sensors that were sending their measures via the removed sensor (such as s_4) are instead connected to a sink connector, which loses all messages. This new connection prevents deadlocks that occur when the last element of the isolated chain cannot propagate its message.

Finally, when a sensor is removed, the connected *ZigBee* and *SensorEnv* are composed in a separated architecture. This architecture connects the *killZb* connection of the sensor to the *die* connections of the ZigBee connectors, which allows an other branch of the behavior to properly terminate these components.

6.2 Requirements

As previously mentioned, a DynBLTL formula requires bounds on temporal operators to ensure that it can be decided in a finite number of steps. We have two possibilities to express bounds, namely using steps or using time units. Usually, the number of steps executed during a time unit depends on the number of components in the system. For instance, the number of steps executed during a cycle mainly depends on the number of sensors deployed since each sensor reads one value at each cycle. In our model, a time unit correspond to a cycle, thus allowing us to specify bounds that are independent of the number of components in the system.

First, we want to evaluate the correctness of our model with respect to its main goal, i.e., warning about imminent flooding. In this context, a false negative occurs when the system fails to predict a flood.

```
eventually before X time units {                          // FalseNeg(X,Y)
  (gw.alert = "low") and (eventually before Y time units env.flood)
}
```

This property characterizes a false negative: the gateway predicts a low risk and a flood occurs in the next Y time units. The parameters of this formula are X, the time during which the system is monitored, and Y, the time during which the prediction of the gateway should hold. Similarly, a false positive occurs when the system predicts a flood that does not actually occur:

```
eventually before X time units {                          // FalsePos(X,Y)
  gw.alert = "flood detected"
  and always during Y time units not env.flood
}
```

The system is correct if there is no false negatives nor false positives for the expected prediction anticipation (parameter Y).

These two formulas are actually BLTL formulas as they involve simple predicate on the state. However, DynBLTL allows expressing properties about the dynamic architecture of the system. For example, suppose that one wants to check that if a sensor sends a message indicating that it is failing, then it is removed from the system in a reasonable amount of time. This disconnection is needed because the sensor in failure will not pass incoming messages. We characterize the removal of a sensor by a link on the *end* connection, corresponding to the initiation of the sensor termination (not detailed here).

In our dynamic system, sensors may appear and disappear during execution. Therefore, the temporal pattern above needs to be dynamically instantiated at each step for each existing sensor:

```
always during X time units {                          // RemoveSensor(X,Y)
  forall s:allOfType(Sensor) {
      (isTrue s.measure <  0) implies {
        eventually before Y time units {
            exists st:allOfType(StartTerminate)
              areLinked(st.start,s.end)
      }
    }
  }
}
```

This property cannot be stated in BLTL since it does not have a construct such as “forall” for instantiating a variable number of temporal sub-formulas, where the number depend on the current state.

Another property of interest consists in checking if a sensor is available, i.e., at least one sensor is connected to the gateway. More precisely, we require that there is a ZigBee connected to the gateway and to a sensor. If not, we require that such a sensor appear in less than Y time units:

```
always during X time units {                       // SensorAvailable(X,Y)
  (not (exists zb:allOfType(ZigBee) areLinked(zb.output,gw.pass)
```

```
   and (exists s:allOfType(Sensor)  areLinked(s.measure,zb.input))))
  implies (eventually before Y time units {
      exists zb:allOfType(ZigBee) areLinked(zb.output,gw.pass)
      and (exists s:allOfType(Sensor) areLinked(s.measure,zb.input))
  }
}
```

6.3 Experimental Results

In this section, we report some experiments aiming to quantitatively evaluate the efficiency of our approach to support the architectural analysis activity. Considering that the literature already reports that PLASMA and its SMC algorithms outperform other existing approaches (c.f. [4, 11, 14]), we are hereby interested in assessing how efficient is our approach and toolchain to verify properties in dynamic software architectures. In the experiments, we have chosen computational effort in terms of execution time and RAM consumption as metrics, which were used to observe the performance of our toolchain when varying the precision of the verification. As PLASMA is executed upon a Java Virtual Machine, 20 runs were performed for each precision value in order to ensure a proper statistical significance for the results. The experiments were conducted under GNU/Linux on a computer with a quad-core 3 GHz processor and 16 GB of RAM.

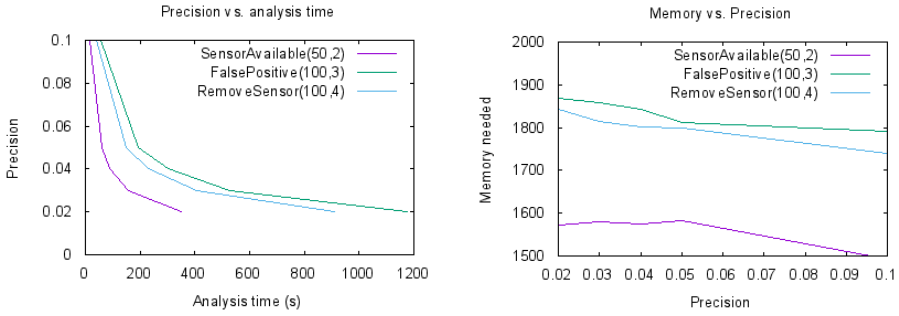
We evaluate our toolchain with the properties from Section 6.2. We rely on the Chernoff algorithm [9] from PLASMA, which requires a precision and a confidence degree as parameters. The algorithm returns an approximation of the probability with an error below the precision parameter, with the given confidence. We choose a confidence of 95% and a precision of either 0.1, 0.05, 0.04, 0.03 or 0.02, requiring respectively 185, 738, 1153, 2050 and 4612 simulations.

Fig. 7(a) shows how the error decreases when the analysis time increases. The property regarding the sensor availability evaluated over a window of 50 time units requires less time than the other properties evaluated over a window of 100 time units because the analysis of each trace is faster. In Fig. 7(b), it is possible to observe that the increase of the amount of RAM required to perform the analyses is not strongly influenced by the precision. This can be explained by the fact that SMC only analyzes one trace at a time. Finally, we can conclude that our SMC approach and toolchain can be regarded as efficient with respect to both execution time and RAM consumption.

Discussion. We rely here on the Chernov bound to compute the number of simulations needed, which increases quadratically in the precision. In case of rare events, i.e., properties that have a very low probability to happen, a better convergence can be obtained by using dedicated methods [11]. In terms of size, our current model contains about 30 processes in total.

7 Final Remarks

In this paper, we have presented our approach on the use of statistical model checking (SMC) to verify properties in dynamic software architectures. Our main



(a) (b)

Fig. 7. Effects of the variation in the precision in the analysis of three properties upon analysis time (a) and amount of RAM required (b).

contribution is an SMC-based toolchain for specifying and verifying such properties atop the PLASMA platform. The inputs for this process are a probabilistic version of an architecture description in the π-ADL language and a set of properties expressed in DynBLTL. We have used a real-world flood monitoring system to show how to specify properties in a dynamic software architectures, as well as it was used in some computational experiments aimed to demonstrate that our approach and toolchain are efficient and hence feasible to be applied on the architectural analysis task. To the best of our knowledge, this is the first work on the application of SMC to verify properties in dynamic software architectures.

As future work, we need to assess the expressiveness and usability of DynBLTL for expressing properties in dynamic software architectures. Finally, we intend to integrate our approach into a framework aimed to support software architects in activities such as architectural representation and formal verification of architectural properties.

References

1. The Go programming language. `https://golang.org/`
2. PLASMA-Lab. `https://project.inria.fr/plasma-lab/`
3. Arnold, A., Boyer, B., Legay, A.: Contracts and behavioral patterns for SoS: The EU IP DANSE Approach. In: Larsen, K.G., Legay, A., Nyman, U. (eds.) Proceedings of the 1st Workshop on Advances in Systems of Systems. EPTCS, vol. 133, pp. 47–60 (2013)
4. Boyer, B., Corre, K., Legay, A., Sedwards, S.: PLASMA-lab: A flexible, distributable statistical model checking library. In: Joshi, K., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) Proceedings of the 10th International Conference on Quantitative Evaluation of Systems, Lecture Notes in Computer Science, vol. 8054, pp. 160–164. Springer Berlin Heidelberg, Germany (2013)
5. Cavalcante, E., Batista, T., Oquendo, F.: Supporting dynamic software architectures: From architectural description to implementation. In: Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture. pp. 31–40. IEEE Computer Society, USA (2015)

6. Cavalcante, E., Oquendo, F., Batista, T.: Architecture-based code generation: From π-adl descriptions to implementations in the Go language. In: Avgeriou, P., Zdun, U. (eds.) Proceedings of the 8th European Conference on Software Architecture, Lecture Notes in Computer Science, vol. 8627, pp. 130–145. Springer International Publishing, Switzerland (2014)
7. Cho, S.M., Kim, H.H., Cha, S.D., Bae, D.H.: Specification and validation of dynamic systems using temporal logic. IEE Proceedings – Software 148(4), 135–140 (Aug 2001)
8. Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model checking. The MIT Press, Cambridge, MA, USA (1999)
9. Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate probabilistic model checking. In: Steffen, B., Levi, G. (eds.) Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Implementations, Lecture Notes in Computer Science, vol. 2937, pp. 73–84. Springer Berlin Heidelberg, Germany (2004)
10. Holzmann, G.J.: The logic of bugs. In: 10th ACM SIGSOFT Symposium on Foundations of Software Engineering. pp. 81–87. ACM, New York, NY, USA (2002)
11. Jegourel, C., Legay, A., Sedwards, S.: A platform for high performance statistical model checking – PLASMA. In: Flanagan, C., König, B. (eds.) Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 7214, pp. 498–503. Springer Berlin Heidelberg, Germany (2012)
12. Kim, Y., Choi, O., Kim, M., Baik, J., Kim, T.H.: Validating software reliability early through statistical model checking. IEEE Software 30(3), 35–41 (May/Jun 2013)
13. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: Barringer, H., et al. (eds.) First International Conference on Runtime Verification, Lecture Notes in Computer Science, vol. 6418, pp. 122–135. Springer Berlin Heidelberg, Germany (2010)
14. Legay, A., Sedwards, S.: On statistical model checking with PLASMA. In: Proceedings of the 2014 Theoretical Aspects of Software Engineering Conference. pp. 139–145. IEEE Computer Society, Washington, DC, USA (2014)
15. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: A survey. IEEE Transactions on Software Engineering 39(6), 869–891 (Jun 2013)
16. Mateescu, R., Oquendo, F.: π-AAL: An architecture analysis language for formally specifying and verifying structural and behavioural properties of software architectures. ACM SIGSOFT Software Engineering Notes 31(2), 1–19 (Mar 2006)
17. Oquendo, F.: π-ADL: An architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. ACM SIGSOFT Software Engineering Notes 29(3), 1–14 (May 2004)
18. Pnueli, A.: The temporal logics of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science. pp. 46–57. IEEE Computer Society, Washington, DC, USA (1977)
19. Quilbeuf, J., Cavalcante, E., Traonouez, L.M., Oquendo, F., Batista, T., Legay, A.: A logic for statistical model checking of dynamic software architectures. In: Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (2016), to appear.
20. Zhang, P., Muccini, H., Li, B.: A classification and comparison of model checking software architecture techniques. Journal of Systems and Software 83(5), 723–744 (May 2010)