https://doi.org/10.1007/s00165-021-00563-2
BCS © 2021
Formal Aspects of Computing (2021) 33: 1147–1172

Formal Aspects of Computing

Statistical model checking for variability-intensive systems: applications to bug detection and minimization

Maxime Cordy[1], Sami Lazreg[1], Mike Papadakis[1] and Axel Legay[2]

[1]SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg
[2]Université Catholique de Louvain, Ottignies-Louvain-la-Neuve, Belgium

Abstract. We propose a new Statistical Model Checking (SMC) method to identify bugs in variability-intensive systems (VIS). The state-space of such systems is exponential in the number of variants, which makes the verification problem harder than for classical systems. To reduce verification time, we propose to combine SMC with featured transition systems (FTS)—a model that represents jointly the state spaces of all variants. Our new methods allow the sampling of executions from one or more (potentially all) variants. We investigate their utility in two complementary use cases. The first case considers the problem of finding all variants that violate a given property expressed in Linear-Time Logic (LTL) within a given simulation budget. To achieve this, we perform random walks in the featured transition system seeking accepting lassos. We show that our method allows us to find bugs much faster (up to 16 times according to our experiments) than exhaustive methods. As any simulation-based approach, however, the risk of Type-1 error exists. We provide a lower bound and an upper bound for the number of simulations to perform to achieve the desired level of confidence. Our empirical study involving 59 properties over three case studies reveals that our method manages to discover all variants violating 41 of the properties. This indicates that SMC can act as a coarse-grained analysis method to quickly identify the set of buggy variants. The second case complements the first one. In case the coarse-grained analysis reveals that no variant can guarantee to satisfy an intended property in all their executions, one should identify the variant that minimizes the probability of violating this property. Thus, we propose a fine-grained SMC method that quickly identifies promising variants and accurately estimates their violation probability. We evaluate different selection strategies and reveal that a genetic algorithm combined with elitist selection yields the best results.

Keywords: Statistical model checking, Variability, Verification, Simulation, Sampling

1. Introduction

We consider the problem of bug detection in Variability Intensive Systems (VIS). This category of systems encompasses any system that can be derived into multiple variants (differing, e.g., in provided functionalities), including software product lines [CN01] and configurable systems [SW98]. Compared to traditional ("single") systems, the complexity of bug detection in VIS is increased: bugs can appear only in some variants, which requires analysing the peculiarities of each variant.

Correspondence to: Maxime Cordy., E-mail: maxime.cordy@uni.lu

Among the number of techniques developed for bug detection, one finds testing and model checking. Testing [BJK+05] executes particular test inputs on the system and checks whether it triggers a bug. Albeit testing remains widely used in industry, the rise of concurrency and inherent system complexity has made system-level test case generation a hard problem. Also, testing is often limited to bounded reachability properties and cannot assess liveness properties.

Model checking [BK08] is a formal verification technique which checks that all behaviours of the system satisfy specified requirements. These behaviours are typically modelled as an automaton, where each node represents a state of the system (e.g. a valuation of the variables of a program and a location in this program's execution flow) and where each transition between two states expresses that the program can move from one state to the other by executing a single action (e.g. executing the next program statement). Requirements are often expressed in temporal logics, e.g. the Linear Temporal Logic (LTL) [Pnu77].

Such logics capture both safety and liveness properties of system behaviours. As an example, consider the LTL formula $\Box(command\_sleep \Rightarrow \Diamond system\_sleep)$. $command\_sleep$ and $system\_sleep$ are logic atoms and represent, respectively, a state where the `sleep` command is input and another state where the system enters sleep mode. The symbols $\Box$ and $\Diamond$ means *always* and *eventually*, respectively. Thus, the whole formula expresses that "it is always the case that when the `sleep` command is input, the system eventually enters sleep mode".

Contrary to testing, model checking is exhaustive: if a bug exists, then the checking algorithm outputs a *counterexample*, i.e. an execution trace of the system that violates the verified property. Exhaustiveness makes model checking an appealing solution to obtain strong guarantees that the system works as intended. It can also nicely complement testing (whose main advantage remains to be applied directly on the running system), e.g. by reasoning over liveness properties or by serving as oracle in test generation processes [ABM98]. Those benefits, however, come at the cost of scalability issues, the most prominent being the *state explosion problem*. This term refers to the phenomenon where the state space to visit is so huge that an exhaustive search is intractable. As an illustration of this, let us remark that the theoretical complexity of the LTL model checking problem is PSPACE-complete [VW86].

Model checking complexity is further exacerbated when it comes to VIS. Indeed, in this case, the model checking problem requires verifying whether *all* the variants satisfy the requirements [CCS+13]. This means that, if the VIS comprises n variation points (n features in a software product line or n Boolean options in a configurable system), the number of different variants to represent and to check can reach 2^n. This exponential factor adds to the inherent complexity of model checking. Thus, checking each variant (or models thereof) separately—an approach known as *enumerative* or *product-based* [TAK+14]—is often intractable. To alleviate this, variability-aware models and companion algorithms were proposed to represent and check efficiently the behaviour of all variants at once. For instance, *Featured Transition Systems* (FTS) [CCS+13] are transition systems where transitions are labelled with (a symbolic encoding of) the set of variants able to exercise this transition. The structure of FTS, if well constructed, allows one to capture in a compact manner commonalities between states and transitions of several variants. Exploiting that information, *family-based* algorithms can check only once the executions that several variants can execute and explore the state space of an individual variant only when it differs from all the others. In spite of positive improvements over the enumerative approach, state-space explosion remains a major challenge.

In this work, we propose an alternative technique for state-space exploration and bug detection in VIS. We use Statistical Model Checking (SMC) [LDB10] as a trade-off between testing and model checking to verify properties (expressed in full LTL) on FTS. The core idea of SMC is to conduct some simulations (i.e. sample executions) of the system (or its model) and verify if these executions satisfy the property to check. The results are then used together with statistical tests to decide whether the system satisfies the property with some degree of confidence. Of course, in contrast with an exhaustive approach, a simulation-based solution does not guarantee a result with 100% confidence. Still, it is possible to bound the probability of making an error. Simulation-based methods are known to be far less memory- and time-consuming than exhaustive ones and are sometimes the only viable option. Over the past years, SMC has been used to, e.g. assess the absence of errors in various areas from aeronautic to systems biology; measure cost average and energy consumption for complex applications such as nanosatellites; or detect rare bugs in concurrent systems [JLS13, CIM+13, LL18].

In the context of VIS, we propose to use SMC in order to support two types of analyses. The first type is a coarse-grained analysis to identify the set of buggy variants and give some confidence that the other variants do

not include bugs—all this without checking the full set of their behaviours. Thus, this first use case considers the problem of finding all variants that violate a given LTL formula. Starting from the FTS modelling all variants' behaviours, we apply a *family-based* SMC method to sample executions from all variants at the same time. Doing so, our method avoids sampling twice (or more) executions that exist in multiple variants. Merging the individual state spaces biases the results, though, as it changes the probability distribution of the executions. This makes the problem different from previous methods intended for single systems (e.g. [GS05]) and obliges us to revisit the fundamentals of SMC in the light of VIS. In particular, we want to characterize the number of samples executions required to bound the probability of Type-1 error by a desired degree of confidence. We provide a lower bound and an upper bound for this number by reducing its computation to particular instances of the coupon problem [BH97].

The second type of analysis is a fine-grained analysis that complements the coarse-grained approach. If the coarse-grained analysis reveals that no variant can guarantee to satisfy an intended property in all their executions, it is interesting for engineers to identify the variant that minimizes the probability of violating this property. Such use cases typically occur, e.g., in security systems where there is a trade-off to find between fixing vulnerabilities and the cost of implementing these fixes [WKCK15], or in cyber-physical systems deployed in an uncontrollable environment [BS20]. While such systems cannot guarantee to meet their intended requirements perfectly, it is desirable to minimize their probability of failures. SMC can contribute to this analysis and provide, for reasonable computation costs, an accurate estimation of such probability of failures.

Our second use case generalizes this problem to VISs. Thus, it concerns the identification of the variants that minimize the probability of violating an LTL formula that no other variant can guarantee to satisfy. To address this case, we propose an SMC method that, given a limited simulation budget, samples executions from the variants individually in order to identify the most promising ones quickly. Then, our method conducts additional simulations of these promising variants to compute an accurate estimation of their probability to violate the formula. Due to the aforementioned complexity blow-up inherent to variability, in practice it is not feasible to simulate all variants, a fortiori not without a sufficient simulation budget to provide accurate estimates. Therefore, our method adopts an iterative process to distribute the simulation budget among a selected set of variants. The challenge here is i) balancing the exploration of a large set of variants and ii) exploiting available results to prioritize promising variants. We investigate and compare different selection strategies (i.e., random and based on genetic algorithms) with different parameterizations.

We implemented our methods on top of ProVeLines [CSHL13b], a model checker for VIS. We carry out experiments over the two use cases, based on 3 case studies totalling 59 properties to check. Regarding the first use case, we provide evidence that family-based SMC is a viable approach to verify VIS. Our study shows that our method manages to find all buggy variants in 41 properties and does so up to 16 times faster than state-of-the-art model checking algorithms for VIS [CCS+13]. Moreover, our approach can achieve a median bug detection rate 3 times higher than classical SMC applied to each variant individually.

As for the second use case, we only consider case studies where no variant satisfies the given property resulting in a subset of 2 case studies totalling 12 properties to check. Given a limited budget for simulations, we observe that Genetic Algorithm (GA) combined with SMC and elitism is the most effective approach for identifying suitable variants of VIS (i.e. those with the lowest violation probability). On average, our methods finds a variant that is amongst the top 11% and 15% (depending on the considered case study).

This paper extends our previous work published in the 23rd edition of the Fundamental Approaches to Software Engineering conference ("FASE 2020"). This extension improves the original material with clarified details regarding the first use case. More importantly, the extension adds the second use case, which the conference paper did not consider. The rest of this paper is structured as follows. Section 2 recapitulates essential background and related work. Sections 3 and 4 presents our methods for each respective use case. Section 5 describes our empirical study and Sect. 6 presents its results. We conclude in Sect. 7.

2. Background on model checking

In model checking, the behaviour of the system is often represented as a transition system (S, Δ, AP, L) where S is a set of states, $\Delta \subseteq S \times S$ is the transition relation, AP is a set of atomic propositions[1] and $L : S \to 2^{AP}$ labels any state with the atomic propositions that the system satisfies when in such a state.

[1] Atomic propositions can be seen as basic observable properties of the system state.

2.1. Linear temporal logic

LTL is a temporal logic that allows specifying desired properties over all future executions of some given system. Given a set AP of atomic propositions, an LTL formula ϕ is formed according to the following grammar: $\phi ::= \top \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\, \phi_1 \mid \bigcirc \phi_1 \mid \phi_1 \mathrm{U} \phi_2$ where ϕ_1 and ϕ_2 are LTL formulae, $a \in AP$, $\bigcirc$ is the next operator and U is the until operator. We also define $\Diamond \phi$ (*"eventually"* ϕ) and $\Box \phi$ (*"always"* ϕ) as a shortcut for $\top \mathrm{U} \phi$ and $\neg\, \Diamond \neg\, \phi$, respectively.

Vardi and Wolper have presented an automata-based approach for checking that a system—modelled as a transition system ts—satisfies an LTL formula ϕ [VW86]. Their approach consists of, first, transforming ϕ into a Büchi automaton $\mathcal{B}_{\neg\, \phi}$ whose language is exactly the set of executions that violate ϕ, that is, those that visit infinitely often a so-called *accepting* state. Such execution σ takes the form of a *lasso*, i.e. $\sigma = q_0 \dots q_n$ with $q_j = q_n$ for some j and where q_i is accepting for some $i : j \leq i \leq n$. We name *accepting* any such lasso whose cycle contains an accepting state.

The second step is to compute the synchronous product of ts and $\mathcal{B}_{\neg\, \phi}$, which results in another Büchi automaton $\mathcal{B}_{ts \otimes \neg\, \phi}$. Any accepting lasso in $\mathcal{B}_{ts \otimes \neg\, \phi}$ represents an execution of the system that violates ϕ. Thus, Vardi and Wolper's algorithm comes down to checking the absence of such accepting lasso in the whole state space of $\mathcal{B}_{ts \otimes \neg\, \phi}$. The size of this state space is $\mathcal{O}(\mid ts \mid \times \mid 2^{|\phi|} \mid)$ and the complexity of this algorithm is PSPACE-complete.

2.2. Model checking for variability-intensive systems

Applying classical model checking to VIS requires iterating over all variants, construct their corresponding automata $\mathcal{B}_{ts \otimes \neg\, \phi}$ and search for accepting lasso in each of these. This enumerative method (also named *product-based* [TAK+14]) fails to exploit the fact that variants have behaviour in common.

As an alternative, researchers came up with models able to capture the behaviour of multiple variants and distinguish between the unique and common behaviour of those variants [CDEG03, CCS+13, tBFGM16]. Among such models, we focus on *featured transition systems* [CCS+13] as those can link an execution to the variants able to execute it more directly than the alternative formalisms. In a nutshell, FTS extend the standard transition system by labelling each transition with a symbolic encoding of the set of variants able to exercise this transition. Then, the set of variants that can produce an execution π is the intersection of all sets of variants associated with the transitions in π.

To check which variants violate a given LTL formula ϕ, one can adapt the procedure of Vardi and Wolper and build the synchronous product of the featured transition system with $\mathcal{B}_{\neg\, \phi}$ [CCS+13]. This product is similar to the Büchi automaton obtained in the single system case, except that its transitions are also labelled with a set of variants.[2] Then, the buggy variants are those that are able to execute the accepting lassos of this automaton. This generalized automaton is the fundamental formalism we work on in this paper.

Definition 1 Let V be a set of variants. A *Featured Büchi Automaton* (FBA) over V is a tuple $(Q, \Delta, Q_0, A, \Theta, \gamma)$ where Q is a set of states, $\Delta \subseteq Q \times Q$ is the transition relation, $Q_0 \subseteq Q$ is a set of initial states, $A \subseteq Q$ is the set of accepting states, Θ is the whole set of variants, and $\gamma : \Delta \to 2^{\Theta}$ associates each transition with the set of variants that can execute it.

Figure 1 shows an FBA with two variants and eight states. State 5 is the only accepting state. Both variants can execute the transition from State 3 to State 4, whereas only variant v_2 can move from State 3 to State 6.

The Büchi automaton corresponding to one particular variant v is derived by removing the transitions not executable by v. That is, we remove all transitions $(q, q') \in \Delta$ such that $v \notin \gamma(q, q')$. The resulting automaton is named the *projection* of the FBA onto v. For example, one obtains the projection of the FBA in Fig. 1 onto v_2 by removing the transition from State 3 to State 7 and those between State 7 and State 8.

[2] Those labels are equal to those found in the corresponding transitions of the featured transition system.

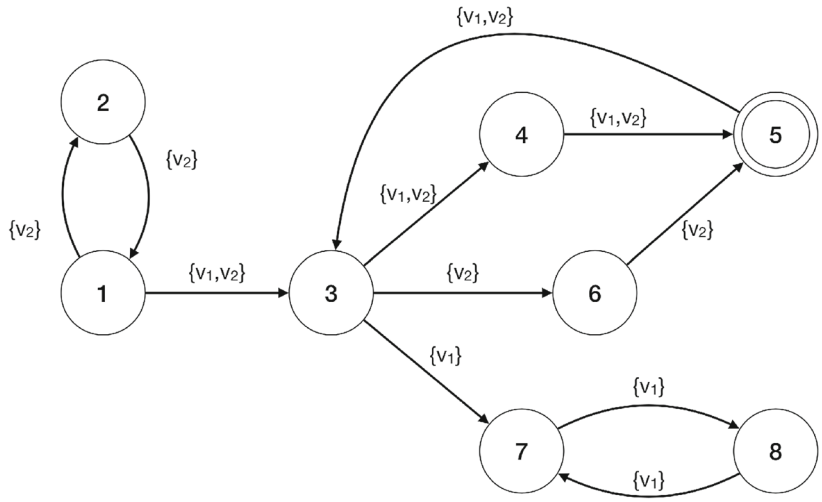


Fig. 1. An example of FBA with two variants.

2.3. Statistical model checking

Originally, SMC was used to compute the probability to satisfy a bounded LTL property for stochastic system [YS02]. The idea was to monitor the properties on bounded executions represented by Bernoulli variables and then use Monte Carlo to estimate the resulting property. SMC also applies to non-stochastic systems by assuming an implicit uniform probability distribution on each state successor.

Grosu and Smolka [GS05] lean on this and propose an SMC method to address the full LTL model-checking problem. Their sampling algorithm walks randomly through the state space of $\mathcal{B}_{ts\otimes\neg\,\phi}$ until it finds a lasso. They repeat the process M times and conclude that the system satisfies the property if and only if none of the M lassos is accepting. They also show that, given a confidence ratio δ and assuming that the probability p for an execution of the system exceeds an error margin ϵ, setting $M = \frac{\delta}{1-\epsilon}$ bounds the probability of a Type-1 error (rejecting the hypothesis that the system violates the property while it actually violates it) by δ. Thus, M can serve as a minimal number of samples to perform. Our work extends theirs in order to support VIS instead of single systems. Other work on applying SMC to the full LTL logic can be found in [DHKP17, YCZ10].

2.4. Related work

2.4.1. VIS verification

There are numerous models proposed to VIS verification. For instance, Classen et al. proposed *Featured Transition System* [CCS+13] (FTS) formalism which is an automata-based model that relies on transitions labelled by a features expression. Consequently, this formalism determine which variants can exercice the transition. Using this information, the fact that variants have behaviour in common could be exploited, leading to significant speedup in terms of verification time. Accordingly, Classen et al. proposed variability-aware *exhaustive* algorithms to model-check FTS. By contrast, our approach samples execution traces from FTS to reduce the verification effort.

In addition to FTS, other models have been extended to capture the behaviour of multiple variants such as modal transition system [tBFGM16], product-line-CCS [GLS08], featured Petri-nets [MCP10]. Each formalism has a different syntax and semantics. Modal transition systems or modal I/O automata use optional "may" transitions to model variability. Similarly, product-line-CSS is a process algebra with alternative choice operators between two process to model the behavior of a set of variants. All these approaches are valid solutions for VIS verification. However, most of them are isolated efforts that do not come with mature companion tools. Our work is, therefore, based on FTS. Ter Beek et al.'s [tBFGM16] solution based on modal transition systems is another mature approach. However, it requires the use of a separate logic to link variants to their behaviour in the model, which we found to be less practical than the explicit variability information contained in FTS. This information makes it easy and efficient to determine the variants that can execute a given buggy behaviour [CHS+10].

2.4.2. SMC for VIS

Recent work has applied SMC in the context of VIS. Vandin et al. [VtBLL18, tBLLV20], proposed an algebraic language to formally model behaviour with dynamic variability (i.e. where the system can adapt its configuration during its execution). Vandin et al. also proposed a product-based SMC approach to check properties on the reconfigurable system. Contrary to this work, our approach assumes static variability (the variants are distinct systems) and relies on family-based algorithms to reason more efficiently on the whole set of variants.

Dubslaff et al. [DKB14] and Nunes et al. [RAN+15] have studied VIS subject to stochastic behaviour and proposed exhaustive model-checking algorithms to check the probabilistic properties of such systems. Because of their exhaustive nature and the inherent computational cost of stochastic model checking approaches, these algorithms suffer from scalability limitations. To overcome this scalability issue, Delahaye et al. applied SMC to stochastic parametric systems [DFL19]. Their approach opens the possibility to verify stochastic VIS, where each variant is a valuation of the parameters. More precisely, Delahaye et al. target the verification of quantitative reachability properties. By contrast, we support non-quantitative but more general properties (expressed in LTL).

2.4.3. Optimizations of VIS

The application of optimization methods to VIS are numerous (see, e.g., [HPHLT15, OSCR12, SRK+12]). These applications generally rely on constraint satisfaction problem, satisfiability modulo theory and integer linear programming. While these methods were effective in their particular context of use,they generally do not scale to large VIS and require complete knowledge of the function to optimize [ORGC14].

Our second use case is an optimization problem—we aim to find the variants that minimize the likelihood that a bug occurs. In this case, the function to optimize is dynamically valued during the simulation of the variants' behaviour. Therefore, we cannot characterize this function a priori. Approximate optimizations fill these gaps by applying meta-search heuristics such as genetic algorithm (GA) [GWW+11] or filtered cartesian flattening [HPHLT15]. Such methods return an approximate solution close to the optimum with more scalability and flexibility. Therefore, meta-heuristics like genetic algorithms are appropriate to solve our use case; these algorithms can explore the search space of a VIS even with partial knowledge—which we build through the simulation of variants' behaviour. We opted for genetic algorithms because they offer a balance between convergence and disruption in the search space. Moreover, they can exploit hidden relationships between the features of the variants [GWW+11] and the probability to minimize. Our experiments reveal that our genetic approach provides good results, though other meta-heuristics could work as well [HJK+14].

2.4.4. Sampling of VIS

Another related, yet different area of research is the sampling of VIS variants (e.g. [OGB19, PAP+19]). Some papers consider the problem of sampling uniformly variants in order to study their characteristics (e.g. performance [KGS+19] and other quality requirements [CLLC19]) and infers the characteristics of the other variants. Others attempt to compute a sample that covers all combinations of VIS variation points [PTR+19]. Recently, Thüm et al. [TvHA+19] survey different strategies for the performance analysis of VIS, including the sampling of variants and family-based test generation, which is based on the same idea of executing test cases common to multiple variants. Contrary to us, such works do not consider temporal/behavioural properties, and most of them perform the sampling based on a static representation of the variant space (i.e. a feature model [KCH+90]). In our second use case (see Sect. 4), we propose to combine such variant sampling techniques with GA and classical (product-based) SMC to check only promising variants of the family.

3. Family-based statistical model checking for bug detection

The problem of bug detection in VIS using model checking can be phrased as follows: given an FBA over a set of variants V, compute the subset $V_{bug} \subseteq V$ such that $v \in V_{bug}$ if and only if v can execute an accepting lasso in the FBA (or equivalently, the projection of the FBA onto v has a reachable accepting lasso).

The purpose of SMC is to reduce the verification effort (when visiting the state space of the system model) by sampling a given number of executions (here, lassos). This gain in efficiency, however, comes at the risk of Type-1 errors.

Input: $fba = (Q, \Delta, Q_0, A, \Theta, \gamma)$
Output: $(\sigma, \Theta_\sigma, accept)$ where σ is a lasso of fba and Θ_σ is the set of the variants able to execute σ and $accept$ is true iff σ is accepting.

1 $q_0 \leftarrow$ pick from Q_0 with probability $\frac{1}{|Q_0|}$;
2 $q \leftarrow q_0$;
3 $\sigma \leftarrow q_0$;
4 $\Theta_\sigma \leftarrow \Theta$;
5 $depth \leftarrow 0$;
6 $a \leftarrow 0$;
7 **while** $hash(q) = \perp$ **do**
8 $\quad depth \leftarrow depth + 1$;
9 $\quad hash(q) \leftarrow depth$;
10 $\quad$ **if** $q \in A$ **then**
11 $\quad\quad a \leftarrow depth$;
12 $\quad$ **end**
13 $\quad Succ_\sigma \leftarrow \{q' \in Q \mid (q, q') \in \Delta \wedge (\gamma(q, q') \cap \Theta_\sigma) \neq \emptyset\}$;
14 $\quad q' \leftarrow$ pick from $Succ_\sigma$ with probability $\frac{1}{|Succ_\sigma|}$;
15 $\quad \sigma \leftarrow \sigma\, q'$;
16 $\quad \Theta_\sigma \leftarrow \Theta_\sigma \cap \gamma(q, q')$;
17 $\quad q \leftarrow q'$;
18 **end**
19 **return** $(\sigma, \Theta_\sigma, hash(q) \leq a)$

Algorithm 1: Random Lasso Sampling

Indeed, while the discovery of a counterexample leads with certainty to the conclusion that the variants able to execute it violate the property ϕ, the fact that the sampling did not find a counterexample for some variant v does not entail a 100% guarantee that v satisfies ϕ. The more lassos we sample, the more confident we can get that the variants without counterexamples satisfy ϕ. Thus, designing a family-based SMC method involves answering three questions: (1) how to sample executions; (2) how to choose a suitable number of executions; (3) what is the associated probability of Type-1 error.

3.1. Random sampling in featured Büchi automata

One can sample a lasso in an FBA by randomly walking through its state space, starting from a randomly-chosen initial state and ending as soon as a cycle is found. A particular restriction is that this lasso should be executable by at least one variant; otherwise, we would sample a behaviour that does not actually exist. The set of variants able to execute a given lasso are those that can execute all its transitions, i.e. the intersection of all $\gamma(q, q')$ met along the transitions of this lasso. More generally, we define the lasso sample space of an FBA as follows.

Definition 2 Let $fba = (Q, \Delta, Q_0, A, \Theta, \gamma)$ be a featured Büchi automaton. The lasso sample space L of fba is the set of executions $\sigma = q_0 \dots q_n$ such that $q_0 \in Q_0$, $(q_i, q_{i+1}) \in \Delta$ for all $0 \leq i \leq n-1$, $(\bigcap_{0 \leq i \leq n-1} \gamma(q_i, q_{i+1})) \neq \emptyset$, $q_j = q_n$ for some $0 \leq j \leq n-1$ and $a \neq b \Rightarrow q_a \neq q_b$ for all $0 \leq a, b \leq n-1$. Moreover, σ is said to be an accepting lasso if $\exists\, q_a \in A$ for some $j \leq a \leq n$.

Algorithm 1 formalises the sampling of lassos in a deadlock-free FBA.[3] After randomly picking an initial state (Line 1), we walk through the state space by randomly choosing, at each iteration, a successor state among those available (Line 7–18). Throughout the search, we maintain the set of variants Θ_σ that can execute σ so far (Line 16). Then, we use this set as a filter when selecting successor states to ensure that σ remains executable by at least one variant. At Line 13, $Succ_\sigma$ is the set of successors q' of q (last state of σ) that can be reached. We stop the search as soon as we reach a previously visited state (Line 7). If this state was visited before the last accepting state, it means that the sampled lasso is accepting (Line 19).

A motivated criticism [ODG+11] of the use of random walk to sample lasso is that shorter lassos receive a higher probability to be sampled. To counterbalance this, we implemented a heuristic named *multi-lasso* [GS05]. It

[3] We assume that no variant may remain stuck in a state without outgoing transition that this variant can execute. Should this happen, we assume that the variant self-loops in the state wherein it is stuck, yielding an immediate lasso.

consists of ignoring backward transitions that do not lead to an accepting lasso if there are still forward transitions to explore. This is achieved by modifying Line 13 such that backward transitions leading to a non-accepting lasso are not considered in the successor set.

Assuming a uniform selection of outgoing transitions from each state, one can compute the probability that a random walk samples any given lasso from the sample space.

Definition 3 The probability $P(\sigma)$ of a lasso $\sigma = q_0 \dots q_n$ is inductively defined as follows: $P[q_0] = \mid Q_0 \mid^{-1}$ and $P[q_0 \dots q_j] = P[q_0 \dots q_{j-1}] \times \mid Succ_{q_0 \dots q_{j-1}} \mid^{-1}$.

In the case of classical Büchi automata, Grosu et al. [GS05] shows that $(L, \mathcal{P}(\mathcal{L}), P)$ defines a probability space. The proof derives from the observation that the lasso sample space contains all non-subsuming finite prefixes of all infinite paths of the automaton. Therefore, one can reduce the problem of sampling from an infinite space of infinite executions to sampling from a finite set of finite prefixes. This proof remains valid for our procedure to sample lassos in an FBA. Indeed, consider the Büchi automaton obtained by removing the feature expressions on the FBA transitions and removing the transitions that no variant can execute. The lasso sample space of this automaton is exactly the lasso sample space from which our procedure samples. Indeed, the only necessary condition for our procedure to sample a lasso is that at least one variant should be able to execute this lasso (see Line 9 of Algorithm 1).

Let us consider an example. In the FBA from Fig. 1, there are two non-accepting lassos ($l_1 = (1, 2, 1)$ and $l_2 = (1, 3, 7, 8, 7)$) and two accepting lassos ($l_3 = (1, 3, 4, 5, 3)$ and $l_4 = (1, 3, 6, 5, 3)$). Both variants can execute lassos l_3, while only v_1 can execute l_2 and only v_2 can execute l_1 and l_4. The probability of sampling l_1 is $\frac{1}{2}$, whereas $P[l_2] = P[l_3] = P[l_4] = \frac{1}{6}$. Thus, the probability of sampling a counterexample executable by v_2 is $\frac{1}{3}$, whereas it is only $\frac{1}{6}$ for v_1.

Next, we characterise the relationship between this probability space and any individual variant v. Let L_v be the set of lassos executable by v. Since $L_v \subseteq L$, the probability p_v to sample such a lasso is $\sum_{\sigma_v \in L_v} P(\sigma)$. Note that p_v can be different from the probability $\hat{p}_v$ of sampling an accepting lasso from the automaton modelling the behaviour of v only (i.e. the projection of the FBA onto v). This is because, in the FBA, the probability of selecting an outgoing transition from a given state is assigned uniformly regardless of the number of variants able to execute that transition. This balance-breaking effect increases more as the variants have different numbers of unique executions.

Let $\sigma = q_0 \dots q_n$ be a lasso in L_v. Then $P_v(\sigma)$ is inductively defined as follows: $P_v[q_0] = P[q_0]$ and $P_v[q_0 \dots q_j] = P_v[q_0 \dots q_{j-1}] \times \mid \{(q_{j-1}, q) \in \Delta_v : q \in Q\} \mid^{-1}$ where $\Delta_v = \{(q, q') \in \Delta : v \in \gamma(q, q')\}$. In our example, $P_{v_1}[l_3] = \frac{1}{2}$, as opposed to $P[l_3] = \frac{1}{6}$. This implies that it is more likely to sample an accepting lasso executable by v_1 from its projection in one trial than it is from the whole FBA in two trials. This illustrates the case where merging the state spaces of the variants can have a negative impact on the capability to find bugs specific to one variant.

Thus, sampling lassos from the FBA allows finding one counterexample executable by multiple products but introduces a bias. Overall, it tends to decrease the probability of sampling lassos from variants that have a smaller state space. This can impact SMC's results and parameter choices, like the number of samples required to get confident results and the associated Type-1 error.

3.2. Hypothesis testing

Remember that addressing the model checking problem for VIS requires to find a counterexample for every buggy variant v. Thus, one must sample a number M of lassos such that one gets an accepting lasso for each such buggy variant with a confidence ratio δ. Let *fba* be a featured Büchi automaton, v be a variant and $p_v = \sum \sigma \in L_v^\omega P(\sigma)$ where L_v^ω is the set of accepting lasso executable by v. Let Z_v denote a Bernoulli random variable such that $Z_v = 1$ with probability p_v and $Z_v = 0$ with probability $q_v = 1 - p_v$. Now, let X_v denote the geometric random variable with parameter p_v that encodes the number of independent samples required until $Z_v = 1$. For a set of variants $V = \{v_1 \dots v_{|V|}\}$, we have that $X_{v_1} \dots X_{v_{|V|}}$ are *not* independent since one may sample a lasso executable by more than one variant.

We define $X = \max_{i=1..|V|} X_{v_i}$. We aim to find a number of sample M such that $P[X \leq M] \geq 1 - \delta$ for a confidence ratio δ. This is analogous to the coupon collector's problem [BH97], which asks how many boxes are needed to collect one instance of every coupon placed randomly in the boxes. It differs from the standard formulation in that the probability of occurrence of coupons are neither independent nor uniform, and a single

box can contain 0 to $\mid V \mid$ coupons. Even for simpler instances of the coupon problem, computing $P[X \leq M]$ analytically is known to be hard [Shi07]. Thus, existing solutions rather characterise a lower bound and an upper bound. We follow this approach as well.

3.3. Lower bound (minimum number of samples)

To compute a lower bound for the number of samples to draw, we transform the family-based SMC problem to a simpler form (in terms of verification effort). We divide our developments into two parts. First, we show that assigning equal probabilities p_{v_i} to every variant v_i (obtained by averaging the original probability values) reduces the number M of required samples. As a second step, we show that assuming that all variants share all their executions also reduces M. Doing so, we reduce the family-based SMC problem to its single-system counterpart, which allows us to reuse the theoretical results of Grosu and Smolka [GS05], and obtain the desired lower bound.

3.3.1. Averaged probabilities

Let $p_{avg} = \frac{1}{|V|} \sum_{v=1..|V|} p_v$ and $X_{\textbf{even}}$ be the counterpart of X where all probabilities p_{v_i} have been replaced by p_{avg}.

Lemma 4 For any number N, it holds that $P[X_{\textbf{even}} \leq N] \geq P[X \leq N]$.

Intuitively, the value of X depends mainly on the variants whose accepting lassos are rarer. By averaging the probability of sampling accepting lassos, we raise the likelihood of getting those rarer lassos. Thus, the number of samples required to get an accepting lasso for all variants. Shioda [Shi07] proves a similar result for the coupon collector problem. He does so by showing that the vector $\mathbf{p}_{\textbf{even}}$ *majorizes* $\mathbf{p} = \{p_{v_1} \dots p_{v_n}\}$ and that the *ccdf*[4] of X is a Schur-concave function of the sampling probabilities. Even though our case is more general than the non-uniform coupon collector's problem, the result of Lemma 4 still holds. Indeed, we observe that the theoretical proof of [Shi07] (a) does not assume the independence of the random variables Z_{v_i}; (b) still applies to the dependent case; and (c) supports the case where the sum of the probability values p_{v_i} is less than one.

3.3.2. Maximized commonalities

Next, let $X_{\textbf{all}}$ be the particular case of $X_{\textbf{even}}$ where all accepting lassos are executable by all variants and are sampled with probability p_{avg}. The expected number of samples required to find a counterexample that any variant can execute is less than the number required to find a counterexample for each variant. Hence the following Lemma.

Lemma 5 It holds that $P[X_{\textbf{all}} \leq N] \geq P[X_{\textbf{even}} \leq N]$.

Thus, the number of samples to find an accepting lasso for every variant is reduced to the number of samples required to find any accepting lasso. Moreover, let us note that $X_{\textbf{all}}$ is a geometric random variable with parameter p_{avg}. This reduces our problem to sampling an accepting lasso in a classical Büchi automaton and allows us to reuse the results of Grosu and Smolka [GS05].

Lemma 6 For a confidence ratio δ and an error margin ϵ, it holds that

$$p_{avg} \geq \epsilon \Rightarrow P[X_{\textbf{all}} \leq M] \geq P[X_{\textbf{all}} \leq N] = 1 - \delta$$

where $M = \frac{ln(\delta)}{ln(1-\epsilon)}$ and $N = \frac{ln(\delta)}{ln(1-p_{avg})}$.

This leads us to the central result of this section.

Theorem 7 Assuming that $p_{avg} \geq \epsilon_{avg}$ for a given error margin ϵ_{avg}, a lower bound for the number of samples required to find an accepting lasso for each buggy variant is $M = \frac{ln(\delta)}{ln(1-\epsilon_{avg})}$ with a Type-1 error bounded by δ.

[4] *ccdf* stands for complementary cumulative distribution function

3.4. Upper bound (maximum number of samples)

We follow a similar two-step process to characterise an upper bound for M. In the first step, we replace the probabilities p_{v_i} of every variant by their minimum. In the second step, we alter the model so that the variants have no common behaviour. Then we show that, given a desired degree of confidence, the obtained model requires a higher number of samples than the original one.

3.4.1. Minimum probability

Let $p_{min} = \min_{v=1..|V|} p_v$ and $X_{\textbf{min}}$ be the counterpart of X where all probabilities p_{v_i} have been replaced by p_{min}. The *ccdf* of X being a decreasing function of the sampling probabilities, we have the following lemma.

Lemma 8 It holds that $P[X_{\textbf{min}} \leq N] \leq P[X \leq N]$.

3.4.2. No common counterexamples

Let $\{(X_{\textbf{indep}})_{v_i}\}$ be a set of independent geometric random variables with parameters p_{min} and let $X_{\textbf{indep}} = \max(X_{\textbf{indep}})_{v_i}$. $X_{\textbf{indep}}$ actually encodes the number of samples required to get a counterexample for all buggy variants when those have no common counterexamples. Since the number of samples to perform cannot be reduced by sampling a counterexample executable by multiple variants, we have the following result.

Lemma 9 It holds that $P[X_{\textbf{indep}} \leq N] \leq P[X_{\textbf{min}} \leq N]$.

Now, let us note that $X_{\textbf{indep}}$ is an instance of the uniform coupon problem with $\mid V \mid$ coupons to collect. A lower bound for $P[X_{\textbf{indep}} \leq M]$ is known to be $1- \mid V \mid \times(1-p_{min})^M$ [Shi07]. Assuming p_{min} greater than some error margin ϵ_{min}, we have $P[X_{\textbf{indep}} \leq M] \geq 1- \mid V \mid \times(1-\epsilon_{min})^M$. Setting a confidence ratio δ, we want to find a M such that $P[X_{\textbf{indep}} \leq M] \geq 1-\delta$. By solving $1- \mid V \mid (1-\epsilon_{min})^M = 1-\delta$, we obtain $M = \frac{ln(\delta)-ln(|V|)}{ln(1-\epsilon_{min})}$, which we can use as the upper bound for the number of samples to perform.

Theorem 10 Assuming that $p_{min} \geq \epsilon_{min}$ for a given error margin ϵ_{min}, an upper bound for the number of samples required to find an accepting lasso for each buggy variant is $M = \frac{ln(\delta)-ln(|V|)}{ln(1-\epsilon_{min})}$ with a Type-1 error is bounded by δ.

4. Product-based statistical model checking for minimizing bug likelihood

Our second use case focuses on the scenario where no variant can completely avoid triggering a bug (i.e. the violation of an LTL formula). The goal is, then, to find the variants minimizing the probability of revealing the bug with a limited amount of sampling execution budget. Formally, let *fts* be an FTS over a set of variant V, ϕ be an LTL formula, and $fba = fts \otimes \mathcal{B}_{\neg\phi}$. One should find $\text{argmin}_{v \in V} P(fba_{|v} \models_k \phi)$ where $\models_k$ means *"satisfies after executing k transitions"*. Assuming a uniform distribution over executable transitions, one can approximate this probability with a maximum likelihood estimator, i.e.

$$P(fba_{|v} \not\models_k \phi) \approx \frac{\#\{\sigma \in Paths_{[k]}(fba_{|v}) \texttt{ s.t. } \sigma \texttt{ is an accepting lasso}\}}{\#(Paths_{[k]}(fba_{|v}))}$$

where $Paths_{[k]}(fba_{|v})$ is a sufficiently large sample of path prefixes of length k in $fba_{|v}$. If $Paths_{[k]}(fba_{|v})$ can be computed, a method that would solve $\text{argmin}_{v \in V} P(fba_{|v} \not\models_k \phi)$ for each variant v would naturally return the variant that minimizes the probability of revealing the bug. However, computing $Paths_{[k]}(fba_{|v})$ could require an intractable number of samples for an intractable number of variants. Thus, our scenario is concerned with finding an appropriate budget allocation to maximize the likelihood of discovering variants with a low violation probability.

4.1. Genetic algorithm

To solve this problem, we propose an elitist Generic Algorithm (GA) that combines variant sampling with statistical model checking. This family of algorithms are convenient to solve our problem as they can identify promising variants through evolution mechanism [GWW+11, HJK+14]. This allows to generate more promising variants based on previous ones. Moreover, such algorithms have been employed to solve search problems with inaccurate evaluation methods (i.e., noisy fitness function [RKD17]). The *noise* of our fitness function originates from the estimation error in violation probabilities (which is reduced as more path prefixes are sampled for a given variant).

Over the last decade, genetic algorithms (GA) have been extensively used as search and optimization tools in various problem domains, including sciences, commerce, and engineering. A GA is a stochastic algorithm that mimics biological evolution. First, the algorithm creates an initial set of solutions called individuals (or chromosomes). This set represents the population. Individuals are encoded as binary vectors where each bit represents the absence or presence of a gene. The evolutionary process begins with this initial population and proceeds over multiple generations. In each generation, the algorithm evaluates the best individuals in the population through a problem-specific *fitness* function, i.e. an objective function that measures the quality of individuals and enables their comparison. Alterers (i.e. mutation and crossover operators) are used to alter the best individuals (mimicking species' evolution) to produce the individuals of the next generation. This evolution process continues until an individual with a desired level of quality is produced or until a fixed number of generations reaches. In our case, our GA generates variants over multiple iterations (generations) aiming to find the ones with the lowest probability to violate the given LTL formula. At each generation, the algorithm allocates a sampling budget to each individual (i.e., each variant) of the current population, samples that many executions from each of these variants, and updates by averaging the violation probability of the variants accordingly. The next generation is created using two mechanisms. The elitism mechanism consists of keeping some of the most promising variants whereas the evolution mechanism will fill the rest of the next generation using mutation and crossover alterers.

Population and genotype. We adopt a binary encoding of the set of variants, that is, we associate each variant with a binary vector that uniquely identifies the variant. Since the set V of variants is finite, one can always define such an encoding using at least $\lceil \log_2(|\ V\ |) \rceil$ bits. The resulting binary vectors form the genotype of the individuals, i.e., each bit is a gene. The only requirement for an individual to be valid is that its associated vector corresponds to an actual variant. In practice, VIS are often defined as a set of boolean features representing the variation points between variants, together with propositional logic formulae defining dependency constraints over these features. In such a case, the binary encoding uses more than the minimal number of bits, which requires a more careful checking of its validity (e.g. using SAT solvers). [BSRC10]

Fitness Function. The fitness function returns, for any variant, its estimated probability of violating the input LTL formula. At each generation, the algorithm samples a predefined number of executions for each variant of the current population. Then, it updates its estimated violation probability according to the number of those executions that violate the property. Thus, the estimated probability values evolve over the generations. Formally, the fitness function of a variant v_i is given by:

$$fitness_{v_i} = \frac{\#\{\sigma \in Paths_{[k]}(v_i), \sigma \models \phi\}}{\#(Paths_{[k]}(v_i))}$$

where $Paths_{[k]}(v_i)$ is the set of k-length paths that have been sampled from $fba_{|v}$. Therefore, the fitness of a variant is the current estimate of its violation probability. Thus, in the long run, it is equivalent to the function that we seek to minimize. The heuristic nature of our algorithm lies in the fact that the fitness function is only an estimation (computed from simulation runs) which becomes more accurate over the iterations of the algorithm.

Selectors and alterers. At each iteration of the genetic algorithm, a subset of the individuals is selected to pursue evolution, and some of them are being altered by mutation to produce new individuals. Generally speaking, an appropriate choice of selection and alteration operators largely depends on the modelled problem. In our context, we focus on finding and keeping a promising variant with the lowest amount of simulation budget. As for alterers, two individuals are combined to form an offspring based on single-point crossover with a high 0.8 recombination probability, while gene mutation in the offspring occurs with a probability of 0.4. For selection, we keep the 1/3 of the *elite* variants with the best fitness (i.e., variants that minimize the property violation) and fill the rest with altered variants from the *elite* variant.

4.2. Generation process

Algorithm 2 formalizes the generation process of our genetic algorithm. As a first step, we initialize the set of sampled paths for each variant to the empty set (Line 1).We also generate an initial population P including L (6 in our experiments) variants selected randomly (Line 2). Then, we make the population evolve for a predefined number N of generations (Lines 3–24) set to 12. At each iteration (generation), we first update the estimated violation probability of each variant of the current population P (i.e. the fitness function) after sampling the predefined number of executions in their projection (Lines 4–6).

Input: V, a set of variants;
fba, an FBA over V;
$k \in \mathbb{N}_0$, the execution length ;
M, the number of samples;
N, the number of generations;
L, the size of the population;
Output: v_{best}, the most promising variant

1 $\forall\, v_i \in V, Paths_{[k]}(v_i) \leftarrow \emptyset$;
2 $P \leftarrow$ `pick randomly` $v_1 \dots v_L$ `from` V;
3 **while** $N \neq 0$ **do**
4 **for** $v_i \in P$ **do**
5 $Paths_{[k]}(v_i) \leftarrow Paths_{[k]}(v_i) \cup \{$`sample` M `paths from` $Paths_{[k]}(fba_{|v})\}$;
6 $fitness_{v_i} = \frac{\#\{\sigma \in Paths_{[k]}(v_i), \sigma \not\models \phi\}}{\#(Paths_{[k]}(v_i))}$
7 **end**
8 **while** $||\ P'\ || < L/elitism\_ratio$ **do**
9 $v \leftarrow \{v' \in P \mid prob(v') = min_{v_i \in P}\ \ prob(v_i)\}$;
10 $P' \leftarrow P' \cup v$;
11 **end**
12 **while** $||\ P'\ || < L$ **do**
13 $v_1, v_2 \leftarrow$ `pick randomly from` P;
14 **if** $p_{crossover} \leq rand(0, 1)$ **then**
15 $v_1, v_2 \leftarrow crossover(v_1, v_2)$;
16 **end**
17 **if** $p_{mutation} \leq rand(0, 1)$ **then**
18 $v_1 \leftarrow mutate(v_1)$, $v_2 \leftarrow mutate(v_2)$;
19 **end**
20 $P' \leftarrow P' \cup v_1 \cup v_2$;
21 **end**
22 $P \leftarrow P'$;
23 $N \leftarrow N - 1$;
24 **end**
25 **return** $v_{best} \leftarrow \{v' \in P \mid fitness(v') = min_{v_i \in P}\ \ fitness(v_i)\}$

Algorithm 2: Genetic algorithm combining variant sampling with statistical model checking.

What follows is the application of selectors and alterers to form the next generation P'. We first use elitist selection that keeps some of the best individuals according to the fitness function (Lines 8–11). As for alterers (Lines 12–21), we randomly apply crossover and mutation operators to fill the new generation. For the crossover, we may (probability of $p_c = 0.8$ in our experiments) randomly pick pairs of individuals from the previous generation and use a simulated binary crossover [RKD17] to create two new offsprings. We assign the same probabilistic importance to each parent . Next, we may ($p_m = 0.4$) also apply binary mutation to alter randomly the genes of each selected individual (Line 21). Each feature (gene) has a probability $p_f = 0.1$ to be altered. At the end of the mutation process, we obtain a new population P' to proceed in the next generation.

After the specified number of generations passed, the algorithm returns the variant with the best fitness from the last generation.

5. Experimental study

5.1. Objectives and methodology

Our experiments concern the evaluation of our SMC methods covering the two use cases. They aim to answer six research questions, i.e. the first three related to the first use case (bug detection) and the last three related to the second use case (bug likelihood estimation and minimization).

One can regard SMC as a means of speeding up verification while risking missing counterexamples. Our first question studies this trade-off and analyses the empirical Type-1 error rate. More precisely, we compute the detection rate of our family-based SMC method, expressed as the number of buggy variants that it detects over the total number of buggy variants.

RQ1 *What is the empirical buggy variant detection rate of family-based SMC?*

We compute the detection rate for different numbers M of samples lying between the lower and upper bounds as characterised in Sect. 3. To get the ground truth (i.e. the true set of all buggy variants), we execute the exhaustive LTL model checking algorithms for FTS designed by Classen et al. [CCS+13]. For the lower bound, we assume that the average probability to sample an accepting lasso for any variant is higher than $\epsilon_{avg} = 0.01$. Setting a confidence ratio $\delta = 0.05$ yields $\frac{ln(0.05)}{ln(0.99)} = 298$. We round up and set $M = 300$ as our lower bound. For the higher bound, we assume that the minimum probability to sample a counterexample in a buggy variant is higher than $\epsilon_{min} = 3.10^{-4}$ and also set $\delta = 0.05$. For a model with 256 variants[5], this yields $M = \frac{ln(0.05)-ln(256)}{ln(0.9997)} = 18478$. For convenience, we round it up to $19,200 = 300 \cdot 2^6$. In the end, we successively set M to $300, 600, \ldots, 19200$ and observe the detection rates.

Next, we investigate a complementary scenario where the engineer has a limited budget of samples to check. We study the smallest budget required by SMC to detect all buggy variants (in the cases where it can indeed detect all of them) and what is the incurred computation resources compared to an exhaustive search of the state space. Thus, our second question is:

RQ2 *How much efficient is SMC with a minimal sample budget compared to an exhaustive search?*

Finally, we compare family-based SMC with the alternative of sampling in each variant's state space separately. We name this alternative method *enumerative SMC*. Hence, our third research question is:

RQ3 *How does family-based SMC compares with enumerative SMC?*

As before, we compare the two techniques w.r.t. detection rate. We set M to the same values as in RQ1. In enumerative SMC, this means that each variant receives a budget of samples of $\frac{M}{|V|}$ where M is the number of samples used in family-based SMC and V is the set of variants.

Once we have shown that SMC can accurately identify buggy variants, we focus on the second use case. We evaluate the capability of our method to estimate the probability of variants to violate given LTL formulae and identify the variants that have the lowest violation probability. We ask:

RQ4 *Can SMC identify the optimal variants minimizing bug likelihood?*

To answer this question, we compare our elitist genetic algorithm with a non-elitist variation and two random baselines (random selection with and without elitism). The main baseline, named *random with elitism*, randomly picks and evaluates variants while keeping the best variants across iterations. This method is similar to our genetic algorithm and differs only by the selection of new variants to evaluate (e.g. it performs random selection rather than mutations). The other baselines are non-elitist versions of the random and the genetic approaches (i.e. they may not retain the best variants of each generation to the next one).

We apply each strategy and record the probability of the variant returned by each technique. We compare the returned variant with the best one using ground truth (i.e. the real violation probability of each variant). The ground truth is computed using a large enough sampling budget per variant.

The previous question compares the genetic approach with baselines with a determined budget. We also aim to understand the impact of this budget on the effectiveness of our SMC methods. A higher simulation budget should obviously improve the quality of variants (i.e., minimizing bug likelihood) returned by methods. We ask:

[5] 256 is the maximum number of variants in our case studies

RQ5 *How does the sample budget impact the effectiveness of our SMC methods?*

To answer this question, we repeat our RQ4 experiments with different simulation budgets. Doing so allows us to observe the effect of providing SMC with more budget and the trend for each alternative method.

Finally, the effectiveness of our genetic algorithm may depend on meta-parameters like the elitism ratio and the probabilities of alterers. Our last question investigates the impact of these parameters. We ask:

RQ6 *How do genetic parameters impact our SMC method to identify the variants minimizing bug?*

To answer this question, we repeat our RQ5 experiments with four configurations of the aforementioned parameters and we observe the trend of each configuration.

5.2. Experimental setup

Implementation. We implemented our SMC algorithms (family-based and enumerative-based) in a prototype tool developed in C and Python. More precisely, the SMC algorithms were implemented in C by reusing the Promela parser of ProVeLines [CSHL13b], a state-of-the-art model checker for VIS. The tool takes as input an FTS, an LTL formula and a sample budget. Then it performs SMC until all samples are checked or until all variants are found to violate the formula. The Python component implements our generic algorithm on top of `pyeasy-ga`[6]. To compare with the exhaustive search, we use ProVeLines [CSHL13b], to ensure a fair comparison by guaranteeing that all approaches explore the same state space.

Dataset. We consider three systems that were used in past research to evaluate VIS model checking algorithms [CCS+13, CSHL13a, CHL+14]. Table 1 summarizes the characteristics of our case studies and their related properties. The first system is a minepump system [KMSL83, CCS+13] with 128 variants. The underlying FTS comprises 250,561 states, while the state space of all variants taken individually reaches 889,124 states. The second model is an elevator model inspired by Plath and Ryan [PR01]. It is composed of eight configuration options, which can be combined into 256 different variants, and its FTS has 58,945,690 states to explore. The third and last is a case study inspired by the CCSDS File Delivery Protocol (CFDP) [Con07], a real-world configurable spacecraft communication protocol [BCH+10]. The FTS modelling the protocol consists of 1,732,536 states to explore and 56 variants (individually totalling 2,890,399 states).

For the RQ related to the first case study (RQ1, RQ2 and RQ3), we discarded the properties that are satisfied by all variants. Those are: Minepump #17, #33, #40; Elevator #13, CFDP #5. Indeed, these properties are not relevant for RQ1 and RQ3 since SMC is trivially correct in such cases. As for RQ2, any small sample budget would return correct results while being more efficient than the exhaustive search. This leaves us with 59 properties.

For the RQ related to the second use case (RQ4, RQ5 and RQ6), we keep only the properties that all variants violate and where the difference between the highest and lowest violation probabilities is at least 0.05. This results in 12 properties in 2 case studies. 5 for the minepump case study and 7 for the elevator.

Infrastructure and repetitions. We run our experiments on a MacBook Pro 2018 with a 2.9 GHz Core-i7 processor and macOS 10.14.5. To account for random variations in the sampling, we execute 100 runs of each experiment and compute the average detection rates for each property.

6. Results

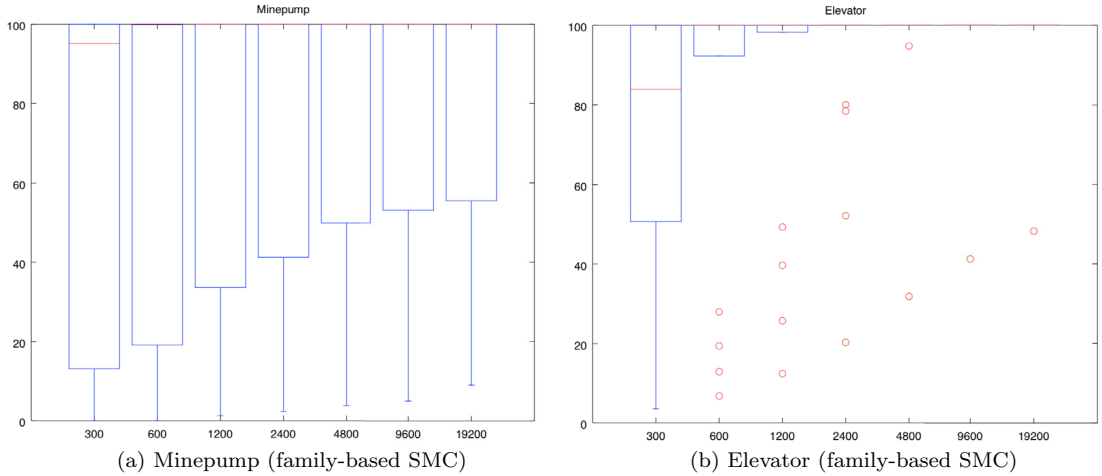
6.1. RQ1: detection rate

Figure 2 shows as boxplots, for each case study and over all checked properties, the percentage of buggy variants for which family-based SMC found a counterexample. We provide those boxplots for different number M of samples.

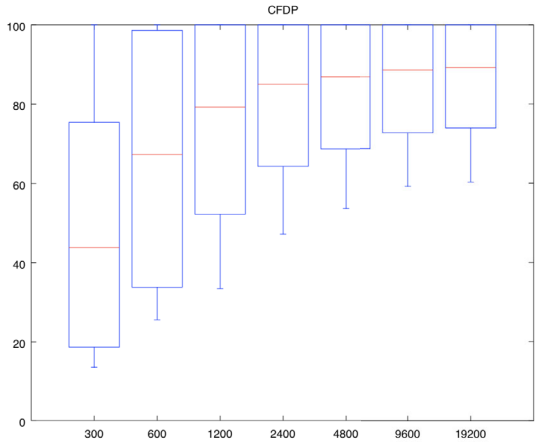
[6] https://github.com/remiomosowon/pyeasyga

Table 1. Models and LTL formulae used in our experiments.

Minepump (250,561 FTS states, 128 valid variants)	
#1	$\neg\,(\Box\Diamond(stateReady \wedge highWater \wedge userStart))$
#2	$\neg\,(\Box\Diamond stateReady)$
#3	$\neg\,(\Box\Diamond stateRunning)$
#4	$\neg\,(\Box\Diamond stateStopped)$
#5	$\neg\,(\Box\Diamond stateMethanestop)$
#6	$\neg\,(\Box\Diamond stateLowstop)$
#7	$\neg\,(\Box\Diamond readCommand)$
#8	$\neg\,(\Box\Diamond readAlarm)$
#9	$\neg\,(\Box\Diamond readLevel)$
#10	$\neg\,((\Box\Diamond readCommand) \wedge (\Box\Diamond readAlarm) \wedge (\Box\Diamond readLevel))$
#11	$\neg\,(\Box\Diamond pumpOn)$
#12	$\neg\,(\Box\Diamond\neg\, pumpOn)$
#13	$\neg\,((\Box\Diamond pumpOn) \wedge (\Box\Diamond\neg\, pumpOn))$
#14	$\neg\,(\Box\Diamond methane)$
#15	$\neg\,(\Box\Diamond\neg\, methane)$
#16	$\neg\,((\Box\Diamond methane) \wedge (\Box\Diamond\neg\, methane))$
#17	$\Box(\neg\, pumpOn \vee stateRunning)$
#18	$\Box(methane \Rightarrow (\Diamond stateMethanestop))$
#19	$\Box(methane \Rightarrow \neg\,(\Diamond stateMethanestop))$
#20	$\Box(pumpOn \vee \neg\, methane)$
#21	$\Box((pumpOn \wedge methane) \Rightarrow \Diamond\neg\, pumpOn)$
#22	$((\Box\Diamond readCommand) \wedge (\Box\Diamond readAlarm) \wedge (\Box\Diamond readLevel)) \Rightarrow \Box((pumpOn \wedge methane) \Rightarrow \Diamond\neg\, pumpOn)$
#23	$\neg\,\Diamond\Box(pumpOn \wedge methane)$
#24	$((\Box\Diamond readCommand) \wedge (\Box\Diamond readAlarm) \wedge (\Box\Diamond readLevel)) \Rightarrow \neg\,\Diamond\Box(pumpOn \wedge methane)$
#25	$\Box((\neg\, pumpOn \wedge methane \wedge \Diamond\neg\, methane) \Rightarrow ((\neg\, pumpOn)\, U \neg\, methane))$
#26	$\Box((highWater \wedge \neg\, methane) \Rightarrow \Diamond pumpOn)$
#27	$\neg\,(\Diamond(highWater \wedge \neg\, methane))$
#28	$((\Box\Diamond readCommand) \wedge (\Box\Diamond readAlarm) \wedge (\Box\Diamond readLevel)) \Rightarrow (\Box((highWater \wedge \neg\, methane) \Rightarrow \Diamond pumpOn))$
#29	$\Box((highWater \wedge \neg\, methane) \Rightarrow \neg\,\Diamond pumpOn)$
#30	$\neg\,\Diamond\Box(\neg\, pumpOn \wedge highWater)$
#31	$((\Box\Diamond readCommand) \wedge (\Box\Diamond readAlarm) \wedge (\Box\Diamond readLevel)) \Rightarrow (\neg\,\Diamond\Box(\neg\, pumpOn \wedge highWater))$
#32	$\neg\,\Diamond\Box(\neg\, pumpOn \wedge \neg\, methane \wedge highWater)$
#33	$((\Box\Diamond readCommand) \wedge (\Box\Diamond readAlarm) \wedge (\Box\Diamond readLevel)) \Rightarrow (\neg\,\Diamond\Box(\neg\, pumpOn \wedge \neg\, methane \wedge highWater))$
#34	$\Box((pumpOn \wedge highWater \wedge \Diamond lowWater) \Rightarrow (pumpOn\, U lowWater))$
#35	$\neg\,\Diamond(pumpOn \wedge highWater \wedge \Diamond lowWater)$
#36	$\Box(lowWater \Rightarrow (\Diamond\neg\, pumpOn))$
#37	$((\Box\Diamond readCommand) \wedge (\Box\Diamond readAlarm) \wedge (\Box\Diamond readLevel)) \Rightarrow (\Box(lowWater \Rightarrow (\Diamond\neg\, pumpOn)))$
#38	$\neg\,\Diamond\Box(pumpOn \wedge lowWater)$
#39	$((\Box\Diamond readCommand) \wedge (\Box\Diamond readAlarm) \wedge (\Box\Diamond readLevel)) \Rightarrow (\neg\,\Box\Diamond(pumpOn \wedge lowWater))$
#40	$\Box((\neg\, pumpOn \wedge lowWater \wedge \Diamond highWater) \Rightarrow ((\neg\, pumpOn)\, U highWater))$
#41	$\neg\,\Diamond(\neg\, pumpOn \wedge lowWater \wedge \Diamond highWater)$
Elevator (58,945,690 FTS states, 256 valid variants)	
#1	$\neg\,\Box\Diamond progress$
#2	$\neg\,\Box\Diamond f0 \vee \neg\,\Box\Diamond f1 \vee \neg\,\Box\Diamond f2 \vee \neg\,\Box\Diamond f3$
#3	$\neg\,\Box\Diamond p0at0 \vee \neg\,\Box\Diamond p0at1 \vee \neg\,\Box\Diamond p0at2 \vee \neg\,\Box\Diamond p0at3$
#4	$\Box(fb2 \Rightarrow (\Diamond f2))$
#5	$\Box\Diamond progress \Rightarrow (\Box(fb2 \Rightarrow (\Diamond f2)))$
#6	$\Box\Diamond progress \Rightarrow (\Box(fb2 \Rightarrow (\Diamond(f2 \wedge dopen))))$
#7	$\Box\Diamond progress \Rightarrow (\neg\,\Diamond\Box f2)$
#8	$\Box\Diamond(progress \vee waiting) \Rightarrow (\neg\,\Diamond\Box f2)$
#9	$\Box\Diamond(progress \vee waiting) \Rightarrow (\neg\,\Diamond\Box f0)$
#10	$\neg\,\Diamond((cb0 \vee cb1 \vee cb2 \vee cb3) \wedge \neg\,(p0in \vee p1in) \wedge dclosed)$
#11	$\Box\Diamond progress \Rightarrow (\neg\,\Diamond\Box dclosed)$
#12	$\Box\Diamond progress \Rightarrow (\neg\,\Diamond\Box(p0to3 \wedge dclosed))$
#13	$\Box\Diamond progress \Rightarrow (\neg\,\Diamond\Box dopen)$
#14	$\Box\Diamond(progress \vee waiting) \Rightarrow (\neg\,\Diamond\Box dopen)$
#15	$((\Box\Diamond(progress \vee waiting)) \wedge (\Box\Diamond(fb0 \vee fb1 \vee fb2 \vee fb3))) \Rightarrow (\neg\,\Diamond\Box dopen)$
#16	$\neg\,\Diamond(p0in \wedge p1in \wedge dclosed)$
#17	$\neg\,\Diamond\Box(p0in \wedge dclosed)$
#18	$\Box\Diamond progress \Rightarrow (\neg\,\Diamond\Box(p0in \wedge dclosed))$
CFDP (1,801,581 FTS states, 56 valid variants)	
#1	$\Diamond fileReceived$
#2	$(\Diamond eofReceived) \Rightarrow \Diamond fileReceived$
#3	$((\Diamond eofReceived) \wedge (\Diamond nakReceived)) \Rightarrow \Diamond fileReceived$
#4	$((\Diamond eofReceived) \wedge (\Box\Diamond nakReceived)) \Rightarrow \Diamond fileReceived$
#5	$\Box(finSend \Rightarrow fileReceived)$



(a) Minepump (family-based SMC) (b) Elevator (family-based SMC)



(c) CFDP (family-based SMC)

Fig. 2. Detection rate of the buggy variants achieved by our SMC method, in the three case studies and using different sample sizes. In each figure, the x-axis is the number of samples.

In the case of Minepump and Elevator, the median detection percentage is 100% starting from $M = 1200$ and $M = 600$, respectively. Further increasing the number of samples raises the 0.25 percentile. In Minepump and for $M = 1200$, there are 18/41 properties for which SMC could not detect all buggy variants. Increasing M improves significantly the percentage of buggy variants detected by SMC for all these properties, although there remain undetected variants in 15 of them even with $M = 19,200$. This illustrates that our assumption regarding p_{min} was inappropriate for those properties: counterexamples are rarer than we imagined. The elevator study yields even better results: at $M = 600$, SMC detects all buggy variants for 10/18 properties; this number becomes 14/18 at $M = 2,400$ and 17/18 at $M = 9,600$. As for the remaining property, SMC with $M = 19,200$ detects 50% of the variants on average and we observe that this percentage consistently increases as we increase M.

The results for CFDP are mixed: while the median percentage goes beyond 80% as soon as $M = 1,200$, it tends to saturate when increasing the number of samples. The 0.25 percentile still increases but also seems to reach an asymptotic behaviour in the trials with the highest M. A detailed look at the results reveals that for $M \geq 1,200$, SMC cannot identify all buggy variants for only two properties: #3 (9 buggy variants) and #4 (4 buggy variants). At $M = 19,200$, SMC detects 5.43 and 3.14 buggy variants for those two properties, respectively.

Further doubling M raises these numbers to 6.36 and 3.26. This indicates that the non-detected variants have few counterexamples, which are rare due to the tinier state space of those variants. The computation resources required by SMC to find such rare counterexamples with high confidence are higher than model-checking the undetected variants thoroughly. An alternative would be to direct SMC towards rare executions, leaning on techniques such as [JLS13, CIM+13].

> SMC can detect all buggy variants for 41 properties out of 59. However, for the remaining properties, SMC was unable to find the rare counterexamples of some buggy variants. This calls for new dedicated heuristics to sample those rare executions.

6.2. RQ2: efficiency

Next, we check how much execution time SMC can spare compared to the exhaustive search. Results are shown in Table 2. Overall, we see that SMC holds the potential to greatly accelerate the discovery of all buggy variants, achieving a total speedup of 526%, 1891% and 356% for Minepump, Elevator and CFDP, respectively. For more than half of the properties, the smallest number of samples we tried (i.e. 300) was sufficient for a thorough detection. Those properties are actually satisfied by all variants. The fact that SMC requires such a small number of samples means that the same bug lies in all the variants (as opposed to each variant violating the property in its own way). On the contrary, Minepump property #31 is also violated by all variants but requires a much higher sample number, which illustrates the presence of variant-specific bugs.

Interestingly, the benefits of SMC are higher in the Elevator case (the largest of the three models), achieving speedups of up to 16,575%. A likely explanation is that the execution paths of the Elevator model share many similarities, which means that a single bug can lead to multiple failed executions. By sampling randomly, SMC avoids exploring thoroughly a part of the state space that contains no bug and, instead, increases the likelihood to move to interesting (likely-buggy) parts. A striking example is property #16 (satisfied by half of the variants), where SMC reduces the verification time from 3 minutes to 4 seconds.

> Where SMC can detect all buggy variants, it can do so with more efficiency compared to exhaustive search, for 33/41 properties, achieving speedups of multiple orders of magnitude.

6.3. RQ3: family-based SMC versus enumerative SMC

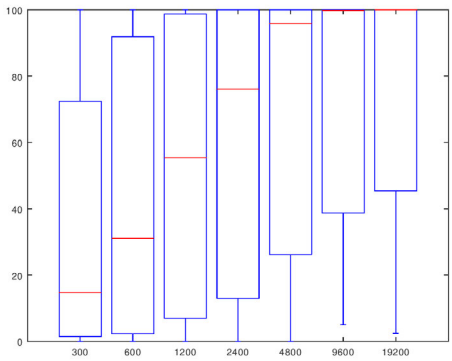
Figure 3 shows the detection rate achieved the enumerative SMC for the three case studies and different numbers of samples, while the results of the family-based SMC were shown in Fig. 2. In the Minepump and Elevator cases, enumerative SMC achieves a lower detection rate than family-based SMC. In both cases, a Student t-test with $\alpha = 0.05$ rejects, with statistical significance, the hypothesis that the two SMC methods yield no difference in error rate. One can observe, for instance, that, with 600 samples, enumerative SMC achieves a median detection rate of 31.13%, while family-based SMC achieved 99.86%. This tends to validate our hypothesis that family-based SMC is more effective as the variants share more executions. Indeed, on average, one state of the Minepump is shared by 3.55 variants.

In the case of CFDP, however, enumerative SMC performs systematically better (up to 13.95% more). Still, the difference in median detection rate tends to disappear as more executions are sampled. Nevertheless, CFDP illustrates the main drawback of family-based SMC: it can overlook counterexamples in variants with fewer behaviours. In fact, previous research [CCS+13] has shown that, compared to enumerative approaches, the benefits of family-based methods diminishes as there are more divergence between the behaviour of the variants and as these divergences occur earlier during verification. These conclusions also apply to our SMC-based method, as it exploits similar heuristics to check common behaviour only once. In other words, the effectiveness of our method diminishes as there is a higher number of behaviours unique to some variants.

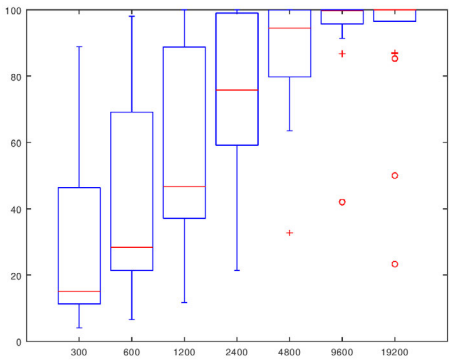
Table 2. Least numbers of samples (in our experiments) that allowed detecting all buggy variants and corresponding execution time. `Full` refers to an exhaustive search of the search space. Only properties that are violated by at least one variant and for which SMC found all buggy variants are shown

Property	# Samples	SMC		Full		Speedup (%)
		# States	Time	# States	Time	
Minepump #1	600	25332	0.18	92469	1.33	739
Minepump #2	300	12553	0.10	24908	1.06	1060
Minepump #4	300	2383	0.03	103933	3.10	10333
Minepump #5	1200	48714	0.32	76040	1.03	322
Minepump #7	300	2469	0.03	18482	0.21	700
Minepump #8	300	2757	0.03	4646	0.05	167
Minepump #9	300	2758	0.03	8263	0.08	267
Minepump #10	600	15191	0.11	55936	0.58	527
Minepump #12	300	2356	0.03	811	0.02	67
Minepump #14	300	2915	0.04	989	0.02	50
Minepump #15	300	2389	0.03	2673	0.05	167
Minepump #16	300	4102	0.04	1917	0.03	75
Minepump #18	300	2604	0.03	125	0.01	33
Minepump #19	600	25027	0.18	143540	2.69	1494
Minepump #20	300	3864	0.03	40	0.01	33
Minepump #25	2400	67620	0.50	346935	6.12	1224
Minepump #26	300	2708	0.03	4382	0.05	167
Minepump #27	300	2450	0.03	3702	0.04	133
Minepump #28	2400	58382	0.43	99780	1.28	298
Minepump #30	300	300	0.03	3648	0.05	167
Minepump #31	9600	165802	1.29	61185	1.03	80
Minepump #32	300	2684	0.03	4110	0.05	167
Minepump #41	300	5732	0.05	3886	0.04	80
Total		**461092**	**3.60**	**1062400**	**18.93**	**526**
Elevator #1	300	4371	0.03	105883	0.52	1733
Elevator #2	600	226813	1.14	437252	2.48	218
Elevator #3	4800	1736781	7.67	14822853	103.22	1346
Elevator #4	300	4403	0.04	1194568	6.63	16575
Elevator #5	300	7719	0.05	1305428	7.76	15520
Elevator #6	300	7061	0.05	1202204	6.89	13780
Elevator #7	600	25021	0.12	732684	4.33	3608
Elevator #8	600	26120	0.13	204934	1.19	915
Elevator #9	300	3142	0.03	39086	0.28	933
Elevator #11	300	3278	0.03	91	0.02	67
Elevator #12	9600	1502419	6.53	1954924	11.12	170
Elevator #14	2400	141753	0.61	7889584	52.88	8669
Elevator #15	2400	142405	0.69	7889753	57.64	8354
Elevator #16	2400	955206	4.02	28551923	182.25	4534
Elevator #17	1200	100755	0.38	516230	3.53	929
Elevator #18	4800	510145	1.94	486694	3.00	155
Total		**5397392**	**23.46**	**67334091**	**443.74**	**1891**
CFDP #1	300	50206	0.20	87937	1.71	855
CFDP #2	1200	117897	0.52	102842	0.85	163
Total		**168103**	**0.72**	**190779.00**	**2.56**	**356**

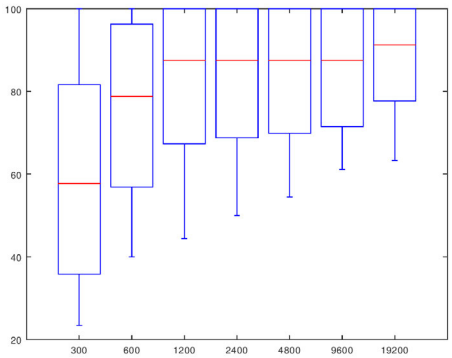
Still, the non-exhaustive nature of our approach adds another dimension to the problem: the rarer such unique behaviour is, the less likely our SMC method samples this behaviour. In such cases, enumerative SMC might complement family-based SMC by sampling from the state space of specific variants.



(a) Minepump (enumerative SMC)



(b) Elevator (enumerative SMC)



(c) CFDP (enumerative SMC)

Fig. 3. Detection rate of the buggy variants achieved by classical SMC applied variant by variant, in the three case studies and using different sample sizes. In each figure, the x-axis is the number of samples.

Family-based SMC can detect significantly more buggy variants than enumerative SMC, especially when few lassos are sampled. Yet, enumerative SMC remains useful for variants that have a tiny state space compared to the others and can, thus, complement the family-based method.

6.4. RQ4: bug likelihood minimization

Table 3 shows, for each case study and LTL formula, the difference δ between the real violation probability of the best variant v_{best} and the variant v_{res} returned by each method. We estimated those real probabilities by running a large sample of executions for each variant. We experimentally found out that 1,920,000 was sufficiently large to get a precise estimate of the violation probability of each variant, with negligible random differences across multiple runs (approximately 1.e-4).

We evaluate each method on a sampling budget $M = 19,200$. Hence, each method will sample a maximum of $19,200$ lassos (total for all variants). Given the random factors induced in each method, we repeated our experiments 100 times and computed the average values obtained for δ. We also compute the average rank of the variant returned by each method.

Table 3. Difference δ between the real violation probability of the best variant v_{best} and the variant v_{res} returned by each method, for each property. Left numbers are the value of δ whereas the right number (between parentheses) are the percentage of error that δ represents. We also show the average ranking of the returned variant (less is better) over 100 runs. Standard deviation of the differences δ is computed over 100 runs

Property	Random	Ran.+El.	Genetic	Gen.+El.	Std. dev.
Minepump #4	0.683 (25%)	0.465 (20%)	0.959 (25%)	0.587 (25%)	.064
Minepump #8	583 (22%)	0.523 (18%)	0.597 (22%)	0.376 (11%)	.044
Minepump #12	1.339 (21%)	0.879 (14%)	1.629 (24%)	1.002 (15%)	.040
Minepump #14	1.327 (25%)	1.150 (21%)	1.504 (25%)	0.778 (14%)	.055
Minepump #18	1.546 (21%)	1.358 (17%)	1.222 (14%)	1.057 (13%)	.085
Avg. Ranking	**Top 22%**	**Top 18%**	**top 22%**	**top 15%**	
Elevator #1	4.197 (9%)	1.050 (6%)	7.614 (13%)	0.550 (3%)	.094
Elevator #4	1.553 (37%)	1.549 (37%)	1.347 (32%)	1.047 (24%)	.024
Elevator #6	0.981 (20%)	0.824 (17%)	1.775 (40%)	0.781 (16%)	.022
Elevator #8	2.741 (14%)	1.681 (4%)	3.196 (20%)	1.816 (5%)	.028
Elevator #9	1.891 (12%)	1.558 (12%)	3.562 (33%)	0.699 (6%)	.021
Elevator #11	0.787 (24%)	0.761 (21%)	2.132 (50%)	0.630 (12%)	.044
Elevator #17	2.323 (20%)	1.289 (15%)	3.360 (30%)	1.440 (16%)	.077
Avg. Ranking	**Top 19%**	**Top 16%**	**top 31%**	**top 11%**	

Overall, the results show that the genetic algorithm with elitism performs better than the other strategies for both Minepump (128 variants) and Elevator (256 variants). For Minepump properties, it returns on average a product in the top 15%; by comparison, the random method with elitism (the second-best approach) returns a product in the top 18%. In the Elevator case study, the average ranking becomes 11% (our genetic approach with elitism) against 16% (random with elitism).

The absence of elitism substantially degrades the results for both random and our genetic algorithm, with random search performing a bit better in this setting. The genetic selection mechanism seems more efficient than a random one. This could be explained by the fact that exploiting previous variants features to select other variants is more efficient than random selection.

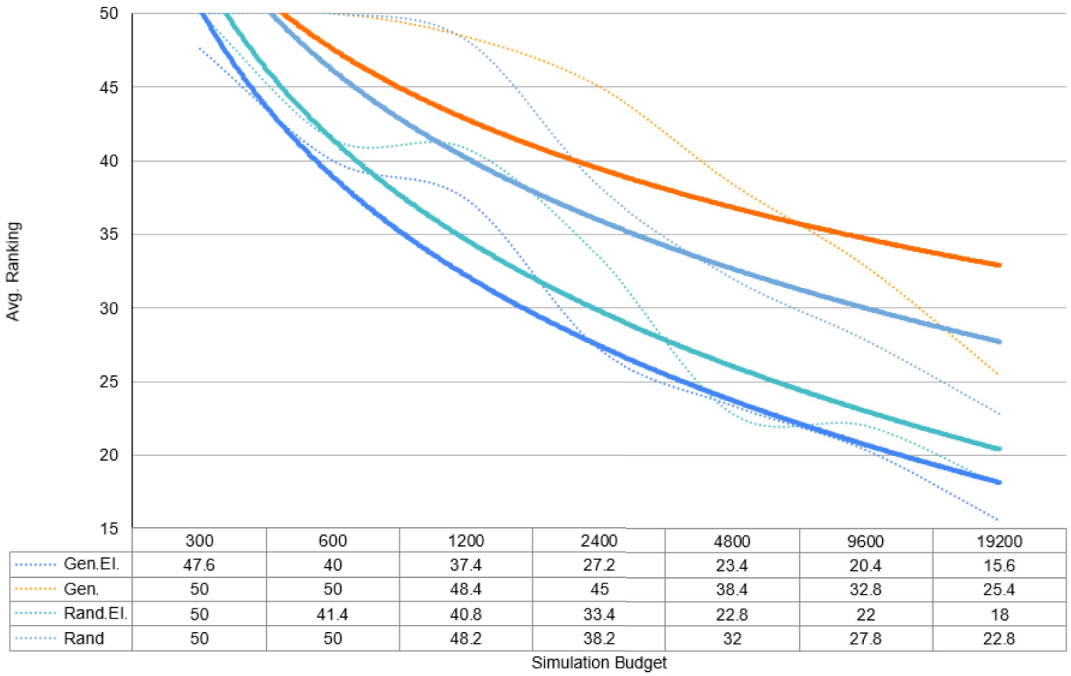
We observe that the efficiency of our product-based methods depends on the studied property. For example, the GA with elitism offers impressive effectiveness for Elevator#1,#8,#9. However, the same techniques has disappointing effectiveness for some cases, e.g. Minepump#4 and Elevator#5.

SMC can effectively identify the variants that minimize bug likelihood using non-exhaustive methods while remaining scalable.

6.5. RQ5: sample budget impact on bug likelihood minimization

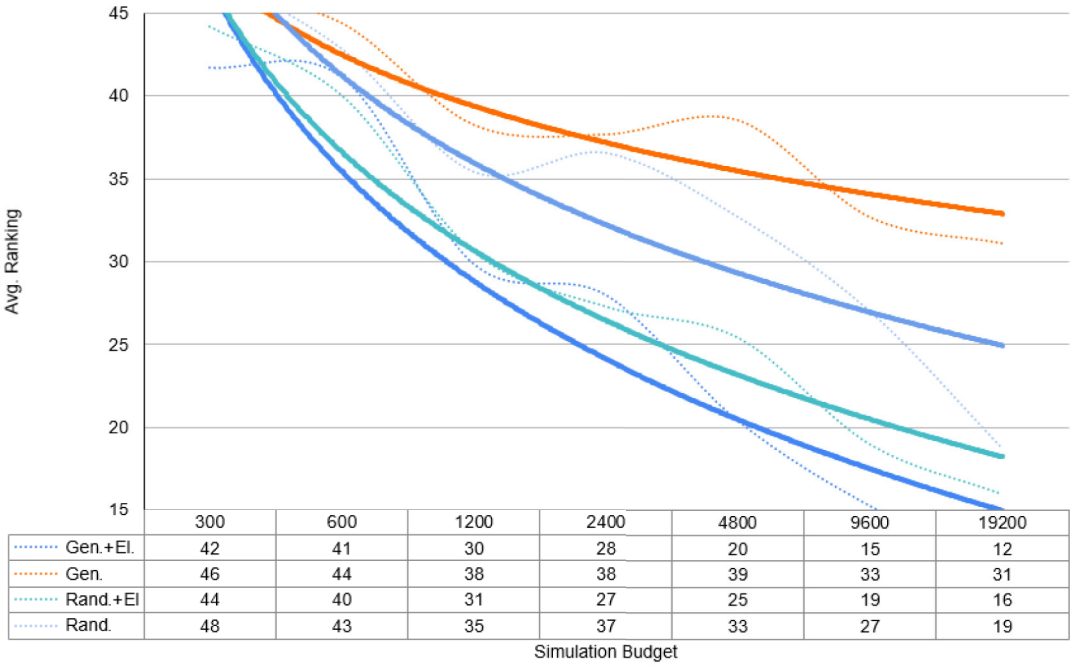
Figure 4 shows, for different simulation budgets, the effectiveness of the SMC methods. Unsurprisingly, increasing the simulation budget improves the efficiency of every method. The heuristics with elitism generally outperform the others. This is mainly due to the fact that, without elitism, the algorithm may replace the current best variants of the population with worse ones. The lack of elitism, furthermore, seems to annihilate the benefits of mutation and cross-over operators.

When elitism is added, the generic algorithm performs better than the random heuristics and much better than non-elitist versions. In fact, the results of non-elitist versions are reached by elitist ones with approximately 4 times less simulation budget. When elitism mechanism is added, the mutation-based selection performs better than the random heuristics.



	300	600	1200	2400	4800	9600	19200
Gen.El.	47.6	40	37.4	27.2	23.4	20.4	15.6
Gen.	50	50	48.4	45	38.4	32.8	25.4
Rand.El.	50	41.4	40.8	33.4	22.8	22	18
Rand	50	50	48.2	38.2	32	27.8	22.8

(a) Minepump



	300	600	1200	2400	4800	9600	19200
Gen.+El.	42	41	30	28	20	15	12
Gen.	46	44	38	38	39	33	31
Rand.+El	44	40	31	27	25	19	16
Rand.	48	43	35	37	33	27	19

(b) Elevator

Fig. 4. Average ranking of the variant returned by our product-based SMC methods to identify variants that minimize the probability of violating the property.

This is because mutating and breeding the best variants of the current population allows the exploitation of the neighbourhood of those current best variants. On the contrary, random favours exploration over exploitation and, therefore, fails to exploit combinations of features that are effective.

> The genetic algorithm with elitism outperforms the other methods and the differences increase as the budget increases as well.

6.6. RQ6: genetic parameters impact on bug likelihood minimization?

We evaluate different configurations of the genetic algorithm using different elitism ratios and alterer parameters. We consider two elitism ratios: 1/3 (E−) and 2/3 (E+). We also consider two sets of alterer parameters: high alteration (M+) and low alteration (M−). In M+, crossover probability was set to 0.8 and mutation to 0.4/0.1—meaning that 40% of the genes have 10% probability to be mutated. In M-, are set to 0.2 for crossover and 0.1/0.01 for mutation. These sets of elitism ratios and alterer parameters yield four configurations in total.

We measure again the effectiveness of our method—for each configuration—as the average ranking of the selected variant, for different simulation budget. Figure 5 shows the results. We can observe that the E−/M− (low elitism and low mutation) version of our algorithm performs much worse than the other (in both case studies). Furthermore, E+/M+ and E+/M− perform similarly. Finally, E−/m+ seems to perform slightly better than the other configurations. This shows that a good balance between elitism and alteration yield the best results. High mutation will provide more disruption in variants produced from one generation to the next. On the contrary, the role of elitism is to stabilize the set of variants from one generation to the next—by keeping the best variants. Low alteration tends to perform worse as the method may spend simulation budget on uninteresting variants across multiple generations regardless of the elitism ratio.

> Our genetic algorithm tends to perform better when it keeps a low ratio of elitism—1/3—and sets higher alteration probabilities.

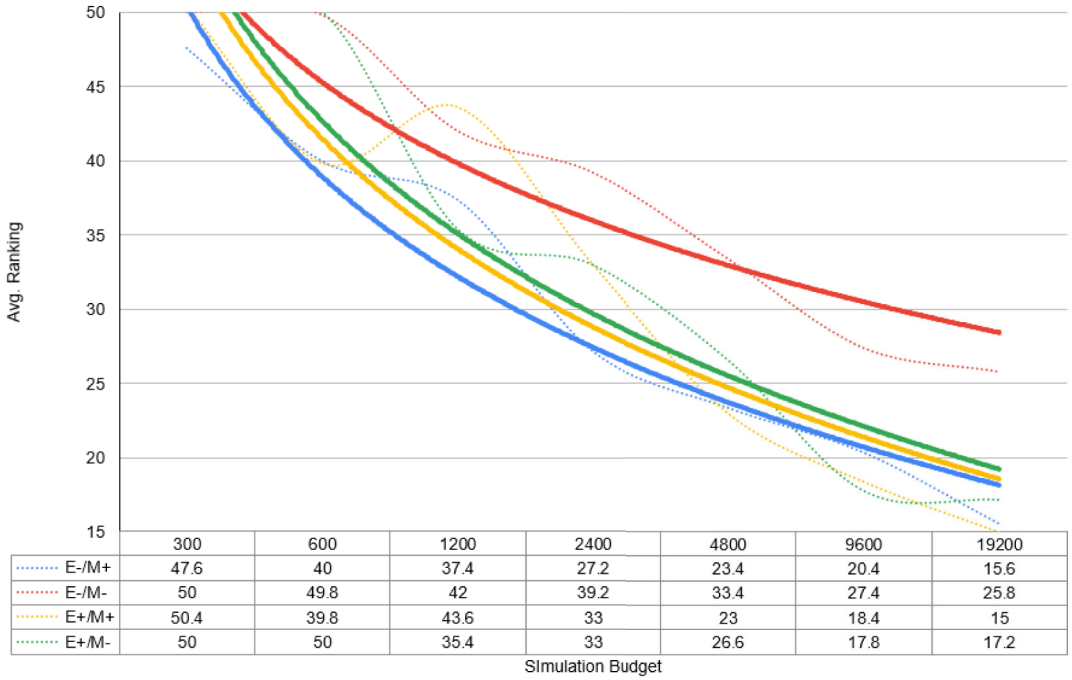
7. Conclusion

We proposed a new simulation-based approach for finding bugs in VIS. It applies statistical model checking to FTS, an extension of transition systems designed to model concisely multiple VIS variants.

In the first use case we investigated, our method results in either collecting counterexamples for multiple variants at once or proving the absence of bugs. The algorithm always converges, up to some confidence error which we quantify on the FTS structure by relying on results for the coupon collector problem. After implementing the approach within a state-of-the-art tool, we study its benefits and drawbacks empirically. It turns out that a small number of samples is often sufficient to detect all variants, outperforming an exhaustive search by an order of magnitude. On the downside, we were unable to find counterexamples for some faulty variants and properties. This calls for future research, exploiting techniques to guide the simulation towards rare bugs/events [JLS13, CIM+13, BDH15] or towards uncovered variants relying, e.g., on distance-based sampling [KGS+19] or light-weight scheduling sampling [DHS18].

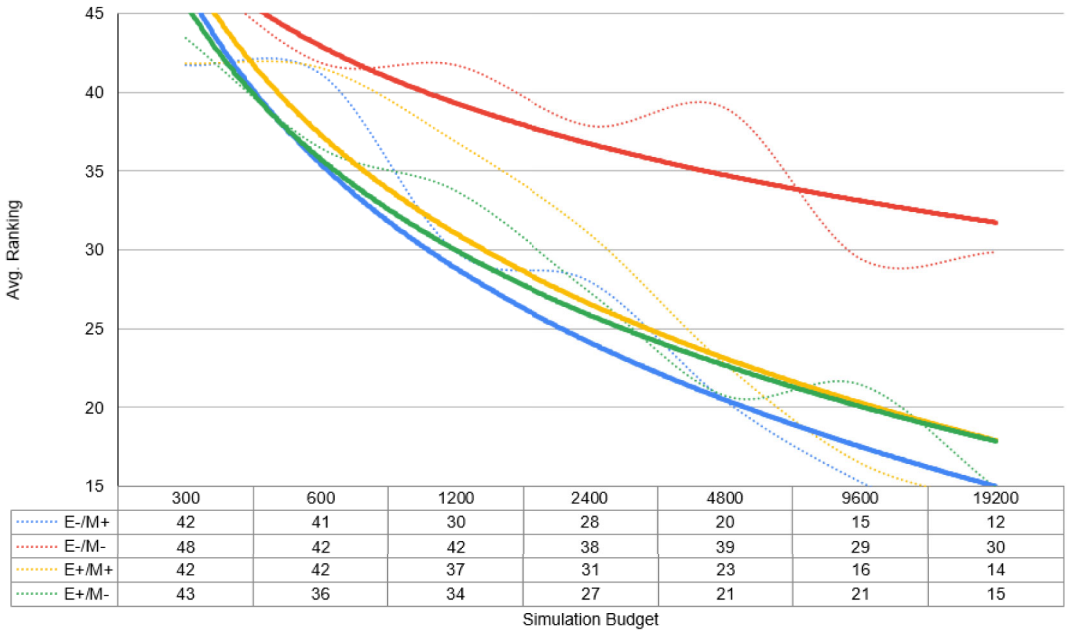
In the second use case, our method combining SMC with genetic algorithms showed the best capability in finding variants with lower violation probabilities. Elitism proved to be an essential constituent to make our approach successful. One particular challenge in allocating the sampling budget resides in that the fitness function is only an approximation of the real violation probabilities (yet increasingly more accurate as more samples are allocated). Interesting directions for the future involve further improvements of our approach with heuristics to deal with such noise in fitness functions.

Nevertheless, the positive outcome of our study is to show that SMC can act as a low-cost-high-reward alternative to exhaustive verification, which can provide thorough results in a majority of cases.



	300	600	1200	2400	4800	9600	19200
E-/M+	47.6	40	37.4	27.2	23.4	20.4	15.6
E-/M-	50	49.8	42	39.2	33.4	27.4	25.8
E+/M+	50.4	39.8	43.6	33	23	18.4	15
E+/M-	50	50	35.4	33	26.6	17.8	17.2

(a) Minepump



	300	600	1200	2400	4800	9600	19200
E-/M+	42	41	30	28	20	15	12
E-/M-	48	42	42	38	39	29	30
E+/M+	42	42	37	31	23	16	14
E+/M-	43	36	34	27	21	21	15

(b) Elevator

Fig. 5. Average ranking of the variant returned by different parameters configurations (i.e., elitism and mutation rate) of genetic algorithm SMC method to identify variants that minimize the probability of violating the property.

Acknowledgements

Maxime Cordy and Sami Lazreg are supported by FNR Luxembourg (grant C19/IS/13566661/BEEHIVE/Cordy).

References

[ABM98] Ammann PE, Black PE, Majurski W (1998) Using model checking to generate tests from specifications. In: Proceedings second international conference on formal engineering methods (Cat.No.98EX241), pp 46–54

[BCH+10] Boucher Q, Classen A, Heymans P, Bourdoux A, Demonceau L (2010) Tag and prune: a pragmatic approach to software product line implementation. In: ASE'10. ACM, pp 333–336

[BDH15] Budde CE, D'Argenio PR, Hermanns H (2015) Rare event simulation with fully automated importance splitting. In: Beltrán M, Knottenbelt WJ, Bradley JT (eds) Computer performance engineering–12th European workshop, EPEW 2015, Madrid, Spain, August 31–September 1, 2015, Proceedings, volume 9272 of Lecture Notes in Computer Science. Springer, pp 275–290

[BH97] Boneh A, Hofri M (1997) The coupon-collector problem revisited—a survey of engineering problems and computational methods. Commun Stat Stoch Models 13(1):39–66

[BJK+05] Broy M, Jonsson B, Katoen J-P, Leucker M, Pretschner A (2005) (eds) Model-based testing of reactive systems, advanced lectures [the volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004], volume 3472 of Lecture Notes in Computer Science. Springer

[BK08] Baier C, Katoen J-P (2008) Principles of model checking. MIT Press

[BS20] Budde CE, Stoelinga M (2020) Automated rare event simulation for fault tree analysis via minimal cut sets. In: Hermanns H (eds) Measurement, modelling and evaluation of computing systems—20th international GI/ITG conference, MMB 2020, Saarbrücken, Germany, March 16–18, 2020, Proceedings, volume 12040 of Lecture Notes in Computer Science. Springer, pp 259–277

[BSRC10] Benavides D, Segura S, Ruiz-Cortés A (September 2010) Automated analysis of feature models 20 years later: a literature review. Inf Syst 35(6):615–636

[CCS+13] Classen A, Cordy M, Schobbens P-Y, Heymans P, Legay A, Raskin J-F (2013) Featured transition systems: foundations for verifying variability-intensive systems and their application to LTL model checking. In: Transactions on software engineering, pp 1069–1089

[CDEG03] Chechik M, Devereux B, Easterbrook SM, Gurfinkel A (2003) Multi-valued symbolic model-checking. ACM Trans Softw Eng Methodol 12(4):371–408

[CHL+14] Cordy M, Heymans P, Legay A, Schobbens P-Y, Dawagne B, Leucker M (2014) Counterexample guided abstraction refinement of product-line behavioural models. In: FSE'14. ACM

[CHS+10] Classen A, Heymans P, Schobbens P-Y, Legay A, Raskin J-F (2010) Model checking lots of systems: efficient verification of temporal properties in software product lines. In: ICSE'10. ACM, pp 335–344

[CIM+13] Chockler H, Ivrii A, Matsliah A, Rollini SF, Sharygina N (2013) Using cross-entropy for satisfiability. In: Shin SY, Maldonado JC (eds) Proceedings of the 28th annual ACM symposium on applied computing, SAC '13, Coimbra, Portugal, March 18–22. ACM, pp 1196–1203

[CLLC19] Cordy M, Legay A, Lazreg S, Collet P (2019) Towards sampling and simulation-based analysis of featured weighted automata. In: Proceedings of the 7th international workshop on formal methods in software engineering, FormaliSE@ICSE 2019, Montreal, QC, Canada, May 27, 2019. pp 61–64

[CN01] Clements PC, Northrop L (August 2001) Software product lines: practices and patterns. SEI Series in Software Engineering. Addison-Wesley

[Con07] Consultative Committee for Space Data Systems (CCSDS). CCSDS file delivery protocol (CFDP): Blue Book, Issue 4. NASA (2007)

[CSHL13a] Cordy M, Schobbens P-Y, Heymans P, Legay A (2013) Beyond Boolean product-line model checking: dealing with feature attributes and multi-features. In: ICSE'13. IEEE, pp 472–481

[CSHL13b] Cordy M, Schobbens P-Y, Heymans P, Legay A (2013) Provelines: a product-line of verifiers for software product lines. In: SPLC'13. ACM, pp 141–146

[DFL19] Delahaye B, Fournier P, Lime D (2019) Statistical model checking for parameterized models

[DHKP17] Daca P, Henzinger TA, Kretínský J, Petrov T (2017) Faster statistical model checking for unbounded temporal properties. ACM Trans Comput Log 18(2):12:1–12:25
[DHS18] D'Argenio PR, Hartmanns A, Sedwards S (2018) Lightweight statistical model checking in nondeterministic continuous time. In: Margaria T, Steffen B (eds) Leveraging applications of formal methods, verification and validation. Verification—8th international symposium, ISoLA 2018, Limassol, Cyprus, November 5–9, 2018, Proceedings, Part II, volume 11245 of Lecture notes in computer science. Springer, pp 336–353
[DKB14] Dubslaff C, Klüppelholz S, Baier C (2014) Probabilistic model checking for energy analysis in software product lines. In: Binder W, Ernst E, Peternier A, Hirschfeld R (eds) 13th International conference on modularity, Modularity '14, Lugano, Switzerland, April 22–26, 2014. ACM, pp 169–180
[GLS08] Gruler A, Leucker M, Scheidemann K (2008) Modeling and model checking software product lines. In: International conference on formal methods for open object-based distributed systems. Springer, pp 113–131
[GS05] Grosu R, Smolka SA (2005) Monte Carlo model checking. In: Halbwachs N, Zuck LD (eds) Tools and algorithms for the construction and analysis of systems. Berlin, Heidelberg, pp 271–286
[GWW+11] Guo J, White J, Wang G, Li J, Wang Y (2011) A genetic algorithm for optimized feature selection with resource constraints in software product lines. J Syst Softw 84(12):2208–2221
[HJK+14] Harman M, Jia Y, Krinke J, Langdon WB, Petke J, Zhang Y (2014) Search based software engineering for software product line engineering: a survey and directions for future work. In: Proceedings of the 18th international software product line conference-volume 1, pp 5–18
[HPHLT15] Henard C, Papadakis M, Harman M, Le TY (2015) Combining multi-objective search and constraint solving for configuring large software product lines. In: Proceedings of ICSE '15. IEEE Press, pp 517–528
[JLS13] Jégourel C, Legay A, Sedwards S (2013) Importance splitting for statistical model checking rare properties. In: Sharygina N, Veith H (eds) Computer aided verification—25th international conference, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013. Proceedings, volume 8044 of lecture notes in computer science. Springer, pp 576–591
[KCH+90] Kang K, Cohen S, Hess J, Novak W, Peterson S (1990) Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21
[KGS+19] Kaltenecker C, Grebhahn A, Siegmund N, Guo J, Apel S (2019) Distance-based sampling of software configuration spaces. In: Atlee JM, Bultan T, Whittle J (eds) Proceedings of the 41st international conference on software engineering, ICSE 2019, Montreal, QC, Canada, May 25–31, 2019. IEEE/ACM, pp 1084–1094
[KMSL83] Kramer J, Magee J, Sloman M, Lister A (1983) Conic: an integrated approach to distributed computer control systems. Comput Digit Tech IEE Proc E 130(1):1–10
[LDB10] Legay A, Delahaye B, Bensalem S (2010) Statistical model checking: an overview. In: Runtime verification—first international conference, RV 2010, St. Julians, Malta, November 1–4, 2010. Proceedings, pp 122–135
[LL18] Larsen KG, Legay A (2018) Statistical model checking the 2018 edition! In: Margaria T, Steffen B (eds) Leveraging applications of formal methods, verification and validation. Verification—8th international symposium, ISoLA 2018, Limassol, Cyprus, November 5–9, 2018, Proceedings, Part II, volume 11245 of lecture notes in computer science. Springer, pp 261–270
[MCP10] Muschevici R, Clarke D, Proença J (2010) Feature petri nets. In: Proceedings of the 14th international software product line conference (SPLC 2010), volume 2. Lancaster University; Lancaster, United Kingdom
[ODG+11] Oudinet J, Denise A, Gaudel M-C, Lassaigne R, Peyronnet S (2011) Uniform Monte-Carlo model checking. In: Giannakopoulou D, Orejas F (eds) Fundamental approaches to software engineering—14th international conference, FASE 2011, held as part of the joint European conferences on theory and practice of software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings, volume 6603 of lecture notes in computer science. Springer, pp 127–140
[OGB19] Oh J, Gazzillo P, Batory DS (2019) *t*-wise coverage by uniform sampling. In: Berger T, Collet P, Duchien L, Fogdal T, Heymans P, Kehrer T, Martinez J, Mazo R, Montalvillo L, Salinesi C, Tërnava X, Thüm T, Ziadi T (eds) Proceedings of the 23rd international systems and software product line conference, SPLC 2019, Volume A, Paris, France, September 9–13, 2019. ACM, pp 15:1–15:4
[ORGC14] Olaechea R, Rayside D, Guo J, Czarnecki K (2014) Comparison of exact and approximate multi-objective optimization for software product lines. In: Proceedings of the 18th international software product line conference-volume 1, pp 92–101
[OSCR12] Olaechea R, Stewart S, Czarnecki K, Rayside D (2012) Modelling and multi-objective optimization of quality attributes in variability-rich software. In: Proceedings of the fourth international workshop on nonfunctional system properties in domain specific modeling languages, pp 1–6
[PAP+19] Plazar Q, Acher M, Perrouin G, Devroey X, Cordy M (2019) Uniform sampling of SAT solutions for configurable systems: are we there yet? In: 12th IEEE conference on software testing, validation and verification, ICST 2019, Xi'an, China, April 22–27, 2019. IEEE, pp 240–251
[Pnu77] Pnueli A (1977) The temporal logic of programs. In: FOCS'77, pp 46–57
[PR01] Plath M, Ryan M (2001) Feature integration using a feature construct. SCP 41(1):53–84
[PTR+19] Pett T, Thüm T, Runge T, Krieter S, Lochau M, Schaefer I (2019) Product sampling for product lines: the scalability challenge. In: Proceedings of the 23rd international systems and software product line conference—Volume A, SPLC '19. Association for Computing Machinery, New York, pp 78–83
[RAN+15] Rodrigues GN, Alves V, Nunes V, Lanna A, Cordy M, Schobbens P-Y, Sharifloo AM, Legay A (2015) Modeling and verification for probabilistic properties in software product lines. In: HASE 2015, Daytona Beach, FL, USA, January 8–10, 2015, pp 173–180
[RKD17] Rakshit P, Konar A, Das S (2017) Noisy evolutionary optimization algorithms—a comprehensive survey. Swarm Evol Comput 33:18–45
[Shi07] Shioda S (2007) Some upper and lower bounds on the coupon collector problem. J Comput Appl Math 200(1):154–167
[SRK+12] Siegmund N, Rosenmüller M, Kuhlemann M, Kästner C, Apel S, Saake G (2012) Spl conqueror: toward optimization of non-functional properties in software product lines. Softw Qual J 20(3):487–517
[SW98] Sabin D, Weigel R (Jul 1998) Product configuration frameworks-a survey. IEEE Intell Syst Their Appl 13(4):42–49
[TAK+14] Thüm T, Apel S, Kästner C, Schaefer I, Saake G (2014) A classification and survey of analysis strategies for software product lines. ACM Comput Surv 47(1):6:1–6:45

[tBFGM16] ter Beek MH, Fantechi A, Gnesi S, Mazzanti F (2016) Modelling and analysing variability in product families: model checking of modal transition systems with variability constraints. J Log Algebra Methods Programm 85(2):287–315

[tBLLV20] ter Beek MH, Legay A, Lluch-Lafuente A, Vandin A (2020) A framework for quantitative modeling and analysis of highly (re)configurable systems. IEEE Trans Softw Eng 46(3):321–345

[TvHA+19] Thüm T, van Hoorn A, Apel S, Bürdek J, Getir S, Heinrich R, Jung R, Kowal M, Lochau M, Schaefer I, Walter J (2019) Performance analysis strategies for software variants and versions. In: Managed software evolution, pp 175–206

[VtBLL18] Vandin A, ter Beek MH, Legay A, Lluch-Lafuente A (2018) Qflan: A tool for the quantitative analysis of highly reconfigurable systems. In: Havelund K, Peleska J, Roscoe B, de Vink EP (eds) Formal methods—22nd international symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15–17, 2018, Proceedings, volume 10951 of Lecture Notes in Computer Science. Springer, pp 329–337

[VW86] Vardi MY, Wolper P (1986) An automata-theoretic approach to automatic program verification. In: LICS'86. IEEE CS, pp 332–344

[WKCK15] Wagner J, Kuznetsov V, Candea G, Kinder J (2015) High system-code security with low overhead. In: Proceedings of the 2015 IEEE symposium on security and privacy, SP '15. IEEE Computer Society, USA, pp 866–879

[YCZ10] Younes HLS, Clarke EM, Zuliani P (2010) Statistical verification of probabilistic properties with unbounded until. In: Davies J, Silva L, da Silva SA (eds) Formal methods: Foundations and applications—13th Brazilian symposium on formal methods, SBMF 2010, Natal, Brazil, November 8–11, 2010, Revised Selected Papers, volume 6527 of Lecture Notes in Computer Science. Springer, pp 144–160

[YS02] Younes HLS, Simmons RG (2002) Probabilistic verification of discrete event systems using acceptance sampling. In Brinksma E, Larsen KG (eds) Computer aided verification, 14th international conference, CAV 2002, Copenhagen, Denmark, July 27–31, 2002, Proceedings, volume 2404 of lecture notes in computer science. Springer, pp 223–235

Received 1 January 2021

Accepted in revised form 16 July 2021 by Jordi Cabot, Heike Wehrheim and Eerke Boiten

Published online 15 December 2021