JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2016; **00**:1–22
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

Formal Verification of Probabilistic SystemC Models with Statistical Model Checking

Van Chan Ngo[1]*Axel Legay[2]

[1]*Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA*
[2]*Inria Rennes - Bretagne Atlantique, Rennes, 35042, France*

SUMMARY

Transaction-level modeling with SystemC has been very successful in describing the behavior of embedded systems by providing high-level executable models, in which many of them have inherent probabilistic behaviors, i.e., random data, unreliable components. It thus is crucial to have both quantitative and qualitative analysis of the probabilities of system properties.
Such analysis can be conducted by constructing a formal model of the system under verification and using *Probabilistic Model Checking* (PMC). However, this method is infeasible for real-life systems which are large and complex, due to the state space explosion. In this article, we demonstrate the successful use of *Statistical Model Checking* (SMC) to carry out such analysis directly from large SystemC models and allow designers to express a wide range of useful properties.
The first contribution of this work is a framework to verify properties in *Bounded Linear Temporal Logic* (BLTL) for SystemC models with both timed and probabilistic characteristics. The framework contains two main components: a *monitor* observing a set of execution traces of the model-under-verification (MUV) and a *statistical model checker* implementing a set of statistic analysis algorithms.
The second contribution is a method that allows users to expose a rich set of user-code primitives in form of atomic propositions in BLTL. These propositions help users exposing the state of the SystemC simulation kernel and the full state of a SystemC source code model. In addition, users can define their own fine-grained time resolution that is used to reason on the semantics of the logic expressing the properties rather the boundary of clock cycles in the SystemC simulation.
The third contribution, we implemented a tool in which the properties of interest are expressed using BLTL. The various features of the tool including automatic generation of monitor for generating execution traces of the MUV, the mechanism to instrument automatically the MUV, and the interaction with statistical model checking algorithms are presented. We demonstrate our approach through a running example, as well as the performance of the tool through some experiments. Moreover, we also implemented a probabilistic scheduler to make our verification perform on possible execution orders of the MUV than a fixed and deterministic one in the current SystemC scheduler implementation. Copyright © 2016 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Runtime Verification; Probabilistic Temporal Assertion; Statistical Model Checking; Program Verification; SystemC Models

1. INTRODUCTION

Transaction-level modeling (TLM) with SystemC has been become increasingly prominent in describing the behavior of embedded systems [7], i.e., System-on-Chips (SoCs). It allows complex

*Correspondence to: Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA.
E-mail: *channgo@cmu.edu*
†Work partly done while the author was at Inria Rennes - Bretagne Atlantique, Rennes, France.

electronic components and software control units can be combined into a single model, enabling simulation of the whole system at once. In many cases, models include probabilistic and non-deterministic characteristics, i.e, random data, reliability of the system's components. Hence, it is crucial to evaluate the quantitative and qualitative analysis of the probabilities of system properties.

1.1. Motivation

We consider a safety-critical system, i.e., a control system for air-traffic, automotive, or medical device. The reliability and availability of the system can be modeled as a stochastic process, in which it exhibits both timed and probabilistic characteristics. For instance, the reliability and availability model of an embedded control system [21] that contains an input processor connected to groups of sensors, an output processor connected to groups of actuators, and a main processor that communicates with the I/O processors through a bus. Suppose that the sensors, actuators, and processors can be failed and the I/O processors have transient and permanent faults. When a transient fault occurs in a processor, rebooting the processor repairs the fault. The times to failure and reboot's delay of processors are exponentially distributed. Thus, the reliability of the system cam be modeled by a *Continuous-Time Markov Chain* (CTMC) [40, 12] (a special case of a discrete-state *stochastic process* in which the probability distribution of the next state depends only on the current state). The analysis can be quantifying the probabilities or rates of all safety-related faults: How likely the system is available to meet a demand for service? What is the probability that the system repairs after a failure (e.g., the system conforms to the existent and prominent standards such as the *Safety Integrity Levels* (SILs))?

In order to conduct such analysis, a general approach is modeling and analyzing a probabilistic model of the system (i.e., Markov chains, stochastic processes), in which the algorithm for computing the measures in properties depends on the class of systems being considered and the logic used for specifying the property. Many algorithms with the corresponding mature tools are based on model checking techniques that compute the probability by a numerical approach [5, 8, 34, 16]. Timed automata with mature verification tools such as UPPAAL [22] are used to verify real-time systems. For a variety of probabilistic systems, the most popular modeling formalism is Markov chain or Markov decision processes, for which *Probabilistic Model Checking* (PMC) tools such as PRISM [17] and MRMC [20] can be used. It is widely used and has been successfully applied to the verification of a range of timed and probabilistic systems. One of the main challenges is the complexity of the algorithms in terms of execution time and memory space due to the size of the state space that tends to grow exponentially, also known as the state space explosion. As a result, the analysis is infeasible. In addition, these tools cannot work directly with the SystemC source code, meaning that a formal model of SystemC model needs to be provided.

An alternative way to evaluate these systems is *Statistical Model Checking* (SMC), a simulation-based approach. Simulation-based approaches produce an approximation of the value to be evaluated, based on a finite set of system's executions. Clearly, comparing to the numerical approach, a simulation-based solution does not provide an exact answer. However, users can tune the statistical parameters such as the absolute error and the level of confidence, according to the requirements. Simulation-based approaches do not construct all the reachable states of the model-under-verification (MUV), thus they require far less execution time and memory space than numerical approaches. For some real-life systems, they are the only one option [43] and have shown the advantages over other methods such as PMC [16, 19].

1.2. Contribution

In this article, we demonstrate the successful use of SMC to carry out directly qualitative and quantitative analysis for probabilistic temporal properties from large SystemC models and allows designers to express a wide range of useful properties. The work makes the following contributions.

We proposed a framework to verify bounded temporal properties for SystemC models with both timed and probabilistic characteristics. The framework contains two main components: a *monitor* observing a set of execution traces of the MUV and a *statistical model checker* implementing a set of hypothesis testing algorithms. We use the similar techniques proposed by Tabakov et al. [37] to

DOI: 10.1002/smr

automatically generate the monitor. The statistical model checker is implemented as a plugin of the checker Plasma Lab [3], in which the properties to be verified are expressed in *Bounded Linear Temporal Logic* (BLTL).

We presented a method that allows users to expose a rich set of user-code primitives in form of atomic propositions in BLTL. These propositions help users exposing the state of the SystemC simulation kernel and the full state of the SystemC source code model. In addition, users can define their own fine-grained time resolution that is used to reason on the semantics of the logic expressing the properties rather the boundary of clock cycles in the SystemC simulation.

We implemented a tool, in which the properties of interest are expressed using BLTL. The various features of the tool including automatic generation of monitor for generating execution traces of the MUV, mechanism to instrument automatically the MUV, and the interaction with statistical model checking algorithms are presented. We demonstrate our approach through a running example, as well as the performance of the framework through some experiments.

To make our verification perform on possible execution orders of the MUV, we implemented a probabilistic scheduler. Specifically, the source of process scheduling is the evaluation phase in the current SystemC scheduler implementation, in which one of the runnable processes is executed. Given a set of N runnable processes in the queue every time at the evaluation phase, our probabilistic scheduler randomly chose one of these processes to execute. The random algorithm is implemented by generating a random integer number uniformly over a range $[0, N-1]$.

2. BACKGROUND

This section introduces the SystemC modeling language and reviews the main features of statistical model checking for stochastic processes as well as bounded linear temporal logic which is used to express system properties.

2.1. SystemC Language

2.1.1. Language Features SystemC[†] is a C++ library [13] providing primitives for modeling hardware and software systems at the level of transactions. Every SystemC model can be compiled with a standard C++ compiler to produce an executable program called executable specification. This specification is used to simulate the system behavior with the provided event-driven simulator. A SystemC model is hierarchical composition of modules (sc_module). Modules are building blocks of SystemC design, they are like modules in Verilog [39], classes in C++. A module consists of an interface for communicating with other modules and a set of processes running concurrently to describe the functionality of the module. An interface contains ports (sc_port) that are similar to the hardware pins. Modules are interconnected using either primitive channels (i.e., the signals, sc_signal) or hierarchical channels via their ports. Channels are data containers that generate events in the simulation kernel whenever the contained data changes.

Processes are not hierarchical, so no process can call another process directly. A process is either a thread or a method. A thread process (sc_thread) can suspend its execution by calling the library statement wait or any of its variants. When the execution is resumed, it will continue from that point. Threads run only once during the execution of the program an are not expected to terminate. On the other hand, a method process (sc_method) cannot suspend its execution by calling wait and is expected to terminate. Thus, it only returns the control to the kernel when reaching the end of its body.

An event is an instance of the SystemC event class (sc_event) whose occurrence triggers or resumes the execution of a process. All processes which are suspended by waiting for an event are resumed when this event occurs, we say that the event is notified. A module's process can be sensitive to a list of events. For example, a process may suspend itself and wait for a value change

[†]IEEE Standard 1666-2005

DOI: 10.1002/smr

of a specific signal. Then, only this event occurrence can resume the execution of the process. In general, a process can wait for an event, a combination of events, or for an amount time to be resumed.

In SystemC, integer values are used as discrete time model. The smallest quantum of time that can be represented is called *time resolution* meaning that any time value smaller than the time resolution will be rounded off. The available time resolutions are femtosecond, picosecond, nanosecond, microsecond, millisecond, and second. SystemC provides functions to set time resolution and declare a time object.

The example in Listing 1 describes a simple delay flip-flop, in which sc_in and sc_out are predefined primitive input and output ports. The sensitivity list contains the edge sensitivity sensitive_pos specified on clk, which indicates that only on the rising edge of clk does din gets transferred to dout. The sensitivity list and process instantiation are define in the class constructor (sc_ctor).

Listing 1 An Implementation of Flip-Flop in SystemC

```
#include "systemc.h"

SC_MODULE(d_ff) {
  /* input ports */
  sc_in<bool> din;
  sc_in<bool> clk;
  /* output port */
  sc_out<bool> dout;

  /* module's functionality */
  void flip() {
    dout = din;
  }

  /* constructor */
  SC_CTOR(d_ff) {
    SC_METHOD(flip);        // method process
    sensitive_pos << clk;   // edge sensitivity
  }
};
```

2.1.2. Simulation Kernel The SystemC simulator is an event-driven simulation [1, 28]. It establishes a hierarchical network of finite number of parallel communicating processes which are under the supervision of the distinguished simulation kernel process. Only one process is dispatched by the scheduler to run at a time point, and the scheduler is non-preemptive, that is, the running process returns control to the kernel only when it finishes executing or explicitly suspends itself by calling wait. Like hardware modeling languages, the SystemC scheduler supports the notion of delta-cycles [24]. A delta-cycle lasts for an infinitesimal amount of time and is used to impose a partial order of simultaneous actions which interprets zero-delay semantics. Thus, the simulation time is not advanced when the scheduler processes a delta-cycle. During a delta-cycle, the scheduler executes actions in two phases: the *evaluate* and the *update* phases. The semantics of the scheduler is given in Listing 2. Each phase of the scheduler is explained as follows.

Listing 2 Semantics of SystemC kernel

```
PC // all primitive channels
P  // all processes
R = ∅ // set of runnable processes
D = ∅ // set of pending delta notifications
U = ∅ // set of update requests
T = ∅ // set of pending timed notifications

/* start elaboration: collect all update requests in U */
for all chan ∈ U do
  run chan.update()
end for
```


DOI: 10.1002/smr

```
for all p ∈ P do
  if p is initialized and p is not clocked thread then
    R = R ∪ p // make p runnable
  end if
end for
for all p ∈ P do
  if p is triggered by an event in D then
    R = R ∪ p
  end if
end for // end of initialization phase

repeat
  while R ≠ ∅ do // new delta−cycle begins
    for all r ∈ R do // evaluation phase
      R = R \ r
      run r until it calls wait or returns
    end for
    for all chan ∈ U do // update phase
      run chan.update()
    end for
    for all p ∈ P do // delta notification phase
      if p is triggered by an event in D then
        R = R ∪ p // make p runnable
      end if
    end for // end of delta−cycle
  end while

  if T ≠ ∅ then
    Advance the simulation clock to the earliest timed delay t
    T = T \ t
    for all p ∈ P do // timed notification phase
      if t triggers p then
        R = R ∪ p // make p runnable
      end if
    end for
  end if
until end of simulation
```

- *Initialize*. During the initialization, each process is executed once unless it is turned off by calling dont_initialize() , or until a synchronization point (i.e., a wait) is reached. The order in which these processes are executed is unspecified.
- *Evaluate*. The kernel starts a delta-cycle and run all processes that are ready to run one at a time. In this same phase a process can be made ready to run by an event notification.
- *Update*. Execute any pending calls to update() resulting from calls to request_update() in the evaluate phase. Note that a primitive channel uses request_update() to have the kernel call its update() function after the execution of processes.
- *Delta-cycle Notification*. The kernel enters the delta notification phase where notified events trigger their dependent processes. Note that immediate notifications may make new processes runable during step (2). If so the kernel loops back to step (2) and starts another evaluation phase and a new delta-cycle. It does not advance simulation time.
- *Simulation-cycle Notification*. If there are no more runnable processes, the kernel advances simulation time to the earliest pending timed notification. All processes sensitive to this event are triggered and the kernel loops back to step (2) and starts a new delta-cycle. This process is finished when all processes have terminated or the specified simulation time is passed.

2.2. *Bounded Linear Temporal Logic*

We here recall the syntax and semantics of BLTL [35], an extension of *Linear Temporal Logic* (LTL) with time bounds on temporal operators. A formula φ is defined over a set of atomic propositions AP as in LTL. A BLTL formula is defined by the following BNF grammar, in which true and false are Boolean constants. The time bound T is an amount of time or a number of states in the execution trace.

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

DOI: 10.1002/smr

The temporal modalities F (the "eventually", sometimes in the future) and G (the "always", from now on forever) can be derived from the "until" U as follows.

$$\mathsf{F}_{\leq \mathsf{T}}\ \varphi = \mathsf{true}\ \mathsf{U}_{\leq \mathsf{T}}\ \varphi \text{ and } \mathsf{G}_{\leq \mathsf{T}}\ \varphi = \neg\ \mathsf{F}_{\leq \mathsf{T}}\ \neg\varphi$$

The semantics of BLTL is defined with respect to execution traces of a model $\mathcal{M}$. Let $\omega = (s_0, t_0)(s_1, t_1)...(s_{N-1}, t_{N-1})$, where $N \in \mathbb{N}$, be a sequence of pairs states and time points, where the model stays in the state s_i for $t_{i+1} - t_i = \delta t_i \in \mathbb{R}_{\geq 0}$ duration of time. The sequence ω is called an execution trace of $\mathcal{M}$. We use ω_k and ω^k to denote the prefix and suffix of ω respectively. We denote the fact that ω satisfies the BLTL formula φ by $\omega \models \varphi$. The semantics is given as follows.

- $\omega^k \models \mathsf{true}$ and $\omega^k \not\models \mathsf{false}$
- $\omega^k \models p, p \in \mathsf{AP}$ iff $p \in L(s_k)$, where $L(s_k)$ is the set of atomic propositions which are true in state s_k
- $\omega^k \models \varphi_1 \wedge \varphi_2$ iff $\omega^k \models \varphi_1$ and $\omega^k \models \varphi_2$
- $\omega^k \models \neg\varphi$ iff $\omega^k \not\models \varphi$
- $\omega^k \models \varphi_1\ \mathsf{U}_{\leq \mathsf{T}}\ \varphi_2$ iff there exists $i \in \mathbb{N}$ such that $\omega^{k+i} \models \varphi_2$, $\Sigma_{0<j\leq i}(t_{k+j} - t_{k+j-1}) \leq \mathsf{T}$, and for each $0 \leq j < i, \omega^{k+j} \models \varphi_1$

Here is a simple example of temporal property expressed in BLTL that can be verified with SMC:

$$\varphi = \mathsf{G}_{\leq \mathsf{T}_1}\,(\mathsf{A} \rightarrow \mathsf{F}_{\leq \mathsf{T}_2}\,(\mathsf{B}\ \mathsf{U}_{\leq \mathsf{T}_3}\ \mathsf{C}))$$

The meaning of $\mathsf{Pr}\,(\varphi)$ is: What is the probability that during the T_1 time units of the system operation, if A holds then, starting from T_2 time units after, B happens before C within T_3 time units.

2.3. *Statistical Model Checking*

Let $\mathcal{M}$ be the formal model of the MUV (i.e., a stochastic process, a CTMC) and φ be a property expressed as a BLTL formula. BLTL ensures that the satisfaction of a formula by a trace can be decided in a finite number of steps. The statistical model checking [23] problem consists in answering the following questions:

- *Qualitative Analysis*. Is the probability that $\mathcal{M}$ satisfies φ greater or equal to a threshold θ with a specific level of statistical confidence?
- *Quantitative Analysis*. What is the probability that $\mathcal{M}$ satisfies φ with a specific level of statistical confidence?

They are denoted by $\mathcal{M} \models \mathsf{Pr}\,(\varphi)$ and $\mathcal{M} \models \mathsf{Pr}_{\geq\theta}(\varphi)$, respectively. Many statistical model checkers have been implemented [3, 42, 16, 19] that have shown their advantages over other methods such as PMC on several case studies.

This is done by associating each execution trace of $\mathcal{M}$ with a discrete random Bernoulli variable B_i, in which the outcome for B_i, denoted by b_i, is 1 if the trace satisfies φ and 0 otherwise.

The predominant statistical method for verifying $\mathcal{M} \models \mathsf{Pr}_{\geq\theta}(\varphi)$ is based on *hypothesis testing*. Let $p = \mathsf{Pr}\,(\varphi)$, to determine whether $p \geq \theta$, we test the hypothesis $\mathsf{H}_0 : p \geq p_0 = \theta + \delta$ against the alternative hypothesis $\mathsf{H}_1 : p \leq p_1 = \theta - \delta$ based on the observations of B_i. The size of *indifference region* is defined by $p_0 - p_1 = 2\delta$. If we take acceptance of H_0 to mean acceptance of $\mathsf{Pr}_{\geq\theta}(\varphi)$ as true and acceptance of H_1 to mean rejection of $\mathsf{Pr}_{\geq\theta}(\varphi)$ as false, then we can use *acceptance sampling* (e.g., Younes in [41] has proposed two solutions, called *single sampling plan* and *sequential probability ratio test*) to verify $\mathsf{Pr}_{\geq\theta}(\varphi)$. An acceptance sampling test with *strength* (α, β) guarantees that H_1 is accepted with probability at most α when H_0 holds and H_0 is accepted with probability at most β when H_1 holds, called a Type-I error and Type-II error, respectively.

To answer the quantitative question, $\mathcal{M} \models \mathsf{Pr}\,(\varphi)$, an alternative statistical method, based on *estimation* instead of hypothesis testing, has been developed. For instance, the probability estimations are based on results derived by Chernoff and Hoeffding bounds [18]. This approach uses n observations $b_1, ..., b_n$ to compute an approximation of p: $\tilde{p} = \frac{1}{n}\Sigma_{i=1}^{n} b_i$. The approximation

DOI: 10.1002/smr

satisifies that $\mathsf{Pr}[|\tilde{p} - p| < \delta] \geq 1 - \alpha$. Based on the theorem of Hoeffding, the number of observations which is determined from the absolute error δ and the confidence $1 - \alpha$ is $n = \lceil \frac{1}{2\delta^2} log \frac{2}{\alpha} \rceil$.

Although SMC can only provide approximate results with a user-specified level of statistical confidence, it is compensated for by its better scalability and resource consumption. Since the models to be analyzed are often approximately known, an approximate result in the analysis of desired properties within specific bounds is quite acceptable. SMC has recently been applied in a wide range of research areas including software engineering [16] (e.g., verification of critical embedded systems), system biology, or medical area [19].

3. A RUNNING EXAMPLE

We will use a simple case study with a *First-In First-Out* (FIFO) channel as a running example (see Fig. 1 with the graphical notations in [13]). This example illustrates how designers can create hierarchical channels that encapsulate both design structure and communication protocols. In the design, once a nanosecond the producer will write one character to the FIFO with probability p_1, while the consumer will read one character from the FIFO with probability p_2. The FIFO which is derived from sc_channel encapsulates the communication protocol between the producer and the consumer.

Figure 1. Every 1 nanosecond, the producer writes 1 character to the FIFO with probability p_1, while the consumer reads 1 character with probability p_2. The FIFO channel is designed to ensure that all data is reliably delivered despite the varying rates of production and consumption

The FIFO channel is designed to ensure that all data is reliably delivered despite the varying rates of production and consumption. The channel uses an event notification hanshake protocol for both the input and output. It uses a circular buffer implemented within a static array to store and retrieve the items within the FIFO. We assume that the sizes of the messages and the FIFO buffer are fixed. Hence, it is obvious that the time required to transfer completely a message, or message *latency*, depends on the production and consumption rates, the FIFO buffer size, the message size, and the probabilities of successful writing and reading. The full implementation of the example can be obtained at the website of our tool [33, 31], in which the probabilities of writing and reading are implemented with the Bernoulli distributions with probabilities p_1 and p_2 respectively from *GNU Scientific Library* (GSL) [14].

The quantitative analysis under consideration is: What is the probability that each single message is transferred completely (e.g., including the message delimiters) within T_1 nanoseconds during T nanoseconds of operation? This kind of analysis can, thus, be conducted in the early design steps. To formulate the underlying property more precisely, we have to take into account the agreement protocol between the producer and consumer, i.e., a simple protocol can be every message has special starting delimiter with the character '&' and ending delimiter with the character '@'. Thus, the property can be translated in BLTL as follows:

$$\varphi = \mathsf{G}_{\leq \mathsf{T}}((\mathsf{c\_read} = '\&') \rightarrow \mathsf{F}_{\leq \mathsf{T}_1}(\mathsf{c\_read} = '@'))$$

where c_read is the character read in the FIFO by the consumer. The input providing to the SMC checker is $\mathsf{Pr}(\varphi)$. This property is expressed in terms of the characters read in the FIFO by the consumer, but the communication protocal between the producer and the consumer is abstracted at a very high level. It is an illustration of a type of properties that can be checked on TLM specifications. The verification of such a property at the transaction level can be connected to its counterpart at *Register-Transfer Level* (RTL) in order to check the correctness of RTL implementations.

DOI: 10.1002/smr

4. SMC FOR SYSTEMC MODELS

In order to apply SMC for SystemC models which exhibit timed and probabilistic characteristics, this section presents the concepts of state and execution trace for SystemC models. It also shows that the operational semantics of this class of SystemC models can be considered as stochastic processes.

4.1. SystemC Model State

Temporal logic formulas are interpreted over execution traces and traditionally a trace has been defined as a sequence of states in the execution of a model. Thus before we can define an execution trace we need a precise definition of the state of a SystemC model simulation. We are inspired by the definition of system state in [37], which consists of the state of the simulation kernel and the state of the SystemC model user-code. We consider the external libraries as black boxes, meaning that their states are not exposed.

Let Sp be the set of simulation kernel phases, Ev be the set of all events, and X be the set of all variables in the model that includes all module attributes, ports, and channels. Let F be the set of all functions in the model, a configuration of the call stack is represented by a tuple (f, f_p, f_r) where $f \in F$ is the executing function, f_p is its parameters, and f_r is its return values. Let Cs be the set of all call stack configurations. Let Loc be the set of all locations, i.e., a specific statement reached during the execution and $Proc$ is the set of the status of all module processes in the model. Then a state is formally represented by a tuple $(sp, ev, \sigma, l, cs, proc)$ where:

- $sp \in Sp$ is the current phase of the simulation kernel, i.e., delta-cycle notification, simulation-cycle simulation,
- $ev \in 2^{Ev}$ is the set of events that are notified during the execution of the model,
- $\sigma : X \to \mathbb{D}_X$ is a map from variables to its domain values,
- $l \in Loc$ is the current location,
- $cs \in Cs$ is the current configuration of the call stack,
- $proc \in Proc$ is the current status of the module processes, i.e., suspended or runnable.

We consider here some examples about states of the simulation kernel and the SystemC model. Assume that a SystemC model has an event named `e`, then the model state can contain information such as the kernel is at the end of simulation-cycle notification phase and the event `e` is notified. Consider the running example again, a state can consist of the information about the characters received by the consumer, represented by the variable with type char, `c_read`.

It also contains the information about the location of the program counter right before and after a call of the function `send()` in the module `Producer` that are represented by two Boolean variables `send_start` and `send_done`, respectively, meaning that they hold the value `true` immediately before and after a call of the function `send()`. Another example, we consider a module that consists several statements at different locations in the source code, in which these statements contain the division operator "/" followed by zero or more spaces and the variable "`a`" (e.g., the statement `y = (x + 1)/a`). Then, a Boolean variable which holds the value `true` right before the execution of all such statements can be used as a part of the states.

Let S be the (finite or infinite) set of all states of the model. We use $V = \{v_0, ..., v_{n-1}\}$ to denote the finite set of variables of primitive type (e.g, usual scalar or enumerated type in C/C++) whose value domain $\mathbb{D}_V = \mathbb{D}_{v_0} \times \cdots \times \mathbb{D}_{v_{n-1}} \subseteq S$ represents the states of a SystemC model.

We have discussed so far the state of a SystemC model execution. It remains to discuss how the semantics of the temporal operators is interpreted over the states in the execution of the model. That means how the states are sampled in order to make the transition from one state to another state. The following definition gives the concept of *temporal resolution*, in which the states are evaluated only at instances in which the temporal resolution holds. It allows the user to set granularity of time.

Definition 1 (Temporal Resolution)
A temporal resolution $\mathcal{T}_r$ is a disjunction of a finite set of Boolean expressions defined over V. It specifies when the set of variables V is evaluated to generate a new state.

DOI: 10.1002/smr

Temporal resolution can be used to define a more fine-grained model of time than a coarse-grained one provided by a cycle-based simulation. We call the expressions in $\mathcal{T}_r$ *temporal events*. Whenever a temporal event is satisfied, or we say that the temporal event occurs, V is sampled.

For example, in the producer and consumer model, assume that we want the satisfaction of the underlying BLTL formula φ to be checked whenever at either the end of simulation-cycle notification or immediately after the event write_event is notified during a run of the model. Hence, we can define a temporal resolution as $\mathcal{T}_r = \{$ end_sc , we_notified $\}$, where end_sc and we_notified are Boolean expressions that have the value true whenever the kernel phase is at the end of the simulation-cycle notification and the event write_event notified, respectively.

We denote the set of occurrences of temporal events from $\mathcal{T}_r$ along an execution of a SystemC model by $\mathcal{T}_r^s$, called a *temporal resolution set*. The value of a variable $v \in V$ at an event occurrence $e_c \in \mathcal{T}_r^s$ is defined by a mapping $\xi_{val}^v : \mathcal{T}_r^s \to \mathbb{D}_v$, where $\mathbb{D}_v$ is the value domain of v.

A mapping $\xi_t : \mathcal{T}_r^s \to \mathcal{T}$ is called a *time event* that identifies the simulation time at each occurrence of an event from the temporal resolution. Hence, the set of time points, called *time tag*, which corresponds to a temporal resolution set $\mathcal{T}_r^s = \{e_{c_0}, ..., e_{c_{N-1}}\}, N \in \mathbb{N}$, is given as follows.

Definition 2 (Time Tag)
Given a temporal resolution set $\mathcal{T}_r^s$, the *time tag* $\mathcal{T}$ corresponding to $\mathcal{T}_r^s$ is a finite or infinite set of non-negative reals $\{t_0, t_1, ..., t_{N-1}\}$, where $t_{i+1} - t_i = \delta t_i \in \mathbb{R}_{\geq 0}$ and $t_i = \xi_t(e_{c_i})$.

Therefore, the state of the SystemC model at an event occurrence $e_c \in \mathcal{T}_r^s$ (with the corresponding simulation time $t = \xi_t(e_c)$) is formally defined as follows.

Definition 3 (Model State)
Let $V = \{v_0, ..., v_{n-1}\}$ be a finite set of variables representing a SystemC model states and $\mathcal{T}_r$ be a temporal resolution. Then the state of the model within an execution at the simulation time $t = \xi_t(e_c)$, written (s, t), is defined by a tuple $(\xi_{val}^{v_0}, ..., \xi_{val}^{v_{n-1}})$.

4.2. Model and Execution Trace

A SystemC model can be viewed as a hierarchical network of parallel communicating processes. Hence, the execution of a SystemC model is an alternation of the control between the model's processes, the external libraries and the kernel process. The execution of the processes is supervised by the kernel process to concurrently update new values for the signals and variables with respect to the cycle-based simulation. For example, given a set of runnable processes in a simulation-cycle, the kernel chooses one of them to execute first in a non-deterministic manner as described in the prior section.

Let V be the set of variables whose values represent the states of a SystemC model. The values of variables in V are samples of the set of states S. A state can be considered as a random variable following an unknown probability distribution defined from all probability distributions of random variables in the model. Given a temporal resolution $\mathcal{T}_r$ and its corresponding temporal resolution set along an execution of the model $\mathcal{T}_r^s = \{e_{c_0}, ..., e_{c_{N-1}}\}, N \in \mathbb{N}$, the evaluation of V at the event occurrence e_{c_i} is defined by the tuple $(\xi_{val}^{v_0}, ..., \xi_{val}^{v_{n-1}})$, or a state of the model at e_{c_i}, denoted by $V(e_{c_i}) = (V(e_{c_i})(v_0), V(e_{c_i})(v_1), ..., V(e_{c_i})(v_{n-1}))$, where $V(e_{c_i})(v_k) = \xi_{val}^{v_k}(e_{c_i})$ with $k = 0, ..., n-1$ is the value of the variable v_k at e_{c_i}.

We denote the set of all possible evaluations by $V_{\mathcal{T}_r^s} \subseteq \mathbb{D}_V$, called the *state space* of the random variables in V. State changes are observed only at the moments of event occurrences. Hence, the operational semantics of a SystemC model is represented by a *stochastic process* $\{(V(e_{c_i}), \xi_t(e_{c_i})), e_{c_i} \in \mathcal{T}_r^s\}_{i \in \mathbb{N}}$, taking values in $V_{\mathcal{T}_r^s} \times \mathbb{R}_{\geq 0}$ and indexed by the parameter e_{c_i}, which are event occurrences in the temporal resolution set $\mathcal{T}_r^s$. An execution trace is a realization of the stochastic process is given as follows.

Definition 4 (Execution Trace)
An execution trace of a SystemC model corresponding to a temporal resolution set $\mathcal{T}_r^s = \{e_{c_0}, ..., e_{c_{N-1}}\}, N \in \mathbb{N}$ is a sequence of states and event occurrence times, denoted by $\omega = (s_0, t_0)...(s_{N-1}, t_{N-1})$, such that for each $i \in 0, ..., N-1$, $s_i = V(e_{c_i})$ and $t_i = \xi_t(e_{c_i})$.

DOI: 10.1002/smr

The value N is called the length (finite or infinite) of the execution, also denoted by $|\omega|$. Given $V' \subseteq V$, the *projection* of ω on V', written $\omega \downarrow_{V'}$, is an execution trace such that $|\omega \downarrow_{V'}| = |\omega|$ and $\forall v \in V', \forall e_c \in \mathcal{T}_r^s, V'(e_c)(v) = V(e_c)(v)$.

4.3. Properties Expressing

Our framework accepts input properties of the forms $\mathsf{Pr}(\varphi)$, $\mathsf{Pr}_{\geq\theta}(\varphi)$, and $\mathsf{X}_{\leq\mathsf{T}}(rv)$, where φ is a BLTL formula. The former is used to compute the probability that φ satisfied by the model. The later asserts that this probability is at least equal to the threshold θ. The last one returns the mean value of random variable rv.

The set of atomic propositions AP in the logic which describes SystemC code features and the simulation semantics is a set of Boolean expressions defined over a subset of V called *observed variables* and the standard operators $(+, -, *, /, >, \geq, <, \leq, !=, =)$. The semantics of the temporal operators in BLTL formulas interpreted over states is defined by a temporal resolution. In other words, the temporal resolution determines how the states are sampled in order to make the transition from one state to another state. The observed variables and temporal resolutions supported by the framework are summarized below; see the tool manual[‡] for the full syntax and semantics. It is noticed that the current implementation only supports the use of simulation phases, locations, and events to define temporal resolutions. The implementation provides a mechanism that allows users to declare observed variables in order to define the set of propositions AP via a high-level language in a configuration file as the input of our tool.

- *Attribute*. Users can define an observed variable whose value and type are equal to the value and type of a module's attribute in the user code. Attributes can be public, protected, or private. For example, `attribute a.t a_t` defines a variable named `a_t` whose value and type are equal to the value and type of the private attribute `t` of the module instance `a`.
- *Function*. Let `f()` be a C++ function with k arguments in the user code. Users can refer to locations in the source code that contain the function call, immediately after the function call, immediately before the first executable statement, and immediately after the last executable statement in `f()` by using Boolean observed variables `f:call`, `f:return`, `f:entry`, and `f:exit`, respectively. Moreover, users can define an observed variable `f:i`, with $i = 0...k$, whose value and type are equal to the value and type of the return object (with $i = 0$) or i^{th} argument of function `f()` before executing the first statement in the function body. For example, if the function `int div(int x, int y)` is defined in the user code, then the formula $\mathsf{G}_{\leq\mathsf{T}}(\mathsf{div:entry} \rightarrow \mathsf{div:2} != 0)$ asserts that the divisor is nonzero whenever the `div` function starts execution.
- *Simulation Phase*. There are 18 predefined Boolean observed variables which refer to the 18 kernel states [33]. These variables are usually used to define a temporal resolution. For example, the formula $\mathsf{G}_{\leq\mathsf{T}}(p = 0)$ which is accompanied with the temporal resolution `MON_DELTA_CYCLE_END` requires the value of variable p to be zero at the end of every delta-cycle.
- *Event*. For each SystemC event e, the framework provides a Boolean observed variable `e.notified` that is true only when the simulation kernel actually notifies e. For example, the formula $\mathsf{G}_{\leq\mathsf{T}}(\mathsf{e.notified})$ which is accompanied with the temporal resolution `MON_UPDATE_PHASE_END` says that the event e is notified at the end of every update phase.

Referring to the running example, the declaration `location send_start "%Producer::send()":call` declares a Boolean variable `send_start` that holds the value `true` immediately before the execution of the model reaches a call site of the function `send()` in the module `Producer`. The characters received by the consumer which is represented by the variable `c_read` can be declared as `attribute pnt_con → c_int c_read`, where `pnt_con` is a pointer to the

[‡]http://project.inria.fr/pscv/en/documentation/

DOI: 10.1002/smr

Consumer object and c_int is an attribute of the Consumer module representing the received character. Then the considered property of the running example can be constructed by using the following predicates defined over the variable c_read: c_read $=' \&'$ and c_read $=' @'$.

Another example, assume that we want to answer the following question: *"Over a period of* T *time units, is the probability that the number of elements in the FIFO buffer is between n_1 and n_2 greater or equal to θ with the confidence α"*? The predicates need to be defined in order to construct the underlying BLTL formula are $n_1 \leq \mathsf{n_{elements}}$ and $\mathsf{n_{elements}} \leq n_2$, where $\mathsf{n_{elements}}$ is an integer variable that represents the current number of elements in the FIFO buffer (it capptures the value of the num_elements attribute in the Fifo module). Then, the property can be translated in BLTL with the operator "always" as follows. The input which is given to the checker is $\mathsf{Pr}_{\geq\theta}(\varphi)$ along with the confidence α.

$$\varphi = \mathsf{G}_{\leq \mathsf{T}}\left((n_1 \leq \mathsf{n_{elements}}) \,\&\, (\mathsf{n_{elements}} \leq n_2)\right)$$

5. IMPLEMENTATION

We have implemented a SMC-based verification tool [30, 31], PSCV, that contains two main original components: a *Monitor* and *Aspect-advice Generator* (MAG) and a *Statistical Model Checker* (SystemC Plugin). The tool whose architecture is depicted in Fig. 2 can be considered as a runtime verification tool for probabilistic temporal properties.

5.1. The Architecture

It consists of off-the-self, modified and original components:

- An off-the-self component, AspectC++ [11], a C++ *Aspect Oriented Programming* (AOP) compiler for instrumenting the MUV.
- A modified component, a patch SystemC-2.3.0 for facilitating the communication between the kernel and the monitor and implementing a random scheduler.
- Two original components are MAG, a C++ tool for automatically generating monitor and aspect-advices for instrumentation, and SystemC plugin, a plugin of the statistical model checker Plasma Lab [3].

5.1.1. Execution Trace Extraction In principle, the full state can be observed during the simulation of the model. In practice, however, users define a set of variables of interest, according to the properties that the users want to verify, and only these variables appear in the states of an execution trace. Given a SystemC model, we use $V_{\mathsf{obs}} \subseteq V$ to denote the set of variables, called *observed*

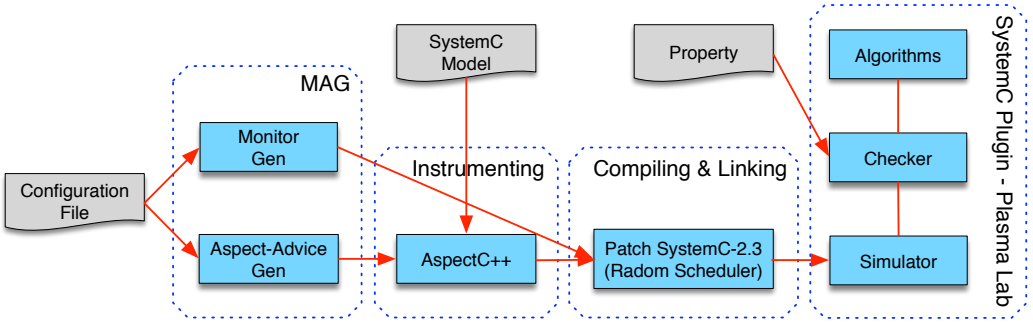

Figure 2. PSCV consists of off-the-self components (AspectC++ and Plasma Lab), modified component (patch SystemC-2.3.0), and original components (MAG and SystemC plugin)

variables, to expose the states of the SystemC model. Then, the observed execution traces of the model are the projections of the execution traces on V_{obs}, meaning that for every execution trace

DOI: 10.1002/smr

ω, the corresponding observed execution trace is $\omega \downarrow_{V_{obs}}$. In the following, when we mention about execution traces, we mean observed execution traces.

In PSCV, based on the techniques in [37], the set of observed variables and temporal resolution are converted into a C++ monitor class and a set of aspect-advices. MAG generates three files: aspect_definitions.ah containing a set of AspectC++ *aspect* definitions, monitor.h, and monitor.cc implementing a monitor as a C++ class and another class called local_observer that is responsible for invoking the callback functions. The callback functions will invoke the sampling function at the right time point during the MUV simulation.

The monitor has step(), a sampling function. It waits for a request from the SystemC Plugin. If the request is stopping the current simulation, it then terminates the MUV's execution. If the plugin requests a new state, then the current values of all observed variables and the simulation time are sent. The step() function is called at every time point defined by the temporal resolution. These time points can be kernel phases, event notifications, or locations in the MUV code control flow. In such cases, the patch SystemC kernel needs to communicate with the local_observer, i.e., when a delta-cycle ends, via a class called mon_observer to invoke the step() function of the monitor. In case of locations in the MUV's user-code, the advice code generated by MAG will call the callback function to invoke the step() function.

The aspect in AspectC++ is an extension of the class concept of C++ to collect advice code implementing a common crosscutting concern in a modular way. For example, to access all attributes of a module called A and the location immediately before the first executable statement of the function foo in A (defined in the configuration file with the syntax % A :: foo() : entry). MAG will generate the aspect definition in Listing 3. The full syntax and semantics of aspects can be found at the web-site of AspectC++ [2].

Listing 3 Example of generated AspectC++ aspect

```
aspect Automatic {
  /* declare the monitor as friend class for accessing the private data */
  pointcut reveal() = "A";

  advice reveal() : slice class {
    friend class monitor_A; // generated monitor is a friend class of A
  }; // class

  /* Instrumentation code for loc1="% A::foo()":entry */
  advice execution("% A::foo()"): before() {
    mon_observer* observer = local_observer::getInstance();
    monitor_A* mon = (monitor_A*) observer->get_monitor_by_index(0);
    mon->callback_userloc_loc1(); // invoke callback function
  } // advice code
}; // automatic aspect
```

5.1.2. Statistical Model Checker The statistical model checker is implemented as a plugin of Plasma Lab [3] that establishes a communication, in which the generated monitor transmits execution traces of the MUV. In the current version, the communication is done via the standard input and output. When a new state is requested, the monitor reports the current state (the values of observed variables and the current simulation time) to the plugin. The length of traces depends on the satisfaction of the formula to be verified, which is finite due to the bounded temporal operators.

Similarly, the required number of traces depends on the statistic algorithms in use (e.g., sequential hypothesis testing or 2-sided Chernoff bound). The full set of statistic algorithms implemented in Plasma Lab can be found at [32].

5.1.3. Random Scheduler Verification does not only depend on the probabilistic characteristics of the MUV, but it also can be significantly affected by *scheduling policy*. Consider a simple module A consisting of two thread processes as shown in Listing 4, where x is initialized to be 1.

Listing 4 Example of deterministic execution order

DOI: 10.1002/smr

```
/* thread t1 functionality */
void A::t1() {
  if (x <= 0)
    x := x + 1;
}

/* thread t1 functionality */
void A::t2() {
  if (x > 0)
    x := x - 1;
}

SC_CTOR(A) {
  /* Declare t1 and t2 as thread processes
   * SystemC kernel always execute t1 first and then t2
   */
  SC_THREAD(t1);
  SC_THREAD(t2);
}
```

Assume that we want to compute the probability that x is always equal to 1. Obviously, x depends on the execution order of two threads, i.e., its value is 1 if `t2` is executed before the execution of `t1` and 0 if the order is `t1` then `t2`.

The current scheduling policy is deterministic, in which it always picks the process that is first added into the queue (the current open-source SystemC kernel implementation uses a queue to store a set of runnable processes). Hence, only one execution order, `t1` then `t2`, is verified instead of two possible orders. As a result, the computed probability is 0, however, ideally it should be 0.5. Therefore, it is more interesting if a verification is performed on all possible execution orders than a fixed one. In many cases, there is no decision or an a priori knowledge of the scheduling to be implemented. Moreover, verification of a specification should be independent of the scheduling policy to be finally implemented.

To make our verification perform on possible execution orders of the MUV, we have implemented a random scheduler by patching the current open-source SystemC kernel implementation. The source of process scheduling is the evaluation phase, in which one of the runnable processes will be executed. Given a set of N runnable processes in the queue every time at the evaluation phase, our random scheduler randomly choses one of them to execute. The random algorithm is implemented by generating a random integer number uniformly over a range $[0, N-1]$. For more simulation efficiency and implementation simplicity, we employ the `rand()` function and $\%$ operator in C/C++.

5.2. *The Verification Flow*

The verification flow using PSCV consists of three steps, as shown in Fig. 3. In the first step,

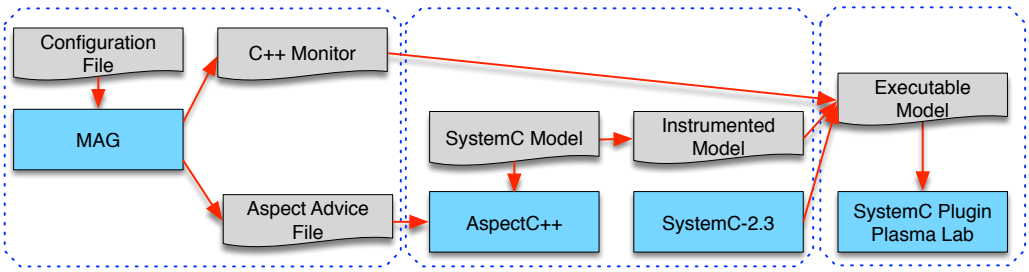

Figure 3. The verification flow consists of 3 steps: writing a configuration file, instrumenting the MUV using AspectC++, and verifying the instrumented MUV using the SystemC plugin

users write a configuration file containing a set of typed variables (*observed variables*), a Boolean expression (*temporal resolution*), and all properties to be verified. MAG translates the configuration file into a C++ monitor and a set of aspect-advices. The set of aspect-advices is then used by

DOI: 10.1002/smr

AspectC++ as input to automatically instrument the MUV to expose the user-codes state and simulation kernel's state in the second step. The instrumented model and the generated monitor are compiled and linked together with the modified SystemC kernel into an executable model.

Referring to the running example, users will define the set of observed variables $V_{\mathsf{obs}} = \{\, \mathsf{c\_read}\,, \mathsf{n_{elements}}\,\}$, where $\mathsf{c\_read}$ is the character read in the FIFO and $\mathsf{n_{elements}}$ is the number of characters in the FIFO buffer. The temporal resolution will be defined as $\mathcal{T}_r = \{\, \mathsf{MON\_TIMED\_NOTIFY\_PHASE\_END}\,\}$, where $\mathsf{MON\_TIMED\_NOTIFY\_PHASE\_END}$ is one of 18 predefined Boolean observed variables which refer to the 18 kernel states [33]. That means that a new state in execution traces is produced whenever the simulation kernel is at the end of simulation-cycle notification phase or every one nanosecond in the example since the time resolution is one nanosecond. The full configuration file is included in the source code of the example[§].

Finally, SystemC plugin, a plugin of the statistical model checker Plasma Lab [3], simulates independently the executable model in order to make the monitor produce execution traces with inputs provided by users. The inputs can be generated using any standard stimuli-generation technique. These traces are finite in length since the BLTL semantics [35] is defined with respect to finite execution traces. The number of simulations is determined by the statistical algorithm used by the plugin based on the absolute error and the level of confidence. Given these execution traces and the user-defined absolute error and confidence, the SystemC plugin employs the statistical model checking to produce an estimation of the probability that the property is satisfied or an assertion that this probability is at least equal to a threshold.

6. EXPERIMENTAL RESULTS

We report the experimental results for the running example and also demonstrate the use of our verification tool to analyze the dependability of a large embedded control system. The number of components in this system makes numerical approaches such as PMC unfeasible. In both case studies, we used the 2-sided Chernoff bound algorithm with the absolute error $\epsilon = 0.02$ and the confidence $\alpha = 0.98$. The experiments were run on a machine with Intel Core i7 2.67 GHz processor and 4GB RAM under the Linux OS with SystemC 2.3.0, in which the checking of the properties in the running example took from less than one minute to several minutes. The analysis of the embedded and control system case study took almost 2 hours, in which 90 properties were verified.

6.1. Producer and Consumer

Let us go back to the running example in Section 3, recall that we want to compute the probability that the following property φ satisfies every 1 nanosecond, with the absolute error 0.02 and the level of confidence 0.98. In this verification, both the FIFO buffer size and message size are 10 characters including the starting and ending delimiters, and the production and consumption rates are 1 nanosecond.

$$\varphi = \mathsf{G}_{\leq \mathsf{T}}\,((\,\mathsf{c\_read}\; =\; '\&')\rightarrow \mathsf{F}_{\leq \mathsf{T}_1}\,(\,\mathsf{c\_read}\; =\; '@'))$$

First, we compute $\mathsf{Pr}\,(\varphi)$ with various values of p_1 and p_2. The results are given in Table I with $\mathsf{T} = 5000$ and $\mathsf{T}_1 = 25$ nanoseconds. Obviously, the probability that the message latency is smaller than T_1 time units increases when p_1 and p_2 increase, where p_1 and p_2 are probabilities that the producer write to and consumer reads from the FIFO successfully . That means in general the latency is shorter when either the probability that the producer successfully writes to the FIFO increases, or the probability that the consumer successfully reads from the FIFO increases.

Second, we compute the probability that a message is sent completely (or the message latency) from the producer to the consumer within T_1 time units over a period of T time units of operation, in which the probabilities p_1 and p_2 are fixed at 0.9. Fig. 4 shows this probability with different

[§]http://project.inria.fr/pscv/producer-and-consumer/

DOI: 10.1002/smr

$p_1 \backslash p_2$	0.3	0.6	0.9
0.6	0	0.0194	0.0720
0.9	0	0.0835	1

Table I. The probability that the message latency is smaller than 25 in the first 5000 nanoseconds of operation

values of T_1 over $\mathsf{T} = 10000$ nanoseconds. It is observed that the message latency is almost smaller than 18 nanoseconds.

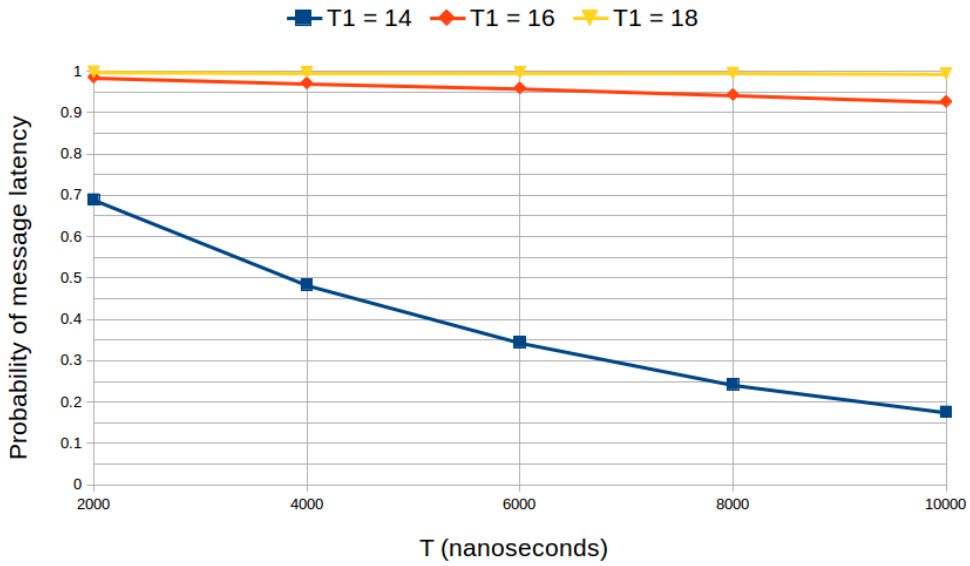

Figure 4. The probability that the message latency is smaller than T_1 in the first T nanoseconds of operation

6.2. *Embedded Control System*

This case study is closely based on the one presented in [29, 21] but contains many more components. The system consists of an input processor (I) connected to 50 groups of 3 sensors, an output processor (O), connected to 30 groups of 2 actuators, and a main processor (M), that communicates with I and O through a bus. At every cycle of 1 minute, the main processor polls data from the input processor that reads and processes data from the sensor groups. Based on this data, the main processor constructs commands to be passed to the output processor for controlling the actuator groups.

The reliability of the system is affected by the failures of the sensors, actuators, and processors. The probability of bus failure is negligible, hence we do not consider it. The sensors and actuators are used in $37-\text{of}-50$ and $27-\text{of}-30$ modular redundancies, respectively. That means if at least 37 sensor groups are functional (a sensor group is functional if at least 2 of the 3 sensors are functional), the system obtains enough information to function properly. Otherwise, the main processor is reported to shut the system down. In the same way, the system requires at least 27 functional actuator groups to function properly (a actuator group is functional if at least 1 of the 2 actuators is functional). Transient and permanent faults can occur in processors I or O and prevent the main processor(M) to read data from I or send commands to O. In that case, M skips the current cycle. If the number of continuously skipped cycles exceeds the limit K, the processor M shuts the system down. When a transient fault occurs in a processor, rebooting the processor

DOI: 10.1002/smr

repairs the fault. Lastly, if the main processor fails, the system is automatically shut down. The mean times to failure for the sensors, the actuators, and the processors are 1 month, 2 months and 1 year, respectively. The mean time to transient failure is 1 day and I/O processors take 30 seconds to reboot. In the model we represent 30 seconds as 1 time unit.

The reliability of the system is modeled as a CTMC [26, 40, 12] that is realized in SystemC, in which a sensor group has 4 states $(0, 1, 2, 3$, the number of working sensors), 3 states $(0, 1, 2$, the number of working actuators) for an actuator group, 2 states for the main processor (0: failure, 1: functional), and 3 states for I/O processors (0: failure, 1: transient failure, 2: functional). A state of the CTMC is represented as a tuple of the component's states, and the mean times to failure define the delay before which a transition between states is enabled. The delay is sampled from a negative exponential distribution with parameter equal to the corresponding mean time to failure. Hence, the model has about 2^{155} states comparing to the model in [21] with about 2^{10} states, that makes the PMC technique unfeasible. That means the state explosion likely occurs, even with some abstraction, i.e., symbolic model checking is applied. The full implementation of the SystemC code and experiments of this case study can be obtained at the website of our tool¶.

We define four types of failures: $\mathsf{failure}_1$ is the failure of the sensors, $\mathsf{failure}_2$ is the failure of the actuators, $\mathsf{failure}_3$ is the failure of the I/O processors and $\mathsf{failure}_4$ is the failure of the main processor. For example, $\mathsf{failure}_1$ is defined by $(\mathsf{number\_sensors} < 37) \wedge (\mathsf{proci\_status} = 2)$. It specifies that the number of working sensor groups has decreased below 37 and the input processor is functional, so that it can report the failure to the main processor. number_sensors and proci_status are observed variables that have same type and value as the number of working sensor groups and the current status of the input processor, respectively. We define $\mathsf{failure}_2$, $\mathsf{failure}_3$, and $\mathsf{failure}_4$ in a similar way.

In our analysis is based on the one in [21] with $\mathsf{K} = 4$ where the properties are checked every 1 time unit. First, we study the probability that each of the four types of failure eventually occurs in the first T time units of operation. This is done using the following BLTL formula $\mathsf{F}_{\leq \mathsf{T}}(\mathsf{failure_i})$. The following figure plots these probabilities over the first 30 days of operation. We observe that the probabilities that the sensors and I/O processors eventually fail are more than the probability that the other components do. In the long run, they are almost the same and approximate to 1, meaning that the sensors and I/O processors will eventually fail with probability 1. The main processor has the smallest probability to eventually fail.

Second, we try to determine which kind of component is more likely to cause the failure of the system, meaning that we determine the probability that a failure related to a given component occurs before any other failures. The atomic proposition $\mathsf{shutdown} = \bigvee_{i=1}^{4} \mathsf{failure_i}$ indicates that the system has shut down because one of the failures has occurred, and the BLTL formula $\neg\, \mathsf{shutdown}\ \mathsf{U}_{\leq \mathsf{T}}\ \mathsf{failure_i}$ states that the failure i occurs within T time units and no other failures have occurred before it. Fig. 6 shows the probability that each kind of failure occurs first over a period of 30 days of operation. It is obvious that the sensors are likelier to cause a system shutdown. At $\mathsf{T} = 20$ days, it seems that we reached a stationary distribution indicating for each kind of component the probability that it is responsible for the failure of the system.

For the third part of our analysis, we divide the states of system into three classes: "up", where every component is functional, "danger", where a failure has occurred but the system has not yet shut down (e.g., the I/O processors have just had a transient failure but they have rebooted in time), and "shutdown", where the system has shut down [21]. We aim to compute the expected time spent in each class of states by the system over a period of T time units. To this end, we add in the model, for each class of state c, a random variable reward_c that measures the time spent in the class c. In our tool, the formula $\mathsf{X}_{\leq \mathsf{T}}\ \mathsf{reward\_c}$ returns the mean value of reward_c after T time of execution. The results are plotted in Fig. 7. From $\mathsf{T} = 20$ days, it seems that the amounts of time spent in the "up" and "danger" states are converged at $10^{1.063} = 11.57$ days and $10^{-1.967} = 0.01$ days, respectively.

¶https://project.inria.fr/pscv/embedded-control-system/

DOI: 10.1002/smr

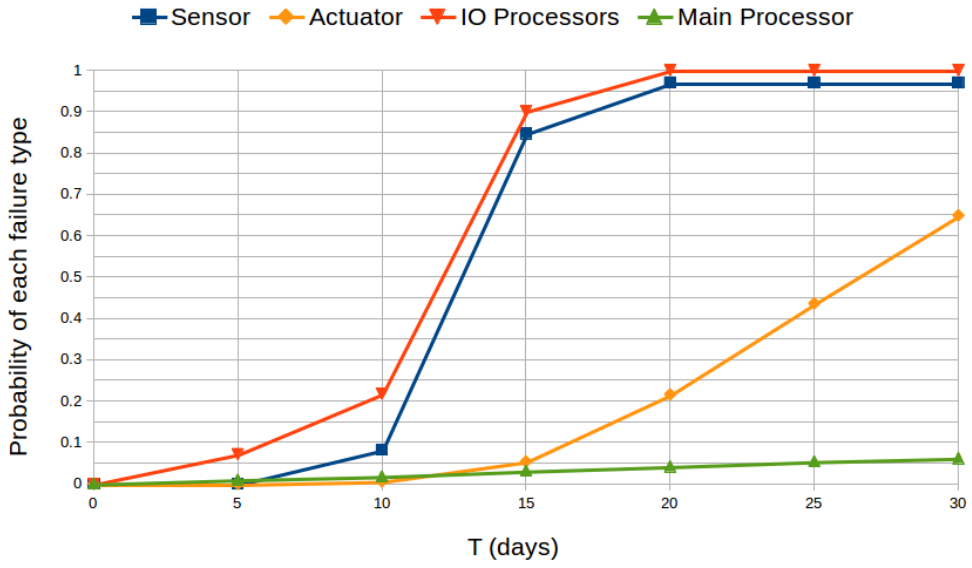

Figure 5. The probability that each of the 4 failure types is the cause of system shutdown in the first T time of operation

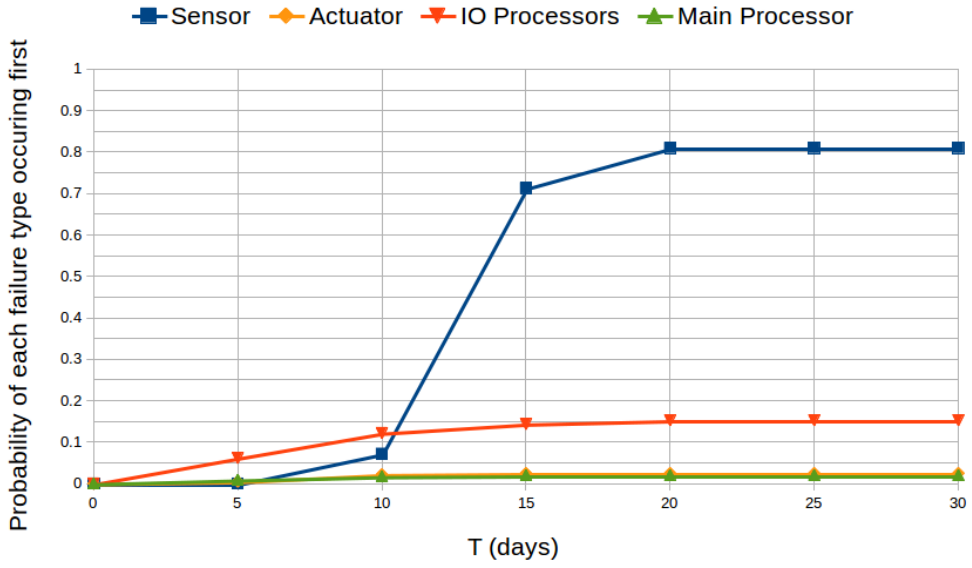

Figure 6. The probability that each of the 4 failure types is the cause of system shutdown in the first T time of operation

Finally, we approximate the number of reboots of the I/O processors, and the number sensor groups, actuator groups that are functional over time by computing the expected values of random variables that count the number of reboots, functional sensor and actuator groups. The results are plotted in Fig. 8 and Fig. 9. It is obvious that the number of reboots of both processors doubles the number of reboots of each processor since they have the same behavior model.

DOI: 10.1002/smr

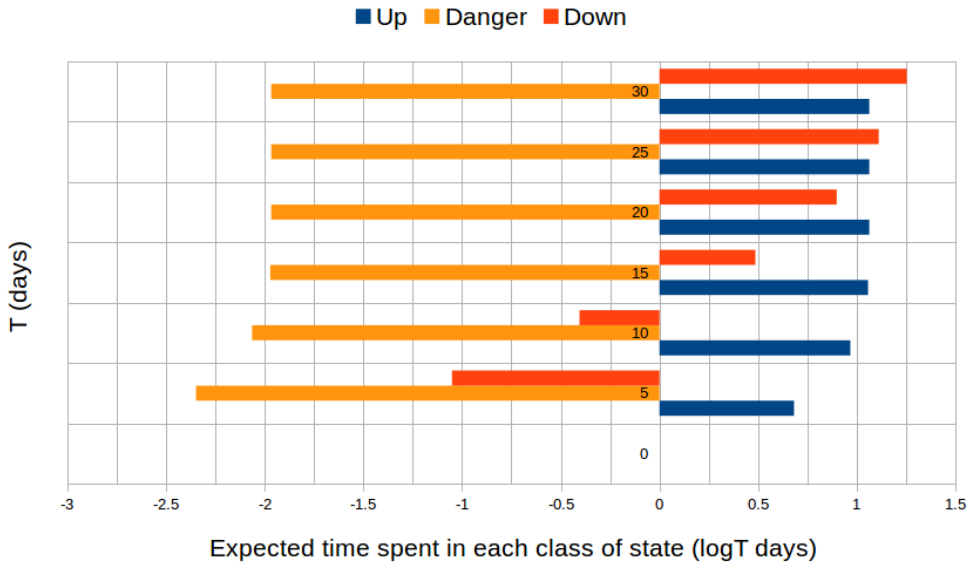

Figure 7. The expected amount of time spent in each of the states: “up”, “danger” and “shutdown”

6.3. *Random Scheduler*

This case study[‖] consisting of 3 examples evaluates our random scheduler implementation. In the example 1 and example 2, we implement a simple module with 2 thread processes. The processes in the example 1 has no synchronization point using `wait` statement, while there is one synchronization point in the example 2. When these examples are run, we can see that there are 2 execution orders in the first example and there are totally 4 execution orders in the second example.

Example 3 implements a SystemC model containing 3 threads. Each thread has 2 synchronization points using `wait` statements with other 2 threads. As a result, each thread has 3 execution segments. Therefore, there are totally $(3!)^3 = 216$ execution orders. In the first evaluation, we run the model 216 times with our random scheduler in order to compute its *coverage*. We got about 136 different execution orders or the scheduler coverage is $136/216 \sim 63\%$. Second, we run the model until we get all 216 different execution orders. Then we roundly need to run the model 1382 times. In other words, the *dynamic random verification efficiency* of the random scheduler is $216/1382 \sim 15.6\%$. It seems that the implementation with the pseudo random number generator (PRNG), by using the `rand()` function, and $\%$ operator in C/C++ is not efficient. We are planning to investigate the Mersenne Twister generator [27] that is by far the most widely used general-purpose PRNG in order to better deal with this issue.

7. RELATED WORK

Some work has been carried out for analyzing stochastic systems with PMC, for example, the dependability analysis of control system with PRISM [21]. PRISM supports construction and analysis of models as Markov chains, determining whether the model satisfies each property expressed in LTL. For example, the exact probabilities can be computed by PRISM. However, the

[‖] https://project.inria.fr/pscv/random-scheduler-examples/

DOI: 10.1002/smr

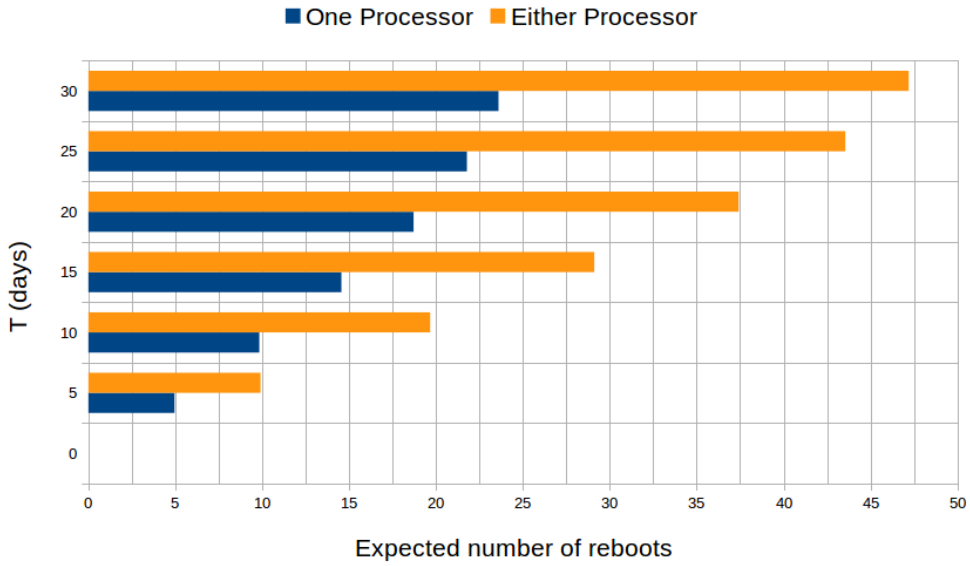

Figure 8. Expected number of reboots that occur in the first T time of operation

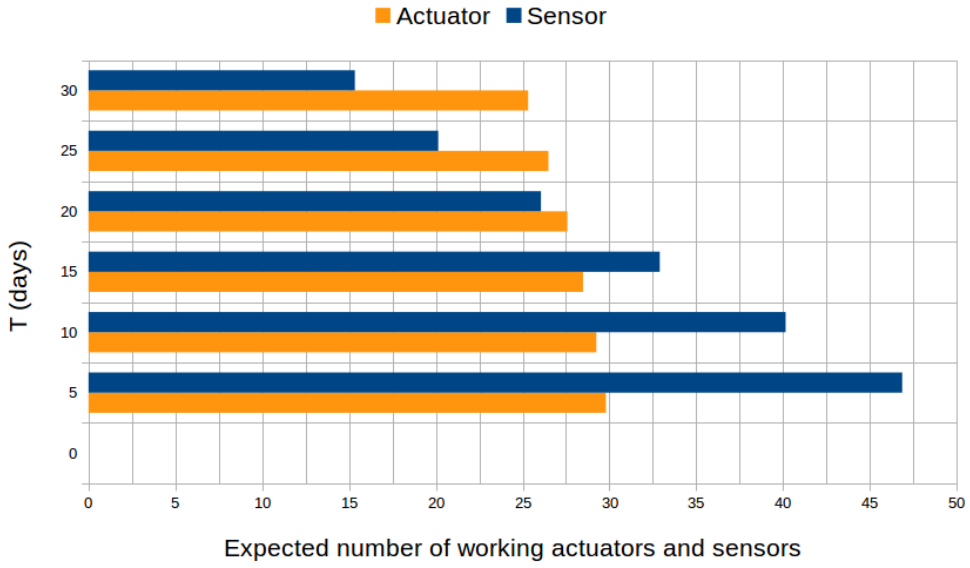

Figure 9. Expected number of functional sensor and actuator groups in the first T time of operation

main drawback of this approach is that when it deals with real-world large systems which make the PMC technique is unfeasible, even with some abstraction, i.e., symbolic model checking, is applied.

SMC has been recently applied in a wide range of research areas including software engineering (e.g., verification of critical embedded systems) [16, 36, 44], system biology [19], or medical area [10]. For instance, in [9], a framework of verifying properties of mixed-signal circuits, in which analog and digital quantities interact, was studied. The bounded linear temporal logic is used to express the properties, then the design of circuit and the verifying properties is connected to a statistical model checker. This approach consists of evaluating the properties on a representative

DOI: 10.1002/smr

subset of behaviors, generated by simulation, and answering the question of whether the circuit satisfies the properties with a probability greater than or equal to some value.

Several verification techniques have been proposed for SystemC, for example, SystemC Verification Working Group proposed the *SystemC Verification Standard* (SCV) [1] that provides APIs to verify functionality of programs. However, SCV does not mention about temporal specifications. One can use SCV as a complementatry of our verification framework in terms of automatically generating inputs for the MUV.

One of the earliest attempt at checking temporal specification properties for SystemC models is carried out by Braun et al. [4]. The temporal properties are expressed as *Finite Linear Temporal Logic* (FLTL) and the temporal resolution is defined by the boundary of the simulation-cycle notification. FLTL is linear temporal logic that interprets formulas over finite traces and supports temporal operators with respect to the temporal resolution. Then they proposed two approaches to extend test-bench features to SystemC for checking temporal properties: the checking is implemented directly within the SystemC language as an add-on library as SystemC itself. The other approach is to interface SystemC with an existing external testbench environment, TestBuilder [6]. As described in [38], the temporal resolution defined at boundary of the the simulation-cycle notification is inadequate for SystemC verifcation. Since it does not expose fully the semantics of the SystemC simulator kernel, which gives much finer grained temporal resolution. For example, the simulation may consist of a single delta-cycle if it is driven by immediate event notifications. As a result, the simulation clock never advances.

There has been a lot of work on the formalization of SystemC [15, 25, 47, 46, 45]. In general, the goal of the formalization process is to extract a formal model from a SystemC program, so that tools like model-checkers can be applied. For example, Zeng et al. in [45] translate a subset of the operational semantics of SystemC as a guarded assignment systems and use this translated model as an input for symbolic executors and checkers. However, all these formalizations consider semantics of SystemC and its simulator in some form of *global model*, and they also suffer from the state space explosion when dealing with industrial and large systems.

Tabakov et al. [37, 38] proposed a dynamtic-analysis-based framework for monitoring temporal SystemC properties. This framework allows users express the verifying properties by fully exposing the semantics of the simulator as well as the user-code. They extend LTL by providing some extra primitives for stating the atomic propositions and let users define a much finer temporal resolution. Their implementation consists of a modified simulation kernel, and a tool to automatically generate the *monitors* and aspect-advices for instrumenting SystemC programs automatically with AOP.

8. CONCLUSION

This article presents the first attempt to verify non-trivial probabilistic and temporal properties of SystemC model with statistical model checking techniques. In comparison with the probabilistic technique, our technique allows us to handle large insdustrial systems modeling in SystemC as well as to expose a rich set of user-code primitives by automatically instrumenting the user-code with AspectC++. The framework contains two main components: a *generator* that automatically generates a monitor and instruments the MUV based on the properties to be verified, and a *statistical model checker* implementing a set of hypothesis testing algorithms. In comparison to the probabilistic model checking, our approach allows users to handle large industrial systems, expose a rich set of user-code primitives in form of atomic propositions in BLTL, and work directly with SystemC models. For instance, our verification framework is used to analyze the dependability of large industrial computer-based control systems as shown in the case study.

Currently, we consider an external library as a “black box”, meaning that we do not consider the states of external libraries. Thus, arguments passed to a function in an external library cannot be monitored. For future work, we would like to allow users to monitor the states of the external libraries. We also plan to apply statistical model checking to verify temporal properties of SystemC-AMS (Analog/Mixed-Signal).

DOI: 10.1002/smr

To improve the dynamic random verification efficiency of the random scheduler's implementation, we are also planning to use the Mersenne Twister generator [27] that is by far the most widely used general-purpose PRNG instead of the default C/C++ generator.

REFERENCES

1. Accellera. http://www.accellera.org/downloads/standards/systemc.
2. AspectC++. http://www.aspectc.org/doc/ac-languageref.pdf, 2012.
3. B. Boyer, K. Corre, A. Legay, and S. Sedwards. Plasma Lab: A flexible, distributable statistical model checking library. In *QEST'13*, pages 160–164, 2013.
4. A. Braun, J. Gerlach, and W. Rosentiel. Checking temporal properties in SystemC specifications. In *HLDVT'02*, pages 23–27. IEEE International, 2002.
5. D. Bustan, S. Rubin, and M. Vardi. Verifying omega-regular properties of Markov chains. In *CAV'04*, 2004.
6. Cadence. Design Systems, Inc.: TestBuilder user guide. In *Product Version 1.0*, 2001.
7. H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd. Surviving the SoC revolution: A guide to platform-based design. In *Kluwer Academic Publishers, Norwell, USA*, 1999.
8. F. Ciesinski and M. Grober. On probabilistic computation tree logic. In *Validation of Stochastic Systems*, 2004.
9. E. Clarke, A. Donze, and A. Legay. Statistical model checking of mixed-analog circuits with an application to a third order delta-signam modulator. In *HVC'08*, 2008.
10. Cmacs. http://cmacs.cs.cmu.edu/.
11. A. Gal, W. Schroder-Preikschat, and O. Spinczyk. AspectC++: Language proposal and prototype implementation. In *OOPSLA'01*, 2001.
12. A. Goyal and et al. Probabilistic modeling of computer system availability. In *Annals of Operations Research*, volume 8, pages 285–306, 1987.
13. T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, USA, 2002.
14. GSL. http://www.gnu.org/software/gsl/.
15. P. Herber, J. Fellmuth, and S. Glesner. Model checking SystemC designs using timed automata. In *CODES/ISSS'08*, 2008.
16. H. Hermanns, B. Watcher, and L. Zhang. Probabilistic cegar. In *CAV'08*. LNCS, Springer, 2008.
17. A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *TACAS'06*. LNCS, Springer, 2006.
18. W. Hoeffding. Probability inequalities for sums of bounded random variables. In *American Statistical Association*, 1963.
19. S. Jha, E. Clarke, C. Langmead, A. Legay, A. Platzer, and P. Zuliani. A Bayesian approach to model checking biological systems. In *CMSB'09*. LNCS, Springer, 2009.
20. J. Katoen, E. Hahn, H. Hermanns, D. Jansen, and I. Zapreev. The ins and outs of the probabilistic model checker MRMC. In *QEST'09*, 2009.
21. M. Kwiatkowska, G. Norman, and D. Parker. Controller dependability analysis by probabilistic model checking. In *Control Engineering Practice*. Elsevier, 2007.
22. K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.
23. A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: An overview. In *RV'10*, 2010.
24. R. Lipsett, C. Schaefer, and C. Ussery. *VHDL: Hardware description and design*. Kluwer Academic Publishers, 1993.
25. F. Maraninchi, M. Moy, C. Helmstetter, J. Cornet, C. Traulsen, and L. Maillet-Contoz. SystemC/TLM semantics for heterogeneous SoCs validation. In *NEWCAS/TAISA'08*, 2008.
26. M. A. Marsan and M. Gerla. Markov models for multiple bus multiprocessor systems. In *IEEE Transactions on Computer*, volume 31(3), 1982.
27. M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. In *ACM Transactions on Modeling and Computer Simulation*, volume 8(1), pages 3–30, 1998.
28. W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiehl. The simulation semantics of SystemC. In *DATE 2001*, 2001.
29. J. Muppala, G. Ciardo, and K. Trivedi. Stochastic reward nets for reliability prediction. In *Communications in Reliability, Maintainability and Serviceability*, 1994.
30. V. C. Ngo, A. Legay, and V. Joloboff. PSCV: A runtime verification tool for probabilistic systemc models. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, pages 84–91, 2016.
31. V. C. Ngo, A. Legay, and J. Quilbeuf. Statistical model checking for SystemC models. In *HASE'16*, 2016.
32. PlasmaLab. https://project.inria.fr/plasma-lab/, 2016.
33. PSCV. https://project.inria.fr/pscv/, 2016.
34. J. Rutten, M. Kwiatkowska, G. Norman, and D. Parker. Mathematical techniques for analyzing concurrent and probabilistic systems. In *CRM Monograph Series*. American Mathematical Society, Providence, 2004.
35. K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In *CAV'05*, 2004.
36. K. Sen, M. Viswanathan, and G. Agha. Vesta: A statistical model-checker and analyzer for probabilistic systems. In *QUEST'05*. IEEE Computer Society, 2005.
37. D. Tabakov and M. Vardi. Monitoring temporal SystemC properties. In *Formal Methods and Models for Codesign*, 2010.
38. D. Tabakov and M. Vardi. Automatic aspectization of SystemC. In *MISS'12*. ACM, 2012.

DOI: 10.1002/smr

39. D. Thomas and P. Moorby. The Verilog hardware description language. In *Springer. ISBN 0-3878-4930-0*, 2008.
40. K. S. Trivedi. Probability and statistics with reliability, queueing, and computer science applications. In *Englewood Cliffs, NJ: Prentice-Hall*, 1982.
41. H. Younes. Verification and planning for stochastic processes with asynchronous events. In *PhD Thesis, Carnegie Mellon*, 2005.
42. H. Younes. Ymer: A statistical model checker. In *CAV'05*, 2005.
43. H. Younes, M. Kwiatkowska, G. Norman, and D. Parker. Numerical vs statistical probabilistic model checking. In *STTT'06*, 2006.
44. H. Younes and R. Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. In *COMPUT'06*, volume 204(9), pages 1368–1409, 2006.
45. N. Zeng and W. Zhang. A systemc semantics in guarded assignment systems and its applications with verds. In *Proceedings of the 2013 Asia-Pacific Software Engineering Conference*, APSEC '13, pages 371–379, Washington, DC, USA, 2013. IEEE Computer Society.
46. N. Zeng and W. Zhang. A symbolic partial order method for verifying systemc. In *Proceedings of the 2014 21st Asia-Pacific Software Engineering Conference - Volume 01*, APSEC '14, pages 271–278, Washington, DC, USA, 2014. IEEE Computer Society.
47. H. Zhu, J. He, S. Qin, and P. J. Brooke. Denotational semantics and its algebraic derivation for an event-driven system-level language. *Formal Aspects of Computing*, 27(1):133–166, 2015.

DOI: 10.1002/smr