Statistical Model Checking with Change Detection

Axel Legay, Louis-Marie Traonouez

To cite this version:

Axel Legay, Louis-Marie Traonouez. Statistical Model Checking with Change Detection. Transactions on Foundations for Mastering Change, 1, pp.157-179, 2016, 978-3-319-46507-4. 10.1007/978-3-319-46508-1_9 . hal-01242138

Statistical Model Checking with Change Detection

Axel Legay and Louis-Marie Traonouez

Inria Rennes – Bretagne Atlantique
263 Avenue du Général Leclerc - Bât 12
Tel.: +33-2-9984-7456 Fax: +33-2-99847171
firstname.lastname@inria.fr

Abstract. Statistical Model Checking (SMC) is a powerful and widely used approach that consists in estimating the probability for a system to satisfy a temporal property. This is done by monitoring a finite number of executions of the system, and then extrapolating the result by using statistics. The answer is correct up to some confidence that can be parameterized by the user. It is known that SMC mitigates the state-space explosion problem and allows to approximate undecidable queries. The approach has been implemented in several toolsets such as Plasma Lab, and successfully applied in a wide range of diverse areas such as systems biology, robotic, or automotive. In this paper, we add two new modest contributions to the cathedral of results on SMC. The first contribution is an algorithm that can be used to monitor changes in the probability distribution to satisfy a bounded-time property at runtime. Concretely, the algorithm constantly monitors the execution of the deployed system, and rise a flag when it observes that the probability has changed significantly. This is done by extending the applicability of the CUSUM algorithm used in signal processing into the formal validation setting. Our second contribution is to show how the programming interface of Plasma Lab can be exploited in order to make SMC technology directly available in toolsets used by designers. This integration is done by exploiting simulation facilities of design tools. Our approach thus differs from the one adopted by other SMC/formal verification toolsets which assume the existence of formal semantics for the design language, as well as a compiling chain to the rather academic one used by validation tool. The concept results in the integration of Plasma Lab as a library of the Simulink toolset. The contributions are illustrated by using Plasma Lab to verify two Simulink case studies, modeling a fuel control system and a pig shed.

Keywords: Satistical Model Checking, CUSUM, Monitoring, Optimisation, MATLAB/Simulink

1 Introduction and Motivations

Complex systems such as cyber-physical systems are large-scale distributed systems, often viewed as networked embedded systems, where a large number of computational components are deployed in a physical environment. Each component collects information and offers services to its environment (e.g., environmental monitoring and control, health-care monitoring and traffic control). This information is processed either at the component, in the network or at a remote location (e.g., the base station), or in any combination of these.

Characteristic of nowadays complex systems is that they have to meet a multitude of quantitative constraints, e.g., timing constraints, power consumption, memory usage, communication bandwidth, QoS, and often under uncertainty of the behavior of the environment. There is thus the need for new mathematical foundation and supporting tools allowing to handle the combination of quantitative aspects concerning, for example, time, stochastic behavior, hybrid behavior including energy consumption. The main difficulties being that the state space of nowadays systems is too large to be analyzed with classical validation technique. Another problem being that the interleaving of information eventually leads to undecidability.

In a series of recent works, the formal methods community has studied *Statistical Model Checking techniques* (SMC) [15,25,28,20] as a way to reason on quantitative (potentially undecidable) complex problems. SMC can be seen as a trade-off between testing and formal verification. The core idea of the approach is to conduct some simulations of the system and then use results from the statistic area in order to estimate the probability to satisfy a given property. Of course, in contrast with an exhaustive approach, a simulation-based solution does not guarantee a correct result. However, it is possible to bound the probability of making an error. Simulation-based methods are known to be far less memory and time intensive than exhaustive ones, and are sometimes the only option. SMC gets widely accepted in various research areas such as systems biology [8,18] or software engineering, in particular for industrial applications. There are several reasons for this success. First, it is very simple to implement, understand and use. Second, it does not require extra modeling or specification effort, but simply an operational model of the system, that can be simulated and checked against state-based properties. Third, it allows to verify properties that cannot be expressed in classical temporal logics, for instance the one presented in [7]. SMC algorithms have been implemented in a series of tools such as Ymer [28], PRISM [19], or UPPAAL [11]. Recently, we have implemented a series of SMC techniques in a flexible and modular toolset called Plasma Lab [5].

In this paper, we propose two new contributions to SMC. As usual with SMC techniques, we focus on requirements that can be represented by bounded temporal properties, i.e., properties that can be decided on a finite sequence of states. Classical SMC algorithms are interested in estimating the probability to satisfy such a property starting from an initial state, which is done by monitoring a finite set of executions from this state. In this paper, we also consider the case where one can only observe the current execution of the system. In this context, we are interested in observing the evolution of the probability to satisfy the property at successive positions of the execution, and detecting positions where it drastically changes from original expectation. In summary, our first contribution is a methodology that can be used to monitor changes in probability distributions to satisfy a bounded property at runtime. Given a possibly infinite sequence of states that represents the continuous execution of the system, the algorithm monitors the property at each position and rises a flag when the proportion of satisfaction has changed significantly. The latter can be used to monitor, e.g., emergent behaviors. To achieve this objective, we adapt CUSUM [23,3], an algorithm that can be used to detect changes in signal monitoring. Our ambition is not to propose a new version of CUSUM, but rather to show how the algorithm can be used in the monitoring context. This is to our knowledge the first application of CUSUM with SMC.

Our second contribution is to show how the programming interface of Plasma Lab can be exploited in order to make SMC technology directly available in toolsets used by designers. Our approach differs from the one adopted by other SMC/formal verification toolsets which assume the existence of formal semantics for the design language, as well as a compiling chain to the rather academic languages used by validation tool. The concept is illustrated with an integration of Plasma Lab as a library of the Simulink toolset of the MATLAB environment. Concretely, we show that the recently developed Plasma Lab can directly be integrated as a Simulink library, hence offering the first in house tool for the verification of stochastic Simulink models – this tool completes the panoply of validation toolsets already distributed with Simulink. Another advantage of our approach is that any advance on SMC that we will implement within Plasma Lab in the future will directly be available to the Simulink users.

Structure of the paper. The paper is organized as follows. In Section 2, we discuss related work about formal verification of Simulink models. In Section 3, we introduce our formal model of systems and define the statistical model checking problems we want to solve. Section 4 discusses solutions to those problems. Section 5 discusses the integration of Plasma Lab within Simulink. Section 6 and 7 illustrate our approach on two case-studies, modeled with Simulink and verified with Plasma Lab. Finally Section 8 concludes the paper.

2 Related Work

Simulink includes the *Simulink Design Verifier* to formally verify and validate the models and the generated code. The framework is based on the analysis tools Polyspace [21] and Prover plug-in [24].

All the modeling is done using Simulink : Formal specifications (requirements), operational and structural specifications, and it is possible to annotate the code with assertions. The different analyses exploit the results of the integrated analyzers (Polyspace & Prover Plug-in). To summarize, Polyspace analyses the C-Code of the atomic components and Prover Plug-in verifies the operational specification against the requirements and generate some counter-examples when possible. The Framework can also generate some test cases for different covering mode (Modified Condition/Decision Coverage).

Polyspace is a static analyzer for C/C++ code based on abstract interpretation. Unlike Plasma Lab the analyses are dedicated to some special classes of properties to ensure the C-Code executions are safe for the data handled. But, Polyspace is not able to verify other kind of properties, like liveness properties. It also has apparently no support for models with probabilistic behavior.

Prover Plug-in [24] is a model checker for the verification of embedded systems, integrated circuits, and more generally for systems designed in languages like Verilog, C, Simulink, . . . , and also for the UML modelling. A priori, the checker only handles deterministic and sequential systems; it implies neither parallelism, nor probabilities. It is however able to generate counter-examples if the verification of the property fails. Unlike Polyspace, Prover Plug-in is a general model checker like Plasma Lab: it can be used for any kind of properties.

Other formal verification approaches imply translating the Simulink model in the specific language of the model checker. Simulink models can be semantically translated to hybrid automata [1]. However the model checking problem of these models is in general undecidable and intractable for complex systems. More efficient translations can be achieved by restricting the type of blocks that can be used in the Simulink models: in general by removing continuous behaviors in order to obtain a finite state machine. For instance Honeywell presents in [22] a tool that translates certain Simulink models in the input language of the model checker NuSMV [6]. [2] also presents a tool chain that translates Simulink models in the input language of the LTL model checker DiViNE. This tool chain uses the tool HiLiTe [12], also developed by Honeywell, that can perform semantic analyses of Simulink models. Finally translations exists from Simulink to synchronous languages, like Lustre in the tool SCADE. These translations however are too restrictive to be applied to the avionics proposed by Honeywell for instance [22]. Contrary to these model checking approach, SMC techniques are not restricted by the model, and our Simulink plugin for Plasma Lab is able to handle any type of Simulink and Stateflow diagrams, with both continuous and discrete behaviors.

A first experiment with SMC and Simulink was presented in [29]. Their approach consists in programming one SMC algorithm within the Simulink toolbox. On the contrary, the flexibility of our tool will allow us to incrementally add new algorithms to the toolbox without new programming efforts. The authors used a Bayesian statistical analysis for verifying the Fault-Tolerant Fuel Control System, an example from the basic Simulink examples database. In the paper they consider a version of the model where the failures are triggered following a Poisson distribution. We are able to use Plasma Lab to verify the same example in Section 6 and we do not limit to verification analyses since we also present in Section 7 a case study on which we perform optimization and change detection.

Finally, our approach is also different from the one in [9] that consists in translating parts of Simulink models into the Uppaal language (which makes it difficult for analyzing counter examples). Therefore Plasma Lab for Simulink offers the first integrated verification tool for Simulink models with stochastic information.

3 Systems and Problems

We denote $\mathbb{R}$ the set of real numbers and $\mathbb{Q}_{\geq 0}$ the set of positive rational numbers. Consider a set of states S and a set of state's variables SV. Assume that each state variable $x \in SV$ is assigned to domain $\mathbb{D}_x$, and define the valuation function V, such that $V(s, x) \in \mathbb{D}_x$ is the value of x in state s. Consider also a time domain $T \subseteq \mathbb{R}$. We propose the following definition to capture the behavior of a large class of stochastic systems.

Definition 1 (Stochastic Process). *A* stochastic process (S, T) *is a family of random variables $\mathcal{X} = \{X_t \mid t \in T\}$, each X_t having range S.*

An *execution* for a stochastic process (S, T) is any sequence of observations $\{x_t \in S \mid t \in T\}$ of the random variables $X_t \in \mathcal{X}$. It can be represented as a sequence $\pi = (s_0, t_0), (s_1, t_1), \ldots, (s_n, t_n)$, such that $s_i \in S$ and $t_i \in T$, with time stamps monotonically increasing, e.g. $t_i < t_{i+1}$. Let $0 \leq i \leq n$, we denote $\pi^i = (s_i, t_i), \ldots, (s_n, t_n)$ the suffix of π starting at position i. Let $\overline{s} \in S$, we denote $Path(\overline{s})$ the set of executions of $\mathcal{X}$ that starts in state $(\overline{s}, 0)$ (also called initial state) and $Path^n(\overline{s})$ the set of executions of length n.

In [28], Youness showed that the executions set of a stochastic process is a measurable space, which defines a probability measure μ over $Path(\overline{s})$. The precise definition of μ depends on the specific probability structure of the stochastic process being studied. We now define the general structure for *stochastic discrete event systems.*

Definition 2. *A* stochastic discrete event system *(SDES) is a stochastic process extended with initial state and variable assignments, i.e., $\mathcal{S}ys = \langle S, I, T, SV, V\rangle$, where (S, T) is a stochastic process, $I \subseteq S$ is the set of initial states, SV is a set of state variables and V is the valuation function.*

We denote $Path(Sys)$ the set of executions of Sys that starts from an initial state in I. Properties over the executions of $\mathcal{S}ys$ are defined via the so-called Bounded Linear Temporal Logic (BLTL) [4]. BLTL restricts Linear Temporal Logic by bounding the scope of the temporal operators. Syntactically, we have

$$\varphi, \varphi' := \ \mathsf{true} \mid x \sim v \mid \varphi \wedge \varphi' \mid \neg\varphi \mid X_{\leq t} \mid \varphi\, U_{\leq t}\, \varphi'$$

where φ, φ' are BLTL formulas, $x \in SV$, $v \in D_x$ and $t \in \mathbb{Q}_{\geq 0}$ and $\sim \in \{<, \leq, =, \geq, >\}$. As usual, we define $F_{\leq t}\varphi \equiv \mathsf{true}\ U_{\leq t}\varphi$ and $G_{\leq t}\varphi \equiv \neg F_{\leq t}\neg\varphi$. The semantics of BLTL is defined with respect to an execution $\pi = (s_0, t_0), (s_1, t_1), \ldots, (s_n, t_n)$ of a SDES using the following rules:

- $\pi \models X_{\leq t}\, \varphi$ iff $\exists i,\ i = max\{j \mid t_0 \leq t_j \leq t_0 + t\}$ and $\pi^i \models \varphi$
- $\pi \models \varphi_1\, U_{\leq t}\, \varphi_2$ iff $\exists i,\ t_0 \leq t_i \leq t_0 + t$ and $\pi^i \models \varphi_2$ and $\forall j,\ 0 \leq j < i,\ \pi^j \models \varphi_1$
- $\pi \models \varphi_1 \wedge \varphi_2$ iff $\pi \models \varphi_1$ and $\pi \models \varphi_2$
- $\pi \models \neg\varphi$ iff $\pi \not\models \varphi$
- $\pi \models x \sim v$ iff $V(s_0, x) \sim v$

- $\pi \models \mathsf{true}$

In the rest of the paper, we consider two problems that are 1. the quantitative (optimization) problem for BLTL, and 2. the detection of changes. The first problem has largely be discussed in SMC papers, and the second problem is a new comer in the SMC area. The motivation to reintroduce the quantitative problem is that it can be used to calibrate the detection algorithm.

3.1 Quantitative and Optimization Problems

Given a SDES $\mathcal{S}ys$ and a BLTL property φ, the existence of a probability measure μ over $Path(Sys)$ allows to define the probability measure $Pr[\mathcal{S}ys \models \varphi] = \mu\{\pi \in Path(Sys) \mid \pi \models \varphi\}$. The *quantitative problem* consists in computing the value of $Pr[\mathcal{S}ys \models \varphi]$.

We will also study the *optimization problem*, that is the one of finding an initial state that maximizes/minimizes the value of a given observation. Consider a set $\mathcal{O}$ of observations over $\mathcal{S}ys$. Each observation $o \in \mathcal{O}$ is a function $o : Path^n(\overline{s}) \rightarrow \mathbb{D}_o$ that associates to each run of length n and starting at $\overline{s}$ a value in a domain $\mathbb{D}_o$. We denote $(\tilde{o})^{\overline{s}}_n$ the average value of $o(\pi)$ over all the executions $\pi \in Path^n(\overline{s})$. The *optimization problem* for $\mathcal{S}ys$ is to determine an initial state $\overline{s} \in I$ that minimizes or maximizes the value $(\tilde{o})^{\overline{s}}_n$, for all $o \in \mathcal{O}$.

As an example, an observation can simply be the maximal value of a given parameter, like a cost or reward, along an execution. The average observation then becomes the sum of those observations divided by the number of runs. In this context, the optimization could be to find the initial state that minimizes the value of the parameters.

3.2 Change Detection Problem

In this section, we monitor the system in order to detect an expected event by looking at the variation of a probability measure over a set of samples of an execution. Therefore, contrary to the previous SMC problems, we consider a single execution on which we checked a BLTL property at regular intervals. On this execution we want to determine the time at which the probability measure of the BLTL property changes sufficiently to characterize an expected event on the system.

More precisely, we consider a (potentially infinite) execution $\pi = (s_0, t_0), (s_1, t_1), \ldots, (s_n, t_n), \ldots$ of a system $\mathcal{S}ys$. We monitor a BLTL property φ from each position (s_i, t_i) of this execution (the monitoring involves a finite sequence of states as BLTL formulas are time bounded) and we compute an ingenious proportion on the numbers of satisfaction and non satisfaction of the property. This proportion is used to detect changes in the probability to satisfy the property at a given point of the execution. Concretely, assuming that this probability is originally $p{<}k$, we detect a change index in the execution when the probability becomes $p{\geq}k$.

Example 1. Consider the firefighting services in city like London. Assume that under normal traffic conditions, the firemen can extinguish a fire within three hours with a probability greater than 0.7. It is expected that this probability decreases when the traffic increases. The challenge is to detect the time t when this change happens.

Formally, we consider a sequence of Bernoulli variables X_i such that $X_i = 1$ iff $\pi^i \models \varphi$. We define that an execution π satisfies a change $\tau = Pr[\pi \models \varphi] \geq k$, iff $Pr[X_i = 1]{<}k$ for $t_i < t$ and $Pr[X_i = 1]{\geq}k$ for $t_i \geq t$. Given an execution π, we use $\tau!$ to denote the index $i = (s_i, t_i)$ in π at which the execution is subject to the change. We assume an implicit change detection maximal time set by the user. If no change is detected after this time has passed, then we set up the evaluation of τ to ∞. In case the execution is subject to several changes, we take the first time. Using those notations, one can define Boolean propositions over changes and their respective time. One can also combine changes propositions with BLTL formulas, providing that those propositions are not in the scope of temporal operators. We now introduce extended BLTL change-based relations, an extension of BLTL that incorporates a change detection operator.

Definition 3. *Given an execution π of $\mathcal{S}ys$, an extended BLTL change relation is defined as:*

$$\begin{aligned} prop &:= \texttt{let}\ \tau_1 = change\ \texttt{and}\ \ldots\ \texttt{and}\ \tau_n = change\ \texttt{in}\ \delta \\ change &:= Pr[\pi \models \varphi] \star k \\ \delta, \delta' &:= \tau_i! \sim \tau_j! + t \mid \tau_i! \sim t \mid \varphi' \mid \delta \wedge \delta' \mid \neg\delta \end{aligned}$$

where $k \in]0,1[$, $t \in \mathbb{Q}_{\geq 0}$, $\star \in \{\leq, \geq\}$, $\sim \in \{<, \leq, =, \geq, >\}$, φ and φ' are BLTL formulae, τ_i and τ_j are change identifiers defined in the prop rule.

This extension allows us, e.g., to express conditions such as “if a change occurs at time t, then the system shall reach a state x in less than 10 units of time”. The semantics of extended BLTL change relation easily follows from the one of BLTL and the description of the change operator.

4 A Statistical Model Checking Approach

In this section, we detail our statistical model checking algorithmic solutions to the problems described in Section 3. SMC solutions to the quantitative verification and optimization problems are well-known and will only briefly be surveyed. SMC solution for extended BLTL change relations is new.

4.1 Quantitative Verification

We first focus on the problem of computing the probability $Pr[\mathcal{S}ys \models \varphi]$ for a SDES $\mathcal{S}ys$ to satisfy a BLTL property φ. With SMC we estimate this probability using a number of statistically independent simulation traces of an executable model. The idea is to monitor the property on each simulation, and to represent the outcome of the ith monitoring with a Bernoulli variable X_i that takes the value 1 if the execution satisfies the property and 0 otherwise. We then use an algorithm from the statistic area to compute the probability of the Bernoulli variable (which corresponds to the probability for the system to satisfy the property). Those algorithms include Monte Carlo, or importance sampling/splitting [16]. Algorithms for monitoring BLTL properties on a given execution can be found in [14]. In this paper, the quantitative problem will mainly be solved in the context of calibrating a change algorithm as well as to validate BLTL properties without change.

Optimization We now show that a simulation approach can also be used to perform an optimization of the model by varying the model parameters and evaluating the observable quantities to optimize. We consider a SEDS $\mathcal{S}ys$, with a set of initial states I, and a set of observations $\mathcal{O}$ and a bound $n \in \mathbb{N}$.

For each initial state $\overline{s} \in I$ we perform N random simulations π^i from $Path^n(\overline{s})$ and we compute the average value of the observed quantities at the end of the simulations. Therefore, for each observation $o \in \mathcal{O}$ we compute an estimation $\frac{1}{N}\sum_{i=1}^{N} o(\pi_i)$ of the average value $(\tilde{o})_n^{\overline{s}}$.

To solve the optimization problem, we must determine the configurations in I that optimize (minimize or maximize) these quantities. When the problem is defined with several observable quantities, we are faced with a multi-objective problem, and the best configurations are then selected by computing the Pareto frontier of the set of observations [10].

4.2 Change Detection with CUSUM

In this section, we consider SMC solutions for verifying extended BLTL properties with changes. We first present an SMC algorithm for change detection, and then briefly discuss the monitoring of extended BLTL. For change detection, we resort to the CUSUM algorithm [23,3], whose principles have already been formalized in other contexts[26]. This algorithm, originally developed in the signal theory world, is used to detect the probability changes during the execution of a stochastic system. The main purpose of the change detection is to detect when some changes in some parameters, not easily observable or measurable, will perturb the measures and the observations done over the system. The principle is to compare the probability p when the system is working normally against the probability p' resulting of the change.

Let $\mathcal{S}ys$ be a SDES and $\pi = (s_0, t_0), (s_1, t_1), \dots$ be an execution of $\mathcal{S}ys$. We consider the change $\tau = Pr[\pi \models \varphi] \geq k$ with φ a BLTL property and $k \in]0, 1[$. Let $X_1, \dots X_N$ be a finite set of Bernoulli variables such that X_i takes the value 1 iff $\pi^i \models \varphi$. We note $p_n = Pr[Xi = 1 | i <= n]$ the probability of satisfying φ from (s_0, t_0) to the state (s_n, t_n). We will use the CUSUM algorithm to decide between the two following hypothesis:

- $H_0 :\ \forall\, n,\ 0 \leq n \leq N, p_n < k$, *i.e.*, no change occurs

- $H_1 : \exists\, m,\ 0 \leq m \leq N$ such that the change occurs at time t_m: $\forall n,\ 0 \leq n \leq N$, we have $t_n < t_m \Rightarrow p_n < k$ and $t_n \geq t_m \Rightarrow p_n \geq k$.

We assume that we know the initial probability $p_{init} < k$ of $Pr[\pi \models \varphi]$ before the change occurs. One solution is to estimate this probability with the Monte Carlo algorithm using an ideal version of the system in which not change occurs. The CUSUM algorithm will use the two probabilities p_{init} and k to decide between the two hypothesis and determine the time of the change, if it occurs.

Like the Sequential Probability Ratio Test (SPRT) [27,20], the CUSUM comparison is based on a likelihood-ratio test: it consists in computing the cumulative sum S_n of the logarithm of the likelihood-ratios s_i over the sequence of samples $X_1, \ldots X_n$ and detecting the change decision as soon as S_n satisfies the stopping rule.

$$S_n = \sum_{i=1}^{n} s_i \qquad\qquad s_i = \begin{cases} \ln \frac{k}{p_{init}}, & \text{if } X_i = 1 \\ \ln \frac{1-k}{1-p_{init}}, & \text{otherwise} \end{cases}$$

The typical behavior of the cumulative sum S_n is a global decreasing before the change, and a sharp increase after the change. Then the stopping rule's purpose is to detect when the positive drift is sufficiently relevant to detect the change. It consists in saving $m_n = \min_{1 \leq i \leq n}\ S_i$, the minimal value of CUSUM, and comparing it with the current value. If the distance is sufficiently great, the stopping decision is taken, *i.e.*, an alarm is raised at time $t_a = \min\{t_n : S_n - m_n \geq \lambda\}$, where λ is a sensitivity threshold.

The CUSUM proportion can only be computed during a finite amount of time, which is set by the user. In case there is no detection, we set $t_a = +\infty$. Note that we presented CUSUM monitoring for the case $p \geq k$, but it could be set up for $p \leq k$ by defining the stopping rule for the maximum value of CUSUM instead.

CUSUM Calibration It is important to note that the likelihood-ratio test assumes that the considered samples are independent. This assumption may be difficult to ensure over a single execution of a system, but several heuristic solutions exist to guarantee independence. One of them consists in finding a location frequently visited during the execution of the system. Collecting exactly one sample each time such a state is visited, ensures independence between samples. In our context, such a state can be the initial location from which the execution is constantly restarted. However this solution cannot be applied to continuous-time systems. Another solution is to introduce delays between the samples. In that case Monte Carlo SMC analyses can evaluate the correlation between the samples, and help to select appropriate delays.

The CUSUM sensitivity depends on the choice of the threshold λ. A smaller value increases the sensitivity, *i.e.*, the false alarms rate. A false alarm is a change detection at a time when no relevant event actually occurs in the system.

Conversely, big values may delay the detection of the changes. The false alarms rate of CUSUM is defined as $E[t_a]$, the expected time of an alarm raised by CUSUM while the system is still running before the change occurs. Ideally, this value must be the biggest as possible $E[t_a] \to +\infty$. The detection delay is defined as the expected time between the actual change of time t and the alarm time t_a raised by CUSUM: $E[t_a - t \mid t < t_a]$. Ideally, this value has to be small as possible. In Section 7, we will propose a heuristic that uses the quantitative model checking problem in order to calibrate the algorithm.

The empirical way to choose the stopping rule One of the main difficulties in applying CUSUM is to compute the minimal duration needed to trigger an alarm. Indeed, the algorithm may be subjected to brief local changes that should not impact the final result. Theoretically, the properties of the CUSUM are based on the computation of the Average Run Length function (ARL) [3]. In a very few cases, this function may be computed or approximated using some approximating techniques (Wald or Siegmund) but most of the time, it is too complex to be used and to deduce λ. In this paper we propose a variant of the methodology proposed in [26]. Our approach consists in exploiting $\mathcal{S}ys_0$, that is a version of the system for which the change does not occur. We first compute the probability p_{init} for this system to satisfy the property. We then compute several CUSUM on $\mathcal{S}ys_0$ in order to compute the average frequency of a false alarm. The latter is obtained by observing the mean time between positive drift in the CUSUM as well as its duration in term of samples (observations of the CUSUM ratio). We then compute the minimal sample duration to exceed the change probability k. This value is multiplied by the logarithm of k divided by p_{init} (*i.e*, the minimal value of a drift).

Monitoring executions for Change Relation Satisfiability We now briefly discuss the monitoring of extended BLTL with changes. Let us consider the change relation γ based on $\tau_1, \ldots, \tau_n$ changes. Using the syntax introduced in Section 3.2, it is expressed as `let` τ_1 `and` ... `and` τ_n `in` γ, where γ contains Boolean operations over changes and BLTL formulas. We use the following monitoring procedure for each atom:

1. For each change τ_i, we set a CUSUM monitor that splits the monitoring into sub-monitors, one for each random variable, *i.e.*, one to monitor the BLTL formula involved in the change from a given position of the execution. Note that classical tableau-based heuristics allows us to reuse information between monitoring actions.
2. The proposition $\tau_i!$ holds iff $t_i \neq +\infty$. The proposition $\tau_i! \sim t$ holds iff $t_i \sim t$. Similarly, the proposition $\tau_i! \sim \tau_j! + t$ holds only if $t_i \sim t_j + t$ but it is undefined if $t_i = t_j = +\infty$.
3. BLTL formulas can be monitored with classical techniques.

In practice, the tool generates monitors on demand for the given atoms and combines their answers in a Boolean manner.

5 Plasma Lab and Simulink Integration

The results presented in Section 4 have been implemented in the Plasma Lab SMC toolbox[1]. In this section, we first recap the main features of the tool, and then show how the architecture of the implementation can be exploited in order to integrate Plasma Lab within Simulink, hence providing an in shell new verification theory for this widely used language. The main contribution in this section with respect to [5] is to show how the architecture can be exploited to perform the integration.

5.1 On Plasma Lab

Plasma Lab is a compact, efficient and flexible platform for statistical model checking of stochastic models. The tool offers a series of SMC algorithms which includes rare events simulation, distributed SMC, non-determinism, or optimization. The main difference between Plasma Lab and other SMC tools is that Plasma Lab proposes an API abstraction of the concepts of stochastic model simulator, property checker (monitoring) and SMC algorithm. In other words, the tool has been designed to be capable of using external simulators, input languages, or SMC algorithms. This not only reduces the effort of integrating new algorithms, but also allows us to create direct plug-in interfaces with industry used specification tools. The latter being done without using extra compilers.

Fig. 1 presents Plasma Lab architecture. More specifically, the relations between model simulators, property checkers, and SMC algorithms components. The simulators features include starting a new trace and simulating a model step by step. The checkers decide a property on a trace by accessing to state values. They also control the simulations, with a *state on demand* approach that generates new states only if more states are needed to decide the property. A SMC algorithm component, such as the CUSUM algorithm, is a runnable object. It collect samples obtained from a checker component. Depending on the property language, their checker either returns Boolean or numerical values. The algorithm then notifies progress and sends its results through the Controller API.

In coordination with this architecture, we use a plugin system to load models and properties components. It is then possible to support new model or property languages. Adding a simulator, a checker or an algorithm component is pretty straightforward as they share a similar plugin architecture. Thus, it requires only a few classes and methods to get a new component running. Each plugin contains a factory class used by Plasma Lab to instantiate component objects. These components implement the corresponding interface defining their behavior. Some companion objects are also required (results, states, identifiers) to allow communication between components and the Controller API.

One of the goal of Plasma Lab is also to benefit from a massive distribution of the simulations, which is one of the advantage of the SMC approach. Therefore Plasma Lab API provides generic methods to define distributed algorithms. We

[1] Available at `https://project.inria.fr/plasma-lab/`

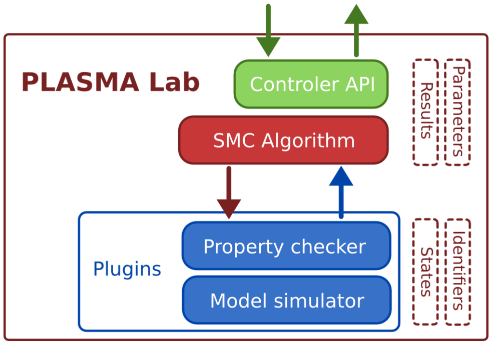

Fig. 1: Plasma Lab architecture

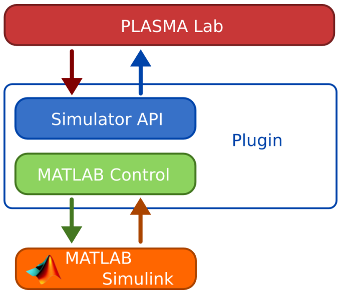

Fig. 2: Interface between Plasma Lab and Simulink

have used these functionalities to distribute large number of simulations over a computer grid [2].

5.2 On Integrating Plasma Lab within Simulink

We now show how to integrate Plasma Lab within Simulink, hence lifting the power of our simulation approaches directly within the tool. We will focus on those Simulink models with stochastic information, as presented in [29]. But our approach is more flexible because the user will directly use Plasma Lab within the Simulink interface, without third party.

Simulink is a block diagram environment for multi-domain simulation and Model-Based Design approach. It supports the design and simulation at the system level, automatic code generation, and the testing and verification of embedded systems. Simulink provides a graphical editor, a customizable set of block libraries and solvers for modeling and simulation of dynamic systems. It is integrated within MATLAB. The Simulink models we considered have special extensions to randomly behave like failures. By default the Simulink library provides some random generators that are not compatible with statistical model checking: they always generate the same random sequence of values at each execution. To overcome this limitation we use some C-function block calls that generate independent sequences of random draws.

Our objective was to integrate Plasma Lab as a new Simulink library. For doing so, we developed a new simulator plugin whose architecture is showed in Fig. 2. One of the key points of our integration has been to exploit MATLAB Control[3], a library that allows to interact with MATLAB from Java. This library uses a proxy object connected to a MATLAB session. MATLAB invokes, *e.g.* functions `eval`, `feval` . . . as well as variables access, that are transmitted and

[2] `https://project.inria.fr/plasma-lab/documentation/tutorial/igrida-experimentation/`

[3] `https://code.google.com/p/matlabcontrol/`

executed on the MATLAB session through the proxy. This allowed us to implement the features of a model component, controlling a Simulink simulation, in MATLAB language. Calls to this implementation are then done in Java from the Plasma Lab plugin.

Regarding the monitoring of properties, we exploit the simulation output of Simulink. More precisely, BLTL properties are checked over the executions of a SDES, *i.e.*, sequences of states and time stamps based on the set of state variables SV. This set must be defined by declaring in Simulink signals as log output. During the simulation these signals are logged in a data structure containing time stamps and are then retrieved as states in Plasma Lab. One important point is that Simulink discretizes the signals trace, its sample frequency being parameterized by each block. In terms of monitoring this means that the sample frequency must be configured to observe any relevant change in the model. In practice, the frequency can be set as a constant value, or, if the model mixes both continuous data flow and state flow, the frequency can be aligned on the transitions, *i.e.*, when a state is newly visited.

6 Fault-Tolerant Fuel Control System

This model is taken from the Simulink/Stateflow examples library. It describes the fuel control system of a gasoline engine. The system is made robust by detecting failures in sensors and dynamically re-configuring its behavior to maintain a continuous operation. This is a typical example of hybrid system. It is modelled in Simulink by using Sateflow diagrams to to handle the discrete changes of the control system, and linear differential equations to model the continuous behaviors.

The system contains four separate sensors: a throttle sensor, a speed sensor, an oxygen sensor, and a pressure sensor. Each of these sensors is represented by a parallel state in Stateflow, that is say finite state machines concurrently active. In total the entire logic of the systems is implemented by six parallel states. Each parallel state of a sensor contains two sub-states, a normal state and a fail state (the exception being the oxygen sensor, which also contains a warm-up state). If any of the sensor readings is outside an acceptable range, then a fault is registered, and the state of the sensor transitions to the failed sub-state. If the sensor recovers, it can transition back to the normal state.

In the original model, sensors faults are decided by the user using manual switch block for each sensor. The interest of the SMC approach comes from the possibility to observe a large set of execution traces produced by a probabilistic procedure. Therefor we replaced the Speed, EGO and MAP manual switches by custom probabilistic switches. These switches use a Poisson distribution and are parameterized by a rate to decide when a fault happen. A sensor will repair itself after a duration of 1 second. This modified model is similar to the one use in [29].

The Poisson distribution block that we use draws a random time T in seconds, that is the time before the next fault happens, and we use a Stateflow diagram as a timer. The signal from the Poisson block is then used by the sensor's switch. A Stateflow repair timer is used to maintain the fault signal for a duration of 1 second.

SMC analysis The system uses its sensors to maintain the air-fuel ratio at a constant value. When one sensor fails, a higher ratio is targeted to allow a smoother running. If another sensors fails the engine is shutdown for safety reasons, which is detected by a zero fuel rate.

We estimate the probability of a long engine shutdown. We use the following BLTL property to monitors executions over a period of 100 t.u., and to check if the fuel remains at zero for 1 t.u. :

$$\Phi = \neg F_{\leq 100}(G_{\leq 0.999}\mathsf{Fuel} = 0)$$

We try to reproduce with this property the results of [29]. In this paper they use a Bayesian SMC technique to estimate the probability of this property with the bound 1 for G operator. We can almost reproduce their results using the Monte Carlo algorithm on our own implementation of the Simulink model with stochastic distributions, but only if we use the approximated bound 0.999. Indeed the property is false, mainly when the three sensors are faulty at the same time. In that case the second sensor to fail remains in fault condition for exactly one second, with at least one other sensor. When this second sensor is repaired, there remains only one faulty sensor and the engine is restarted. Whether the Fuel variable in the sample after exactly one second is monitored at 0 or 1 by the SMC checker, changes the evaluation of the property. By using the value 0.999 we avoid these approximation issues. Table 1 recaps our results and the one of [29] for different values of the sensors fault rates (expressed in seconds). Our results are obtained with Plasma Lab Monte Carlo (MC) algorithm after 1000 simulations. It takes approximately 2500 seconds to complete on a 2.7GHz Intel Core i7 with 8GB RAM and running MATLAB R2014b on Linux.

Fault rates	**Plasma Lab MC**	**Bayesian SMC [29]**
(3 7 8)	0.396	0.356
(10 8 9)	0.748	0.853
(20 10 20)	0.93	0.984
(30 30 30)	0.985	0.996

Table 1: Probability estimation of Φ with Plasma Lab and the results from [29]. The fault rates in seconds correspond to the Speed, EGO and MAP sensors, respectively.

7 A Pig Shed Case study

We now illustrate the change detection contribution of this paper on the model of a temperature controller in a pig shed. This model is inspired by similar studies [17,13,10]. The system under control is a pig shed equipped with a fan and a heater to regulate the air temperature. Air temperature in the shed is subjected to random variations due to the variation of external temperature and the variation of the number of pigs that produce heat. The objective of the controller is to counter these variations such that the temperature remains within a given comfort zone. To do so, the controller can activate the heater to increase the temperature, and the fan to bring external air and therefore cool the shed. Then the temperature T of the shed is given by the following differential equation:

$$T' = T_{ext} * Q - T * Q + W_{heater} + W_{pigs}$$

where T_{ext} is the external temperature, $Q = Q_{min} + Q_{fan}$ is the air flow created by a minimal flow Q_{min}, and an additional flow Q_{fan} when the fan is activated, W_{heater} is the heat produced by the heater, when activated, and W_{pigs} is the heat produced by the pigs. This equation is modeled by the Simulink subsystem of Fig. 3.

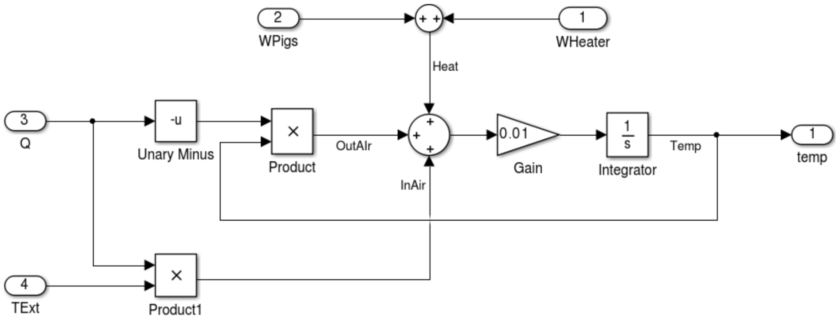

Fig. 3: Simulink model of the differential equation controlling the temperature

The controller that we study applies a *bang-bang* (also called *on-off*) strategy that is specified by four temperature thresholds, that is (1) when the temperature goes above TFanOn, the fan is turned on, (2) when the temperature returns below TFanOff, the fan is turned off, (3) when the temperature goes below THeaterOn, the heater is turned on, (4) when the temperature returns above THeaterOff, the heater is turned off. This controller is implemented by Stateflow automata given in Fig. 4.

The fan and the heater are subjected to random failures when they are in use. Exponential distributions control the occurrence time of a failure. After a failure a reparation process allows to restart the fan or the heater, but it also takes a random time, exponentially distributed. These failures are modeled by

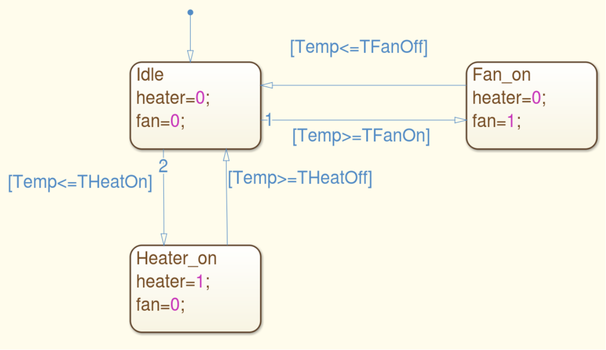

Fig. 4: Temperature controller

two Stateflow automata, as shown in Fig. 5. In this automaton, rnd is a random number between 0 and 1, and tuse is the duration of use of the fan or heater. The timings tfail and trepair corresponds respectively to the time of next failure, and the repair time, each chosen according to an exponential distribution with parameter lambdaFail and lambdaRepair, respectively. Additionally, the failure rate increases with usage due to wear and tear. This continues until a replacement is performed, which resets the rate.

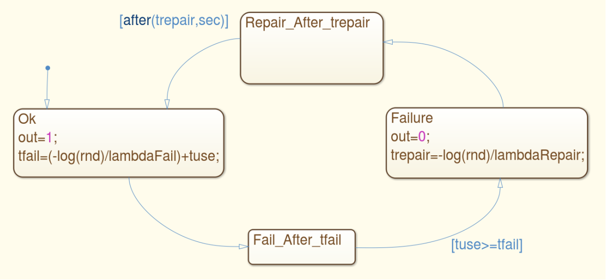

Fig. 5: Failure generator

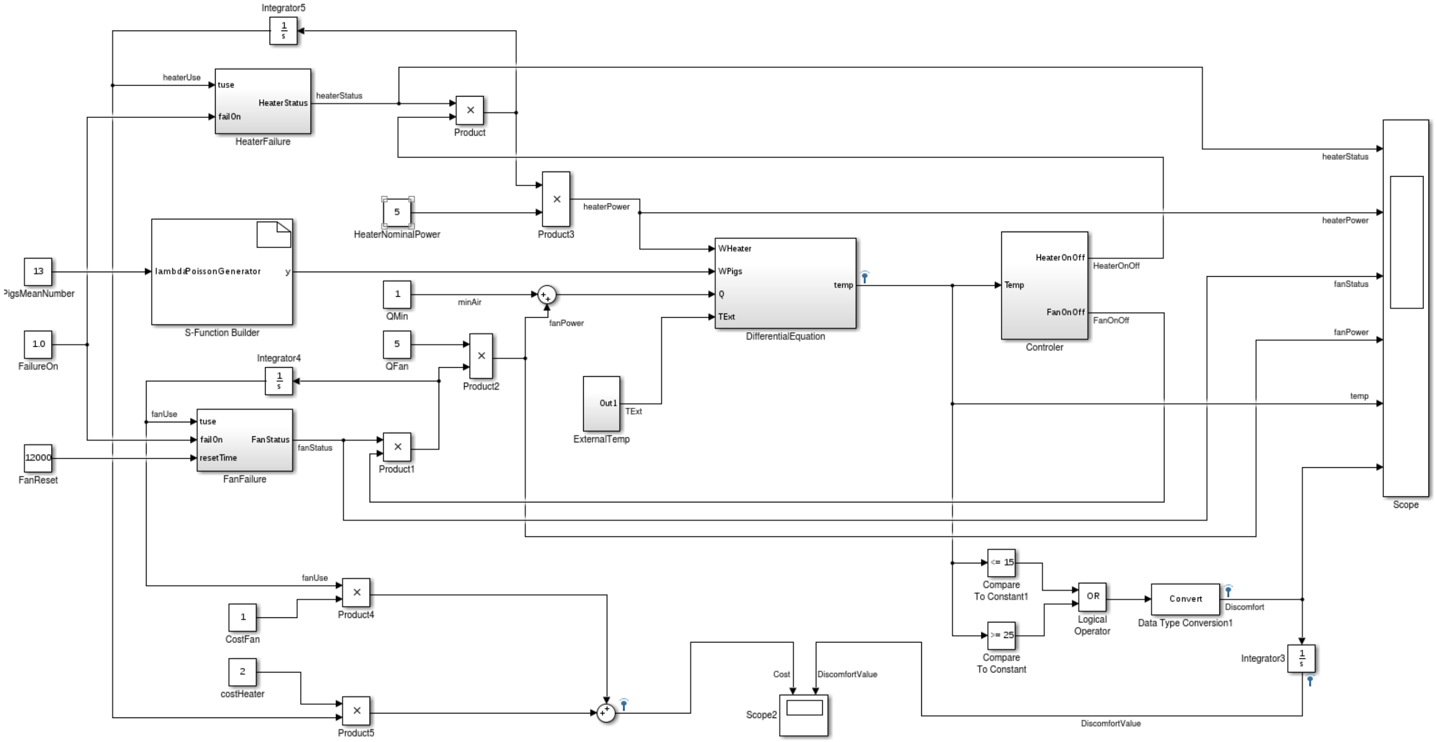

Fig. 6: Pig shed Simulink model

An overview of the complete Simulink model is shown in Fig. 6.

7.1 Quantitative Verification and Optimization

The controller goal is to maintain the temperature within a comfort zone specified by a minimum and a maximum temperature (resp. $\mathsf{T}_{\mathsf{min}} = 15\,°\mathrm{C}$ and $\mathsf{T}_{\mathsf{max}} = 25\,°\mathrm{C}$). The system contains a predicate Discomfort that is true when the temperature of the system is outside this comfort zone. We first consider the following values for the controller thresholds: $\mathsf{TFanOn} = 22\,°\mathrm{C}$, $\mathsf{TFanOff} = 20\,°\mathrm{C}$, $\mathsf{THeaterOn} = 18\,°\mathrm{C}$ and $\mathsf{THeaterOff} = 20\,°\mathrm{C}$.

We apply statistical model checking to evaluate the efficiency of the controller both in the presence and absence of failures. The first BLTL property that we monitor checks that the system is never in discomfort for an excessive period of time. This is expressed by the following property:

$$\Phi_1 = G_{\leq t_1} F_{\leq t_2} \neg \mathsf{Discomfort}$$

where t_1 is the simulation time, t_2 is the accepted discomfort time. Another safety specification is to check if there exists long periods without discomfort. This is possible with:

$$\Phi_2 = F_{\leq t_1} G_{\leq t_2} \neg \mathsf{Discomfort}$$

Finally, a third BLTL property checks that each period of discomfort is followed by a period without discomfort:

$$\Phi_3 = G_{\leq t_1} \Big(G_{\leq t_2} \mathsf{Discomfort} \Rightarrow F_{\leq t_3} (G_{\leq t_4} \neg \mathsf{Discomfort}) \Big)$$

Here t_1 and t_2 are as previously, while $t_3 \geq t_2$ is the expected time at which the system returns to normal situation, and t_4 is the duration of the period without discomfort.

We use Plasma Lab to estimate the probability to satisfy these properties for different values of the timing constraints, on both models with and without failures. Each property is evaluated over a period of time $t_1 = 12000$ time units (t.u.) with precision $\epsilon = 0.01$ and confidence $\delta = 0.01$. Φ_1 and Φ_2 are evaluated for several values of t_2. Note that for $t_2 = 0$, Φ_1 resumes to checking $G_{\leq t_1} \neg \mathsf{Discomfort}$. Φ_3 is evaluated with $t_2 = 25$ *t.u.* and several values of t_3 and t_4.

The results for properties Φ_1 and Φ_2 are presented in Figs. 7 and 8, respectively. While the probabilities of satisfying Φ_1 show a significant difference between the models with and without failures, the results for Φ_2 are almost identical. This means that discomfort is as frequent in the two models, but it tends to last longer in the presence of failures. The results for Φ_3 are presented in Figs. 9 and 10. It shows again that the model without failures recovers quicker from a discomfort period.

Instead of estimating a probability using SMC techniques, we can compute the average value of two quantities in the model, namely the *discomfort time*, that is the cumulative time when the model is in a discomfort state, and the

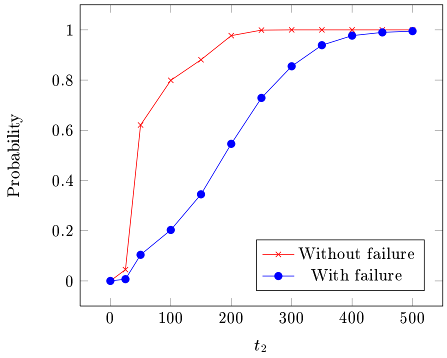

Fig. 7: Probability estimation with SMC of satisfying Φ_1

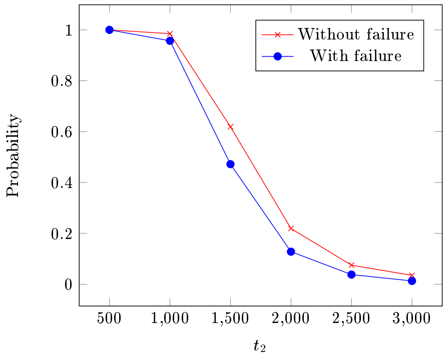

Fig. 8: Probability estimation with SMC of satisfying Φ_2

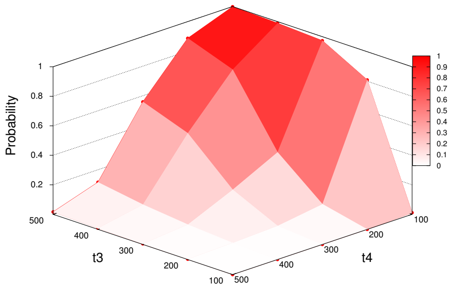

Fig. 9: Probability estimation with SMC of satisfying Φ_3 without failures

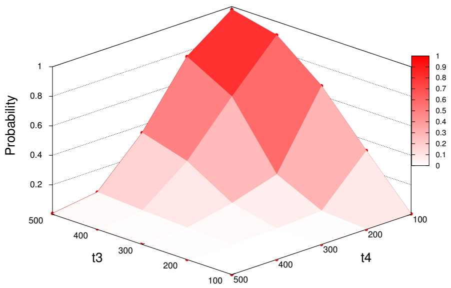

Fig. 10: Probability estimation with SMC of satisfying Φ_3 with failures

energy *cost*, computed with the duration of use of the heater and the fan. The cost is 1 per t.u. for the fan and 2 per t.u. for the heater. We aim at minimizing these two values by choosing adequate values of the model parameters.

Using Plasma Lab we can automatically instantiate the model with a range of values for the four temperature thresholds. We specify the ranges $[15, 20]$ for $\mathsf{THeaterOn}$ and $\mathsf{THeaterOff}$, and $[20, 25]$ for TFanOn and $\mathsf{TFanOff}$, with an increment of 1. We additionally specify the following constraints to select a subset of the possible values of the parameters:

$$\begin{aligned} \mathsf{TFanOff} &< \mathsf{TFanOn} \\ \mathsf{THeaterOn} &< \mathsf{THeaterOff} \\ \mathsf{THeaterOn} &< \mathsf{TFanOn} \end{aligned}$$

Using these constraints Plasma Lab generates a set of 225 possible configurations, for each variant of the models, with and without failures. Each configuration is automatically analyzed with 100 simulations. We then plot the average values of the cost and the discomfort in Fig. 11 and Fig. 12. These graphs helps to select the best values of the parameters by looking at the points that lie on the Pareto frontier of the data.

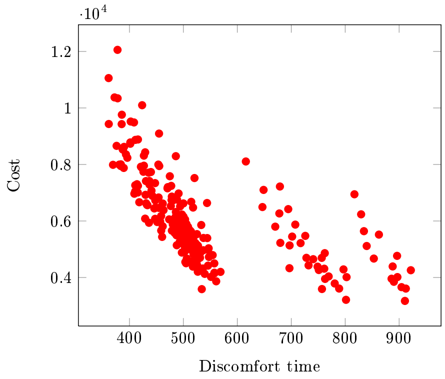

Fig. 11: Optimization of the thresholds parameters without failures

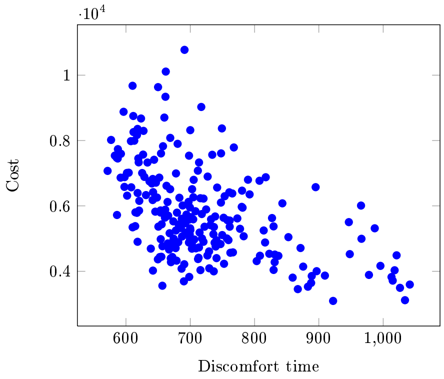

Fig. 12: Optimization of the thresholds parameters with failures

7.2 Change Detection: Detection and Calibration

In our pig shed, the equipment may sometimes fail (heater or fan may break). In such situation, the shed may be too frequently in the discomfort zone, which may lead to the death of several pigs.

As we have seen, the probability of being in the discomfort zone is nominally very low. However, to avoid problems, one should be able to rise a flag as soon as the probability to be in the discomfort zones crosses a given threshold. Our objective is to detect that when such a change happens, there is a maintenance procedure that moves the shed out of the discomfort zone. In our example, this maintenance feature is modeled as a procedure that is regularly applied to the pig shed. Initially, the time between each maintenance is set to a very large value (500000 $t.u.$). The final objective is to set this time value in order to have an acceptable maintenance delay when the death risk is too heavy for the pigs (emergent behavior). This will be done by detecting changes.

We modeled the property using the change property language we proposed and we used the CUSUM algorithm to check it. We first define τ to be the following change: " the probability to be in the discomfort zone more than $t_1 = 100$ $t.u.$ is greater than 0.35". We are now ready to propose a property that expresses that when the change occurs, then the maintenance must be done in less than $t_2 = 1000$ $t.u.$ Formally,

$$\phi_4 = \left| \begin{array}{ll} \texttt{let} & \tau = \; Pr\big[\pi \models G_{\leq t_1}\mathsf{Discomfort}\big] \geq 0.35 \\ \texttt{in} & \tau! \Rightarrow F_{\leq \tau! + t_2}\mathsf{Reparation} \end{array} \right.$$

In order to perform the analysis, the CUSUM algorithm needs a calibration step. We first require an estimate of p_{init}, the initial probability of being in the discomfort zone before the change occurs, and we determine a minimum delay between the samples that ensures independence between the analyses. We disable failures of the temperature regulation system (fans + heaters) in the shed model and we simulate a 200000 $t.u.$ long trace. We sample the trace with a fixed delay between each sample. For each sample we perform a Monte Carlo analysis of the property $G_{\leq t_1}\mathsf{Discomfort}$ by restarting 600 simulations from the initial state of the sample. For sample delays lower than 100 $t.u.$, the probabilities computed for each sample differ, but they converge to 0 (with a precision 0.05 and a confidence 0.9) for the delays 150 $t.u.$ and 200 $t.u.$ Therefore we will select a sample delay of 200 $t.u.$ and an initial probability of $p_{init} = 0.05$ for the CUSUM analysis.

Next step is to set the stopping sensitivity λ on which depends the false alarm probability and the detection delay. This is done again by observing the model without failures: we simulate 100 executions of the CUSUM and observe 1000 samples during each execution. We compute for each samples the CUSUM cumulative ratio. Since there is no failure, the curve of the cumulative ratio should always decrease. Indeed, it should only increase when failures happen, *i.e.*, when the change happens. In practice, even without failure, the curve may locally increase for a short amount of time, which is due to the uncertainty introduced in the model. The objective is to characterize those local drifts to avoid false alarms.

To do so we analyze the CUSUM simulations and we observed that the mean time between positive drifts is 127.88 $t.u.$ and the mean duration of positive drift is 1.2 $t.u.$. The frequency of positive drifts is thus $1.2/(127.88 + 1.2)$, which is in the interval $[0, 0.05]$ as predicted by Monte Carlo algorithm. In order to observe a real alarm one needs to push this quotient to 0.35, which is the probability one wants to observe. This amounts to varying the duration of a positive sample, *i.e.*, to replace 1.2 by a higher value in the above quotient. Doing so, we conclude that the probability will become greater than 0.35 when the positive drift is longer that 52 samples. From the definition of CUSUM, we compute that the drift is $\ln\frac{0.35}{0.05}$ for each positive sample. We finally set the stopping rule to $\lambda = 52 * \ln\frac{0.35}{0.05} \approx 101$.

We then launched the CUSUM on the model with failures over an execution of 200000 $t.u.$ that is checked against the property $G_{\leq t_1}\mathsf{Discomfort}$ every 200 $t.u.$ Figure 13 displays the values obtained with Plasma Lab for the CUSUM cumulative ratio and the minimum value reached. From these values Plasma Lab detected that the stopping rule was satisfied after the sample 580, that corresponds to the simulation time 105837 $t.u.$ We reproduced the same experiment several times (20): we determined that the change occurred at 115104 $t.u.$ in average and in earlier at 101847 $t.u.$ We conclude that to satisfy Property ϕ_4 the maintenance operation must be scheduled at 100000 $t.u.$

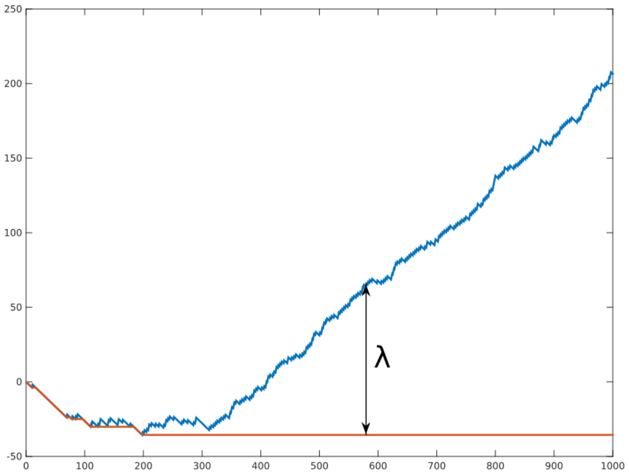

Fig. 13: CUSUM monitoring of $G_{\leq t_1}\mathsf{Discomfort}$. CUSUM cumulative ratio (in blue) and minimum value reached (in red). The horizontal axis is the simulation time in samples number and the vertical axis is the value of the CUSUM ratio. The black arrows denote the time of the change when the cumulative ratio exceeds the minimum value by λ.

8 Conclusion

The paper presents two modest contributions to SMC. The first contribution takes the form of an algorithm used to detect changes on the probability to satisfy a bounded property at runtime. The second contribution illustrates the power of Plasma Lab via a Simulink library integration. This integration constitutes one of the first proof of concept that SMC can indeed be integrated as feature library in a tool largely used in industry.

Future work include an integration of Plasma Lab with the FMI standard in order to verify complex heterogeneous systems. Another future work is to extend the power of distributed computing to Plasma Lab/Simulink. The latter is technically challenging as it would require to duplicate compiled code to avoid license duplication and costs.

References

1. Agrawal, A., Simon, G., Karsai, G.: Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. Electron. Notes Theor. Comput. Sci. 109, 43–56 (Dec 2004)
2. Barnat, J., Beran, J., Brim, L., Kratochvíla, T., Ročkai, P.: Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs. In: Formal Methods

for Industrial Critical Systems, Lecture Notes in Computer Science, vol. 7437, pp. 78–92. Springer (2012)

3. Basseville, M., Nikiforov, I.V.: Detection of Abrupt Changes: Theory and Application. Prentice-Hall, Inc. (1993)
4. Biere, A., Heljanko, K., Junttila, T.A., Latvala, T., Schuppan, V.: Linear Encodings of Bounded LTL Model Checking. Logical Methods in Computer Science 2(5) (2006)
5. Boyer, B., Corre, K., Legay, A., Sedwards, S.: PLASMA-lab: A Flexible, Distributable Statistical Model Checking Library. In: Proceedings of the 10th International Conference on Quantitative Evaluation of Systems (QEST). Lecture Notes in Computer Science, vol. 8054, pp. 160–164. Springer (2013)
6. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Proceedings of the 14th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 2404, pp. 359–364. Springer (2002)
7. Clarke, E.M., Donzé, A., Legay, A.: Statistical Model Checking of Mixed-Analog Circuits with an Application to a Third Order Delta-Sigma Modulator. In: Proceedings of the 3rd Haifa Verification Conference (HVC). Lecture Notes in Computer Science, vol. 5394, pp. 149–163. Springer (2008)
8. Clarke, E.M., Faeder, J.R., Langmead, C.J., Harris, L.A., Jha, S.K., Legay, A.: Statistical Model Checking in BioLab: Applications to the Automated Analysis of T-Cell Receptor Signaling Pathway. In: Proceedings of the 6th International Conference on Computational Methods in Systems Biology (CMSB). Lecture Notes in Computer Science, vol. 5307, pp. 231–250 (2008)
9. David, A., Du, D., Larsen, K.G., Legay, A., Mikucionis, M., Poulsen, D.B., Sedwards, S.: Statistical Model Checking for Stochastic Hybrid Systems. In: Proceedings of the 1st International Workshop on Hybrid Systems and Biology (HSB). EPTCS, vol. 92, pp. 122–136 (2012)
10. David, A., Du, D., Larsen, K.G., Legay, A., Mikucionis, M.: Optimizing Control Strategy Using Statistical Model Checking. In: Proceedings of the 5th International Symposium NASA Formal Methods (NFM). Lecture Notes in Computer Science, vol. 7871, pp. 352–367. Springer (2013)
11. David, A., Larsen, K.G., Legay, A., Mikucionis, M., Wang, Z.: Time for Statistical Model Checking of Real-Time Systems. In: Proceedings of the 23rd International Conference on Computer Aided Verification (CAV). vol. 6806, pp. 349–355. Springer (2011)
12. Devesh Bhatt, Gabor Madl, David Oglesby, Kirk Schloegel: Towards Scalable Verification of Commercial Avionics Software. In: AIAA Infotech@Aerospace 2010. Infotech@Aerospace Conferences, American Institute of Aeronautics and Astronautics (Apr 2010)
13. Grabiec, B., Traonouez, L., Jard, C., Lime, D., Roux, O.H.: Diagnosis Using Unfoldings of Parametric Time Petri Nets. In: Proceedings of the 8th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS). Lecture Notes in Computer Science, vol. 6246, pp. 137–151. Springer (2010)
14. Havelund, K., Rosu, G.: Synthesizing Monitors for Safety Properties. In: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 2280, pp. 342–356. Springer (2002)

15. Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate Probabilistic Model Checking. In: Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), Lecture Notes in Computer Science, vol. 2937, pp. 73–84. Springer (2004)
16. Jégourel, C., Legay, A., Sedwards, S.: Importance Splitting for Statistical Model Checking Rare Properties. In: Proceedings of the 25th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 8044, pp. 576–591. Springer (2013)
17. Jessen, J.J., Rasmussen, J.I., Larsen, K.G., David, A.: Guided Controller Synthesis for Climate Controller Using Uppaal Tiga. In: Proceedings of the 5th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS). Lecture Notes in Computer Science, vol. 4763, pp. 227–240. Springer (2007)
18. Jha, S.K., Clarke, E.M., Langmead, C.J., Legay, A., Platzer, A., Zuliani, P.: A Bayesian Approach to Model Checking Biological Systems. In: Proceedings of the 7th International Conference on Computational Methods in Systems Biology (CMSB). Lecture Notes in Computer Science, vol. 5688, pp. 218–234. Springer (2009)
19. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Proceedings of the 23rd International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer (2011), http://www.prismmodelchecker.org
20. Legay, A., Delahaye, B., Bensalem, S.: Statistical Model Checking: An Overview. In: Proceedings of the 1st International Conference on Runtime Verification (RV). Lecture Notes in Computer Science, vol. 6418, pp. 122–135. Springer
21. Mathworks: Polyspace a static analysis tools for C/C++ and Ada (Dec 2014), http://www.mathworks.fr/products/polyspace/
22. Meenakshi, B., Bhatnagar, A., Roy, S.: Tool for Translating Simulink Models into Input Language of a Model Checker. In: Formal Methods and Software Engineering, Lecture Notes in Computer Science, vol. 4260, pp. 606–620. Springer (2006)
23. Page, E.S.: Continuous inspection schemes. Biometrika 41(1/2), 100–115 (1954), http://www.jstor.org/stable/2333009
24. Prover: Prover-Plugin (Dec 2014), http://www.prover.com/products/prover_plugin/
25. Sen, K., Viswanathan, M., Agha, G.: Statistical Model Checking of Black-Box Probabilistic Systems. In: Proceedings of the 16th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 3114, pp. 202–215. Springer (2004)
26. Verdier, G., Hilgert, N., Vila, J.: Adaptive threshold computation for CUSUM-type procedures in change detection and isolation problems. Computational Statistics & Data Analysis 52(9), 4161–4174 (2008)
27. Wald, A.: Sequential Tests of Statistical Hypotheses. The Annals of Mathematical Statistics 16(2), 117–186 (1945)
28. Younes, H.L.S.: Verification and Planning for Stochastic Processes with Asynchronous Events. Ph.D. thesis, Carnegie Mellon (2005)
29. Zuliani, P., Platzer, A., Clarke, E.M.: Bayesian statistical model checking with application to Stateflow/Simulink verification. Formal Methods in System Design 43(2), 338–367 (2013)