A Logic for the Statistical Model Checking of Dynamic Software Architectures

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

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

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

Abstract. Dynamic software architectures emerge when addressing important features of contemporary systems, which often operate in dynamic environments subjected to change. Such systems are designed to be reconfigured over time while maintaining important properties, e.g., availability, correctness, etc. Verifying that reconfiguration operations make the system to meet the desired properties remains a major challenge. First, the verification process itself becomes often difficult when using exhaustive formal methods (such as model checking) due to the potentially infinite state space. Second, it is necessary to express the properties to be verified using some notation able to cope with the dynamic nature of these systems. Aiming at tackling these issues, we introduce DynBLTL, a new logic tailored to express both structural and behavioral properties in dynamic software architectures. Furthermore, we propose using statistical model checking (SMC) to support an efficient analysis of these properties by evaluating the probability of meeting them through a number of simulations. In this paper, we describe the main features of DynBLTL and how it was implemented as a plug-in for PLASMA, a statistical model checker.

1 Introduction

Dynamic software architectures are those that encompass evolution rules for a software system and its elements during runtime [20]. Their relevance is due to the fact that dynamism is an important concern for contemporary systems, which often operate on environments subjected to change. In a dynamic software architecture, constituent elements may be created, interconnected or removed, or may even undergo a complete rearrangement at runtime, ideally with minimal or no disruption. For this reason, supporting dynamism is important mainly in the case of certain safety-and mission-critical systems, such as air traffic control, energy, disaster management, environmental monitoring, and health systems. Systems in these scenarios are required to maintain a high level of availability,

so that stopping and restarting them is not an option due to financial costs, physical damages, or even threats to the life and safety of people.

One of the major challenges in the design of software-intensive systems consists in verifying the correctness of their software architectures, i.e., if the envisioned architecture is able to fully realize the established requirements [26]. Ensuring correctness and other relevant system properties becomes more important mainly for evolving systems since such a verification needs to be performed before, during, and after evolution. The requirements to be verified are typically concerned with the relationship between the system behavior (e.g., a particular value is received or sent) and an architectural property, such as checking if a component is connected to or disconnected from another component. For illustrative purposes, consider a sensor-based system in which sensors measure a given value from the environment and transmit it to a base station, possibly via other sensors. A requirement of interest in this context would be that a sensor signaling its failure (a behavioral property) gets disconnected from the other sensors (an architectural property).

The automated analysis of architectural properties can be performed by means of formal verification, which determines if a software system satisfies properties capturing the system requirements. However, such a process is challenging in the context of dynamic systems. As the number of components to appear and be connected to the system is unbounded a priori, its state space is likely to be infinite. Therefore, exhaustive methods that explore the whole state space are unfeasible for dynamic software architectures unless the number of components that will be part of the system is known in advance. Nonetheless, the state space might still be quite large and hence the use of such techniques may be a prohibitive choice in terms of execution time and computational resources. This is the case of model checking [8], a formal verification technique that is among the most frequently used ones in the analysis of software architectures [30].

To face the state space explosion observed in traditional verification techniques, we propose the use of statistical model checking (SMC) to support the formal verification of architectural properties in dynamic systems. 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 [18]. This technique requires a stochastic executable model for the system, in which the choice of the next action to execute is done according to a probability distribution. With a stochastic system, the property to verify might be satisfied by some executions and not satisfied by some others. SMC then executes a number of stochastic simulations of the system and evaluates the approximated probability of the system to meet the property under verification. Requiring the execution to be probabilistic is not a limitation because dynamic systems can be reasonably described by assigning probabilities for new components to appear or for the existing components to fail and be disconnected, for example. Moreover, probability distributions can be used to model input values.

Besides an executable probabilistic model of the system, SMC requires a language for expressing properties to be verified and a monitor for deciding

them on finite traces, which is obtained by bounding temporal operators [24]. The particular nature of dynamic software architectures is that architectural elements (components and connectors) may appear, disappear, be connected or be disconnected at runtime. Therefore, expressing behavioral and structural properties regarding a dynamic software architecture needs to take into account architectural elements that are dynamically created and removed, i.e., they may exist at a given instant in time and no longer exist at another.

To cope with these characteristics, this paper brings as main contribution DynBLTL, a new logic aimed to express properties in dynamic software architectures using bounded temporal operators. DynBLTL is designed to handle the absence of an architectural element in a given formula expressing a property by means of the *undefined* value ($\mathfrak{U}$), which is returned when reading values from components that are no longer in the system. We have implemented DynBLTL as a plug-in for PLASMA [1, 13], a flexible, modular statistical model checker.

This paper is organized as follows. In Section 2, we provide an overview of how SMC works. Section 3 presents how to formalize a trace of a dynamic system. Section 4 defines DynBLTL and describes its semantics for execution traces. In Section 5, we describe how DynBLTL was implemented atop the PLASMA statistical model checker. Section 6 discusses related work. Finally, Section 7 contains some concluding remarks.

2 Statistical Model Checking: An Overview

SMC provides a number of advantages in comparison to traditional formal verification techniques such as model checking. First, SMC does not suffer from the exponential growth of the state space (the so-called *state space explosion problem*) as it does not build the entire representation of the state space, thus making it a promising approach for verifying complex large-scale and critical software systems [15]. Second, SMC can be applied as soon as a simulator of the system to verify is available. Third, the proliferation of parallel computer architectures allows producing multiple independent simulations, thereby speeding up the verification of large-scale systems even though it is still necessary to make the simulation procedure as efficient as possible [18]. Fourth, despite the results of SMC are approximations, the technique is more scalable and consumes less computation resources. In some cases, obtaining quickly an approximation of the result is more valuable than obtaining the exact result after a long computation [19]. As the verification accuracy parameterizes the analysis, the user can set the trade-off between verification speed and accuracy.

A statistical model checker basically consists of a simulator for running the system under verification, a checker for verifying properties on a trace, 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 may satisfy φ and others may not satisfy it, depending on the probabilistic choices made during these executions. We denote by p the probability that a trace satisfy φ. The goal of a statistical model checker is to approximate p. The simulator produces traces that are analyzed by the checker. For each trace, the result of the checker (i.e. whether the trace satisfies φ or not) is recorded. Based on the precision parameters and the results obtained so far, the statistical analyzer determines when enough traces have been seen to produce an accurate enough approximation of p. A more accurate approximation requires more traces.

SMC answers two types of questions. The first one is qualitative: *Is the probability p for M to satisfy φ greater or equal than a certain threshold θ?* The second question is quantitative: *What is the probability p for M to satisfy φ?* [27]. In both cases, producing a trace σ_i and checking if it satisfies φ (i.e., $\sigma_i \models \varphi$) is modeled as a random variable B_i following a Bernoulli distribution of parameter p [17]. The possible values of B_i are either 0 (if $\sigma_i \not\models \varphi$) or 1 (if $\sigma_i \models \varphi$), with probability functions $Pr[B_i = 1] = p$ and $Pr[B_i = 0] = 1 - p$. The variable B_i is associated with the i-th simulation of M.

Qualitative approach. The main existing SMC approaches for the qualitative question [28, 29] rely on *hypothesis testing* as means of inferring if the simulated execution traces provide statistical evidence on the satisfaction or violation of a property [25]. To determine if $p \geq \theta$, two hypotheses can be considered, namely (i) $H : p \geq \theta$ and (ii) $K : p < \theta$. The test is parameterized by two bounds, α and β. The probability of accepting the hypothesis K when the hypothesis H holds is bounded by α, and the probability of accepting H when K holds is bounded by β. Such algorithms sequentially perform simulations until either H or K can be returned with confidence of α or β. Other sequential hypothesis testing algorithms are based on the Bayesian approach [14].

Quantitative approach. In order to compute the probability p for M to satisfy φ, Hérault et al. [11] and Laplante et al. [16] propose an estimation procedure based on the Chernoff-Hoeffding bound [12], which provides the minimum number of simulations required to ensure the desired confidence level. Given a precision ε, this procedure computes an estimate p' of p with confidence δ, thereby ensuring $Pr(|p' - p| \leq \varepsilon) \geq 1 - \delta$.

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

3 Representing Traces of Dynamic Systems

Typical operations performed on dynamic software architectures comprise creating, removing, attaching, and detaching components and connectors. In order

to express architectural properties, we have to represent the set of components and their interconnections. Furthermore, we need to capture the behavior of the system by observing the messages exchanged between elements of the architecture.

We define a *state* of a dynamic software architecture as a directed graph $g = (V, E)$ comprising a finite set of nodes V and a finite set of edges E. Each node $v \in V$ represents an architectural element (component or connector) of the system. In turn, each edge $e \in E$ represents a communication channel between two architectural elements and is labeled by the value, if any, exchanged between the connected nodes. We represent the set of all possible values by *Val*, which contains the *undefined* value $\mathfrak{U}$ to represent the absence of value.

Definition 1 (State). *A* state *of a dynamic system is a directed graph* $g = (V, E)$ *where:*

- *Each node* $v \in V$ *is defined by a tuple* (id, T, C) *in which* id *is a globally unique identifier for the architectural element,* T *is the declared type of the architectural element, and* C *is a finite set representing its connections.* $id(v)$ *returns the identifier for node,* $T(v)$ *returns its type,* $C(v)$ *denotes the set of connections of the node* v*, and* $v.c$ *denotes a connection* $c \in C(v)$*.*
- *Each edge* $e \in E$ *connecting two nodes in the graph is labeled by the values exchanged between them. These values are contained into Val, the set of all possible values that can be exchanged between two nodes. Formally,* $E \subseteq \{(v_1.c_1, x, v_2.c_2) \mid x \in Val \wedge \bigwedge_{i=1}^{2} v_i \in V \wedge c_i \in C(v_i)\}$*. For each connection, the set of edges connected to it contains at most one edge labeled by a value different of* $\mathfrak{U}$*.*

Given a state graph g, $V(g)$ and $E(g)$, respectively, denote its sets of nodes and of edges. Note that we do not forbid edges between connections of the same node, because they may be allowed in the language describing dynamic architectures.

The SMC technique relies on checking multiple *execution traces* resulting from simulations of the system under verification against the specified properties. A simulation ω results in a trace σ composed of a finite sequence of state graphs.

Definition 2 (Trace). *An untimed trace* σ^{ut} *is a sequence* $g_0, g_1, \ldots, g_n$ *of states. In turn, a timed trace* σ *is a sequence* $((t_0, g_0), \ldots, (t_n, g_n))$ *of states with timestamps* t_i*, such that* $\forall i : t_i \in \mathbb{R} \wedge t_i \leq t_{i+1}$*.*

SMC allows verifying systems that are *stochastic processes*. For this reason, traces have to be produced by a stochastic process, i.e., each state in the trace is the restriction of a complete system state and the choice of next complete state is governed by a probability distribution. For verifying timed systems, we require that for any value $M \in \mathbb{R}$, the system eventually produces a state (t_i, g_i) with $t_i > M$ for some i. In other words, we require that the time converges towards $+\infty$ during the execution of the system.

As an example, consider a simple client-server architecture that dynamically adapts to the demand. In such a system, clients may appear and interact with

a server by sending requests and receiving answers. We assume that each server can handle a limited number of clients (two in our example). If all servers have reached that limit and a new client appears, the systems spawns a new server to handle the new client. Whenever the client has completed its interaction with the server, it disconnects and disappears from the system. If a server has no client left, it is shutdown and disappears from the system. At last, if the overall utilization of the servers is low, one tries to shutdown some servers in order to save energy. This is done by reallocating clients so that some severs become unused.

Fig. 1 presents an execution trace of the client-server system. Initially, only one server is present in the system and a server has four connections ($r1$, $r2$, $a1$ and $a2$). At $t = 5$, three new clients appear and two of them are directly connected to the server. At $t = 6$, a new server is spawned and is connected to the third client, while the two first clients send their requests (requests and answers are represented as numbers). At $t = 7$, the client C_2 receives the answer to its request while the client C_3 sends a request to server S_2. At $t = 9$, the client C_3 receives the answer to its request and the client C_2 has disappeared. At $t = 10$, the client C_3 is relocated to server S_1 and the server S_2 is removed.

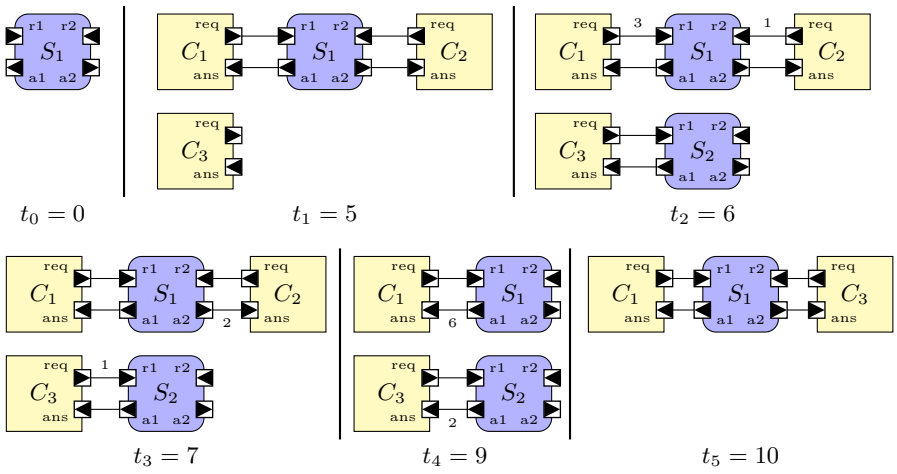

Fig. 1. An execution trace of the client-server example.

4 Expressing Properties about Dynamic Systems

Zhang et al. [30] report that linear temporal logic (LTL) [24] has been often used in the literature as underlying formalism for specifying temporal architectural properties and verifying them through model checking. LTL extends classical Boolean logic with *temporal operators* (a.k.a. modalities) that allow reasoning on the temporal dimension of the execution of the system. In this perspective,

LTL expresses properties about the future of the execution (sequences of states), e.g., a condition that will be eventually true, a condition that will be true until another fact becomes true, etc.

As SMC relies on simulation, it verifies bounded properties, i.e., properties that can be decided on a finite execution of the system under verification. While LTL-based formulas aim at specifying the infinite behavior of the system, a time-bounded form of LTL called BLTL [14] considers finite sequences of execution states of the system. The bounds are specified on the temporal operators, such as for instance the *always* operator. In LTL, this operator states that a property must be verified at every step of a (potentially infinite) trace. In BLTL, it has a bound and states that the property must hold until the bound is reached.

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. Furthermore, we may want to verify that the new components respect a particular behavior. BLTL is unable to handle this characteristic since it would require to statically know the set of components that will appear and write a dedicated formula for each of them. To tackle such a limitation, we introduce DynBLTL, a logic for expressing linear temporal properties over traces of dynamic systems. DynBLTL can express the required behavior of new components by having quantifiers over the set of existing components. In order to specify a behavior for the quantified components, DynBLTL allows interleaving quantifiers and temporal operators. Note that these quantifiers are not quantifiers in computation tree logic (CTL), but quantifiers over a finite set. In DynBLTL, all temporal operators are bounded, thereby making properties decidable on finite traces.

DynBLTL is designed to handle the absence of an architectural element in a given formula expressing a property. In practice, a Boolean expression can take three values, namely *true*, *false* or *undefined* ($\mathfrak{U}$). The additional undefined value refers to the fact that an expression may not be evaluated at a given execution state depending on the current runtime configuration of the system. This is necessary for situations in which it is not possible to evaluate an expression at the considered state, e.g., a statement about an architectural element that does not exist at that moment. As an example, the expression `c1.req > 3.2` cannot be evaluated if the component `c1` does not exist (as at t_0 in Fig. 1) or the connection `c1.req` is not involved in a communication at that state (as at t_1 in Fig. 1).

Fig. 2 shows the concrete syntax of DynBLTL by using the Extended Backus-Naur Form (EBNF). DynBLTL is not typed, so that a property can be evaluated to any type, i.e., Boolean, integer, string or undefined. As SMC requires a Boolean value as the result of the evaluation of a property on a trace, we add a syntactical constraint on properties to enforce that the returned value is Boolean. The `until` or `isTrue` operators always return a Boolean value. Consequently, we require that the root operator of a property is either `until`, `isTrue` or a Boolean combination of them.

```
node → ID
connection → ID . ID
function → ID ( (value(,value)*)+ )
value → value OP value | − value | connection | function | node | LITERAL
predicate → value CMP value | value
bound → FLOAT time units | INT steps
property →
    exists ID : function property     | count ID : function property
  | in bound property                 | property until bound property
  | not property                      | property or property
  | isTrue property
  | predicate
```

Fig. 2. Concrete textual syntax of DynBLTL. `ID` is an identifier, `OP` is an arithmetic operator, `LITERAL` is a Boolean, float, integer or string literal, `CMP` is a comparison operator, `FLOAT` is a float-pointing number, and `INT` is an integer number.

The semantics of a property φ is a function $[\![\varphi]\!]$ that takes a trace σ as argument and returns a value in *Val*. We define the semantics for a timed trace $\sigma = (t_0, g_0), \ldots, (t_n, g_n)$. If the system is untimed, we can only evaluate temporal operators whose bound is expressed in steps. Assume that φ is a property in which all temporal operators bounds are expressed in steps. Evaluating an untimed trace $\sigma^{ut} = g_0, \ldots, g_n$ falls back to evaluating a timed trace with the same states and arbitrary timestamps. Indeed, timestamps are only relevant for temporal operators whose bound is expressed in time units.

Section 4.1 describes the main elements of DynBLTL whereas Section 4.2 shows some examples on how to express architectural properties in dynamic systems using our logic.

4.1 DynBLTL Elements

A property can be specified by a formula containing *literals*, *identifiers* referring to nodes and connections in the state graph, *operations* (arithmetic, logical, comparison), predefined *functions*, *quantified expressions*, and *temporal operators*. These elements are briefly described in the following.

Literals and identifiers. As basic elements, a formula expressing a property can contain (i) a literal, which can be a Boolean value, numerical value or a string, (ii) an identifier representing a node of the state graph, or (iii) a connection of a node of the state graph. The evaluation of these literals only takes into account the first state of the trace, as follows:

- if φ is a literal l, then $[\![\varphi]\!](\sigma) = l$, i.e., the formula is evaluated to the respective value of l;
- if φ is an identifier *idt* representing a node, then $[\![\varphi]\!]((t_0, g_0), \ldots, (t_n, g_n))$ = `true` if there exists a node with that name at the current state, i.e. if $\exists v \in V(g_0)\ id(v) = idt$; otherwise, it evaluates to $\mathfrak{U}$;

– if φ is a connection c of a node v of a state graph ($v.c$), then $[\![\varphi]\!]((t_0, g_0), \ldots, (t_n, g_n))$ is evaluated to the only non-undefined value labeling any edge of g_0 attached to the connection $v.c$, or to $\mathfrak{U}$ otherwise.

Operations and comparisons. Arithmetic operations as well as inequalities and equalities are evaluated as usual or set to $\mathfrak{U}$ if at least one argument is out of their definition domain. DynBLTL supports the usual arithmetic operators (`+`,−,∗,`/`), and the usual comparisons (`<`,`<=`,`>`,`>=`,`=`,`!=`). Note that both $\mathfrak{U}$`!=`$\mathfrak{U}$ and $\mathfrak{U}$`=`$\mathfrak{U}$ evaluates to $\mathfrak{U}$.

Usual Boolean operators are also supported. The `not` operator acts as usual on Boolean values and returns $\mathfrak{U}$ with other values. The `or` operator returns `true` if at least one of the operands evaluates to true, `false` if both operands evaluate to false, and $\mathfrak{U}$ otherwise. Note that it may return true even if one of the operands is $\mathfrak{U}$. Other usual Boolean operators are obtained as follows: φ_1 `and` $\varphi_2 \stackrel{\text{def}}{=}$ `not` (`not` φ_1 `or` `not` φ_2) and φ_1 `implies` $\varphi_2 \stackrel{\text{def}}{=}$ `not` φ_1 `or` φ_2.

Functions. DynBLTL provides four predefined functions that can be used to explore the architectural configuration, i.e., the nodes of a state graph:

– `allOfType(`T`)` returns a collection with all nodes of type T;
– `areConnected(`v_1`,`v_2`)` returns true if nodes v_1 and v_2 are connected by an edge in the state graph, false if v_1 and v_2 exist in the state graph, but they are not connected by an edge, or $\mathfrak{U}$ otherwise;
– `areLinked(`$v_1.c_1$`,`$v_2.c_2$`)` returns true if the connection c_1 of node v_1 and the connection c_2 of node v_2 are connected by an edge in the state graph, false if both $v_1.c_1$ and $v_2.c_2$ exist in the state graph, but they are not connected by an edge, or $\mathfrak{U}$ otherwise; and
– `lastValue(`$v.c$`)` returns the last non-undefined value of the connection c of node v or $\mathfrak{U}$ if its value was always undefined.

Quantified expressions. In DynBLTL, three types of quantified expressions can be used to specify formulas expressing properties, namely the existential and universal quantified expressions traditionally used in predicate logic, as well as expressions involving an additional quantifier for counting elements upon the satisfaction of a predicate. These quantified expressions comprise an identifier r, a function f that returns a collection of elements, and a formula φ with free occurrences of r. In the sequel we assume that $[\![f]\!](\sigma) = e = \{e_1, \ldots, e_n\}$ and we denote by $\varphi[r \leftarrow e_i]$ the formula φ where each free occurrence of r is replaced by e_i. Quantifiers are defined as follows ($[\![\cdot]\!](\sigma)$ is omitted for readability):

– `exists` r`:` $f\varphi$ returns `true` if $\varphi[r \leftarrow e_i]$ evaluates to true for at least one element e_i ($1 \leq i \leq n$) or to `false` if $\varphi[r \leftarrow e_i]$ evaluates to false for all elements e_i, or $\mathfrak{U}$ otherwise.
– `forall` r`:` $f\varphi$ returns `true` if $\varphi[r \leftarrow e_i]$ evaluates to `true` for all elements e_i ($1 \leq i \leq n$) or to `false` if $\varphi[r \leftarrow e_i]$ evaluates to false for at least one element e_i, or $\mathfrak{U}$ otherwise.
– `count` r`:` $f\varphi$ returns the number of elements $e_i \in e$ such that $\varphi[r \leftarrow e_i]$ evaluates to `true`.

Temporal operators. Similarly to traditional BLTL, DynBLTL provides four temporal operators, namely `in`, `until`, `eventually before`, and `always during`. These operators are parametrized by a bound expressed either in `steps` or in `time units`. We provide here their definition:

- The `in` operator (a.k.a. *next*) evaluates its argument at a later point specified by the bound. If the bound is expressed in steps, we translate the trace by that number of steps:

$$[\![\texttt{in } b \texttt{ steps } \varphi]\!]((t_0, g_0), \ldots, (t_n, g_n)) = [\![\varphi]\!]((t_b, g_b), \ldots, (t_n, g_n))$$

  We assume that n is always bigger than b. In practice, it falls back to asking the simulator to perform more steps and complete the trace. If the bound is expressed in terms of time units, we translate the trace by the amount of time units provided as argument:

$$[\![\texttt{in } b \texttt{ time units } \varphi]\!]((t_0, g_0), \ldots, (t_n, g_n)) = [\![\varphi]\!]((t_k, g_k), \ldots, (t_n, g_n))$$

  where $k = min(\{0 \leq i \leq n \mid t_i - t_0 > b\})$.
- The `until` operator returns a Boolean value. An `until` expression evaluates to `true` if its right argument is evaluated to true within the bound and if the left argument evaluates to true or to $\mathfrak{U}$ until the right argument becomes true. We introduce new notations: $\sigma \models \varphi \equiv [\![\varphi]\!](\sigma) = \texttt{true}$ and $\sigma \not\models \varphi \equiv [\![\varphi]\!](\sigma) = \texttt{false}$[1]. If the bound is expressed in steps, we have:

$$\begin{gathered}((t_0, g_0), \ldots, (t_n, g_n)) \models \varphi_1 \texttt{ until } b \texttt{ steps } \varphi_2 \text{ iff} \\ \exists 0 \leq i \leq b \;.\; ((t_i, g_i), \ldots, (t_n, g_n)) \models \varphi_2 \wedge \\ \forall 0 \leq j < i . \neg((t_j, g_j), \ldots, (t_n, g_n)) \not\models \varphi_1\end{gathered}$$

  If the bound is expressed in time units, we have:

$$\begin{gathered}((t_0, g_0), \ldots, (t_n, g_n)) \models \varphi_1 \texttt{ until } b \texttt{ time units } \varphi_2 \text{ iff} \\ \exists 0 \leq i \leq n \;.\; (t_i - t_0 \leq b) \;\wedge ((t_i, g_i), \ldots, (t_n, g_n)) \models \varphi_2 \wedge \\ \forall 0 \leq j < i \;.\; \neg((t_j, g_j), \ldots, (t_n, g_n)) \not\models \varphi_1\end{gathered}$$

- The `eventually before` operator can be defined by reusing the previous definition of the `until` operator as:

$$\texttt{eventually before } b \; \varphi \stackrel{\text{def}}{=} \texttt{true until } b \; \varphi$$

- The `always during` operator can be defined by reusing the previous definition of the `eventually before` operator as:

$$\texttt{always during } b \; \varphi \stackrel{\text{def}}{=} \texttt{not eventually before } b$$

The reader may have noticed that we treat the value $\mathfrak{U}$ in a particular way when defining the `until` operator. Indeed, when $\mathfrak{U}$ appears on the left side of `until`, it is treated as `true`. However, when it appears on the right side, it is

[1] Note that if φ does not evaluate to a Boolean, then neither $\sigma \models \varphi$ nor $\sigma \not\models \varphi$ holds.

treated as `false`. We made this choice for the sake of intuitiveness. For instance, the property `c1.req < 2 until 10 steps c2.req = 5` can return true, even if `c1.req < 2` evaluates to $\mathfrak{U}$ during the 10 steps. We consider that evaluating to $\mathfrak{U}$ on the left hand side of `until` does not invalidate the formula. However, if `c1.req < 2` evaluates to false before `c2.req` evaluates to 5, then the whole expression evaluates to false.

The `isTrue` operator enforces the evaluation of a property to a Boolean value. Formally, $[\![\texttt{isTrue}\varphi]\!](\sigma) = \sigma \models \varphi$. This operator can be used to modify the behavior of `until`: `(isTrue c2.req < 2) until 10 steps c2.req = 5` will evaluate to false if `c2.req` evaluates to $\mathfrak{U}$ before `c2.req` evaluates to 5. We also define its dual operator `isNotFalse` $\varphi \stackrel{\text{def}}{=}$ `not isTrue not` φ.

4.2 Examples

Consider again the client-server example from Section 3. It is possible to express some interesting properties about such an architecture. For instance, we can express the fact that each request is treated in less than three time units:

```
always during 100 time units {
  forall c:allOfType(Client) {
    c.req > 0 implies eventually before 3 time units c.ans > 0
  }
}
```

As previously mentioned, the bound of 100 time steps on the `always during` operator is needed to ensure that the property can be decided on a finite trace. Therefore this property checks only the 100 first time units of the trace.

We can also express properties about the reconfiguration process. For instance, we can require that no client remains disconnected for more than five time units.

```
always during 100 time units {
  forall c:allOfType(Client) {
    not always during 5 time units {
      not exists s:allOfType(Server) areConnected(c,s)
    }
  }
}
```

By interleaving the `forall` quantifier between temporal operators, we require that each client meets a given property.

At last, we require that the reconfiguration effectively reduces the number of unused servers. More precisely, assuming a limit of two clients per server, if the number of servers is more than half the number of clients, then a reconfiguration is needed. We allow five time units for that reconfiguration:

```
not eventually before 100 time units {
  always during 5 time units {
    (count c:allOfType(Client) true) <
```

```
    2 * (count s:allOfType(Server) true) - 1
  }
}
```

5 Implementation

We have implemented DynBLTL as a plug-in for the PLASMA statistical model checker [1, 13].[2] We have also provided a simulator plug-in that interfaces with an external simulator used to produce the traces. The simulator plug-in receives events about the architecture, such as (i) when a node v appears, (ii) when connection c_1 of node v_1 is linked to connection c_2 of node v_2, (iii) when a value x is sent from connection c_1 of node v_1 to connection c_2 of node v_2, or (iv) when a node v disappears. From this information, the simulator plug-in maintains a trace of the current execution as a sequence of states from Definition 1. If these events have a timestamp, the produced trace is also timed. The DynBLTL plug-in asks the simulator plug-in about the particular states of the trace in order to evaluate the property.

Currently, we support simulations of architectural descriptions in π-ADL [23], a formal architecture description language for specifying both structure and behavior of dynamic software architectures. Fig. 3 shows an overview of our SMC-based toolchain for verifying architectural properties. An architecture description in π-ADL is first translated to source code in the Go programming language [5, 4]. As π-ADL is non-deterministic, we enforce stochastic behavior by providing a probabilistic choice function that randomly chooses the next action to execute among the possible ones. Furthermore, some functions can be declared unobservable in π-ADL and implemented directly in Go. Such functions can rely on probability distributions to model inputs. Our methodology is explained in [6].

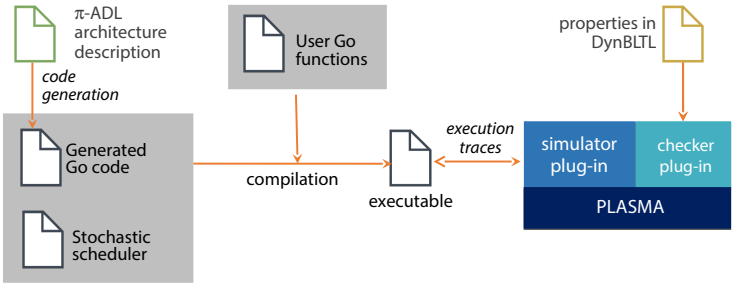

Fig. 3. Overview of our toolchain for verifying properties expressed in DynBLTL from π-ADL architecture descriptions.

Once the probabilistic choice function and the implementation of unobservable functions are provided, the obtained Go code is compiled into an executable

[2] The developed plug-in is available at `http://plasma4pi-adl.gforge.inria.fr/`.

and run by the simulator plug-in. Whenever a new trace is required for the analysis, the simulator plug-in launches a new instance of the executable to generate a new trace. If the trace is not long enough to evaluate the property, the simulator plug-in requests more simulation steps from the executable.

6 Related Work

The idea of interleaving quantifiers and temporal logics is not new and has been used in LTL(MSO), for example [2]. In that model, the number of constituents is constant throughout the execution and therefore this logic is not applicable to dynamic systems.

The Bandera specification language allows model checking multi-threaded Java programs [9]. The dynamicity is handled by bounding the number of classes that can be dynamically created to be able to statically build a representation of the state space, but such an approach requires the user to annotate the Java code. Cho et al. [7] also proposed a logic for dealing with dynamic systems based on freeze quantifiers. In both cases, the logic cannot express architectural properties. The π-AAL language [21] was developed to express properties about π-ADL models, but its semantics is not suitable for performing SMC since properties are evaluated per trace, not per computation tree.

An important part of the verification of dynamic systems deals with validation of reconfiguration operations. In this context, several works have provided ways to specify what a correct reconfiguration means. In the work of Mazzara and Bhattacharyya [22], several frameworks for describing and analyzing dynamic reconfiguration are studied, but they do not handle logics similar to DynBLTL. Basso et al. [3] express architectural properties in CTL with additional predicates encoding the state of the architecture, but this logic does not allow interleaving quantifiers (over sets) and temporal operators. Finally, Dormoy et al. [10] propose a logic where architectural properties are used as predicate and expressed through quantifiers, but quantifiers and temporal operators cannot be interleaved.

7 Conclusion

In this paper, we have presented DynBLTL, a new logic tailored for the statistical model checking of dynamic software architectures. DynBLTL was implemented as a plug-in for the PLASMA statistical model checker, thus benefiting from all SMC algorithms already implemented. The developed toolchain is currently able to verify properties associated to architectural descriptions in the π-ADL language.

As future work, we first plan to improve the developed tools, especially the performance of the monitor. We also intend to identify which SMC algorithm gives the best results for verifying properties of dynamic architectures. Finally, we are interested in verifying properties for systems-of-systems and look forward to use DynBLTL in this context.

Acknowledgments. This work was partially supported by the Brazilian National Agency of Petroleum, Natural Gas and Biofuels through the PRH-22/ANP/MCTI Program (for Everton Cavalcante) and by CNPq under grant 308725/2013-1 (for Thais Batista).

References

1. PLASMA-Lab. `https://project.inria.fr/plasma-lab/`
2. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J., Saksena, M.: Regular model checking for LTL(MSO). International Journal on Software Tools for Technology Transfer 14(2), 223–241 (Apr 2012)
3. Basso, A., Bolotov, A., Basukoski, A., Getov, V., Henrio, L., Urbanski, M.: Specification and verification of reconfiguration protocols in grid component systems. In: Proceedings of the 3rd IEEE Conference on Intelligent Systems (2006)
4. 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)
5. 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)
6. Cavalcante, E., Quilbeuf, J., Traonouez, L.M., Oquendo, F., Batista, T., Legay, A.: Statistical model checking of dynamic software architectures. In: Proceedings of the 10th European Conference on Software Architecture. Springer International Publishing (2016)
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. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Robby: Expressing checkable properties of dynamic systems: The Bandera specification language. International Journal on Software Tools for Technology Transfer 4(1), 34–56 (Oct 2002)
10. Dormoy, J., Kouchnarenko, O., Lanoix, A.: Using temporal logic for dynamic reconfigurations of components. In: Barbosa, L.S., Lumpe, M. (eds.) Proceedings of the 7th International Workshop on Formal Aspects of Component Software, Lecture Notes in Computer Science, vol. 6921, pp. 200–217. Springer Berlin Heidelberg, Germany (2010)
11. 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)
12. Hoeffding, W.: Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association 58(301), 13–30 (Mar 1963)
13. 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)
14. Jha, S.K., Clarke, E.M., Langmead, C.J., Legay, A., Platzer, A., Zuliani, P.: A Bayesian approach to model checking biological systems. In: Degano, P., Gorrieri, R. (eds.) Proceedings of 7th International Conference Computational Methods in Systems Biology, Lecture Notes in Computer Science, vol. 5688, pp. 218–234. Springer Berlin Heidelberg, Germany (2009)
15. 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)
16. Laplante, S., Lassaigne, R., Magniez, F., Peyronnet, S., de Rougemont, M.: Probabilistic abstraction for model checking: An approach based on property testing. ACM Transactions on Computational Logic 8(4) (Aug 2007)
17. Lefebvre, M.: Applied Probability and Statistics. Springer New York, USA (2006)
18. 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)
19. Legay, A., Viswanathan, M.: Statistical model checking: Challenges and perspectives. International Journal on Software Tools for Technology Transfer 17(4), 369–376 (Aug 2015)
20. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering. pp. 3–14. ACM, New York, NY, USA (1996)
21. 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)
22. Mazzara, M., Bhattacharyya, A.: On modelling and analysis of dynamic reconfiguration of dependable real-time systems. In: Proceedings of the Third International Conference on Dependability. pp. 173–181 (2010)
23. 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)
24. 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)
25. Sen, K., Viswanathan, M., Agha, G.: Statistical model checking of black-box probabilistic systems. In: Alur, R., Peled, D.A. (eds.) Proceedings of the 16th International Conference on Computer Aided Verification, Lecture Notes in Computer Science, vol. 3114, pp. 202–215. Springer Berlin Heidelberg, Germany (2004)
26. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, theory, and practice. John Wiley & Sons, Inc., USA (2010)
27. Younes, H.L.S., Kwiatkowska, M., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking. International Journal on Software Tools for Technology Transfer 8(3), 216–228 (Jun 2006)
28. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: Brinksma, E., Larsen, K.G. (eds.) Proceedings of the 14th International Conference on Computer Aided Verification, Lecture Notes in Computer Science, vol. 2404, pp. 223–235. Springer Berlin Heidelberg, Germany (2002)

29. Younes, H.L.S.: Verification and planning for stochastic processes with asynchronous events. Doctoral dissertation, Carnegie Mellon University, Doctoral dissertation (2004)
30. 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)