
Proceedings Paper:
Stefanakos, Ioannis orcid.org/0000-0003-3741-252X, Calinescu, Radu orcid.org/0000-0002-2678-9260 and Gerasimou, Simos (Accepted: 2021) Probabilistic Program Performance Analysis. In: EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2021). IEEE , pp. 148-157. (In Press)

https://doi.org/10.1109/SEAA53835.2021.00027

Probabilistic Program Performance Analysis

Ioannis Stefanakos
Department of Computer Science
University of York, UK
is742@york.ac.uk

Radu Calinescu
Department of Computer Science
University of York, UK
radu.calinescu@york.ac.uk

Simos Gerasimou
Department of Computer Science
University of York, UK
simos.gerasimou@york.ac.uk

***Abstract*—We introduce a tool-supported method for the formal analysis of timing, resource use, cost and other quality aspects of computer programs. The new method synthesises a Markov-chain model of the analysed code, computes this quantitative model's transition probabilities using information from program logs, and employs probabilistic model checking to evaluate the performance properties of interest. Unlike existing solutions, our method can reuse the probabilistic model to accurately predict how the program performance would change if the code ran on a different hardware platform, used a new function library, or had a different usage profile. We show the effectiveness of our method by using it to analyse the performance of Java code from the Apache Commons Math library, the Android messaging app Telegram, and an implementation of the knapsack algorithm.**

***Index Terms*—program quality analysis, software performance, quantitative models, probabilistic model checking**

I. Introduction

Software is among the most flexible engineering artifacts. Computer code can run unmodified on hardware platforms as different as desktop PCs and smartphones, or with different *usage profiles* (i.e., probability distributions of the program inputs). Even when the code is modified, the change can be localised: a function or module is easy to replace with a functionally equivalent one that is, for instance, faster or more reliable. This flexibility is a great strength, but makes the analysis of the performance and other quality aspects of software systems very challenging. Changes in platform, usage profile and individual functions or modules may not affect the functionality of programs, but can impact their execution time and use of resources significantly. Given the importance of these properties, software performance analysis has been the subject of intense research for several decades [1], [2], [3], [4]. Nevertheless, the solutions delivered by this research focus on analysing the performance of software at architectural level, e.g. [5], [6], [7], [8], [9].

The equally important and challenging analysis of software performance at code level is typically carried out through program instrumentation, monitoring and profiling [10], [11], [12], [13], [14]. While these techniques produce accurate results, they have the significant drawback that the code needs to be actually executed for every platform and usage profile of interest, and after every code change.

Our paper introduces a probabilistic program performance analysis (PROPER) method that circumvents this drawback. To this end, we automatically derive a discrete-time Markov chain (DTMC) model of the analysed code, exploiting usage profile information from program logs to calculate the model's transition probabilities. Performance concerns such as the execution time or energy use of individual statements or library function calls are encoded as DTMC reward structures, and the program performance properties of interest are formalised in probabilistic temporal logic and evaluated through the probabilistic model checking of this DTMC. PROPER supports the what-if analysis of program performance in all the scenarios mentioned earlier: before deploying the code on a new platform; for an expected change in the usage profile; and to assess the performance impact of using a new implementation of a function.

As discussed in Section VII, PROPER is the first method that uses probabilistic model checking to automatically evaluate software performance properties at code level. An approach that uses probabilistic modelling for code-level analysis was proposed in [15], [16]. However, unlike our PROPER method, this approach addresses the analysis of program reliability, uses bounded loop unfolding to handle loops, and can only perform approximate analysis for programs that contain loops.

The main contributions of our paper are:

- the theoretical foundation underpinning the generation of the PROPER Markov-chain models;
- a prototype tool that implements our theoretical results, automating the PROPER synthesis of DTMC models for the performance analysis of Java methods;
- an extensive evaluation of the PROPER method and tool for code from an existing Java library, Android application, and optimisation algorithm implementation.

We organised the rest of the paper as follows. Section II introduces a running example used to illustrate the application of our performance analysis method. Section III defines the probabilistic model checking terminology and concepts required to present the PROPER theoretical foundation in Section IV and our prototype tool in Section V. Finally, we present the evaluation of our program performance analysis method in Section VI, we discuss related research in Section VII, and we conclude with a brief summary in Section VIII.

II. Running example

To illustrate the steps and application of our PROPER method, we consider the `distance1` Java method from the Apache Commons Math library.[1] This method calculates the

[1] https://commons.apache.org/proper/commons-math/

```
1  public static int distance1(int[] p1, int[] p2)
2              throws DimensionMismatchException {
3    if (checkEqualLength(p1, p2) == false) {
4      throw new DimensionMismatchException
5                (p1.length, p2.length); // @cost=7
6    }
7    else {
8      int sum = 0;
9      int i = 0;
10     while (i < p1.length) {
11       sum += Math.abs(p1[i]-p2[i]); // @time=2.5
12       i++;
13     }
14     return sum;
15   }
16 }
```

Fig. 1. Java method `distance1` from the Apache Commons Math library

L1 distance between two points in multidimensional space, which is a distance metric widely used in applications such as machine learning. As shown in Figure 1, the method receives as input two integer arrays, and checks whether the arrays have equal length in line 3. An exception is thrown if the arrays have different lengths (line 4). Otherwise, the absolute distance between the points is calculated using the *Math.abs* function (line 11) and is returned in line 14.

We suppose that the method `distance1` is used by an application for which a detailed log reflecting the method's usage profile (i.e., the typical combinations of argument lengths that `distance1` is invoked with) is available. Additionally, we suppose that the application's developers want to assess:

- the expected cost (i.e., the mean cost) for an invocation of the method, given that a cost of 7 is incurred each time when the method throws an exception in line 4;
- the method's expected execution time, if each execution of the statement from line 11 requires 2.5ns on average.

The annotations '*@cost=7*' and '*@time=2.5*' appended as comments to lines 5 and 11, respectively, are used to specify the two performance properties whose evaluation is of interest.

III. Background

Probabilistic model checking (PMC) [17] is a formal verification technique used to establish the correctness, reliability and performance of systems with stochastic behaviour, where this behaviour is formalised using Markov models. From the multiple types of Markov models that PMC can analyse, PROPER generates and uses *discrete-time Markov chains*.

Definition 1. *A discrete-time Markov chain (DTMC) over a set of atomic propositions AP is a tuple $D = (S, s_0, \boldsymbol{P}, L)$ where S is a finite set of states, $s_0 \in S$ is the initial state, $\boldsymbol{P} : S \times S \to [0,1]$ is a transition probability matrix such that, for all $s \in S$, $\sum_{s' \in S} \boldsymbol{P}(s,s') = 1$, and $L : S \to 2^{AP}$ is a state labelling function that maps each state $s \in S$ to the set of atomic propositions $L(s) \subseteq AP$ that hold in state s.*

To enable the analysis of additional types of properties, DTMCs are augmented with *cost/reward structures* [18] that associate non-negative values with their states and transitions.

Definition 2. *A cost/reward structure over a DTMC $D = (S, s_0, \boldsymbol{P}, L)$ is a pair of real-valued functions $(\underline{\rho}, \iota)$ where:*

- *$\underline{\rho} : S \to \mathbb{R}_{\geq 0}$ is a state reward function that defines the value (cost/reward) obtained when D is in state $s \in S$ for one time step.*
- *$\iota : S \times S \to \mathbb{R}_{\geq 0}$ is a transition reward function that defines the value (cost/reward) obtained each time a transition occurs.*

The properties of DTMCs analysed through PMC are formally expressed in probabilistic computation tree logic (PCTL) [19], a temporal logic with the following syntax.

Definition 3. *PCTL state formulae Φ and path formulae ϕ over an atomic proposition set AP are defined by the grammar:*

$$\Phi \ ::= \ true \mid a \mid \neg\, \Phi \mid \Phi \ \wedge \ \Phi \mid P_{\bowtie p}[\phi]$$

$$\phi \ ::= \ X \ \Phi \mid \Phi \ U^{\leq k} \ \Phi$$

and cost/reward state formulae are defined by the grammar:

$$R_{\bowtie r}[C^{\leq k}] \mid R_{\bowtie r}[I^{=k}] \mid R_{\bowtie r}[F \ \Phi]$$

where $a \in AP$, $\bowtie \in \{<, \leq, \geq, >\}$ is a relational operator, $k \in \mathbb{N} \cup \{\infty\}$, $p \in [0,1]$ is a probability bound, and $r \in R_{\geq 0}$ is a reward bound.

The PCTL semantics is defined using a satisfaction relation $\models$. Given a Markov chain $D = (S, s_0, \mathbf{P}, L)$, we have: always $D \models true$; $D \models a$ iff $a \in L(s_0)$; $D \models \neg\Phi$ iff $\neg(D \models \Phi)$; $D \models \Phi_1 \wedge \Phi_2$ iff $D \models \Phi_1$ and $D \models \Phi_2$; and $D \models P_{\bowtie p}[\phi]$ iff the probability x that paths starting at state s_0 (i.e., sequence of states $s_0 s_1 s_2 \ldots$ such that $\forall i \geq 0 : \mathbf{P}(s_i, s_{i+1}) > 0$) satisfy the path property ϕ satisfies $x \bowtie p$. The *next formula* $\mathrm{X}\ \Phi$ holds for a path if Φ is satisfied in the next state on the path; and the *until formula* $\Phi_1 \, U^{\leq k} \, \Phi_2$ holds for a path iff Φ_1 holds in the first $i < k$ path states and Φ_2 holds in the $(i+1)$-th path state. Finally, the three reward state formulae use the reward operator R to verify if the expected reward x accumulated up to timestep k, at timestep k, and accumulated to reach a state that satisfies Φ, respectively, satisfies $x \bowtie r$. Finally, the notation $P_{=?}[\,\cdot\,]$ and $R_{=?}[\,\cdot\,]$ is used to denote the value of the probability and expected reward from a PCTL state and reward state formula, respectively. Detailed descriptions of the PCTL semantics are available in [18], [19].

Our PROPER program performance analysis method uses PCTL reward reachability properties $R_{=?}[\,\cdot\,]$ to formalise performance properties of a program such as execution time, energy consumption and cost, and the probabilistic model checker PRISM [20] to evaluate these properties. However, PROPER can easily be combined with any other probabilistic model checker (e.g., MRMC [21] or Storm [22]) to support the analysis of the performance properties of interest.

IV. PROPER Performance Analysis Method

A. Method Overview

As shown in Figure 2, PROPER carries out the analysis of the performance properties of a program in three steps. In the

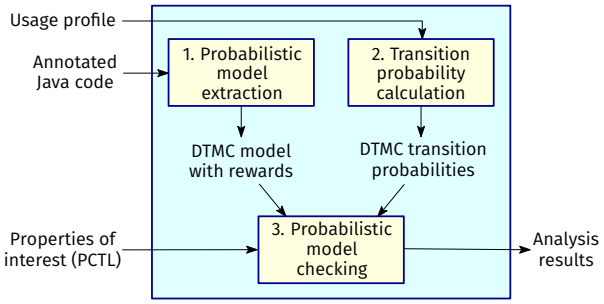

Fig. 2. PROPER program performance analysis

first step, a reward-augmented DTMC model is automatically extracted from the analysed Java code. To that end, the code is first annotated with the performance properties of interest by appending a comment of the form

$$\textit{// @property=value} \tag{1}$$

to the Java statements that these performance properties are associated with. In this PROPER annotation, *property* can be any one-word label (e.g., '*cost*' or '*time*', as shown in Figure 1), and *value* is a positive quantity such as 7 or 2.5. The same property label can be added to as many statements as required, e.g., to indicate that a non-negligible *cost* or execution `time` is associated with multiple statements.

The second PROPER step calculates the transition probabilities associated with the DTMC states that model the conditional statements and the loops from the code. This calculation is carried out based on the usage profile of the analysed code, taken or derived from program logs, where we assume that the code is appropriately instrumented to generate logs containing this information.

In the third PROPER step, the performance properties of interest, specified in PCTL, are analysed by applying probabilistic model checking to the DTMC model obtained in step 1. To enable this analysis, the transition probabilities of the DTMC are set to the probability values calculated in step 2.

The three steps of our method and further types of analyses enabled by the DTMC model are described in detail in the remainder of this section. The PROPER method is applicable to the performance analysis of single-threaded Java code. The current version of our PROPER prototype tool can handle the analysis of single Java methods that use variables declared locally or passed as arguments to the method, and whose invocations of other methods have no side effects (i.e., do not change the analysed method's variables). However, these constraints are only a limitation of the current implementation: the steps of our method do not impose any of these constraints.

B. Probabilistic model synthesis

The synthesis of the DTMC model is carried out by recursively applying the code-to-model transformation rules from Figure 3. We distinguish between four types of statements:

1) Assignment statements and method calls (with no side effects) are modelled using a single DTMC state. This state has one incoming transition (from the DTMC fragment modelling the previous statement in the code) and one outgoing transition (to the DTMC fragment modelling the next statement).
2) Conditional statements are modelled using a state with two outgoing transitions, one to the DTMC fragment modelling the statements from the 'if' branch, and one to the DTMC fragment modelling the 'else' branch. The latter DTMC fragment is empty if the else branch is missing. The derivation of the probability p_{if} from the program logs is described in the next section.
3) Loops are modelled using a state with two outgoing transitions, one leading to the DTMC fragment modelling the statements from the loop body, and one leading to the fragment modelling the statement that comes after the loop. Additionally, the outgoing transition of the DTMC fragment modelling the statements from the loop body leads back to the initial state of the loop. The derivation of the probability p_{while} for the initial state of the loop is described in the next section. Note that we only focus on 'while' loops since other types of loops (e.g., 'for' loops) can easily be converted into 'while' loops.
4) Return statements and exceptions are modelled using a state whose only outgoing transitions leads to the "end" state of the DTMC. This state, shown in dashed line in Figure 3, has a self-loop transition of probability 1, does not correspond to any statement from the code, and is used as the sink state for all outgoing transitions corresponding to final statements from the code.

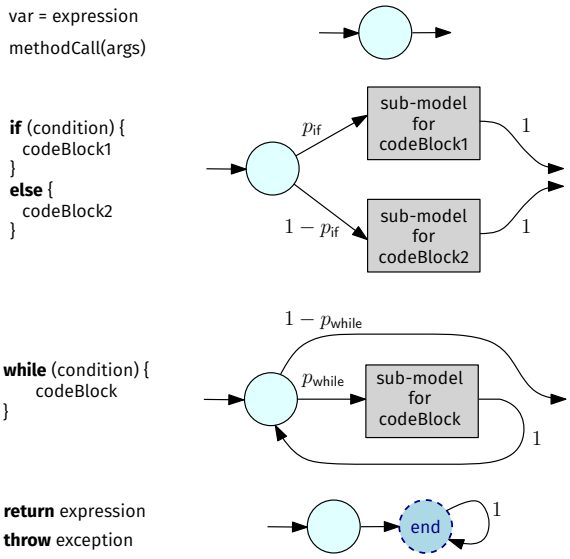

Fig. 3. PROPER code-to-model transformation rules

Example 1. *Figure 4 shows the DTMC obtained by applying these rules to the Java code from our running example. The statement modelled by each DTMC state is mentioned under the state, and the states are numbered* 0 *to* 8*.*

To allow the use of model checkers to analyse its synthesised DTMCs, PROPER uses the rules from Figure 3 to generate these DTMCs in the high-level modelling language

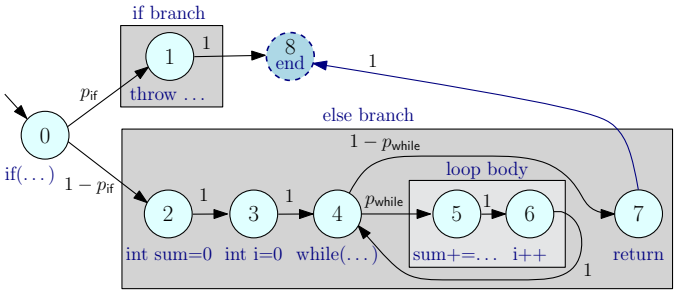

Fig. 4. DTMC model for the `distance1` Java method

```
1  dtmc
2
3  const double p1;
4  const double p2;
5  const int end_state = 8;
6
7  module distance1
8    s : [0..end_state] init 0;
9    [] s=0 -> p1:(s'=1)+(1-p1):(s'=2); //line:3
10   [] s=1 -> 1:(s'=end_state);        //line:4
11   [] s=2 -> 1:(s'=3);                //line:8
12   [] s=3 -> 1:(s'=4);                //line:9
13   [] s=4 -> p2:(s'=5)+(1-p2):(s'=7); //line:10
14   [] s=5 -> 1:(s'=6);                //line:11
15   [] s=6 -> 1:(s'=4);                //line:12
16   [] s=7 -> 1:(s'=end_state);        //line:14
17   [] s=8 -> 1:(s'=8);
18 endmodule
19
20 rewards "cost"
21   s=1 : 7;
22 endrewards
23
24 rewards "time"
25   s=5 : 2.5;
26 endrewards
```

Fig. 5. PRISM model synthesised for the `distance1` Java method

of the PRISM model checker [20], which models a system as the parallel composition of a set of modules. The state of a module is determined by a set of finite-range local variables, and its state transitions are specified by probabilistic guarded commands that modify these variables, and have the form:

$$[action]\ guard \rightarrow\ e_1 : update_1\ +\ \ldots\ +\ e_n : update_n;$$

where $guard$ is a boolean expression over all model variables. If the $guard$ is true, the arithmetic expression $e_i, 1 \leq i \leq n$, gives the probability with which the $update_i$ change of the module variables occurs. When the optional label $action$ is present, all modules comprising commands with the same $action$ must perform one of these commands simultaneously.

The DTMC produced by PROPER comprises a single PRISM module, and is generated by the function BUILDMODEL from Algorithm 1. This function takes as input a Java method, parses its code into an abstract syntax tree ast in line 33, and obtains the PRISM module commands by invoking the function SYNTHESIS. These commands—prefixed with the appropriate model preamble assembled in lines 35 and 36, and followed by the model ending built in line 37—are then returned in line 38.

SYNTHESIS starts with a $model$ comprising an empty sequence of commands (line 3). The $model$'s guarded commands are then generated by the for loop in lines 4–29. The iterations of this loop handle one statement from the ast abstract syntax tree at a time, by using the switch from lines 5–25 to handle each statement according to its type. The four cases of the switch statement correspond to the four types of statements described earlier in this section. This part of the algorithm uses the counters $stateCtr$ and $condCtr$ (initialised in line 1) to keep track of the index for the states and transition probabilities being generated, respectively.

A single guarded command is generated if the processed statement $stmt$ is an assignment or a method call (line 7). If $stmt$ is a conditional, a new state with two outgoing transitions is created (line 9). The first transition, corresponding to the 'if' branch of the conditional, points to the next state with a probability $\mathrm{p}_{condCtr}$. The second transition, corresponding to the 'else' branch (if this branch exists) or to the statement after the conditional (otherwise), has probability $1-\mathrm{p}_{condCtr}$, points to a state identified (in line 12 if the else branch is missing, or in line 14 otherwise) after the $model$ commands for the 'if' branch are obtained by invoking SYNTHESIS recursively in line 10. These commands are appended to the $model$ in line 12 if the 'else' branch is missing, or in line 14 otherwise. In the latter case, the commands for the 'else' branch are then generated (line 15) and added to the $model$ (line 16).

The $model$ commands when $stmt$ is a loop statement are generated in lines 19–22, by following a similar process to that used for a conditional statement, except that the last state modelling the loop body has its only outgoing transition leading back to the first state modelling the loop (line 22). To allow this, the $stateCtr$ value for the first state of the loop commands is recorded in line 19.

Finally, when $stmt$ is a return or an exception statement, a new $model$ state is created (line 24). The only outgoing transition of this state points to the end_state of the $model$. This state is declared in the $model\_preamble$ in line 35 of BUILDMODEL and is generated in the $model\_ending$ in line 37 of BUILDMODEL, after the execution of SYNTHESIS finishes and the index of this state is known.

To enable the generation of the reward structures for the $model$, SYNTHESIS records the reward annotations from all statements (lines 26–28) into the $rewards$ dictionary initialised in line 1. The reward structures are then included in the $model\_ending$ by invoking the auxiliary function ADDREWARDSTRUCTURES in line 37 of BUILDMODEL. Finally, the auxiliary function ADDVARIABLES is invoked in line 35 of BUILDMODEL to create the variable declarations for all unknown transition probabilities generated by SYNTHESIS for conditional statements and loops. The format of the reward structures and variable declarations generated by the two auxiliary functions is illustrated in the following example.

Example 2. *Figure 5 shows the PRISM-encoded DTMC model generated by Algorithm 1 for the* `distance1` *Java method from our running example. The model has two reward*

Algorithm 1: DTMC model synthesis (shaded strings indicate literals included in the model)

```
1  stateCtr=0, condCtr=0, rewards = ()
2  function SYNTHESIS(ast)
3      model = ''
4      for each stmt ∈ ast do
5          switch (stmt)
6              case assignment or methodCall :
7                  model += '[] s=' + (stateCtr++) + '→ 1:(s'=' + (stateCtr) + ');'
8              case conditional :
9                  model += '[] s=' + (stateCtr++) + '→ p' + condCtr + ':(s'=' + (stateCtr) + ')+(1-p' + (condCtr++) + '):(s'='
10                 if_branch_model = SYNTHESIS(stmt.thenStmts);
11                 if ¬stmt.hasElseBranch then
12                     model += (stateCtr) + ');' + if_branch_model
13                 else
14                     model += (++stateCtr) + ');' + if_branch_model + '[] s=' + (stateCtr − 1) + '→ 1:(s'='
15                     else_branch_model = SYNTHESIS(stmt.elseStmts)
16                     model += (stateCtr) + ');' + else_branch_model
17                 end
18             case loop :
19                 loopStartingState=stateCtr
20                 model += '[] s=' + (stateCtr++) + '→ p' + condCtr + ':(s'=' + (stateCtr) + ')+(1-p' + (condCtr++) + '):(s'='
21                 loop_body_model = SYNTHESIS(stmt.loopBody)
22                 model += (++stateCtr) + ');' + loop_body_model + '[] s=' + (stateCtr − 1) + '→ 1:(s'=' + loopStartingState + ');'
23             case return or exception :
24                 model += '[] s=' + (stateCtr++) + '→ 1:(s'=end_state);'
25         end
26         while reward = stmt.getNextReward do
27             rewards[reward.name] += (stateCtr − 1, reward.value)
28         end
29     end
30     return model
31 end
32 function BUILDMODEL(method)
33     ast = PARSE(method)
34     model_commands = SYNTHESIS (ast)
35     model_preamble = 'dtmc'+ ADDVARIABLES(condCtr) + 'const int end_state = ' + stateCtr + '; \n'
36     model_preamble += 'module'+ ast.methodName + '\n s : [0..end_state] init 0; \n'
37     model_ending = '[] s=' + stateCtr + '→ 1:(s'=' + stateCtr + ');\n endmodule' + ADDREWARDSTRUCTURES(rewards)
38     return model_preamble + model_commands + model_ending
39 end
```

structures, corresponding to the `time` *and* `cost` *annotations from the Java code in Figure 1. The transition probabilities* `p1` *and* `p2` *correspond to the 'if' statement and 'while' loop from the Java code. Their values depend on the usage profile of the code, and are determined as described in the next section.*

C. Transition probability calculation

The transition probabilities for the DTMC states modelling conditional statements and loops are calculated from the usage profile of the analysed code. PROPER requires a usage profile that provides, for each conditional statement and loop, the (expected) number of executions of the 'if' branch of the conditional statement or of body of the loop, respectively, over N_0 executions of the analysed code. There are multiple ways in which this usage profile can be obtained:

- directly from the program logs, if the code is instrumented to log this information;
- through a technique called *model counting* [23], which can calculate expected values for these counts from empirical probability distributions of the program inputs, where these distributions are taken from program logs;
- by Monte Carlo simulation applied to a simplified version of the code (from which the statements with no impact on the required execution counts are removed), where the program inputs for the simulation are drawn randomly from logs that reflect the empirical probability distributions of these inputs.

Give a usage profile with these characteristics, consider a set of $n \geq 1$ nested conditional statements and/or loops from the analysed code. If the execution counts for these conditional statements/loops are N_1, N_2, ..., N_n,[2] then the transition probability associated with the i-th conditional statement/loop is calculated as:

$$p_i = \begin{cases} \frac{N_i}{N_{i-1}}, & \text{if statement } i \text{ is a conditional} \\ \frac{N_i}{N_{i-1}+Ni}, & \text{otherwise (if statement } i \text{ is a loop)} \end{cases} \quad (2)$$

where $1 \leq i \leq n$. For conditional statements and loops that are not nested within other conditional statements/loops (such as those from our running example), the number of executions of the analysed code is used in (2), i.e., $N_{i-1} = N_0$.

Example 3. *Suppose that the usage profile for the Java method* `distance1` *from our running example indicates that, across* $N_0 = 10,000$ *invocations of the method, the if branch of the*

[2] For a conditional statement, the count is of the number of executions of the if branch, if this branch is part of the statement nest, or of the else branch, if this branch exists and is part of the statement nest. For a loop, the count is of the number of executions of the statements within the body of the loop.

conditional statement starting in line 3 from Figure 1 was executed $N_1 = 15$ times, and the body of the while loop from lines 10–13 was executed $N_2 = 254,000$ times. Accordingly, the values of the unspecified transition probabilities for the DTMC model from Figure 5 are given by $p1 = \frac{N_1}{N_0} = \frac{15}{10,000} = 0.0015$ *and* $p2 = \frac{N_2}{N_1'+N_2} = \frac{254,000}{(10,000-15)+254,000} = 0.9621$.

The following result shows that the PROPER probabilistic model synthesised in Section IV-B and instantiated with the probabilities calculated above can be used to determine the performance properties of the code under analysis.

Theorem 1. *Given a Java method annotated with a performance property* (1)*, its DTMC D generated by Algorithm 1, and the DTMC transition probabilities* (2) *calculated for a usage profile of the method, the expected value of the property for this usage profile is given by the probabilistic model checking of the reward property $R_{=?}[F\ s = end\_state]$ over D.*

Proof. The performance properties analysed by our PROPER method are *additive*, i.e., if the execution time, cost or resource use under analysis is due to multiple program statements, the analysis can be carried out by adding up the property values determined separately for each of these statements. As such, we only need to prove the theorem for a property that associates a value $v > 0$ with a single program statement. We consider the general case where this statement is part of the body of $n \geq 0$ nested loops and/or conditional statements. Given N_0 program executions representative for the analysed usage profile, let $N_i \geq 0$, $1 \leq i \leq n$, be the total number of executions of the n-th such loop/conditional statement over the N_0 program executions.

The relevant part of the DTMC model D generated for the analysed code (i.e., the part modelling the n loop/conditional statement nest) comprises (a) n nested loop/conditional statement model constructs with the structure from Figure 3 and probabilities $p_{\mathsf{while}} = p_1, p_2, \ldots, p_i$ given by (2); and (b) a reward structure that associates the value v with a state within the innermost of these constructs. As such, the probabilistic model checking of the reward property $R_{=?}[F\, s = end\_state]$ over D yields the expected reward value:

$$r = f_1 f_2 \ldots f_n \cdot v, \tag{3}$$

where f_i is a multiplicative factor associated with the i-th model construct, $1 \leq i \leq n$. For a model construct associated with a loop, this factor is given by

$$\begin{aligned} f_i = p_i(1 + p_i(1 + p_i(\ldots))) &= \lim_{k\to\infty} \left(p_i \tfrac{1-p_i^k}{1-p_i}\right) \\ &= \tfrac{p_i}{1-p_i} = \tfrac{\frac{N_i}{N_{i-1}+N_i}}{1-\frac{N_i}{N_{i-1}+N_i}} = \tfrac{N_i}{N_{i-1}} \end{aligned} \tag{4}$$

due to the repeated execution of i-th loop with probability p_i. For a model construct associated with a conditional statement, the factor is simply $f_i = p_i = \frac{N_i}{N_{i-1}}$. Replacing these factor values in (3) gives an expected reward value

$$r = \tfrac{N_1}{N_0} \cdot \tfrac{N_2}{N_1} \cdot \ldots \cdot \tfrac{N_n}{N_{n-1}} \cdot v = \tfrac{N_n}{N_0} \cdot v, \tag{5}$$

i.e., the mean value of the analysed property for the considered usage profile (because the value v is associated with a statement executed N_i times across N_0 program executions). □

D. Probabilistic model checking

In this PROPER step, we use a probabilistic model checker, e.g., PRISM [20] or Storm [22], to analyse the PCTL-encoded performance properties of interest over the DTMC synthesised by Algorithm 1, with the probabilities computed in (2).

Example 4. *Consider again our running example (Section II). Determining the values of the 'cost' and 'time' properties specified using PROPER annotations in Figure 1 involves the probabilistic model checking of the reward PCTL properties $R\{\text{“}cost\text{”}\}=?[F\ s = end\_state]$ and $R\{\text{“}time\text{”}\}=?[F\ s = end\_state]$ over the DTMC model from Figure 5. To carry out these analyses for the usage profile from Example 3, the unspecified DTMC probabilities need to be initialised such that $p1 = 0.0015$ and $p2 = 0.9651$. The results of these analyses (using PRISM) are $cost = 0.0105$ and $time = 69.0275$.*

E. Further application scenarios

Besides supporting the analysis of the performance properties specified by the initial code annotations, the PROPER DTMC model can be reused for additional analyses in scenarios encountered in software engineering practice. One such scenario occurs when a method invocation from the analysed code is replaced with the invocation of a functionally equivalent method with different performance characteristics.

Example 5. *The impact of replacing the Math.abs function call from line 11 of the `distance1` Java method from Figure 1) with a call to the improved function FastMath.abs can be analysed using the same DTMC model as in Example 4, after only updating the reward value from line 22 of the model (see Figure 5) to match the specifications of the new function.*

Another scenario in which the DTMC model can be reused is when the code needs to be deployed on a new hardware platform with different quality attributes. As shown by the following example, new quality properties can be analysed in this scenario by defining new reward structures for the DTMC.

Example 6. *Suppose that the application using the method `distance1` from our running example needs to be deployed on a smart phone on which its invocations of `checkEqualLength` and `Math.abs` consume 90 and 85 units of energy, respectively. The expected energy consumption of `distance1` can be predicted before actually running the application on the new hardware, by simply augmenting the DTMC model from Figure 5 with the new rewards structure*

```
rewards "energy"
  s=0 : 90;
  s=5 : 85;
endrewards
```

where $s = 0$ and $s = 5$ are the DTMC states modelling the statements that use `checkEqualLength` and `Math.abs`.

V. Implementation

To automate the performance analysis of probabilistic programs using PROPER, we implemented a tool with the architecture in Fig. 2. Our PROPER tool uses JavaParser[3] to parse the Java code of interest and generate the corresponding DTMC models (Section IV-B). We developed a customised Monte Carlo simulation method in Java to calculate the transition probabilities (Section IV-C) and employ the probabilistic model checker PRISM [20] to analyse properties of interest (Section IV-D). The PROPER open-source prototype tool, the full experimental results summarised next, additional information about our approach and the case studies used for its evaluation are available at https://github.com/is742/PROPER.

VI. Evaluation

A. Research Questions

We evaluated PROPER by performing extensive experiments to answer the following research questions.

RQ1 (Accuracy): How accurately does PROPER support the analysis of non-functional properties of interest? We used this research question to establish if our method can achieve the same accuracy levels compared to the standard practice of analysing quality properties of interest via simulation or by running the system in normal working conditions.

RQ2 (Decision-Making): How effective is PROPER to support the intended uses? To support software engineers in their decision-making, our PROPER method should successfully predict the effect of changes within the code and within the code's operating environment.

RQ3 (Efficiency): What are the computational overheads of PROPER? We evaluated the execution time and memory footprint incurred by PROPER and compared them against the overheads incurred by simulation or real system execution.

B. Experimental Setup

We applied PROPER in multiple scenarios using Java source-code adapted from four Java libraries and applications:

1) The `distance1` Java method from the Apache Commons Math library[4] (see running example in Section II).
2) The `getDevicePerfomanceClass` method from the Android messsaging app Telegram[5] (abbreviated '`devPerf`' in this section). Given a mobile device in which Telegram operates, this method identifies the specifications of the operating device and determines its performance class. The performance categories that a device can be linked with are: low, average and high. In our case study, we assumed that based on the result returned by this method, Telegram adapts to the specifications and shifts the performance of some of its features. Additionally, we introduced a new performance category (very high) to show the applicability of our approach in cases where additional code is being introduced.
3) The `fst` method from the Apache Commons Maths library. This method implements the fast sine transformer algorithm for one-dimensional real data sets.
4) An implementation of the widely used dynamic-programming knapsack algorithm (`knapsackDP`) taken from a public tutorial series on GitHub.[6]

Table I provides an overview of our case studies, along with a list of identified performance properties of interest, formally expressed in PCTL [19], that can be evaluated using our tool-supported PROPER method.

For the evaluation of all research questions, we assume that the values of the rewards of interest linked with a service or state, e.g., cost, execution time or energy consumption, are obtained from the service provider, and that logs capturing the program's usage profile are available. In the `distance1` case study, we measure the expected time and cost associated with the *Math.abs* method and to throwing the exception, respectively. In the `devPerf` case study, we are interested in the expected energy consumption of running the code, due to an *Animations* method that sets the level of the application's visual quality. Depending on its input mode, each instance of this method is linked to a different amount of energy (28, 34, 40 or 48). Similarly, in the `fst` case study, we measure the expected time associated with the *FastMath.sin* method (where each execution takes 1.5 time units), together with the expected cost of reaching any of the two exceptions (of cost 5 each). Finally, in the `knapsackDP` case study, we are interested in the expected energy consumption due to a *display* method located in the code (whose executions use 67 units of energy each), and in the expected time associated with the *Math.max* method, each invocation of which takes 2 time units.

All experiments were run on a macOS Big Sur Macbook Pro with 2 GHz Dual-Core Intel Core i5 CPU and 8 GB RAM. The source code, Markov models, data used for the experimental evaluation and full experimental results are publicly available in our GitHub repository https://github.com/is742/PROPER.

C. Results and Discussion

RQ1 (Accuracy). We answer RQ1 by comparing the PROPER results with those produced by simulating the execution of the programs from Table I in a realistic environment and with a suitably instrumented operational profile. To achieve this, we deployed the code of each program in a mobile device in the form of a stand-alone application using the Android studio's emulator and performed simulation directly on the device.

Table II shows the results obtained from the verification of properties of interest using PROPER and simulation. To execute the PMC step of PROPER (Section IV-D), we used the PRISM model checker [20] and provided as input to the DTMC models the probabilities obtained during the transition probability calculation step of the approach (Section IV-C). Then, we quantified the properties shown in Table I. To obtain the values from simulation, we performed 10^4 simulated runs of each case study. The input for the methods during every simulation was randomly selected from the program's log. Due

[3] https://javaparser.org [4] https://commons.apache.org/proper/commons-math/
[5] https://github.com/DrKLO/Telegram/
[6] https://github.com/eugenp/tutorials/

TABLE I
DESCRIPTION OF CASE STUDIES' MODELS AND PROPERTIES OF INTEREST EXPRESSED IN BOTH NATURAL LANGUAGE AND PCTL.

Case studies	#states	#trans.	#linesOfcode	Performance property description	PCTL
distance1	8	10	16	What is the expected time? What is the expected cost?	$R\{\text{“}time\text{”}\} = ?\ [\ F\ s = end\_state\ ]$ $R\{\text{“}cost\text{”}\} = ?\ [\ F\ s = end\_state\ ]$
devPerf	17	21	40	What is the expected energy consumption?	$R\{\text{“}energy\text{”}\} = ?\ [\ F\ s = end\_state\ ]$
fst	30	35	47	What is the expected time? What is the expected cost?	$R\{\text{“}time\text{”}\} = ?\ [\ F\ s = end\_state\ ]$ $R\{\text{“}cost\text{”}\} = ?\ [\ F\ s = end\_state\ ]$
knapsackDP	18	23	29	What is the expected energy consumption? What is the expected time?	$R\{\text{“}energy\text{”}\} = ?\ [\ F\ s = end\_state\ ]$ $R\{\text{“}time\text{”}\} = ?\ [\ F\ s = end\_state\ ]$

TABLE II
COMPARISON IN ACCURACY OF RESULTS OBTAINED USING PROPER AND SIMULATION.

Properties	PROPER				Simulation			
	distance1	fst	knapsackDP	devPerf	distance1	fst	knapsackDP	devPerf
R{“time”}=?[F s=*end_state*]	2.5	1.14	21.96	N/A	2.5	1.11	21.19	N/A
R{“cost”}=?[F s=*end_state*]	4.66	1.91	N/A	N/A	4.63	1.89	N/A	N/A
R{“energy”}=?[F s=*end_state*]	N/A	N/A	735.93	30.96	N/A	N/A	710.24	31.02

to the randomness of selecting data from the log, the results, as expected, were slightly different every time we performed the simulation. To alleviate this validity threat (cf. Section VI-D) and to increase the accuracy of simulation results, we created 10 sets of simulated runs of 10^4 code executions and calculated the average property values.

As can be seen from the results in Table II, the quality properties evaluated using PROPER are within 3.5% of those obtained in simulation. The small differences in the results in Table II are due to the randomness in simulation. Increasing the number of simulation runs would reduce further the delta; experimenting further on this research thread is part of our future work. These results confirm the capability of our approach to accurately analyse performance properties of probabilistic programs without the need to execute the source code in simulation.

RQ2 (Decision-Making). We illustrate the capabilities of PROPER and how it can help software engineers to make informed decisions using two modification scenarios (*Scenario A* and *Scenario B*) that frequently occur in the domains of product obsolescence [24] and software modernisation [25].

In *Scenario A*, software engineers replace one of the external methods used by the program of interest to optimise the requirements defined during the design phase of the program. Such a modification may involve, for example, replacing an existing external method with a faster alternative to reduce response time, or using a less reliable but cheaper method to reduce the operational cost, provided that the method does not critically affect the application's functionality. Since the operational profile of the application does not change, and given the reward values for the new method by the service providers in the form of a service-level agreement, we can use PROPER to quantify quality properties of interest without simulating the code's execution. This will not only save time and effort, but it will also enable engineers to verify additional properties that were not considered during system design.

Table III shows the updated results in bold obtained during *Scenario A*. In `distance1` case study, we used the method *FastMath.abs* that offered improved execution time (=1.8) instead of *Math.abs* whose execution time was 2.5. The expected cost was not affected by this change, as it is associated with the exception. In the `fst` case study, we replaced the *FastMath.sin* method with the slower (=2.2 per invocation) but more reliable *Math.sin* method which resulted in a slight increase in execution time (i.e., 1.14 with *FastMath.sin* vs 1.67 with *Math.sin*). Similarly to `distance1`, the cost was not affected. The change in the `knapsackDP` program affected both the expected time and energy consumption. In particular, we introduced the faster (=1.3) method *FastMath.max* instead of the *Math.max* method, which resulted in reduced execution time (14.27 vs 21.96). We also updated the *display* method to increase performance using a more computationally-expensive method (=78), which led to increased overall energy consumption (735.03 vs 856.76 before and after the change, respectively). Finally, in the `devPerf` program we assumed that the *Animations* method was updated to offer better optimisations making use of the increased number of cores in modern mobile devices (=23,30,35,43). This change resulted in a decrease of energy consumption (30.96 vs 26.27) in all its invocations.

In *Scenario B*, software engineers do not make any internal changes in the code; instead, the application is deployed in a new device with different capabilities and specifications. Such scenarios may arise when transferring the same software between mobile devices or when deploying the same software in robotic systems with different performance, memory, networking and other characteristics (e.g., a robot using a Raspberry Pi 4 and another using a Raspberry Pi Zero).

Since the applied changes are only external and the operational profile of the application does not change, we can employ PROPER and obtain the updated values for the quality attributes of interest. Table III (*Scenario B*) shows in bold the updated values of the performance properties for the four applications assuming that they have been deployed in a device with reduced hardware performance.

The experimental results from both scenarios show that PROPER can provide useful insights on the impact of potential internal changes in the code or external in the operating environment of an application. The impact of such changes can

TABLE III
RESULTS OBTAINED USING PROPER FOR TWO DIFFERENT SCENARIOS. SCENARIO A: REPLACEMENT OF A PROGRAM METHOD WITH A FUNCTIONALLY-EQUIVALENT METHOD WITH DIFFERENT PERFORMANCE CHARACTERISTICS. SCENARIO B: PROGRAM DEPLOYMENT ON A NEW HARDWARE PLATFORM WITH DIFFERENT QUALITY ATTRIBUTES.

Properties	**Scenario A**				**Scenario B**			
	distance1	fst	knapsackDP	devPerf	distance1	fst	knapsackDP	devPerf
R{"time"}=?[F s=*end_state*]	**1.8**	**1.67**	**14.27**	N/A	**3.2**	**1.97**	**30.75**	N/A
R{"cost"}=?[F s=*end_state*]	4.66	1.91	N/A	N/A	4.66	1.91	N/A	N/A
R{"energy"}=?[F s=*end_state*]	N/A	N/A	**856.76**	**26.27**	N/A	N/A	**900.69**	**35.73**

TABLE IV
TIME AND MEMORY CONSUMPTION COMPARISON BETWEEN PROPER AND SIMULATION.

Properties	**PROPER**				**Simulation**			
	distance1	fst	knapsackDP	devPerf	distance1	fst	knapsackDP	devPerf
Execution time (seconds)	0.003	0.005	0.01	0.004	256.2	193.2	2826.6	264.6
Memory consumption (MB)	[12-39]	[12-36]	[12-37]	[11-36]	[3.4-37.7]	[4.3-37.4]	[6-49]	[2.7-36]

be assessed without updating the code or deploying it in the target hardware platform, thus reducing significantly the effort and cost in analysing performance properties of interest. These results provide evidence how PROPER can assist software engineers in making informed decisions.

RQ3 (Efficiency). To answer RQ3, we measured the execution time and memory consumption of running the code in real time with obtaining results using PROPER. To measure the code's execution time we used the *currentTimeMillis* method from Oracle's *System*(https://docs.oracle.com/javase) class and for the probabilistic model checking step of PROPER we used the output log from PRISM [20] to obtain both the time needed for model construction and model checking for each of the specified properties. We measured the memory consumption using the *JavaVisualVM* profiling tool which comes with the Java Development Kit (JDK). Also, we used the method *sleep* from Oracle's *Thread*class to simulate a server response time of 2ms for each function invocation.

The experimental results in Table IV show that PROPER is much faster than executing the code in its operating environment. In terms of memory, PROPER independent of the case study consumes on average the same amount of memory. With simulation, however, the `knapsackDP` method which had a longer execution time than the rest case studies, showed an increase in the min and max values of used memory too.

D. Threats to Validity

Construct validity threats may arise from the construction of the case studies' models based on the selected Java code. To mitigate this threat, all use cases are based on real-world applications, and the produced models refer to parts of these applications' source code.

Internal validity threats can originate from obtaining inaccurate results via simulating the code's execution. To mitigate these threats, we performed simulation up to 10^4 times. Additionally, we created 10 sets of these simulation runs and calculated the average of their output values.

External validity threats might be due to the difficulty of representing part of a Java application's source code as a DTMC model. To mitigate this threat, we carefully evaluated each model to its respective code method, and built an automated implementation of PROPER to assist us in the code-to-model transformation process. However, further experiments are needed to evaluate our method for additional code samples.

VII. RELATED WORK

Probabilistic software analysis (PSA) [26] has been used successfully in domains including testing, cryptographic protocols, cyber-physical systems, and reliability analysis [27]. However, to the best of our knowledge, our method is the first PSA approach that synthesises a probabilistic model directly from source code to verify performance properties of interest. The only related work we are aware of belongs to the areas of *software maintenance* [28] and *software reliability analysis* [29]. Unlike our approach, research in these areas uses mostly techniques such as symbolic execution [30], [31] and simulation [32], [33], rather than probabilistic model checking.

Probabilistic symbolic execution [30] is an extension of symbolic execution that allows probabilistic reasoning. A probabilistic environment for Java based on symbolic execution is proposed in [31]. This framework can handle probabilistic programming features, and be used for the encoding and analysis of DTMCs, Bayesian Networks, etc. Additionally, [15] introduces a general methodology that uses symbolic execution of source code for extracting failure and success paths that can be used for probabilistic reliability assessment, against relevant usage scenarios. [16] extends the previous approach by building upon the symbolic execution framework with the aim of computing a precise numeric characterisation of program changes. However, the focus of these approaches is on reliability. In contrast, PROPER targets the analysis of performance-related quality properties. Also, the bounded exploration depth set during symbolic execution can lead to loss of information necessary for quality property analysis, while our approach achieves precise exploration of loops.

The reliability assessment approach from [34] uses software metrics for reliability modelling. This work differs from ours as it uses DTMC models built around the control transfer relationship between components and it is not directly applied on source code. Furthermore, [35] introduces reduction methods for probabilistic programs that operate purely on a syntactic level and [33] proposes a framework of incorporating path testing into reliability estimation for modular software systems. Also, [32] develops simulation procedures to assess

the impact of individual components on the reliability of an application in the presence of fault detection and repair. These approaches differ from ours as they focus on techniques that improve the calculation and monitoring of reliability.

Finally, architecture-level probabilistic analysis of nonfunctional properties has been proposed, e.g., [36], [37], [38], [39]. These approaches are complementary to PROPER, as they do not support the code-level analysis of software performance and other quality properties.

VIII. Conclusion

We presented PROPER, a tool-supported method for the automated performance analysis of probabilistic programs. PROPER synthesises a DTMC using code annotated with performance properties of interest (e.g., timing, resource use, cost), calculates the transition probabilities of the DTMC using program logs, and executes probabilistic model checking to quantify these properties. We evaluated PROPER on four applications and demonstrated how it can support the performance analysis in scenarios involving changes in hardware platforms, function libraries or usage profile. Our future work includes (1) extending PROPER to support analysis of reliability properties; (2) investigating methods to support the computation of confidence intervals of performance properties [40]; (3) applying PROPER to other applications and scenarios, and assessing its scalability to larger programs; and (4) validating PROPER in studies where it is used by practitioners.

Acknowledgements

This work was supported by Microsoft Research through its PhD Scholarship Programme.

References

[1] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: A survey," *Transactions on Software Engineering*, vol. 30, no. 5, pp. 295–310, 2004.

[2] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Performance evaluation*, vol. 67, pp. 634–658, 2010.

[3] M. Plauth, L. Feinbube, and A. Polze, "A performance survey of lightweight virtualization techniques," in *European Conference on Service-Oriented and Cloud Computing*, 2017, pp. 34–48.

[4] D. Arcelli, "Exploiting queuing networks to model and assess the performance of self-adaptive software systems: a survey," *Procedia Computer Science*, vol. 170, pp. 498–505, 2020.

[5] V. Cortellessa, A. Di Marco, and P. Inverardi, "Integrating performance and reliability analysis in a non-functional MDA framework," in *Fundamental Approaches to Software Engineering*, 2007, pp. 57–71.

[6] J. M. Franco, F. Correia, R. Barbosa, M. Zenha-Rela, B. Schmerl, and D. Garlan, "Improving self-adaptation planning through software architecture-based stochastic modeling," *Journal of Systems and Software*, vol. 115, pp. 42–60, 2016.

[7] C. Ghezzi and A. Molzam Sharifloo, "Model-based verification of quantitative non-functional properties for software product lines," *Information and Software Technology*, vol. 55, no. 3, pp. 508–524, 2013.

[8] K. Cooper, L. Dai, and Y. Deng, "Performance modeling and analysis of software architectures: An aspect-oriented UML based approach," *Science of Computer Programming*, vol. 57, no. 1, pp. 89–108, 2005.

[9] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, no. 1, pp. 3–22, 2009.

[10] N. Kumar, B. R. Childers, and M. L. Soffa, "Low overhead program monitoring and profiling," *SIGSOFT*, vol. 31, no. 1, pp. 28–34, 2005.

[11] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *TOPLAS*, vol. 16, no. 4, pp. 1319–1360, 1994.

[12] P. Arafa, G. M. Tchamgoue, H. Kashif, and S. Fischmeister, "QDIME: QoS-aware dynamic binary instrumentation," in *MASCOTS*, 2017, pp. 132–142.

[13] A. Van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *ICPE*, 2012, pp. 247–248.

[14] S. Schubert, D. Kostic, W. Zwaenepoel, and K. G. Shin, "Profiling software for energy consumption," in *GreenCom*, 2012, pp. 515–522.

[15] A. Filieri, C. Pasareanu, and W. Visser, "Reliability analysis in Symbolic Pathfinder," in *ICSE*, 2013, pp. 622–631.

[16] A. Filieri, C. S. Pasareanu, and G. Yang, "Quantification of software changes through probabilistic symbolic execution," in *ASE*, 2015, pp. 703–708.

[17] J.-P. Katoen, "The probabilistic model checking landscape," in *Symposium on Logic in Computer Science*, 2016, pp. 31–45.

[18] S. Andova, H. Hermanns, and J.-P. Katoen, "Discrete-time rewards model-checked," in *International Conference on Formal Modeling and Analysis of Timed Systems*, 2004, pp. 88–104.

[19] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal Aspects of Computing*, vol. 6, pp. 512–535, 1994.

[20] M. Kwiatkowska, G. Norman, and D. Parker, "Prism 4.0: Verification of probabilistic real-time systems," in *CAV*, 2011, pp. 585–591.

[21] J.-P. Katoen, M. Khattri, and I. S. Zapreevt, "A markov reward model checker," in *Quantitative Evaluation of Systems*, 2005, pp. 243–244.

[22] C. Dehnert, S. Junges, J. Katoen, and M. Volk, "A storm is coming: A modern probabilistic model checker," in *CAV*, 2017, pp. 592–600.

[23] M. Borges, Q.-S. Phan, A. Filieri, and C. S. Păsăreanu, "Model-counting approaches for nonlinear numerical constraints," in *NASA Formal Methods*, 2017, pp. 131–138.

[24] B. Bartels, U. Ermel, P. Sandborn, and M. G. Pecht, *Strategies to the prediction, mitigation and management of product obsolescence*. John Wiley & Sons, 2012, vol. 87.

[25] B. E. Cossette and R. J. Walker, "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries," in *FSE*, 2012, pp. 1–11.

[26] M. B. Dwyer, A. Filieri, J. Geldenhuys *et al.*, "Probabilistic program analysis," in *GTTSE*, 2015, pp. 1–25.

[27] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani, "Probabilistic programming," in *FOSE*, 2014, pp. 167–181.

[28] K. H. Bennett and V. Rajlich, "Software maintenance and evolution: a roadmap," in *ICSE*, 2000, pp. 73–87.

[29] P. N. Misra, "Software reliability analysis," *IBM Systems Journal*, vol. 22, no. 3, pp. 262–270, 1983.

[30] J. Geldenhuys, M. B. Dwyer, and W. Visser, "Probabilistic symbolic execution," in *ISSTA*, 2012, pp. 166–176.

[31] W. Visser and C. Pasareanu, "Probabilistic programming for Java using symbolic execution and model counting," in *SAICSIT*, 2017, pp. 1–10.

[32] S. S. Gokhale and Michael Rung-Tsong Lyu, "A simulation approach to structure-based software reliability analysis," *IEEE Transactions on Software Engineering*, vol. 31, no. 8, pp. 643–656, 2005.

[33] C. Hsu and C. Huang, "An adaptive reliability analysis using path testing for complex component-based software systems," *IEEE Transactions on Reliability*, vol. 60, no. 1, pp. 158–170, 2011.

[34] J. Zhang, Y. Lu, K. Shi, and C. Xu, "Empirical research on the application of a structure-based software reliability model," *IEEE/CAA Journal of Automatica Sinica*, pp. 1–10, 2020.

[35] C. Dubslaff, A. Morozov, C. Baier, and K. Janschek, "Reduction methods on probabilistic control-flow programs for reliability analysis," *CoRR*, 2020.

[36] C. Trubiani, A. Ghabi, and A. Egyed, "Exploiting traceability uncertainty between software architectural models and extra-functional results," *Journal of Systems and Software*, vol. 125, pp. 15–34, 2017.

[37] R. Calinescu, V. Cortellessa, I. Stefanakos, and C. Trubiani, "Analysis and refactoring of software systems using performance antipattern profiles," in *FASE*, 2020, pp. 357–377.

[38] R. Calinescu, M. Autili, J. Cámara *et al.*, "Synthesis and verification of self-aware computing systems," in *Self-Aware Computing Systems*. Springer, 2017, pp. 337–373.

[39] J. Camara, D. Garlan, and B. Schmerl, "Synthesis and quantitative verification of tradeoff spaces for families of software systems," in *Software Architecture*, 2017, pp. 3–21.

[40] R. Calinescu, K. Johnson, and C. Paterson, "FACT: A probabilistic model checker for formal verification with confidence intervals," in *TACAS*, 2016, pp. 540–546.