Automated Repair of Security Errors in C Programs via Statistical Model Checking: A Proof of Concept

Khanh Huu The Dam[1], Fabien Duchene[1(✉)], Thomas Given-Wilson[1], Maxime Cordy[2], and Axel Legay[1]

[1] UCLouvain, Louvain-La-Neuve, Belgium
{khan.dam,fabien.duchene,thomas.given-wilson,axel.legay}@uclouvain.be
[2] University of Luxembourg, Esch-sur-Alzette, Luxembourg
maxime.cordy@uni.lu

Abstract. One major challenge in software development is finding and repairing programming errors. Recently, formal methods such as model checking have become a popular approach to finding errors due to their formal guarantees about error status and evidence of the error in the form of a trace. Another recently growing area, automated program repair, aims to fix errors using automated approaches that do not require programmer intervention. This paper gives a proof of concept that one can combine these two areas using state-of-the-art approaches in both: the discovery of program errors and traces exhibiting these errors; and automated program repair building on test cases generated from traces. This naturally links together the discovery, learning, repairing, and validation of repair in a single package.

1 Introduction

The discovery and repair of errors are major challenges in software engineering that take developers considerable effort. This is why software engineering research has intensively worked on improving techniques to assess program correctness and to repair program errors.

Formal methods have become a popular approach to finding errors in hardware and software systems. Formal methods offer great promise in that they can provide guarantees regarding the presence or absence of errors in the system. *Model Checking* (MC) [7], for instance, exhaustively explores the possible executions of the system (or a model of the system) in search of errors – typically expressed as violations of a given temporal property. Formal methods contrasts with other approaches more prominently used in software development, such as testing techniques whose offered guarantees are limited to the behavior that (manually or generated) test cases cover.

Formal methods were initially restricted to the verification of system models (e.g. transition systems) that would directly be provided by the user. More recent tools enable the verification of source code directly. These tools implement a variety of analysis approaches, including: heuristics [29]; model

T. Margaria and B. Steffen (Eds.): ISoLA 2022, LNCS 13701, pp. 279–298, 2022.
https://doi.org/10.1007/978-3-031-19849-6_17

checking [7] and variations [17]; symbolic verification [10], and runtime analysis [12]. Unfortunately, all those approaches either suffer from the state-space explosion problem (i.e. the program state space is too large to be explored exhaustively) or from difficulties in handling the memory model of the system under verification. These two limitations hinders the practical applicability of formal methods to real-world programs.

Statistical Model Checking (SMC) [14,20,33] resolves these complexity limitations, albeit at the cost of some statistical uncertainty in the results. Instead of exhaustive exploration, SMC simulates multiple executions of a system and monitors these executions in regards to expected properties. Then, SMC uses statistical algorithms to extrapolate its conclusions to the system globally. While SMC can efficiently find errors and avoid the aforementioned complexity issues, SMC provides only a statistical likelihood of the absence of errors. So far, SMC has been mostly applied to verify safety and liveness properties on (stochastic) models of systems. This includes for example, properties of biological systems [9,16], robotics [26], security [30], or of railway systems [3]. On the other hand, with the exception of [18,24], SMC has not been used to verify properties on program code. Program code analysis applications are generally limited to specific reachability properties and restricted fragments of programming languages. In particular, the verification of security properties that depend on space and time memory failures (e.g. out of bounds, overflows, read-only memory access, etc.) has not yet been explored, whereas such failures are prominent in popular languages such as C. The verification of such properties is critical because their violation leads to serious errors that conventional testing techniques (that lean on a test suite) cannot unveil [23].

The main challenge in applying SMC and other formal methods to program code is to accurately and efficiently model the possible values within the program and its complex internal information. The true possible values of a program execution can be difficult to determine because of the large domains of values to consider (e.g. all 64-bit Integers) and the difficulty of tracking complex runtime relationships between successive values. Similarly, building an accurate but abstract model of memory organization is challenging, since the compiler and operating system have specific features that are opaque to the verification engine. All these challenges ultimately lead to inaccuracies in the verification results and, in turn, to an increasing risk of overlooking pernicious security vulnerabilities.

In [5], we have proposed the *C Statistical Model Checker* (C-SMC) approach and tool in order to address these challenges and enable the discovery a wide range of security errors. C-SMC combines the inherent strengths of SMC with a runtime engine to connect the program under verification with a real executions. This allows us to reason on contextual information such as memory, stack, registers, warning flags, etc. The runtime engine feeds the SMC engine with information over the program executions without having to derive a formal model of the entire hardware and operating system's behavior. We, more specifically, use a debugging engine – viz. GDB [1] – to inform the analysis on the true instantiations of memory and internal system hardware to determine concrete values. As C-SMC builds on SMC, this approach can handle many independent

executions of the program and effectively combine the analysis results of these executions in order to determine the likelihood to satisfy given properties.

Automated Program Repair (APR) [11] is a popular area of research that aims to automate code modifications in order to fix existing errors without programmer intervention. APR is typically used once the presence of an error has been detected in the program, e.g. via formal methods or a tool such as C-SMC.

The process of most APR approaches fall in the subarea of "generate and validate". During the *generation* part the APR uses test case results in order to locate suspicious code statements possibly responsible for the error and, then, alter these statements in order to produce candidate patches. During the *validation* part the APR methods execute the program test suite on the patched code and discard all patches that make some tests fail.

Contribution. The present paper is a proof of concept of how to enrich C-SMC with a program repair module. The tool is used both to generate error trace as well as to evaluate potential patches. Concretely, C-SMC can discover errors and produce test cases that can then be the input to APR, and further C-SMC can validate the patches generated as part of the APR itself. We show how an APR approach can benefit from the statistical nature of SMC and the information that C-SMC produces regarding the errors discovered.

Conceptually, we propose to use the set of traces that C-SMC produce as inputs for APR. The traces are divided into positive traces (that raise no error) and negative traces (that exhibit the error). Hence, we lean on the capability of SMC to provide multiple traces that reveal the error – and to do so without guidance from developer-written or automatically generated tests. Based on the traces, we determine the set of suspicious statements through reverse data-flow analysis from the statement where the error was revealed.

We, then, propose a neural agent-based method that, given a set of suspicious statements, looks for the sequence of actions (statement alterations) that maximizes the likelihood to repair the error. Candidate sequences of actions produce in turn potential patches whose correctness is validated via C-SMC to speed up the process.

We evaluate our repair approach on the benchmark from [5]. Our repair approach fixes 8 out of the 9 errors and – compared to a random baseline – does so while producing 23% to 91% fewer candidates patches in all cases but one. Put together, SMC and APR effectively automate the detection and repair of memory usage-related errors to an extent that previous approaches could not achieve. Our approach complements the state of the art on software testing and repair, which has mostly focused on test-based approaches. Here we find a productive union of formal methods – viz. SMC – and APR that yield a complete package for error detection and repair.

To summarize, our main contributions are as follows. (1) Propose the first approach for automated error detection and program repair that builds on statistical model checking. Our approach can detect and repair pernicious security errors that conventional test cases are not likely to trigger. (2) Propose a novel genetic algorithmic searching approach to find an effective program repair efficiently. (3) Demonstrate an implementation of our approach on top of established

tools and technologies. (4) Evaluate our approach using a benchmark of eight memory usage-related errors and one arithmetic error. Our evaluation results suggest that (a) our detection method could find all of the nine errors, and (b) our repair method could fix eight of these errors and do so with fewer generated patches for seven of these eight errors.

Structure. The structure of the paper is as follows. Section 2 provides background and related work that are necessary to understand the paper. Section 3 presents an illustrative example used in this work. Section 4 describes the novel genetic algorithmic repair agent. Section 5 presents our implementation and experimental results. Section 6 concludes.

Note on Content. As said above, this paper is a proof of concept paper that is written to generate discussions with the community. Consequently, many formal details are left out of this presentation.

2 Background and Related Work

We consider systems/programs where executions are represented by traces, i.e., sequences of states. Each state represents the status of the system under verification at a given moment of time. States and traces can be abstracted as mathematical objects such as (sequences of) Boolean variables [7]. When considering languages such as C, a state represents the current location in the program (line of code), its memory, and the set of instructions/statements that are available from this state. A state can also contain information about the execution environment. A trace moves from on state to another state by following a transition. When considering languages such as C, such transition represents the execution of an instruction.

2.1 Essentials of Statistical Model Checking

Statistical Model Checking (SMC) [14,20,33] is a variety of probabilistic model checking (see e.g. [2]) that exploits execution traces to avoid an explicit representation of the state space. The evaluation of properties is instead calculated from an empirical distribution of the executions of the system. Given a number of statistically independent traces of a stochastic model, and the capabilities to decide whether a trace satisfies a property, it is possible to estimate the probability that the model will satisfy the property.

Note that unlike model checking [7], SMC provides results that are not complete and may only provide an approximation of the true properties of the system. Thus, an estimate of the behavior may be obtained with a specified confidence provided by, e.g. the Okamoto bound [14,25]. Further, it is possible to efficiently evaluate the truth of a hypothesis without needing to calculate the actual probability using, e.g. the sequential probability ratio test [31,33]. SMC can also be used to compute the probability of rare errors [21].

To apply SMC requires building a model of the system and expressing the properties to be checked. Typically the model can be defined as a *labeled*

transition system (LTS) and the properties expressed in a logic such as *bounded linear temporal logic* (BLTL) that can express complex behavioral properties with nested temporal causality.

2.2 Trace Execution Properties

Expressing properties formally is key in software verification. Based on definitions above, properties about a program can be defined in terms of properties over states, over individual traces, and over sets of traces. We first focus on the first two types of properties. The last one, which corresponds to the verification of the whole program, will be handled in the next (sub-)section.

A propositional logic can define properties about each state individually. To be able to consider all the states of an execution trace, this propositional logic needs to be extended. One popular extension is *Linear Temporal Logic* LTL [27]. LTL allows us to make hypotheses of unbounded traces via temporal operators. As SMC restricts to finite trace executions, we consider a bounded LTL, where each temporal operator is bounded for the number of states to which it applies.

Bounded Linear Temporal Logic (BLTL) is an enhancement of LTL that adds bounds expressed in step or time units. The syntax of BLTL is as follows:

$$\phi, \psi \quad ::= \quad p \mid \phi \vee \psi \mid \neg\phi \mid \phi \, U_{\leq t} \, \psi \mid X_{\leq t} \, \phi \, . \tag{1}$$

The p is propositional variables, disjunction $\phi \vee \psi$ and negation $\neg\phi$ are all as in LTL. The formula $X\phi$ is true if ϕ is true in the next state from the current state. The formula $\phi \, U_{\leq t} \, \psi$ is true if both: ψ becomes true before t in the sequence from the current state; and ϕ remains true in every state before the state where ψ becomes true. For a formal definition of BLTL semantics, see [35]. A BLTL formula is expressed with respect to a trace. It is also helpful to have conjunction ($\phi \wedge \psi$) and implication ($\phi \Rightarrow \psi$) that are defined in the usual manner. Similarly the *always* (G) and *eventually* (F) operators can be defined using the BLTL syntax above as follows. Eventually is defined as $F_{\leq t}\phi = \texttt{true} \, U_{\leq t} \, \phi$ and means that the formula ϕ should become true before t. Always is defined as $G_{\leq t}\phi = \neg F_{\leq t} \neg\phi$ and means that ϕ must always hold for the next t.

Let $w = s_0, s_1, ..., s_L, ...$ be an execution trace, and denote by $w^j = s_j, ..., s_L, ...$ the portion of the trace starting from j (included). The truthfulness of the formulas can be decided using the rules described in Table 1.

BLTL allows us to express reachability properties such as "the software should eventually reach a state where variable x is equal to 1". The logic can also be used to express more elaborated causalities such as "always, if the software reaches a state where x is equal to 1, it will eventually reach a state where y is equal to 1". This expressive power allows us to express a wide range of safety and security properties. BLTL properties can be verified with monitoring procedures [13]. Such procedures, inspect successive states of an execution trace until it can decide whether the property is satisfied. BLTL properties, which are a fragment of safety properties, can be monitored over finite traces using well-established techniques such as those presented in [13].

Table 1. BLTL rules.

$w \models F_{\leq t}\phi$	iff $w \models$ true $U_{\leq t}\phi$
$w \models G_{\leq t}\phi$	iff $w \models \neg(F_{\leq t}\neg\phi)$
$w \models \phi U_{\leq t}\psi$	iff $\exists i, t_0 \leq t_i \leq t_0 + t$ and $w^i \models \psi$ and $\forall j, 0 \leq j < i, w^j \models \phi$
$w \models X_{\leq t}\phi$	iff $\exists i, i = max(j \vert t_0 \leq t_j \leq t_0 + t)$ and $w^i \models \phi$
$w \models X_{\leq}\phi$	iff $w^1 \models \phi$
$w \models \phi \vee \psi$	iff $w \models \phi$ or $w \models \psi$
$w \models \phi \wedge \psi$	iff $w \models \neg\phi \vee w \models \neg\psi$
$w \models \phi \Rightarrow \psi$	iff $w \models \neg\phi \vee \psi$
$w \models \neg\phi$	iff $w \not\models \varphi$
$w \models$ true	always
$w \models$ false	never

2.3 Probabilistic Verification

We now turn to properties defined over sets of trace executions, that is properties defined on the whole system. We are interested in solving the probabilistic BLTL problem, that is to compute the probability for the system to satisfy a BLTL property ϕ. Such probability being defined as the probability that a random trace of the system satisfies ϕ.

Statistical model checking [19] has been proposed as an efficient approach to solve such problem. SMC statistically measures the truthfulness of properties over a smaller number of traces. This is done by performing a fixed number of simulations of the system and using an algorithm to estimate the probability that the system satisfies the property. As the number of observed executions is finite, this answer comes together with a confidence interval [19].

There is a wide range of statistics algorithms that can be used to estimate the probability to satisfy a given property. This includes, e.g., importance splitting and sampling [19] that can also be used to efficiently exhibit traces that do not satisfy the property. As the study of those algorithms is not the topic of this paper, we propose to work with the most simple one that is based on the Monte Carlo estimator. The estimator relies on the following proportion:

$$\bar{\gamma} = \frac{\sum_{i=1}^{N} \mathbf{1}(w_i \models \phi)}{N} \text{ where } \mathbf{1}(x) = \begin{cases} 1 & \text{if } x \text{ is true} \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

where N is the number of simulation being performed. Let γ be the true probability to satisfy ϕ and P be a probability evaluation. Let ϵ (precision) and δ (confidence)be small values. The Chernoff-Hoeffding bound [15] guarantees that $P(|\bar{\gamma} - \gamma| \geq \epsilon) \leq \delta$ is given by $N = \lceil \frac{ln2 - ln\delta}{2\epsilon^2} \rceil$. Thus, by controlling the number of trace executions verified, the user entirely controls the preciseness of the $\bar{\gamma}$ estimator.

2.4 SMC Tools

As seen in the previous section, SMC mainly depends on sub-parts that include the type of execution trace property that has to be monitored and the statistical algorithm used to compute the stochastic guarantee. Implicitly, SMC also depends on the type of system under verification and on the capacity to generate an arbitrary number of execution traces from this system.

In [34], the authors proposed `YMER`, a tool that can be used to verify BLTL properties of Markov Chains with an hypothesis testing procedure. That is, the tool decides between two hypothesis rather than computing an exact probability. In [8] Monte Carlo is applied to verify properties of Metric Temporal Logic over stochastic timed automata. Those tools have been shown to be very efficient on various problems. However, they are rather static in the sense that they do not exploit the intrinsic modularity of SMC. As an example, `YMER` does not permit replacing the hypothesis testing algorithm by a Monte Carlo one. None of those tools allows replacing classical BLTL with an algorithm that would consider debugger expression. On the top of this, none of those tools consider C code.

In [4,22], Boyer et al. introduced Plasma a Statistical Model Checking (SMC) [19] tool that can provide the ability to create custom statistical model checkers. Especially, Plasma works with open source plugins that can (1) specify the property under verification (this includes BLTL as well as many other logics such as those introduced in [6]), (2) monitor traces, (3) generate traces in an efficient manner, i.e., to minimize the number of traces needed to compute the probability to satisfy a given property. Those plugins can be customized to adapt to a wide range of properties and systems.

In a recent work [5], we have introduced C-SMC, a new open source plugin that allows us to monitor security properties of C program. C-SMC monitors BLTL properties via and extension that exploits states of the GDB debugger as input instead of Boolean abstraction. Using this, BLTL properties can refer to variables, registers, memory, locations and next instructions of a given state of the C program. In addition properties can also refer to special flags of the GDB debugger. C-SMC has been able to detect security and safety flaws such as buffer and integer overflows. It can also be used to detect overflow that depends on contextual information coming e.g. from global variables. The later one shall be showed in our motivating example.

3 Illustrative Example

Consider the illustrative binary search example given in Listing 1.1. Variable `r` should be initialized to `size-1` but is initialized to `size`. This error will lead the search to occasionally go out of the bounds of the array being inspected. Consider an array of size `10` (indexed from `0` to `9`) containing numbers `1` to `10`. If the `search` function is called with a value that is greater than the greatest value of the array (`10` in this example) then it will go out of bounds looking for the value located at `arr[size]` (`10` in this case).

This behavior will often lead to no consequences (because even if the access is out of bounds it is still checking a valid address on the stack) and thus be undetected. However, when looking for a value not contained in the array, `11` for instance, it is *possible* that the memory address located just after the end of the array contains the searched for value. In this case, the output of the search function will be invalid. This error is very hard to reproduce and to express. Indeed, the `search` function itself does not contain the information about the input values (i.e. `arr` and `size` and whether or not `size` is correct) and so most analysis engines can only warn about a *potential* out of bounds access for all instances of `arr[m]` or report no errors.

```
1  int search(int arr[],int size
       ,int elem) {
2   int l = 0;
3   int r = size; //not size-1
4   int m = 0;
5   while (r >= l) {
6      m = (l+r) / 2;
7      if (arr[m] == elem) {
8          return m;
9      } else if (arr[m] > elem)
           {
10         r = m - 1;
11     } else {
12         l = m + 1;
13     }
14  }
15  return -1;
16 }
```

Listing 1.1. Binary Search Example.

Pernicious security errors like the above are difficult to reveal through conventional testing. This is due to the fact that testing focuses on comparing outputs for a given set of inputs. As an example, we show in Listing 1.2 a set of 14 test cases that thoroughly check whether the search function works correctly when invoked on (1) all elements that the array contains and (2) close elements that the array does not contain. For conciseness, we have concatenated all these cases into a single piece of code. In this code, the `report_fail` function is used to report when some test yields an incorrect result.

```
1  int tab[8] = {-3,-1,0,4,5,6,7,8};
2  int input[14] = {-3,-1,0,4,5,6,7,8,-2,1,2,3, -(rand() % (
       MAXINT - 3)) - 4,(rand() % (MAXINT - 8)) + 9};
3  int i = 0;
4  while (i < 8) { // Test good values
5    int y = search(tab,sizeof(tab)/sizeof(tab[0]),input[i]);
6    if (y != i) { report_fail(input[i]); }
7    i++;
8  }
9  while (i < 14) { // Test random values outside the good
       values
10   int y = search(tab,sizeof(tab)/sizeof(tab[0]),input[i]);
11   if (y != -1) { report_fail(input[i]); }
12   i++;
13 }
```

Listing 1.2. Test Cases for the `search` function

Observe that all these test cases will usually pass. Thus, although developers can strenuously test the `search` function and find other potential errors, it is likely that the out of bounds access error will never occur during the test execution. This error would actually occur in the unlikely event where the searched value is not in the array but resides in memory exactly at the memory address located right after the end of the array. In such a case, the search function would return an index value equal to the size of the array, while the special value -1 is expected. This example already demonstrates that test cases are inappropriate to detect errors related to memory usage because the error is independent of the particular inputs on which the function is tested. Even if every possible `int` value was covered in the test suite, the error would likely remain undetected. To detect this error with C-SMC instead of checking the value of the output for a given input, we monitor the value of the variable `m` using the formula $m = 0|(0 <= m \& m < size$. With this formula we can check that at any moment, `m` does not go out of bounds. This allows us to detect an error that is very hard to reproduce or detect with unit tests or analysis engines.

3.1 SMC-Based Validation

Consider again the binary search example shown in Listing 1.1, where the upper bound of the binary search `r` is initialized beyond the end of the array `arr`. A natural check for array data structure is to ensure there is no access (read or write) beyond the limit of the array. This can be formulated as a BLTL property:

$$\neg(\texttt{arr[x]} \wedge \texttt{x} \geq |\texttt{arr}|), \tag{3}$$

where `arr[x]` indicates an access to the array `arr` at offset `x` and $\texttt{x} \geq |\texttt{arr}|$ indicates that `x` is greater than or equal to the size of `arr`.

In the example code, there are two accesses to an array, one at lines 7 and one at line 9. However, the value `m` that records the accessed index is modified at each loop iteration, based on the values of `size`, `elem`, and the values contained in `arr`. Hence, the above property can be reformulated as:

$$\mathrm{G}_{\leq 1000}(line = 7 \vee line = 9) => m \geq 0 \wedge m < size, \tag{4}$$

where *line* indicates the line number in C code, m and $size$ are variables m and $size$ in C code of Listing 1.1.

Assume that C-SMC tested with the following input values for `search` to generate tests (and in particular this limits the scope of our search).

```
1 int tab[8] = {1,2,3,4,5,6,7,8};
2 int v = (rand() % 10);
3 int i = search(tab, 8, v);
```

Listing 1.3. Simple Example of Search C Code Input

After running 4 tests with C-SMC we may discover that the program works properly for $\texttt{v} \in \{6, 2, 3\}$ while 9 causes a problem. Observe that applying SMC

to `search` could yield an infinite number of traces that exhibit both good and bad behavior depending on the input arguments given to the `search` function. As an example, consider the inputs given in Listing 1.2. They will generate traces that satisfy Property 4 when `v` is assigned to a value located between index 0 and 12 of the `input` array (`input[0:12]`) given in line 2 of Listing 1.2. They will generate traces that do not satisfy Property 4 when `v` is assigned to `input[13]`. In fact, all searches for a value $v \leq 8$ will provide a `GOOD` trace, and all searches for values $v > 8$ will provide a `BAD` trace. Note that the capabilities to identify `GOOD` and `BAD` traces depends on the randomized algorithm used to generate inputs. There exists various strategies to reach this objective. Including, e.g., importance splitting and sampling.

BLTL can also be used to describe other classes of properties that are beyond the scope of classical testing. As an example, it is worth checking that the program terminates after a bounded number of steps. The latter can be expressed by:

$$\mathrm{F}_{\leq 1000}(line = 15 \vee line = 16) \tag{5}$$

In addition, we can also use BLTL to specify expected (input, output) pairs just like in typical test cases. Such BLTL formulae would take the form:

$$\mathrm{G}_{\leq 1000}(line = 16 => eax = gt) \tag{6}$$

where *eax* is a reserved symbol to indicate the return value of the function and *gt* is a specific, user-defined target output.

3.2 Using Traces for Automated Program Repair

We now turn to illustrate how to repair the program. Observe that the trace monitoring process of SMC has the potential to locate error statements/lines. Those can be exploited to produce a set of possible changes, i.e., actions, for fixing the program according to the execution traces. The key principle is to start from the statement where SMC detected the violation of a property, and apply a reverse data-flow analysis to determine all previously executed statements that share a (direct or indirect) data dependency with the statement where the violation was located. It then examines the abstract syntax tree of the program in order to determine the constituents of these statements and, in turn, determines the set of actions (statement alterations) that the repair agent will consider. For example, a `BAD` trace given by C-SMC contains lines 2, 3, 5, 6, 7, 9, 10 and 12, and this trace is ended at line 7. Applying the reverse data-flow analysis from line 7, we locates error statements at lines 6, 10, 12, 3 and 2. We can then generate possible changes to each of these lines. This approach is similar to fault-spectrum localisation as presented in [32].

We want to use a repair agent that takes as input the set of possible actions that the trace analysis component identified based on the error trace. It then produces different sequences of actions (based on the taken set) and applies these sequences to generate a set of independent patches that are themselves verified

with C-SMC. This process continues until the faulty program is fixed or until the computational budget (expressed in number of produced patches) is reached.

For instance, the agent has fixed our binary search error example by modifying the line where variable `r` is initialized, as shown in Listing 1.4.

In the rest of this paper, we describe a trace analysis framework that can be used in combination with a genetic algorithm based repair agent. We then report on case studies that implement this proof of concept.

4 Designing a Repair Agent

```
1  int search(int arr[],int
       size,int elem) {
2    int l = 0;
3    int r = size - 1;//
         patched
4    int m = 0;
5    while (r >= l) {
6      m = (l+r) / 2;
7      if (arr[m] == elem) {
8        return m;
9      } else if (arr[m] >
           elem) {
10       r = m - 1;
11     } else {
12       l = m + 1;
13     }
14   }
15   return -1;
16 }
```

Listing 1.4. Binary Search Function C Code

Our objective is to develop a repair agent that exploits the information contained in execution traces generated by the SMC engine. We first show how to reason on a trace. Especially, we give intuition on how a trace can be corrected. Recall that each trace contains lines (i.e. location in program's code) which are reached by the execution and a property field which specifies the property violated by the program. Intuitively, the error statements/lines appear to `BAD` traces. The last line in a `BAD` trace indicates where the property has been violated and the error occurs. Let $\mathcal{L}$ be a set of lines which may cause the error. Let $\mathcal{T}$ be a set of `BAD` traces from C-SMC. $\mathcal{L}$ is computed as follows. Initially, $\mathcal{L} = \{\ell_i | \ell_i \text{ is the last line in the trace} t_i \in \mathcal{T}\}$. Then, $\ell \in t$ is iteratively added to $\mathcal{L}$ if there exist a $\ell' \in \mathcal{L}$, and (ℓ', ℓ) are data dependent (this can be done via classical taint analysis techniques).

Table 2 shows lines appearing in execution traces of the illustrative example (Listing 1.1) computed by C-SMC on the three properties given in Eqs. 4, 5 and 6. The error lines, which are marked with a X, are given by $\mathcal{L} = \{2, 3, 6, 7, 10, 12\}$. In order to fix the faulty program, we propose to apply an action to the error lines. An action is implemented by either replacing or inserting a new statement at a given error line. An action $a = (\ell, c)$ means that a change c is implemented at line ℓ. The change c is generated by three ways as follows. First, one can change operators of an expression. For example, if `<name> <operator> <name>` is an expression, then `<operator>` can be changed by taking an operator in either the arithmetic operators or the comparison operators. Second, one can replace the current patterns of an expression by existing patterns in the program. The replacement is applied to an expression which matches to an expression pattern in the library. The existing expression matches to a pattern if

Table 2. Error lines for the example in Listing 1.1.

Line	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Property 1 (Eq. 4)	-	X	X	-	-	X	X	-	-	X	-	X	-	-	-	-
Property 2 (Eq. 5)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Property 3 (Eq. 6)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

its abstract syntax tree can be transformed to the pattern's abstract syntax tree, or if they contain the same numbers of `<name>` and `<operator>`. For example, if (`<name> <operator> <name>`)`<operator> <name>` is an expression and it matches to the pattern `<name> <operator>`(`<name> <operator> <name>`), a change is made by the `<name>` in the pattern by the `<name>` in the expression. Third, a change can be created by inserting a new statement. Consider, e.g., the insertion of the control statements `if else`, or the addition of special statements such as `break`, `continue`, `exit`, `return`.

4.1 A SMC-Based Program Repair Algorithm

Our objective is to fix the faulty program by finding a subset of actions (called patch) from actions given by the trace analysis. By fixing the program, we mean that all properties under verification are satisfied with probability one. We here propose the use of a genetic algorithm and heuristics to generate the patch in an efficient manner.

Let $\overline{gp}$ denote a faulty program. Let $\mathcal{A}_{\overline{gp}}$ denotes a finite set of actions given by the trace analysis part to fix the faulty program $\overline{gp}$. Let $\Pi_{\overline{gp}}$ be a power set of $\mathcal{A}_{\overline{gp}}$ (the set of patches). A patch $\rho = \{a \in \mathcal{A}_{\overline{gp}}\} \in \Pi_{\overline{gp}}$ is applied to $\overline{gp}$ to transform $\overline{gp}$ to gp, i.e., $\overline{gp} \xrightarrow{\rho} gp$. Let $\mathcal{G}$ be a finite set of programs, i.e., $\mathcal{G} = \{gp | \exists \rho \in \Pi_{\overline{gp}} : \overline{gp} \xrightarrow{\rho} gp\}$. Our task is to find a patch $\rho* \in \Pi_{\overline{gp}}$ such that $\overline{gp} \xrightarrow{\rho*} gp*$ and $gp*$ is a fixed program. Note that the faulty program $\overline{gp}$ is fixed by a patch $\rho*$, i.e., $\overline{gp} \xrightarrow{\rho*} gp*$, if all the properties given in the environment are satisfied by $gp*$.

Let f be an objective function $f : \mathcal{G} \rightarrow [0.0, 1.0]^m$ where $[0, 1]^m$ denotes the output of m properties of a program $gp \in \mathcal{G}$, e.g., $f(gp) = [0.0, 1.0, 0.2]$ indicates that the program gp has 0% chance for holding the property 1, 100% chance for holding the property 2 and 20% chance for holding the property 3. The program $gp*$, i.e., $\overline{gp} \xrightarrow{\rho*} gp*$, satisfies all properties such as $f(gp*) = [1.0]^m$. Let $\mathbf{s} = f(gp)$ be a validation-state of the program gp, i.e., a vector whose components represent the probability to satisfy the properties under verification. Consider $\mathbf{s}_0 = f(\overline{gp})$ to be the initial validation-state, i.e. the initial probability value for each property that the program must eventually satisfy.

Our genetic algorithm works by evaluating a population of patches. In case, those patches do not lead to s_*, mutation and recombination operators shall be called. Given a validation-state, our objective is to derive actions that will lead to better patches and eventually to a better validation-state. For doing so, we

propose to use a Q-function [28] to specify the relation of states and actions, as follows. For every observed patch $\rho \in \Pi_{\overline{gp}}$:

$$Q(\mathbf{s}, a) = r + \gamma \max_{a'} Q(\mathbf{s}', a'), \forall a \in \rho., \tag{7}$$

where γ is a coefficient in $[0, 1]$, r and $\mathbf{s}'$ are reward and validation-state respectively. We assume the reward to be 0 when the system is not repaired and 1 if it is repaired. In practice, we will stick to those two values. In theory, a reward of $0,5$ would mean that the program is half fixed. Initially, we have $Q(\mathbf{s}, a) = 0$ for every state $\mathbf{s}$ and every action $a \in \mathcal{A}_{\overline{gp}}$. The reward r gets 1.0 if the faulty program is fixed. Otherwise, it gets -1.0.

Intuitively, $Q(\mathbf{s}, a)$ indicates the probability to choose action a at validation-state $\mathbf{s}$. The Q-function is updated for every patch and its values later are used to generate a new patch. For instance, if there is a patch $\rho = \{a_1, a_2\}$ and the state transitions $\mathbf{s}_0 \xrightarrow{a_1} (\mathbf{s}_1, r_1) \xrightarrow{a_2} (\mathbf{s}_2, r_2)$, then the Q-function is updated as follows: $Q(\mathbf{s}_0, a_1) = r_1 + max_{a'} Q(\mathbf{s_1}, a')$ and $Q(\mathbf{s}_1, a_2) = r_2 + max_{a'} Q(\mathbf{s_2}, a')$. In this work, we implement a ϵ greedy selection algorithm to choose an action for the patch generation, such that, at a state $\mathbf{s}$ an action is either chosen by $\arg\max_{a \in \mathcal{A}_{\overline{gp}}}(Q(\mathbf{s}, a))$ with a given probability ϵ or chosen any action $a \in \mathcal{A}_{\overline{gp}}$ with probability $1 - \epsilon$.

Input : $\epsilon, Q(\mathbf{s}), \mathcal{A}$
Output: an action a
if *random*() $< \epsilon$ **then**
  $a \leftarrow \arg\max_{a \in \mathcal{A}}(Q(\mathbf{s}, a))$
else
  $a \leftarrow \text{random}(\mathcal{A})$
end

Algorithm 1: ϵ greedy selection algorithm (GreedySelection())

To estimate the values of Q-function, we implement an estimator neural network $\text{NN}(\mathbf{s}, \mathbf{s}')$. The network takes the source validation-state $\mathbf{s}$ and the destination validation-state $\mathbf{s}'$ as inputs. It outputs the value of the Q-function at validation-state $\mathbf{s}$. Output values are used to compute a patch for the transition $\mathbf{s} \rightarrow \mathbf{s}'$. Hence, if there is a transition $\mathbf{s} \rightarrow \mathbf{s}'$, the values of Q-function at the state $\mathbf{s}$, i.e., $Q'(\mathbf{s}) \approx Q(\mathbf{s}, a)_{a \in \mathcal{A}_{\overline{gp}}}$, which are used to compute a patch ρ: $\mathbf{s} \xrightarrow{\rho} \mathbf{s}'$, are computed as follows.

$$Q'(\mathbf{s}) = \text{NN}(\mathbf{s}, \mathbf{s}') \tag{8}$$

For every patch, the neural network is updated by the actual values of actions in this patch with the following Mean Squared Error loss function.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \bar{Y}_i)^2, \tag{9}$$

where $Y_i = Q'(\mathbf{s}, a_i)$ and $\bar{Y}_i = Q(\mathbf{s}, a_i)$ are predicted/actual values of the action a_i at the state $\mathbf{s}$, respectively. n is the number of actions, i.e., $n = |\rho|$. Note that

Input : a set of actions $\mathcal{A}_{\overline{gp}}$, MaxPopulation, the estimator neural network NN, τ is the threshold to create an optimal patch, and the initial/final states of C program $\mathbf{s}_0, \mathbf{s}*$
Output: $\rho*$ is an optimal patch to fix the faulty program
1 $P \leftarrow \{\{a\}_{\mathcal{A}_{\overline{gp}}}\}$
2 $N \leftarrow 0$
3 isdone $\leftarrow$ False
4 **while** *not isdone and* $N \leq$ *MaxPopulation* **do**
5 $N \leftarrow N + |P|$
6 $S \leftarrow \emptyset$
7 **foreach** $\rho \in P$ **do**
8 $\mathbf{s} \leftarrow$ Evaluate(ρ)
9 UpdateQ($\mathbf{s}, \rho$, NN)
10 $S \leftarrow S \cup \{\mathbf{s}\}$
11 **if** $\mathbf{s} = \mathbf{s}*$ **then**
12 isdone $\leftarrow$ True
13 $\rho* \leftarrow \rho$
14 **break**
15 **end**
16 **end**
17 parents $\leftarrow$ SelectParent($P, S, nparent$)
18 $P \leftarrow$ CreatePopulation(parents, S, NN, $\mathbf{s}_0, \mathbf{s}*$)
19 **end**
20 **if** *not isdone* **then**
21 $Q(\mathbf{s}_0) \leftarrow \text{NN}(\mathbf{s}_0, \mathbf{s}*)$
22 $\rho* \leftarrow \{a \in \mathcal{A}_{\overline{gp}} | Q(\mathbf{s}_0, a) \geq \tau\}$
23 **end**

Algorithm 2: The patch generation algorithm

if $\mathbf{s} \xrightarrow{\rho} \mathbf{s}'$, $Q(\mathbf{s}, a_i)_{a_i \in \rho}$ is computed by Eq. 7 and $Q'(\mathbf{s}, a_i)_{a_i \in \rho}$ is a value of $Q'(\mathbf{s})$, which is predicted by NN, i.e., $Q'(\mathbf{s}) = \text{NN}(\mathbf{s}, \mathbf{s}')$.

We are now ready to present the genetic algorithm for patch generation as shown in Algorithm 2. Intuitively, patches are created from one action in the set of actions given by the trace analysis part (line 1). Then, each patch is evaluated by SMC (line 8). The validation-states obtained from this evaluation are used to update the neural network NN (line 9). If there exists a patch which can fix the faulty program, i.e., the state of the current patch is the final state $\mathbf{s}*$, the algorithm terminates (line 11–14). Otherwise, it creates a parent set from the current population, i.e., P, and generates a new population from this set (lines 17–18). If the optimal patch is not found after exceeding the limit number, i.e., MaxPopulation, the threshold τ is used to choose actions for creating an "optimal" patch (line 21–22). The threshold τ is used to select actions in a patch. We use τ to vary the number of actions in a patch. Although the patch chosen by τ may not be a good one for fixing the faulty program, it can be a recommendation for the developers to fix their code. Algorithm 3 presents the parent selection algorithm. This algorithm conserves n selected parents by exploiting an Euclidean distance between the

Input : a population P, a set S of states of population P, n is the number of selected parents
Output: a parent set is the subset of P
1 Compute the Euclidean distance between the state of each patch and the final state $[1.0]^m$.
2 Select n patches from P which have the shortest distance from the final state.

Algorithm 3: The Parent Selection algorithm (SelectParent())

Input : The parent population P,S, NN and the initial/final states of C program $\mathbf{s}_0, \mathbf{s}*$
Output: a population P'
1 $Q(\mathbf{s}_0) \leftarrow \text{NN}(\mathbf{s}_0; \mathbf{s}*)$
2 $a \leftarrow \text{GreedySelection}(\epsilon, Q(\mathbf{s}_0), \mathcal{A}_{\overline{gp}})$
3 $P' \leftarrow \{a\}$
4 **foreach** $\rho \in P$ **do**
5 $\quad \mathbf{s} \leftarrow S[\rho]$ `// the state of the C-program after applying patch` ρ
6 $\quad Q(\mathbf{s}) \leftarrow \text{NN}(\mathbf{s}, \mathbf{s}*)$
7 $\quad a_{max} = \arg\max_{a \in \mathcal{A}_{\overline{gp}}} Q(\mathbf{s}, a)$
8 $\quad a_{min} = \arg\min_{a \in \rho} Q(\mathbf{s}, a)$
9 $\quad P' \leftarrow P' \cup \{\rho \cup \{a_{max}\}\}$
10 $\quad P' \leftarrow P' \cup \{\rho \setminus \{a_{min}\}\}$
11 **end**
12 **foreach** $\rho_1 \in P$ **do**
13 $\quad \mathbf{s}_1 \leftarrow S[\rho_1]$
14 $\quad Q(\mathbf{s}_1) \leftarrow \text{NN}(\mathbf{s}_1, \mathbf{s}*)$
15 $\quad \rho'_1 \leftarrow$ Sort $\rho_1$by $Q(\mathbf{s}_1)$
16 $\quad$ **foreach** $\rho_2 \in P$ **do**
17 $\quad\quad \mathbf{s}_2 \leftarrow S[\rho_2]$
18 $\quad\quad Q(\mathbf{s}_2) \leftarrow \text{NN}(\mathbf{s}_2, \mathbf{s}*)$
19 $\quad\quad \rho'_2 \leftarrow$ Sort $\rho_2$by $Q(\mathbf{s}_2)$
20 $\quad\quad P' \leftarrow P' \cup$ switch a part of ρ'_1 and $\rho'2$
21 $\quad$ **end**
22 **end**

Algorithm 4: The Population Generation algorithm (CreatePopulation())

validation state obtained for each patch and the expected final validation-state. Algorithm 4 presents the recombination and mutation operations. Initially, the new population P' is created from the parent population starting with a patch of one action that is selected by Function GreedySelection. Then, P' is grown up by two operators (lines 1–3). The first operator is the mutation operator. It implements two operations: (1) to add an action with the highest Q-value into ρ, (2) to remove an action $a \in \rho$ with the lowest Q-value (lines 4–11). The second operator called recombination replace a pair (ρ_1, ρ_2) by a pair (ρ'_1, ρ'_2) that is created by exchanging actions which are associated with the highest Q-values in ρ_1 and ρ_2 (lines 12–21).

5 Implementation and Experimental Results

We implemented the APR process described in Sect. 4 with three modules: the environment module, trace analysis module, and agent module.

The *environment module* has two roles: C program evaluation and program repair evaluation. The C program evaluation is done by interfacing with C-SMC and inputting the C program and BLTL properties. If any property of the C program is not satisfied then repair must be triggered (otherwise the program is error free and no further action taken). The traces from C-SMC are gathered along with their result (`GOOD` or `BAD`) and these are sent to the trace analysis module. The program repair role takes a patch from the agent module and produces the state of the C program after applying the patch.

The *trace analysis module* takes a set of traces from the environment module and the C program as inputs, and produces a set of actions to fix the C program. For the motivating example, it analyses the `BAD` traces of Eq. 4 and generates actions to fix the C program at lines 2, 3, 5, 6, 7, 8, 9, 10, 12. Then, these actions are sent to the agent module for patch generation.

The *agent module* takes the lines from the trace analysis module and implements the algorithms detailed in Sect. 4 to generate a patch to apply to the C program. The agent module then sends this to the environment module to evaluate the efficacy of the patch and provide this as data for further patch generation using the genetic techniques described in Sect. 4.

5.1 Experimental Results

In this section, we evaluate the performance of the C-SMC based APR tool with Q-function on the motivating example in Listing 1.1 and in three variants. Our objective is to evaluate the performance of a Q-function with respect to a random strategy to select actions to mutate and recombine in the genetic algorithm.

In this experiment, we choose the maximum population to be 3000 patches, i.e., $MaxPopulation = 3000$ and we applied the randomized strategy for action selection. The results are reported in Table 3. The motivating example in Listing 1.1 starts in validation-state $s_0 = [0.9, 1.0, 1.0]$. It is fixed by the APR tool after 50 patches out of a population of 112 patches by the random strategy. Variant 1 is obtained by replacing $l = m + 1$ with $l = m - 1$ at line 12. The initial validation-state is given by $s_0 = [1.0, 1.0, 0.6]$. It is fixed by our tool after 72 patches while it takes a population of 210 patches to fix it with a random strategy. Variant 2 is obtained by replacing $r = size$ with $r = size - 1$ at line 3, $m = (l + r)/2$ with $m = (r - l)/2$ at line 6 and $l = m + 1$ with $l = m - 1$ at line 12. Its initial validation state is given by $s_0 = [1.0, 1.0, 0.4]$. It is fixed after 290 patches by our tool while the random strategy takes 2259 patches. Variant 3 is obtained by replacing $r = m - 1$ with $r = m + 1$ at line 10 and $l = m + 1$ with $l = m - 1$ at line 12. Its initial validation-state is given by $s_0 = [1.0, 1.0, 0.6]$. It is fixed after 290 patches by our tool. A random strategy could not fix it after a population of 3000 patches. The results show that the use of an optimized Q-function drastically improve the selection of patches.

Table 3. Comparison of the APR tool with Q-function and the random strategy.

	Size of population to fix the fault			
	Motivating example	Variant 1	Variant 2	Variant 3
APR tool with random strategy	112	210	2259	>3000
APR tool with Q function	50	72	290	2012

Table 4. Evaluation of the APR tool on other C code.

	Size of population to fix the fault	
	APR tool with random strategy	APR tool with Q-function
Binary search	112	50
Binary search (variant 1)	210	72
Binary search (variant 2)	2259	290
Binary search (variant 3)	>3000	2012
Static out-of-bounds	4	2
Random out-of-bounds	4	2
Divide by zero	1	3
Buffer overflow	25	3
Local binary search	13	10
Pointer to pointer	2	1
Read-only memory	Failed	Failed
Integer overflow	10	3

The C-SMC based APR tool with Q-function was also applied to the nine benchmark error programs from [5] that are available on GitHub[1]. The results are shown in Table 4. Observe that in all cases (except for "read-only memory") the combination of C-SMC with APR was able to not only find the error, but also successfully repair the error. The failure to repair the "read-only memory" example is due to there being no reasonable way for the patch generation algorithm to create a suitable repair. While this example was relevant to demonstrate the capabilities to detect such errors, it would be difficult to fix given the patch algorithms possible changes, and also to preserve the program's behavior. Further, in all cases (except for "divide by zero") the APR tool with Q-function proved to be more effective than the random strategy.

Due to space limitation, a detailed analysis of the benchmarks is given in the aforementioned GitHub. These results demonstrate that the approach is effective both in concept and in practice, and further that the APR Q-function strategy is an improvement over the default (random) genetic algorithm.

[1] https://github.com/csvl/bugs.

6 Conclusions and Future Work

The challenge of finding and repairing errors in software development is complex and ongoing. Formal methods such as SMC provide a strong basis for finding and proving the existence of errors, including a trace to demonstrate the error. The statistical nature of SMC allows the generation of multiple correct and erroneous traces of the program. These traces form a natural set of test cases that can be the input for APR approaches. Combining SMC with APR is an effective solution to find errors and provide the necessary information to repair them.

This paper presents a proof of concept showing that this approach is effective in discovering, learning, repairing, and validating the repair in a single tool. Further, the proposed Q-function for improving the genetic algorithmic search for a patch improves the efficacy of the patch generation. This is demonstrated by the implementation here exploiting C-SMC and demonstrating effective discovery and repair of errors.

Future Work. We would like to extend our approach to guided simulations approaches such as importance sampling and splitting. This shall help us to find error traces in a more efficient manner and hence improve the efficiency of the repair agent. We also would like to apply the work to a wider class of systems. Especially, we would be interested in applying the approach to real-time version of C. This could be done by exploiting recent work on stochastic timed automata implemented in UPPAAL-SMC [18]. The limitations of the "read-only memory" benchmark also motive the use of more complex repair tools in the agent and other methods of patch generation. Finally, observe that this paper is a proof of concept. Comparison with other tools shall be investiguated in near future.

References

1. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/. Accessed 14 Oct 2020
2. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
3. Basile, D., ter Beek, M.H., Legay, A.: Strategy synthesis for autonomous driving in a moving block railway system with Uppaal Stratego. In: Gotsman, A., Sokolova, A. (eds.) FORTE 2020. LNCS, vol. 12136, pp. 3–21. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50086-3_1
4. Boyer, B., Corre, K., Legay, A., Sedwards, S.: PLASMA-lab: a flexible, distributable statistical model checking library. In: Joshi, K., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) QEST 2013. LNCS, vol. 8054, pp. 160–164. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40196-1_12
5. Chenoy, A., Duchene, F., Given-Wilson, T., Legay, A.: C-SMC: a hybrid statistical model checking and concrete runtime engine for analyzing C programs. In: Laarman, A., Sokolova, A. (eds.) SPIN 2021. LNCS, vol. 12864, pp. 101–119. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-84629-9_6
6. Clarke, E.M., Donzé, A., Legay, A.: On simulation-based probabilistic model checking of mixed-analog circuits. Formal Methods Syst. Des. **36**(2), 97–113 (2010)

7. Clarke, E.M., Henzinger, T.A., Veith, H.: Introduction to model checking. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 1–26. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_1
8. David, A., Larsen, K.G., Legay, A., Mikucionis, M., Poulsen, D.B.: Uppaal SMC tutorial. Int. J. Softw. Tools Technol. Transf. **17**(4), 397–415 (2015)
9. David, A., Larsen, K.G., Legay, A., Mikucionis, M., Poulsen, D.B., Sedwards, S.: Statistical model checking for biological systems. Int. J. Softw. Tools Technol. Transf. **17**(3), 351–367 (2015)
10. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Sarkar, V., Hall, M.W. (eds.) Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, 12–15 June 2005, pp. 213–223. ACM (2005)
11. Goues, C.L., Pradel, M., Roychoudhury, A., Chandra, S.: Automatic program repair. IEEE Softw. **38**(4), 22–27 (2021)
12. Havelund, K.: A scala DSL for rete-based runtime verification. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 322–327. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40787-1_19
13. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_24
14. Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate probabilistic model checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 73–84. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_8
15. Hoeffding, W.: Probability Inequalities for sums of Bounded Random Variables. In: Fisher, N.I., Sen, P.K. (eds.) The Collected Works of Wassily Hoeffding, pp. 409–426. Springer, New York (1994). https://doi.org/10.1007/978-1-4612-0865-5_26
16. Jha, S.K., Clarke, E.M., Langmead, C.J., Legay, A., Platzer, A., Zuliani, P.: A Bayesian approach to model checking biological systems. In: Degano, P., Gorrieri, R. (eds.) CMSB 2009. LNCS, vol. 5688, pp. 218–234. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03845-7_15
17. Karakaya, K., Bodden, E.: Sootfx: a static code feature extraction tool for java and android. In: 21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Luxembourg, 27–28 September 2021, pp. 181–186. IEEE (2021)
18. Kulczynski, M., Legay, A., Nowotka, D., Poulsen, D.B.: Analysis of source code using UPPAAL. In: Proença, J., Paskevich, A. (eds.) Proceedings of the 6th Workshop on Formal Integrated Development Environment, F-IDE@NFM 2021, Held online, 24–25th May 2021. EPTCS, vol. 338, pp. 31–38 (2021)
19. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: an overview. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 122–135. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_11
20. Legay, A., Lukina, A., Traonouez, L.M., Yang, J., Smolka, S.A., Grosu, R.: Statistical model checking. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 478–504. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-91908-9_23
21. Legay, A., Sedwards, S., Traonouez, L.-M.: Rare events for statistical model checking an overview. In: Larsen, K.G., Potapov, I., Srba, J. (eds.) RP 2016. LNCS, vol. 9899, pp. 23–35. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45994-3_2

22. Legay, A., Sedwards, S., Traonouez, L.-M.: Plasma lab: a modular statistical model checking platform. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 77–93. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_6
23. Liu, K., Koyuncu, A., Bissyandé, T.F., Kim, D., Klein, J., Traon, Y.L.: You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In: 12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, 22–27 April 2019, pp. 102–113. IEEE (2019)
24. Ngo, V.C., Legay, A.: Formal verification of probabilistic SystemC models with statistical model checking. J. Softw. Evol. Process. **30**(3), e1890 (2018)
25. Okamoto, M.: Some inequalities relating to the partial sum of binomial probabilities. Ann. Inst. Stat. Math. **10**, 29–35 (1959)
26. Palopoli, L., et al.: Navigation assistance and guidance of older adults across complex public spaces: the DALi approach. Intell. Serv. Robot. **8**(2), 77–92 (2015)
27. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (SFCS 1977), Providence, RI, USA, pp. 46–57. IEEE, September 1977
28. Sutton, R.S., Barto, A.G.: Reinforcement Learning - An Introduction. Adaptive Computation and Machine Learning, MIT Press, Cambridge (1998)
29. Tasharofi, S., Karmani, R.K., Lauterburg, S., Legay, A., Marinov, D., Agha, G.: TransDPOR: a novel dynamic partial-order reduction technique for testing actor programs. In: Giese, H., Rosu, G. (eds.) FMOODS/FORTE -2012. LNCS, vol. 7273, pp. 219–234. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30793-5_14
30. ter Beek, M.H., Legay, A., Lluch-Lafuente, A., Vandin, A.: Quantitative security risk modeling and analysis with RisQFLan. Comput. Secur. **109**, 102381 (2021)
31. Wald, A.: Sequential tests of statistical hypotheses. Ann. Math. Stat. **16**(2), 117–186 (1945)
32. Wen, W.: Software fault localization based on program slicing spectrum. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) 34th International Conference on Software Engineering, ICSE 2012, 2–9 June 2012, Zurich, Switzerland, pp. 1511–1514. IEEE Computer Society (2012)
33. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 223–235. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_17
34. Younes, H.L.S.: Ymer: a statistical model checker. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 429–433. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_43
35. Zuliani, P., Platzer, A., Clarke, E.M.: Bayesian statistical model checking with application to stateflow/simulink verification. Formal Methods Syst. Des. **43**(2), 338–367 (2013)