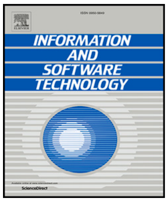
Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Probabilistic program performance analysis with confidence intervals

Ioannis Stefanakos [*], Radu Calinescu, Simos Gerasimou

Department of Computer Science, University of York, York, YO10 5GH, UK

ARTICLE INFO

Keywords:
Program quality analysis
Software performance
Discrete-time Markov chains
Probabilistic model checking
Formal verification with confidence intervals

ABSTRACT

Context: More often than not, the algorithms implemented by software systems continue to operate correctly when executed on different platforms or with different inputs, and can be easily replaced with functionally equivalent ones. However, such changes can have a significant and difficult to predict impact on the software performance, resource use, and other key quality properties.
Objective: The paper introduces a method for the formal analysis of timing, resource use, cost and other quality aspects of computer programs, and a tool that automates the application of the method to Java code.
Method: A tool-supported probabilistic program performance analysis (PROPER) method was developed, and was evaluated using Java code from the Apache Commons Math library, the Android messaging app Telegram, and open-source implementations of the knapsack, binary search, and minimum path sum algorithms. PROPER synthesises a parametric Markov-chain model of the analysed code, uses information from program logs to calculate confidence intervals for the parameters of this model, and employs formal verification with confidence intervals to obtain confidence intervals for the performance properties of interest. A PROPER variant that operates with point estimates instead of confidence intervals can be used when large program logs are available.
Results: The PROPER point estimates for the analysed performance properties were accurate within 7.9% and 1.75% of the ground truth when using program logs with 10^3 and 10^4 entries, respectively. All PROPER confidence intervals for these properties contained the true property value, and became narrower when larger logs were used in the analysis. The analyses were completed in under 15 ms for point estimates, and in between 6.7 s and 7.8 s for confidence intervals on a regular laptop computer.
Conclusion: PROPER can synthesise and reuse a parametric Markov model to accurately predict how software performance would change if the code ran on a different hardware platform, used a new function library, or had a different usage profile—supporting practitioners who are interested in these analyses.

1. Introduction

Software applications can often be executed unmodified on hardware platforms as diverse as servers, laptops and smartphones, and/or with different *usage profiles* (i.e., probability distributions of the program inputs). Even changes such as the replacement of a function or module with a functionally equivalent one that is, for instance, faster or more reliable, can typically be implemented without major efforts without impacting the functionality of the application. While this unique characteristic of software is of major benefit, it makes the analysis of the performance, cost and other quality properties of software systems very difficult [1,2]. Indeed, modifications in the platform, usage profile and individual components of software systems may not impact their functionality, but can significantly affect their execution time, resource use and cost. Given the importance of these quality properties [3–5], the modelling and analysis of software performance have been studied extensively [6–8].

Existing software performance engineering methods are primarily targeting the analysis of software performance at *architectural-level*, with the quality properties of service-based systems, software product lines, and other component-based software systems evaluated through formally modelling the dependencies and interactions between their components [9–13]. In contrast, the equally important evaluation of software performance at code-level is typically carried out using empirical methods involving program instrumentation, monitoring and profiling [14–18]. These methods tend to yield accurate results, but have the major limitation that the code needs to be actually deployed and executed on every platform and for each usage profile under evaluation, and/or after each change to the code.

To address this limitation, we developed a new method for probabilistic program performance analysis (PROPER). PROPER operates in three stages. In the first stage, it models the code under analysis (CUA) as a *parametric discrete-time Markov chain* (pDTMC).

[*] Corresponding author.
E-mail address: ioannis.stefanakos@york.ac.uk (I. Stefanakos).

https://doi.org/10.1016/j.infsof.2022.107143
Received 2 March 2022; Received in revised form 28 September 2022; Accepted 20 December 2022
Available online 26 December 2022

This PROPER pDTMC is generated automatically from the CUA, and is parameterised by the probabilities with which the CUA conditional statements and loops are performed during a CUA execution. In the second stage, PROPER computes either point estimates or confidence intervals for the unknown pDTMC parameters by exploiting information from program logs. Finally, in its last stage, PROPER analyses the CUA performance properties of interest, yielding point estimates or confidence intervals for their values. The analysis based on point estimates is only accurate when large program logs are available to ensure that its results are not affected by epistemic uncertainty.

PROPER represents the first method that uses formal verification with confidence-interval to automatically evaluate software performance properties at code-level. An approach that uses probabilistic modelling for code-level analysis was proposed in [19,20]. 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. More generally, PROPER differs from existing approaches to program performance analysis through its use of confidence intervals. This unique characteristic enables PROPER to reflect the epistemic uncertainty due to the necessarily finite size of the program logs that the CUA usage profile is derived from, with narrower confidence intervals produced from larger logs than from smaller ones. In contrast, any approaches that use point estimates disregard this uncertainty, yielding potentially inaccurate results without any warning when only small program logs are available.

The main contributions of our paper include:

1. The PROPER method for the formal what-if analysis of program performance (i) before deploying the CUA on a new platform; (ii) for an expected change in the CUA usage profile; and/or (iii) to assess the performance impact of using a new implementation of a function.
2. A tool that automates the application of our program performance analysis method to Java code.
3. An extension of the confidence-interval probabilistic model checker FACT [21] that enables its use to analyse PROPER-generated pDTMCs.
4. An extensive evaluation of the PROPER method and tool for Java code from the Apache Commons Math library, the Android messaging app Telegram, and open-source implementations of the knapsack, binary search, and minimum path sum algorithms.

A preliminary version of PROPER that does not include the use of confidence intervals was introduced in [22]. This paper updates and extends the theoretical foundation from [22] with the results required for the derivation of confidence intervals for the PROPER pDTMC parameters and the CUA performance properties. This extension is presented in Section 4 and, as described in Section 5, is implemented by our PROPER tool and modified version of the FACT model checker. Furthermore, we considerably extended and improved the validation of PROPER by evaluating it for additional Java methods that include a more complex running example (Section 2), and by comparing it to a broader range of related work in Section 7.

The remainder of the paper is structured as follows. In Section 2 we provide a running example used to demonstrate the application of PROPER later in the paper. Section 3 provides background information on probabilistic model checking. This is then used to describe the PROPER theoretical foundation in Section 4, and its implementation tool in Section 5. The experiments carried out for the PROPER evaluation are described in Section 6. Finally, we compare PROPER to related research in Section 7, and we present our conclusions and propose directions for further research in Section 8.

2. Running example

We illustrate the steps and application of our PROPER method using a scenario where a software engineer is interested in evaluating performance aspects of the minimum path sum algorithm (`minPathSum`) implemented in Java. The dynamic-programming implementation of the algorithm is adapted from a public repository on GitHub[1] that contains implementations of popular algorithms. Given an $m \times n$ grid with non-negative numbers, where m and n refer to length and width of the input array, respectively, this method finds the path from top left to bottom right such that the sum of numbers along its path is minimised. As shown in Fig. 1, the method receives as input a two-dimensional array (grid), and performs the calculations based on the following rules: (a) moving from the top left corner to the down right corner, and (b) moving one step down or right. We suppose that the method `minPathSum` is used by an application for which a detailed log reflecting the method's usage profile (i.e., the number of executions for each conditional statement and/or loop that appears in the code) 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 0.25 is incurred each time the *Math.min*(…) function is invoked in line 22;
- the expected execution time of the method, under the assumption that the *while* loops from lines 9 and 14 require 0.01 ms on average, and the execution of the *while* loop from line 21 requires 0.03 ms on average.

The annotations '*@cost=0.25*' appended as comment to line 22, '*@time=0.01*' appended as comment to lines 9 and 14, and '*@time=0.03*' appended as comment to line 21, are used to specify the two performance properties whose evaluation is of interest.

3. Theoretical basis

3.1. Probabilistic model checking

Probabilistic model checking (PMC) [23] is a mathematically based technique used to verify the correctness, reliability and performance of systems with stochastic behaviour, where this behaviour is formalised using Markov models. The technique operates with multiple types of Markov models, each of which is suitable for the analysis of different classes of system properties. While probabilistic model checking can be used with multiple types of Markov models, such as discrete-time Markov chains, continuous-time Markov chains and Markov decision processes [23], in this section we focus on (parametric) discrete-time Markov chains, which are the type of models generated and used by our approach.

Definition 1. A discrete-time Markov chain (DTMC) over a set of atomic propositions AP is a tuple $D = (S, s_0, \mathbf{P}, L)$ where S is a finite set of states, $s_0 \in S$ is the initial state, $\mathbf{P} : S \times S \to [0, 1]$ is a transition probability matrix such that, for all $s \in S$, $\sum_{s' \in S} \mathbf{P}(s, s') = 1$, 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 support the analysis of a wider range of properties types, DTMCs are augmented with *cost/reward structures* [24] that associate non-negative values with their states and transitions. The difference between cost and reward is purely semantic. There is no mathematical distinction, just a commonly adopted notion that "cost" should be minimised and "reward" should be maximised.

[1] https://github.com/TheAlgorithms/Java/

```java
1  public static int minPathSum(int[][] grid) {
2    int m = grid.length, n = grid[0].length;
3    if (n == 0) {
4       return 0;
5    }
6    int[][] dp = new int[m][n];
7    dp[0][0] = grid[0][0];
8    int i = 0;
9    while (i < n - 1) {
10      dp[0][i+1] = dp[0][i] + grid[0][i+1]; //@time=0.01
11      i++;
12   }
13   i = 0;
14   while (i < m - 1) {
15      dp[i+1][0] = dp[i][0] + grid[i+1][0]; //@time=0.01
16      i++;
17   }
18   i=1;
19   while (i < m) {
20      int j=1;
21      while (j < n) {
22         dp[i][j] = Math.min(dp[i-1][j], dp[i][j-1]) +
            ↪  grid[i][j]; //@cost=0.25,@time=0.03
23         j++;
24      }
25      i++;
26   }
27   return dp[m - 1][n - 1];
28 }
```

Fig. 1. Java `minPathSum` method calculating the minimum path sum of an $m \times n$ matrix. PROPER annotations, described later in the paper, indicate that the statements in lines 10 and 15 have a mean execution time of 0.01 s, and the statement in line 22 has a cost of 0.25 and a mean execution time of 0.03 s on the platform that will be used to execute the method.

Definition 2. A cost/reward structure over a DTMC $D = (S, s_0, \mathbf{P}, L)$ 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.

Finally, the DTMCs used by PROPER are *parametric*, i.e., they contain transition probabilities that are unknown when these models are derived.

Definition 3. A *parametric discrete-time Markov chain* (pDTMC) is a discrete-time Markov chain comprising one or several unknown state transition probabilities and/or costs/rewards that are specified as rational functions (i.e., as fractions whose numerators and denominators are polynomial functions) over a set of continuous variables [25].

The properties of DTMCs and pDTMCs analysed through PMC are formally expressed in probabilistic computation tree logic (PCTL) [26], a branching-time temporal logic with the following syntax.

Definition 4. 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 are 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* X Φ holds for a path if Φ is satisfied in the next state of 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 [24,26].

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.

3.2. Formal verification with confidence intervals

Formal verification with confidence intervals [27] is a formal technique that computes confidence intervals for the quality properties of systems with stochastic behaviour. Given a pDTMC that models the behaviour of such a system, a PCTL formula $P_{=?}[\,\cdot\,]$ or $R_{=?}[\,\cdot\,]$ associated with a quality property of this system, and a confidence level $\alpha \in (0, 1)$, the technique computes an α confidence interval for the property. For example, the technique can be used to establish that the 95%

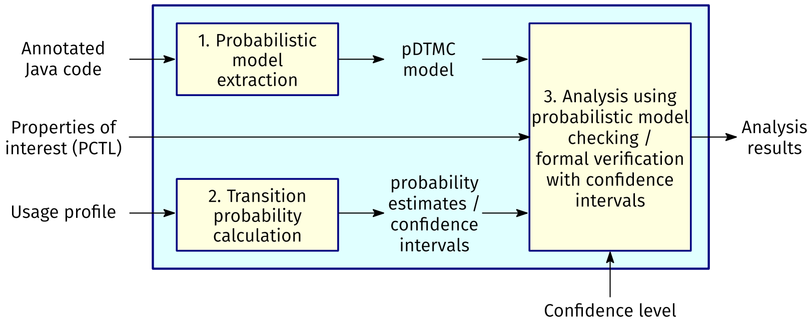


Fig. 2. PROPER probabilistic program performance analysis.

confidence interval for the expected execution time of a Java method is $[150\text{ ms}, 190\text{ ms}]$, or that the 99% confidence interval for the daily energy consumption of a sensor is $[85\text{ J}, 103\text{ J}]$.

This confidence interval computation is performed in three steps. First, a confidence interval is calculated for each pDTMC parameter by using observations of the system related to that parameter. Next, *parametric model checking* is used to obtain a closed-form expression (i.e., an expression containing only the basic arithmetic operators and exponent) for the quality property. Parametric model checking is supported by model checkers including PARAM [28], PRISM [29], Storm [30] and ePMC/fPMC [31,32]. Finally, the expression obtained in this way and the parameter confidence intervals are used to establish the confidence interval for the quality property under analysis. The width of this confidence interval depends on the number of available observations and the confidence level α. In particular, wide confidence intervals are obtained when only few observations are available, and narrow intervals can be computed given large number of observations. For a complete description of the technique and of the FACT model checker that implements it, see [21,27], respectively.

4. PROPER performance analysis method

4.1. Method overview

Fig. 2 shows the stages of our PROPER method for the analysis of performance properties of the program of interest. In a first stage, a reward-augmented parametric DTMC model is automatically extracted from the analysed Java code. This pDTMC is parameterised by the probabilities of executing the conditional statements and loops from the analysed code, which are initially unknown. To enable the generation of this pDTMC, the CUA is annotated by appending comments of the form

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

to its Java statements. In this PROPER annotation, *property* is a one-word label that corresponds to a code property under analysis (e.g., '*cost*' or '*time*', as shown in Fig. 1), and *value* represents the (actual or estimated) mean value of that property for the statement that the annotation is associated with. As an example, the annotation '*// @time=0.01*' from line 10 of the Java method from Fig. 1 indicates that the mean execution time of the statement from that line is 0.01 (time units). The same property label needs to be added to all statements for which the property has a non-negligible value, e.g., to indicate that a non-negligible *cost* or execution *time* is associated with multiple statements. Instances of such annotations can be observed in Fig. 1 for several statements from the Java method we use as a running example.

In the second stage, PROPER calculates the transition probabilities associated with the pDTMC 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. Two techniques for this calculation are available. First, the usage profile can be used to obtain point estimates for the unknown transition probabilities. However, using such probability estimates can yield inaccurate overall results unless the usage profile is based on very large program logs. Therefore, PROPER also supports the calculation of confidence intervals for the unknown transition probabilities. This technique has the advantage that it takes into account the log size by producing wider confidence intervals when only small program logs are available, and narrow confidence intervals when the calculation is based on large logs.

Finally, in the third PROPER stage, the PCTL-formulated performance properties of interest are analysed, again using one of two techniques: (i) standard probabilistic model checking is used when point estimates were calculated for the unknown transition probabilities of the pDTMC model of the CUA; and (ii) formal verification with confidence intervals (at a user-specified confidence level) is used when confidence intervals were computed for these transition probabilities. The result of the analysis is also a point estimate in the first case, and a confidence interval in the second case.

The three stages of our PROPER method and further types of analyses enabled by the pDTMC 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.

Before describing the stages of our method in detail, we summarise the underlying assumptions for its application below:

- The CUA is annotated as shown in (1) in order to convey to PROPER relevant information about the nonfunctional properties of interest.
- Log files (obtained as described further in Section 4.3.1) are available to enable inferring usage profiles, computing and providing point estimates or confidence intervals for the unknown transition probabilities of the produced pDTMC model.
- The logs contain representative samples of the inputs that the analysed code will encounter when deployed.[2]

4.2. Probabilistic model synthesis

PROPER enables the synthesis of pDTMC models through the recursive application of the code-to-model transformation rules shown in Fig. 3. Our method supports the four types of statements below:

[2] As emphasised in software performance evaluation surveys [1,6,33], it is essential (and non-trivial) to ensure that the usage profile obtained from such logs is representative; the use of code instrumentation and other techniques to obtain representative logs is discussed in these surveys.

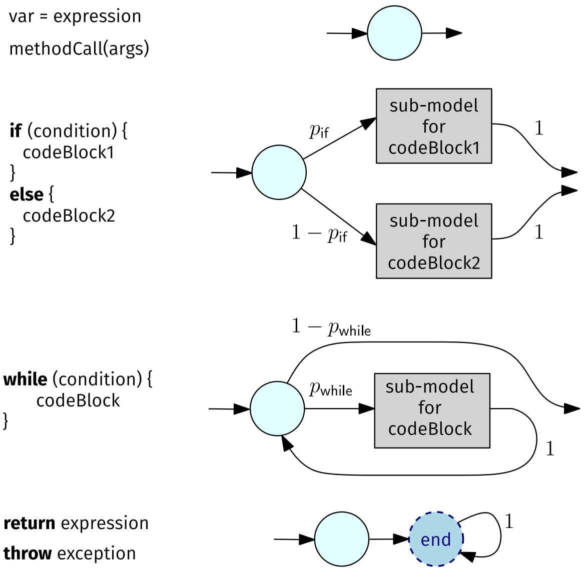


Fig. 3. PROPER code-to-model transformation rules.

1. Assignment statements and method calls (with no side effects) are modelled using a single pDTMC state. This state has one incoming transition (from the pDTMC fragment modelling the previous statement in the code) and one outgoing transition (to the pDTMC fragment modelling the next statement).
2. Conditional statements are modelled using a state with two outgoing transitions, one to the pDTMC fragment modelling the statements from the 'if' branch, and one to the pDTMC fragment modelling the 'else' branch. The latter pDTMC 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 pDTMC 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 pDTMC 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. Although we describe here the transformation for 'while' loops, PROPER also supports other types of loops (e.g., 'for' loops) since they 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 pDTMC. This state, shown in dashed line in Fig. 3, has a self-loop transition with 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.

Example 1. Fig. 4 depicts the pDTMC obtained by applying the above rules to the Java code from our running example (see Fig. 1). The statement modelled by each pDTMC state is mentioned under the state, and the states are numbered 0 to 21. The grey-shaded areas contain states, as shown in the figure, that correspond to blocks of code contained within conditional statements and loops.

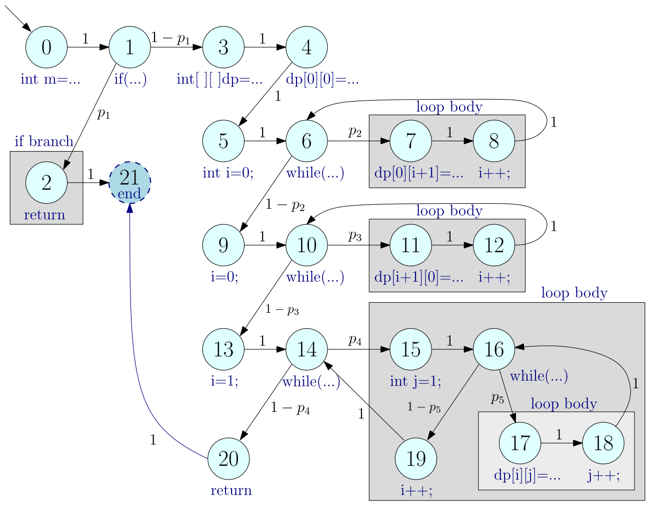


Fig. 4. DTMC model for the `minPathSum` Java method.

To allow the use of model checkers to analyse its synthesised pDTMCs, PROPER uses the rules from Fig. 3 to generate these pDTMCs in the high-level modelling language of the PRISM model checker [29], 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, $1 \le i \le n$, 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 pDTMC produced by PROPER comprises a single PRISM module, and is generated by the function BuildModel shown in 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 $p_{condCtr}$. The second transition, corresponding to the 'else' branch (if this branch exists) or to the statement after the conditional (otherwise), has probability $1-p_{condCtr}$, 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).

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
```

Lines 19–22 produce the *model* commands when *stmt* is a loop statement. The process followed is similar to that employed 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). The *loopStartingState* command in line 19 records the *stateCtr* value for the first state of the loop statement execution.

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.

The following result establishes the complexity of the model synthesis algorithm.

Theorem 1. *The function* BuildModel *from Algorithm 1 requires* $O(n_s n_a)$ *time, where* n_s *and* n_a *represent the number of statements from the* method *supplied as its only argument, and the maximum number of annotations* (1) *for the same statement, respectively.*

Proof. Building the abstract syntax tree in line 22 of BuildModel involves the examination of each statement, to establish its type (as required by the switch from line 5 of the function Synthesis) and to record each of its up to n_a annotations (1). As such, each statement can be parsed in $O(n_a)$ time, and parsing all n_s statements from *method* requires $O(n_s n_a)$ time.

The invocation of the function Synthesis in line 34 of BuildModel also takes $O(n_s n_a)$ time, because: (i) the depth-first search performed by this function (through its for each loop starting in line 4 and the recursive invocations of Synthesis from lines 10, 15 and 21) visits each of the n_s nodes of the *ast* abstract syntax tree once; (ii) each operation that does not invoke Synthesis within the switch statement from lines 5–25 only requires constant time; and (iii) the while loop from lines 26–28 (executed once for each statement) requires $O(n_a)$ time to handle the up to n_a annotations associated to the statement *stmt* under examination.

The creation of the model variables in line 35 of BuildModel takes $O(n_s)$ time since the number of such variables is given the final value of *condCtr* counter, which is incremented at most once (in line 9 or in line 20 of Synthesis) for each of the n_s statements from *method*. Finally, the operation from line 36 of BuildModel requires constant time, the

```
dtmc

const double p1;
const double p2;
const double p3;
const double p4;
const double p5;

const int end_state = 21;

module minPathSum
  s : [0..end_state] init 0;

  [] s=0  -> 1:(s'=1);                    //line:2
  [] s=1  -> p1:(s'=2) + (1-p1):(s'=3);   //line:3
  [] s=2  -> 1:(s'=end_state);            //line:4
  [] s=3  -> 1:(s'=4);                    //line:6
  [] s=4  -> 1:(s'=5);                    //line:7
  [] s=5  -> 1:(s'=6);                    //line:8
  [] s=6  -> p2:(s'=7) + (1-p2):(s'=9);   //line:9
  [] s=7  -> 1:(s'=8);                    //line:10
  [] s=8  -> 1:(s'=6);                    //line:11
  [] s=9  -> 1:(s'=10);                   //line:13
  [] s=10 -> p3:(s'=11) + (1-p3):(s'=13); //line:14
  [] s=11 -> 1:(s'=12);                   //line:15
  [] s=12 -> 1:(s'=10);                   //line:16
  [] s=13 -> 1:(s'=14);                   //line:18
  [] s=14 -> p4:(s'=15) + (1-p4):(s'=20); //line:19
  [] s=15 -> 1:(s'=16);                   //line:20
  [] s=16 -> p5:(s'=17) + (1-p5):(s'=19); //line:21
  [] s=17 -> 1:(s'=18);                   //line:22
  [] s=18 -> 1:(s'=16);                   //line:23
  [] s=19 -> 1:(s'=14);                   //line:25
  [] s=20 -> 1:(s'=end_state);            //line:27
  [] s=21 -> 1:(s'=21);
endmodule

rewards "time"
  s=7  : 0.01;
  s=11 : 0.01;
  s=18 : 0.03;
endrewards

rewards "cost"
  s=18 : 0.25;
endrewards
```

Fig. 5. PRISM model synthesised for the minPathSum Java method.

addition of the model's reward structures (which comprise up to n_a reward values for each of the n_s statements from *method*) in line 37 is completed in $O(n_s n_a)$ time, and returning the three parts of the model in line 38 requires constant time (or $O(n_s n_a)$ if a fully fledged string concatenation is performed).

As each operation from lines 33–38 of BuildModel can be completed in $O(n_s n_a)$, $O(n_s)$ or constant time, we conclude that the entire method requires $O(n_s n_a)$ time. □

Example 2. Fig. 5 shows the PRISM-encoded pDTMC model generated by Algorithm 1 for the minPathSum Java method from our running example. The model has two reward structures, corresponding to the time and cost annotations from the Java code in Fig. 1. The variables p1, … , p5 correspond to the probabilities of executing the 'if' statement (line 3) and 'while' loops (lines 9, 14, 19, and 21). Their values depend on the usage profile of the code, and are determined as described in the next section.

4.3. 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.

4.3.1. Point estimate calculation

To calculate point estimates for the pDTMC transition probabilities, PROPER requires a usage profile that provides, for each of N_0 executions of the analysed code, the observed number of executions of every CUA conditional statement 'if' branch and loop body. 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* [34], which can calculate expected values for these counts from empirical probability distributions of the program inputs, where these distributions are taken from program logs. Model counting has been applied to programs with both linear and non-linear numerical constraints with tools such as Latte [35] and SharpSAT [36], respectively. Application areas include but are not limited to probabilistic software analysis [37,38] and software reliability [39].
- By applying Monte Carlo simulation 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. Regarding the validity of the simplification process, only the statements with no impact on the required execution counts are removed, i.e., statements that do not affect the code's operational profile. The code's simplification is due to the constraints discussed in Section 4.1.

Given 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 $N_1, N_2, \ldots, N_n$,[3] then the transition probability associated with the ith conditional statement/loop is calculated as:

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

where $1 \leq i \leq n$. As we show later in Theorem 2, this result is inspired by the way of calculating the sum of an infinite geometric series, cf. (4). For conditional statements and loops that are not nested within other conditional statements/loops (such as those from our running example, except the nested loop from lines 21–24 in Fig. 1), 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 minPathSum from our running example indicates that, across $N_0 = 10,000$ invocations of the method, the mean number of executions of the if branch of the conditional statement starting in line 3 from Fig. 1 is $N_1 = 912$, and the mean numbers of executions of the bodies of the while loops from lines 9–12, 14–17, 19–26, and 21–24 are $N_2 = 45,000$, $N_3 = 40,000$, $N_4 = 45,000$, and $N_5 = 202,400$ times, respectively. Accordingly, the values of the unspecified transition probabilities for the DTMC model from Fig. 5 are given by $p1 = \frac{N_1}{N_0} = \frac{912}{10,000} = 0.0912$, $p2 = \frac{N_2}{N_0+N_2} = \frac{45,000}{10,000+45,000} = 0.8181$, $p3 = \frac{N_3}{N_0+N3} = \frac{40000}{10,000+40,000} = 0.8$, $p4 = \frac{N_4}{N_0+N_4} = \frac{45,000}{10,000+45,000} = 0.8181$, and $p5 = \frac{N_5}{N_4+N_5} = \frac{202,400}{45,000+202,400} = 0.8181$.

[3] 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.

The following result shows that the PROPER probabilistic model synthesised in Section 4.2 and instantiated with the probabilities calculated above can be used to determine the performance properties for the code under analysis.

Theorem 2. *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 nth 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 Fig. 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 ith model construct, $1 \leq i \leq n$. For a model construct associated with a loop, this factor is given by

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

due to the repeated execution of ith 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

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

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). □

4.3.2. Confidence interval calculation

To calculate confidence intervals for the pDTMC transition probabilities, PROPER requires a usage profile that provides, for each of N_0 executions of the CUA, the number of executions of the 'if' branch of every conditional statement and of the body of every loop. This usage profile needs to be obtained from appropriately instrumented logs of the analysed program, or though *model counting* [34]. Simulation with the program inputs drawn randomly from logs that reflect the empirical probability distributions of these inputs cannot be used, as this technique is able to generate any number of observations, including duplicates that are disallowed in the calculation of confidence intervals. Assume that the number of such executions for a generic conditional statement or loop body x are x_1, x_2, …, x_{N_0}. Using the law of large numbers and the central limit theorem [40], PROPER calculates an α_x confidence interval $[\underline{m_x}, \overline{m_x}]$ for the *true* mean number of executions μ_x of that conditional statement or loop body, with the interval bounds given by:

$$\underline{m_x} = m_x - z(\alpha_x)\frac{s}{\sqrt{N_0}} \tag{6}$$

and

$$\overline{m_x} = m_x + z(\alpha_x)\frac{s}{\sqrt{N_0}} \tag{7}$$

where $m_x = \frac{\sum_{i=1}^{N_0} x_i}{N_0}$ is the sample mean, $s = \sqrt{\frac{\sum_{i=1}^{N_0}(x_i - m_x)^2}{N_0 - 1}}$ is the sample standard deviation, and $z(\alpha_x)$ is the value from the standard normal distribution $\mathcal{N}(0,1)$ corresponding to the selected confidence level α_x. For a generic CUA comprising m conditional statements and loops, PROPER uses this technique to calculate an $\alpha_x = \alpha^{\frac{1}{m}}$ confidence interval for the mean number of executions μ_x of each conditional statement or loop body x, where $\alpha \in (0,1)$ is the user-specified confidence level for the analysis of the properties of interest (see Fig. 2).

Given a generic set of $n \geq 1$ nested conditional statements and/or loops from the analysed code, the unknown transition probabilities p_1, $p_2, \ldots, p_n$ that Algorithm 1 associates with these conditional statements and loops are redefined as

$$p_i = \begin{cases} \frac{\mu_i}{\mu_{i-1}}, & \text{if statement } i \text{ is a conditional} \\ \frac{\mu_i}{\mu_{i-1}+\mu_i}, & \text{otherwise (if statement } i \text{ is a loop)} \end{cases} \tag{8}$$

where $\mu_0 = 1$. The following result shows that this definition is equivalent to (2).

Theorem 3. *The probability definitions* (2) and (8) *are equivalent when $N_0 \to \infty$.*

Proof. Consider a generic probability p_i from (2), $1 \leq i \leq n$. We consider first the case when statement i is a conditional statement. In this case, we have:

$$\lim_{N_0\to\infty} \frac{N_i}{N_{i-1}} = \lim_{N_0\to\infty} \frac{N_i/N_0}{N_{i-1}/N_0} = \frac{\mu_i}{\mu_{i-1}},$$

which shows that the theorem holds in this case. Otherwise, i.e., if statement i is a loop, we have:

$$\lim_{N_0\to\infty} \frac{N_i}{N_{i-1}+N_i} = \lim_{N_0\to\infty} \frac{N_i/N_0}{N_{i-1}/N_0 + N_i/N_0} = \frac{\mu_i}{\mu_{i-1}+\mu_i},$$

which shows that the theorem also holds in this case, completing the proof. □

This theorem allows us to use definition (8) instead of (2) for the transition probabilities from the pDTMC generated by PROPER, and therefore to obtain simultaneous confidence intervals $[\underline{p_i}, \overline{p_i}]$ at confidence level α for the pDTMC probabilities p_i as follows:

$$[\underline{p_i}, \overline{p_i}] = \begin{cases} \left[\frac{\underline{m_i}}{\overline{m_{i-1}}}, \frac{\overline{m_i}}{\underline{m_{i-1}}} \right], & \text{if statement } i \text{ is a conditional} \\ \left[\frac{\underline{m_i}}{\overline{m_{i-1}}+\underline{m_i}}, \frac{\overline{m_i}}{\underline{m_{i-1}}+\overline{m_i}} \right], & \text{otherwise} \end{cases} \tag{9}$$

for $1 \leq i \leq n$, where $[\underline{m_i}, \overline{m_i}]$ represents the confidence interval for the true mean μ_i, and $\underline{\mu_0} = \overline{\mu_0} = 1$.

4.4. Analysis

4.4.1. Probabilistic model checking with point estimates

Given the synthesised pDTMC model (Section 4.2) and the calculated transition probabilities (Section 4.3.1), PROPER uses a probabilistic model checker such as PRISM [29] or Storm [30] to evaluate the PCTL-encoded performance properties of interest.

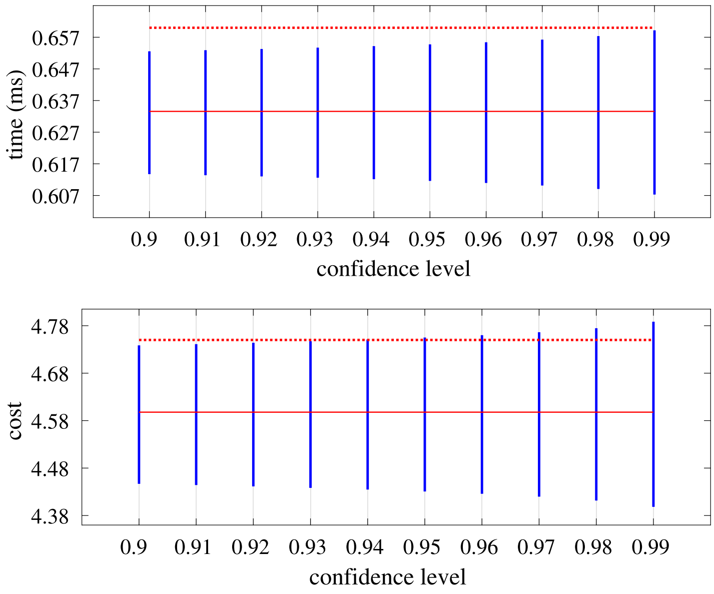


Fig. 6. Confidence intervals (shown as vertical segments) and point estimates (shown as solid horizontal lines) for the properties of interest from our running example from Section 2. The dotted horizontal line shows requirement bounds for the two properties. The vertical blue lines exceeding the bound indicate requirement violation. (For interpretation of thereferences to colour in this figure legend, the reader is referred to the web version of this article.)

Example 4. Consider again our running example (Section 2). Determining the values of the 'cost' and 'time' properties specified using PROPER annotations in Fig. 1 involves the probabilistic model checking of the reward PCTL properties $R\{``cost"\}_{=?}[Fs = end\_state]$ and $R\{``time"\}_{=?}[Fs = end\_state]$ over the pDTMC model from Fig. 5. To carry out these analyses for the usage profile from Example 3, the unspecified DTMC probabilities need to be initialised such that p1 = 0.0912, p2 = 0.8181, p3 = 0.8, p4 = 0.8181, and p5 = 0.8181. Using PRISM, the results of these analyses are time = 0.628 and cost = 4.59.

4.4.2. Probabilistic model checking with confidence intervals

Although PROPER can support the quantitative verification of properties for the CUA using point estimates (Section 4.4.1), the outcome of this formal analysis depends on the accuracy of the (unknown) transition probabilities. Establishing these probabilities as point estimates can be done by involving domain experts or by applying techniques for model fitting to log data or run-time observations [41]. Unavoidably, point estimates are affected by unquantified estimation errors. Given the nonlinear nature of Markov models, the propagation of these errors in formal analysis leads to imprecise results with unknown, but most likely, significant consequences that could invalidate the analysis conclusions or any decisions made.

To alleviate this issue, PROPER also supports the derivation of confidence intervals for the CUA performance properties of interest. To this end, PROPER uses the calculated *simultaneous* confidence intervals for the transition probabilities of the pDTMC at a user-specified confidence level α (Section 4.3.2) and employs an adapted variant of the formal verification with confidence intervals tool FACT [21] to obtain confidence intervals for the performance properties of interest. More specifically, our adapted FACT tool (Section 5) initially uses the synthesised pDTMC and the performance property to obtain an algebraic expression representing this property. Then, an optimisation problem is formed using the estimated confidence intervals for the transition probabilities and the derived algebraic expression, which is solved using a Matlab optimisation package providing the confidence interval for the target performance property.

Example 5. By applying probabilistic model checking with confidence intervals to the pDTMC from our running example, we can obtain confidence intervals for the time and cost properties of interest. Fig. 6 illustrates the output of the formal analysis. The blue vertical lines correspond to the range of confidence intervals for a given confidence level $\alpha \in \{0.9, 0.91, 0.92, \ldots, 0.99\}$, and the solid red horizontal line depicts the point estimate obtained during the analysis in Example 4 (and which falls within the obtained ranges). Assuming that the red dotted horizontal lines specify the time and cost requirement bounds, i.e., 0.66 ms and 4.75, respectively, the use of point estimates would indicate that both requirements are satisfied. In contrast, using the confidence interval-based variant of PROPER with any confidence level $a > 0.95$ would indicate that the cost exceeds the requirement bound, thus enabling software engineers to perform modifications to the CUA and reduce the cost below the requirement bound.

4.5. 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 6. The impact of replacing the *Math.min*(…) function call from line 22 of the `minPathSum` Java method from Fig. 1 with a call to the improved function *FastMath.min*(…) can be analysed using the same pDTMC model as in Example 4, after only updating the reward values from lines 48 and 52 of the model (see Fig. 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 7. Suppose that the application using the method `minPathSum` from our running example needs to be deployed on a smart phone on which each invocation of *Math.min*(…) consumes 60 J. The expected energy consumption of `minPathSum` can be predicted before actually

Table 1
Description of the models and properties of interest (expressed in both natural language and PCTL) for the analysed code.

Analysed code	#states	#transitions	#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?	$R\{\text{“}energy\text{”}\}_{=?}\,[F\ s = end\_state]$ $R\{\text{“}time\text{”}\}_{=?}\,[F\ s = end\_state]$
`minPathSum`	21	26	28	What is the expected time? What is the expected cost?	$R\{\text{“}time\text{”}\}_{=?}\,[F\ s = end\_state]$ $R\{\text{“}cost\text{”}\}_{=?}\,[F\ s = end\_state]$
`binarySearch`	8	11	17	What is the expected time?	$R\{\text{“}time\text{”}\}_{=?}\,[F\ s = end\_state]$

running the application on the new hardware, by simply augmenting the pDTMC model from Fig. 5 with the new rewards structure

```
rewards "energy"
  s=18 : 60;
endrewards
```

where $s = 18$ is the pDTMC state modelling the statement that uses *Math.min*(…).

5. Implementation

To automate the PROPER-driven performance analysis of probabilistic programs, we implemented a tool with its high-level description shown in Fig. 2. Our PROPER prototype tool uses JavaParser[4] to parse the Java code of interest and generate the corresponding pDTMC models (Section 4.2). We implemented a customised Monte Carlo simulation method in Java for the generation of program logs for the CUA. The point estimate calculation variant of PROPER employs the probabilistic model checker PRISM [29] to analyse properties of interest (Section 4.4.1) using the transition probabilities calculation as described in Section 4.3.1. For the estimation of confidence intervals (Section 4.4.2), we extended the FACT model checker [21] and incorporated its extension into PROPER's workflow. In particular, we have adapted the FACT construct 'param double x = ...;' to also accept the number of executions of a conditional statement or loop body x over the N_0 runs of the CUA, i.e., $x_1, x_2, \ldots, x_{N_0}$. We also incorporated into FACT the confidence interval calculation of statement x described in Eqs. (6) and (7). The PROPER open-source prototype tool, the full experimental results summarised in the next section, additional information about our approach and the artefacts from its evaluation are available at https://github.com/is742/PROPER.

6. Evaluation

6.1. 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 nonfunctional properties of interest as the amount of log data increases? We used this research question to establish how the accuracy of our PROPER method is affected as the log size varies and compare the derived confidence intervals against point estimates.

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? To establish the extent to which PROPER can be used in practice, we evaluated the execution time and memory footprint during the execution of the various PROPER stages.

6.2. Experimental setup

We applied PROPER (version 1.0) in multiple scenarios using Java source-code adapted from six Java libraries and open-source applications:

1. The `distance1` Java method from the Apache Commons Math library.[5] This method calculates the L1 distance between two points in a multidimensional space, which is a distance metric widely used in applications such as machine learning. The method receives as input two integer arrays, and checks whether the arrays have equal length.
2. The `getDevicePerfomanceClass` method from the Android messaging app Telegram[6] (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 experimentation, 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.[7]
5. An implementation of the minimum path sum algorithm (`minPathSum`)[8] (see running example in Section 2).
6. An implementation of the binary search algorithm (`binarySearch`) taken from the same public repository of algorithms as `minPathSum` on GitHub. This method finds the position of a target value within a sorted array.

Table 1 provides an overview of the analysed code, along with a list of identified performance properties of interest, formally expressed in PCTL [26], that can be evaluated using our tool-supported PROPER method.

[4] https://javaparser.org

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

[6] https://github.com/DrKLO/Telegram/

[7] https://github.com/eugenp/tutorials/

[8] https://github.com/TheAlgorithms/Java/

Table 2
Property point estimate comparison for different log sizes.

Log size	Analysed code									
	`distance1`		`devPerf`	`fst`		`knapsackDP`		`minPathSum`		`binarySearch`
	Time (ms)	Cost	Energy (J)	Time (ms)	Cost	Time (ms)	Energy (J)	Time (ms)	Cost	Time (ms)
10^2	2.875	4.48	30.24	1.365	1.9	19.08	639.18	0.5759	4.137	0.0947
10^3	2.302	4.78	30.98	1.15	1.82	21.32	714.22	0.64126	4.648	0.0905
10^4	2.518	4.64	31.02	1.16	1.9	21.79	730.16	0.63304	4.5918	0.0924
10^5	2.5	4.66	30.96	1.14	1.91	21.96	735.93	0.63357	4.5978	0.0918
True value	2.5	4.66	30.96	1.14	1.91	21.96	735.93	0.63357	4.5978	0.0918

Table 3
Error percentage of property point estimates for different log sizes (compared to ground truth).

Log size	Analysed code									
	`distance1`		`devPerf`	`fst`		`knapsackDP`		`minPathSum`		`binarySearch`
	Time (ms)	Cost	Energy (J)	Time (ms)	Cost	Time (ms)	Energy (J)	Time (ms)	Cost	Time (ms)
10^2	15%	3.8%	2.3%	19.7%	0.5%	13%	13%	9%	10%	3%
10^3	7.9%	2.5%	0.06%	0.87%	4.7%	2.9%	2.9%	1.2%	1.1%	1.4%
10^4	0.7%	0.4%	0.19%	1.75%	0.5%	0.77%	0.78%	0.08%	0.13%	0.65%
10^5	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%

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. The reward values for each analysed method are detailed below. We used synthetic logs obtained by first establishing appropriate ranges for each input argument of the analysed method, and then using a set of values drawn randomly from a uniform probability distribution over this range.

In the analysis of the `distance1` code, we analyse the expected time and cost associated with the invocations of *Math.abs*(…) function and for throwing an exception, respectively, under the assumption that each *Math.abs*(…) invocation takes 2.5 ms, and that throwing the exception has a cost of 7. In `devPerf` code's analysis, we are interested in the expected energy consumption of running the code, due to an *Animations*(…) function that sets the level of the application's visual quality. Depending on its input mode, each instance of this function is linked to a different amount of energy (28 J, 34 J, 40 J or 48 J). Similarly, in the experimentation with `fst`, we measure the expected time associated with the *FastMath.sin*(…) function (where each execution takes 1.5 ms), together with the expected cost of reaching any of the two exceptions (of cost 5 each). Analysing the `knapsackDP` code, we are interested in the expected energy consumption due to a *display*(…) function located in the code (whose executions use 67 J each), and in the expected time associated with the *Math.max*(…) function, each invocation of which takes 2 ms. Regarding the code from our running example (`minPathSum`), we assess the expected time associated with each loop execution. Specifically, 0.01 ms for the first two while loops in the code, and 0.03 ms for the nested while loop. Also, we are interested in the expected cost associated with the *Math.min*(…) function, each invocation of which has a cost of 0.25. Finally, in the analysis of `binarySearch`, we measure the expected time associated with the *Integer.compare*(…) function, each invocation of which takes 0.238 ms.

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.

6.3. Results and discussion

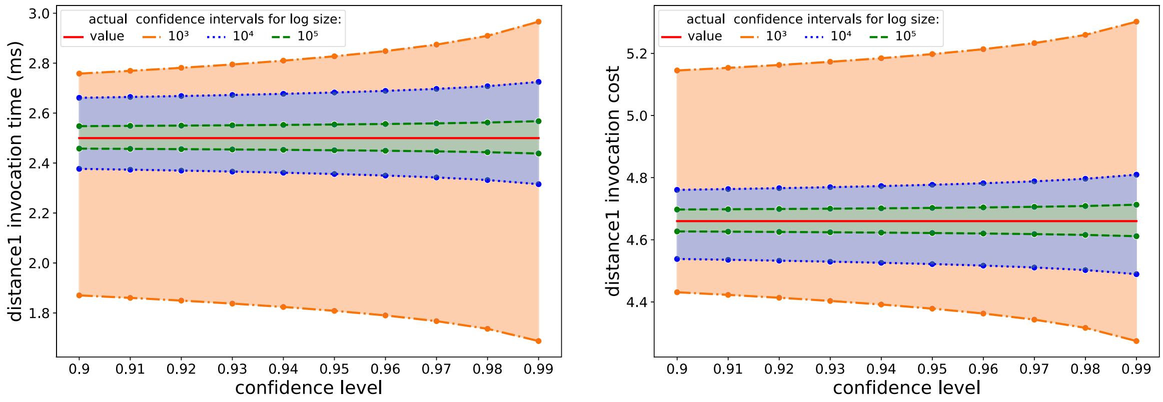
RQ1 (Accuracy). We answer RQ1 by assessing the results obtained using PROPER for various log sizes and by comparing the derived confidence intervals against point estimates. Table 2 shows the results obtained from the generation of point estimates for properties of interest for the CUA of our six code fragments using PROPER and logs of different sizes. To execute the PMC step of PROPER (Section 4.4.1) and quantify the properties shown in Table 1, we used the PRISM model checker [29] and instantiated the synthesised pDTMC models (Section 4.2) using the probabilities obtained after running the transition probability calculation step of the approach (Section 4.3).

To assess the accuracy of our approach, we created instances of logs of various sizes ranging from logs with 10^3 entries to 10^5 entries. Unsurprisingly, as can be derived from Table 2, the accuracy of point estimates with respect to the ground truth[9] heavily depends on the number of log entries. More specifically, the accuracy of properties evaluated using logs with $n < 10^5$ entries varied, deviating up to 19.7% from the ground truth for the smaller logs. The deviation followed a decreasing pattern (i.e., the verification results became more accurate) as the log size increases. We observed that for any property of any CUA with a log size that is $n \geq 10^5$, the results match almost perfectly the anticipated true values. Table 3 provides an overview of the percentage deviation for the analysed code. A general rule that can be established based on this experimentation is that the larger the number of entries of the log, the more accurate the verification results.

We also performed experiments to establish the capacity of PROPER to produce confidence intervals for the properties of interest of a CUA. Figs. 7–11 show the confidence intervals for the properties of each use case from Table 1. On the x-axis we have the confidence level in the range of $[0.9, 0.99]$, which increments with a step of 0.01, and on the y-axis we have the confidence intervals for time (ms), cost or energy (J). The graphs depict the range of these property values for each confidence level (confidence intervals) and for different log sizes (i.e., 10^3, 10^4, 10^5). As expected, while the confidence level increases the range of property values decreases and the upper and lower bounds approach the actual value, indicated by a red sold line.

These graphs reveal two important insights. First, in all use cases the confidence interval for all properties always enclosed the ground truth (illustrated as a solid red line in the figures) irrespective of the log size and the chosen confidence interval. Second, the width of the interval depends both on the selected confidence level α and the log size. As the log size increases, the confidence interval becomes smaller. Similarly, for a specific log file the confidence interval becomes

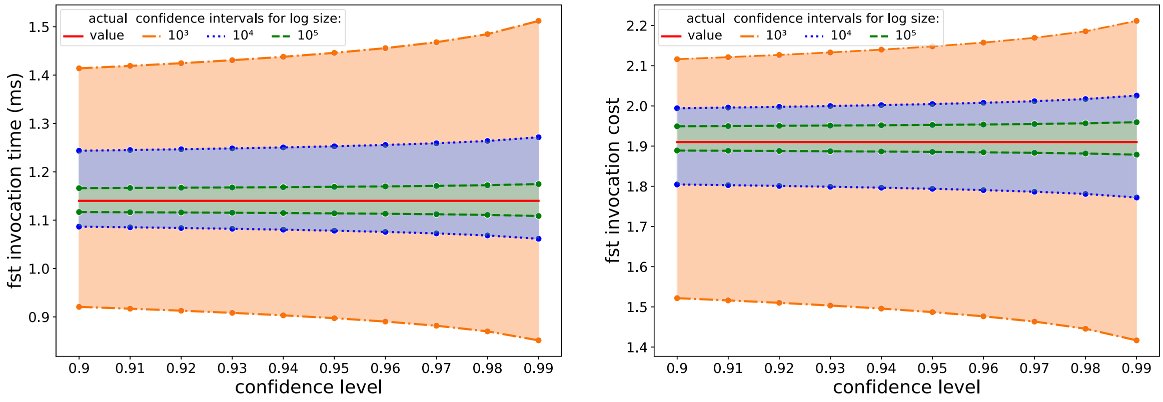
[9] We obtained the ground truth assuming that a log of $n = 10^6$ entries represents sufficiently the usage profile of the CUA. We carried out experiments with logs of larger sizes and confirmed that the results follow a similar pattern. We omit these results for clarity reasons.



(a) *time* property

(b) *cost* property

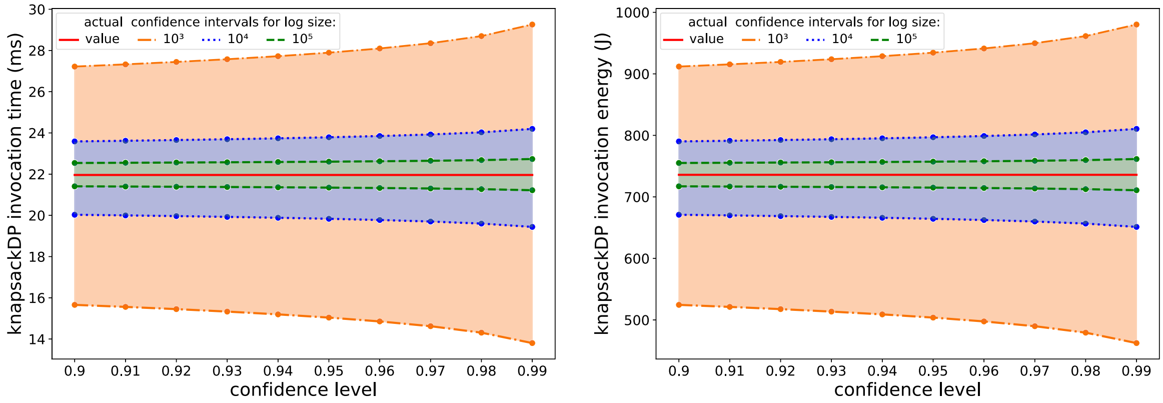
Fig. 7. Confidence intervals for the `distance1` method using logs of various sizes.



(a) *time* property

(b) *cost* property

Fig. 8. Confidence intervals for the `fst` method using logs of various sizes.



(a) *time* property

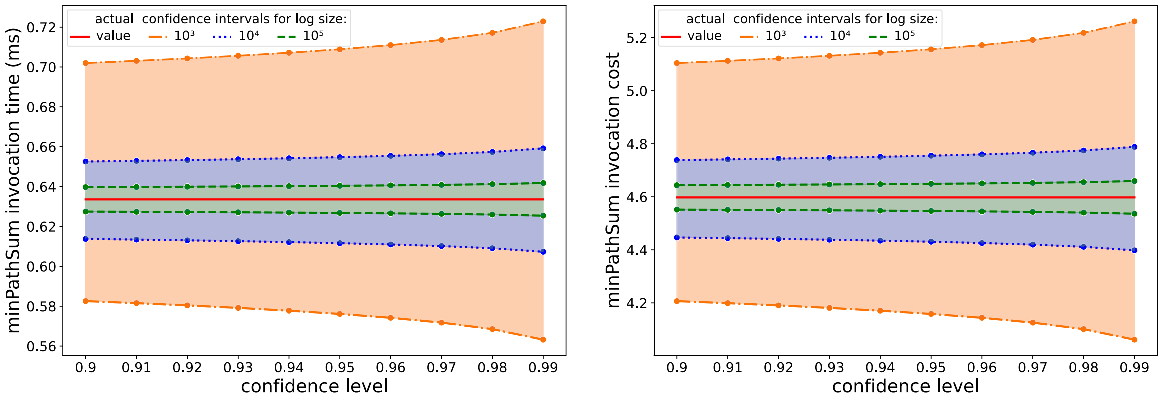
(b) *energy* property

Fig. 9. Confidence intervals for the `knapsackDP` method using logs of various sizes.

larger as the confidence level increases. Accordingly, using confidence intervals instead of point estimates can prevent wrong estimations of property values, especially in cases where the log data is limited.

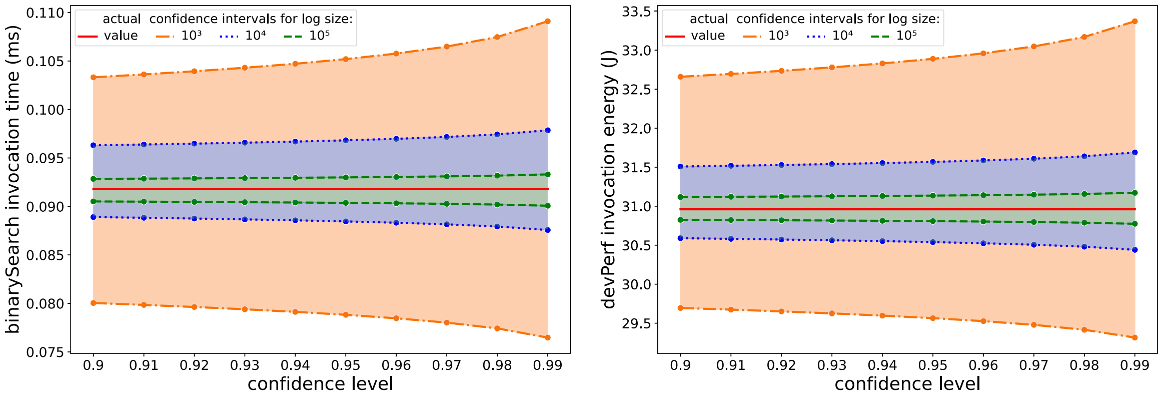
RQ2 (Decision-Making). We illustrate the capabilities of PROPER and how it can help software engineers to make informed decisions using three modification scenarios (*Scenario A*, *Scenario B*, and *Scenario C*) that frequently occur in the domains of product obsolescence [42,43] and software modernisation [44,45].

In *Scenario A*, software engineers replace one of the external methods used within the CUA to optimise the requirements defined during the design phase of the software. 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



(a) *time* property   (b) *cost* property

Fig. 10. Confidence intervals for the `minPathSum` method using logs of various sizes.



(a) *time* property   (b) *energy* property

Fig. 11. Confidence intervals for the `binarySearch` and `devPerf` methods using logs of various sizes.

Table 4
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.

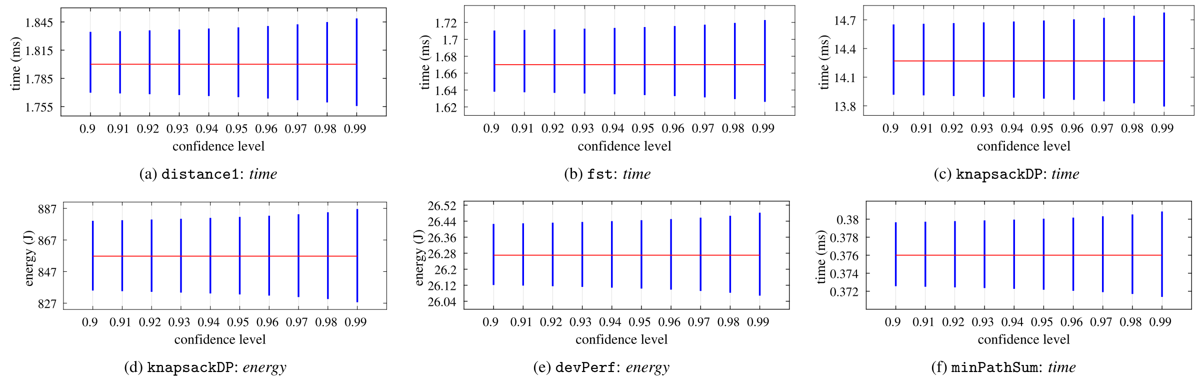
Properties	Scenario A					Scenario B				
	`distance1`	`fst`	`knapsackDP`	`devPerf`	`minPathSum`	`distance1`	`fst`	`knapsackDP`	`devPerf`	`minPathSum`
Time (ms)	**1.8**	**1.67**	**14.27**	N/A	**0.376**	**3.2**	**1.97**	**30.75**	N/A	**0.95**
Cost	4.66	1.91	N/A	N/A	N/A	4.66	1.91	N/A	N/A	N/A
Energy (J)	N/A	N/A	**856.76**	**26.27**	N/A	N/A	N/A	**900.69**	**35.73**	N/A

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 4 shows the updated results in bold obtained during *Scenario A*. In `distance1` method, we used the function *FastMath.abs*(…) that offered improved execution time (=1.8 ms) instead of *Math.abs*(…) whose execution time was 2.5 ms. The expected cost was not affected by this change, as it is only associated with the throwing of an exception earlier in the code. In the `fst` method, we replaced the *FastMath.sin*(…) function with the slower (=2.2 ms per invocation) but more reliable *Math.sin*(…) function which resulted in a slight increase in execution time (i.e., 1.14 ms with *FastMath.sin*(…) vs 1.67 ms 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 ms) function *FastMath.max*(…) instead of the *Math.max*(…) function, which resulted in reduced execution time (14.27 ms vs 21.96 ms). We also updated the *display*(…) function to increase performance using a more computationally-expensive function (=78 J), which led to an increased overall energy consumption (735.03 J vs 856.76 J before and after the change, respectively). Finally, in the `devPerf` program we assumed that the *Animations*(…) function was updated to offer better optimisations making use of the increased number of cores in modern mobile devices (=23 J, 30 J, 35 J, 43 J). This change resulted in a decrease of the energy consumption (30.96 J vs 26.27 J) 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) [46].

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 properties of interest. Table 4 (*Scenario B*) shows in bold the updated values of the performance properties for the five applications assuming that they have been deployed in a device with reduced hardware performance. Additionally, Figs. 12 and 13 depict the confidence intervals for the properties associated



(a) distance1: *time*

(b) fst: *time*

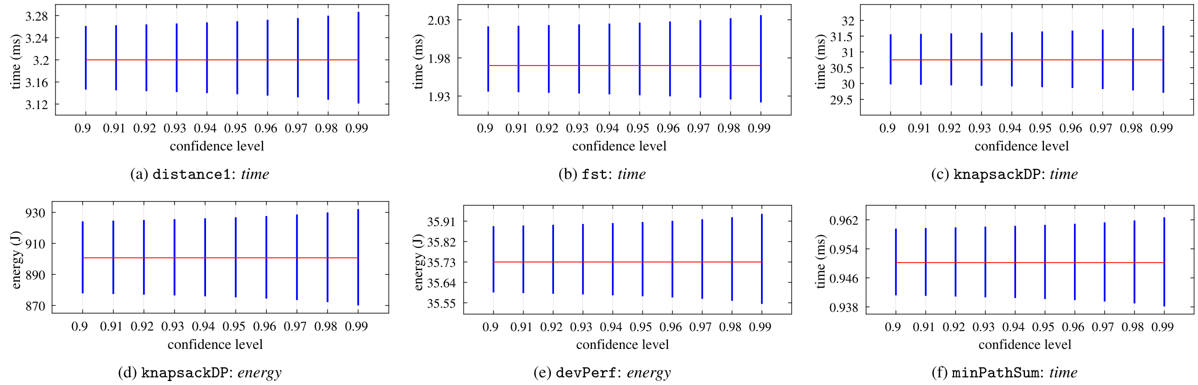
(c) knapsackDP: *time*

(d) knapsackDP: *energy*

(e) devPerf: *energy*

(f) minPathSum: *time*

Fig. 12. Confidence intervals for the evaluated properties in *Scenario A* (replacing an external function of the CUA) using a log with $n = 10^6$ entries. The blue vertical lines correspond to the range of confidence intervals for a given confidence level $\alpha \in \{0.9, 0.91, 0.92, \dots, 0.99\}$, and the solid red horizontal line is the actual value. (For interpretation of thereferences to colour in this figure legend, the reader is referred to the web version of this article.)



(a) distance1: *time*

(b) fst: *time*

(c) knapsackDP: *time*

(d) knapsackDP: *energy*

(e) devPerf: *energy*

(f) minPathSum: *time*

Fig. 13. Confidence intervals for the evaluated properties in *Scenario B* (deploying the CUA in a new device) using a log with $n=10^6$ entries. The blue vertical lines correspond to the range of confidence intervals for a given confidence level $\alpha \in \{0.9, 0.91, 0.92, \dots, 0.99\}$, and the solid red horizontal line is the actual value. (For interpretation of thereferences to colour in this figure legend, the reader is referred to the web version of this article.)

with the two scenarios assuming a log with $n = 10^6$ entries. On the x-axis we have the confidence level in the range of $[0.9, 0.99]$, which increments with a step of 0.01, and on the y-axis we have the property values. The vertical blue lines represent the range of property values for a given confidence level (confidence intervals) and the horizontal red line the actual value. The derived confidence intervals corroborate our findings from research question RQ1 regarding the capacity of PROPER to produce intervals that enclose the true value (solid horizontal red line) and support decision making considering a desirable confidence level.

Scenario C involves using PROPER to analyse the binarySearch method (Fig. 14). The results of this analysis indicated that the 'if' conditional statement in line 13 and the 'else' statement in line 16 have higher probabilities (i.e., are executed more frequently) than the earlier 'if' statement in line 9. The current structure of the method leads to additional execution time as the *compare*(…) function in the condition of the 'if' statement (line 9) will be invoked multiple times without the condition being satisfied. By applying code restructuring, this additional cost could be reduced, and thus, improve the performance of the method. The applied changes can be observed (highlighted) in Fig. 15 depicting the Java code and the synthesised pDTMC model. The 'if' statement in line 9 of Fig. 14 is now moved at the end of the 'if-else' block in line 16 of Fig. 15 as its condition has the lowest probability of being satisfied. By applying PROPER without running the CUA after refactoring the binarySearch method, we confirmed that this change will reduce the execution time of the method from 0.0918 ms (before restructuring) to 0.0737 ms.

The experimental results from the three scenarios provide sufficient evidence that PROPER can produce 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 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 PROPER, both for the calculation of point estimates and of confidence intervals for the properties of interest and the CUA. To this end, we used the *currentTimeMillis* method from Oracle's *System* (https://docs.oracle.com/javase) class for the formal verification with confidence intervals. For the calculation of point estimates, we used the output log from PRISM [29] and obtained both the time needed for model construction and model checking for each of the specified properties. To measure the memory consumption, we used the *JavaVisualVM* profiling tool which comes with the Java Development Kit (JDK).

The experimental results in Table 5 demonstrate that PROPER's analysis by point estimates is significantly faster than obtaining the confidence intervals, up to three orders of magnitude. Despite this performance difference, we can see that the execution time for both PROPER variants is under 10 s. This difference in execution time is not surprising, especially since performing probabilistic model checking using point estimates (Section 4.4.1) involves only the invocation of the model checker (e.g., PRISM [29]) to evaluate the PCTL performance property of interest. On the contrary, deriving confidence intervals for the same property (Section 4.4.2) involves not only calculating *simultaneous* confidence intervals for the transition probabilities of the pDTMC at a user-specified confidence level α but also the invocation of

```
public class BinarySearch {

    public int search(int array[], int key, int
     ↪ left, int right) {
        if (right < left) {
            return -1;
        }
        int median = (left + right) >>> 1;

        if (Integer.compare(key,array[median]) == 0)
         ↪ { //@time=0.0238
            return median;
        }
        else {
            if (Integer.compare(key,array[median]) <
             ↪ 0) { //@time=0.0238
                return search(array, key, left, median
                 ↪ - 1);
            }
            else {
                return search(array, key, median + 1,
                 ↪ right);
            }
        }
    }
}
```

```
dtmc

const double p1;
const double p2;
const double p3;

const int end_state = 8;

module binarySearch
 s : [0..end_state] init 0;

 [] s=0 -> p1:(s'=1)+(1-p1):(s'=2); //line:4
 [] s=1 -> 1:(s'=end_state);        //line:5
 [] s=2 -> 1:(s'=3);                //line:7
 [] s=3 -> p2:(s'=4)+(1-p2):(s'=5); //line:9
 [] s=4 -> 1:(s'=end_state);        //line:10
 [] s=5 -> p3:(s'=6)+(1-p3):(s'=7); //line:13
 [] s=6 -> 1:(s'=0);                //line:14
 [] s=7 -> 1:(s'=0);                //line:17
 [] s=8 -> 1:(s'=8);
endmodule

rewards "time"
 s=3 : 0.0238;
 s=5 : 0.0238;
endrewards
```

Fig. 14. Java method (left) and pDTMC model (right) of the `binarySearch` method used in Scenario C before restructuring.

```
public class BinarySearch {

    public int search(int array[], int key, int
     ↪ left, int right) {
        if (right < left) {
            return -1;
        }
        int median = (left + right) >>> 1;

        if (Integer.compare(key,array[median]) < 0) {
         ↪ //@time=0.0238
            return search(array, key, left, median -
             ↪ 1);
        }
        else {
            if (Integer.compare(key,array[median]) >
             ↪ 0){ //@time=0.0238
                return search(array, key, median + 1,
                 ↪ right);
            }
            else {
                return median;
            }
        }
    }
}
```

```
dtmc

const double p1;
const double p2;
const double p3;

const int end_state = 8;

module binarySearch
 s : [0..end_state] init 0;

 [] s=0 -> p1:(s'=1)+(1-p1):(s'=2); //line:4
 [] s=1 -> 1:(s'=end_state);        //line:5
 [] s=2 -> 1:(s'=3);                //line:7
 [] s=3 -> p2:(s'=4)+(1-p2):(s'=5); //line:9
 [] s=4 -> 1:(s'=0);                //line:10
 [] s=5 -> p3:(s'=6)+(1-p3):(s'=7); //line:13
 [] s=6 -> 1:(s'=0);                //line:14
 [] s=7 -> 1:(s'=end_state);        //line:17
 [] s=8 -> 1:(s'=8);
endmodule

rewards "time"
 s=3 : 0.0238;
 s=5 : 0.0238;
endrewards
```

Fig. 15. Java method (left) and synthesised pDTMC model (right) of the `binarySearch` method used in Scenario C after code restructuring. The shaded regions in the Java code and the pDTMC model indicate the changes compared to their original versions before restructuring from Fig. 14.

Table 5
Time and memory consumption comparison between PROPER's types of analysis (left: confidence interval; right: point estimates).

Analysed code	Probabilistic model checking with CI		Probabilistic model checking with point estimates	
	Execution time (s)	Memory consumption (MB)	Execution time (s)	Memory consumption (MB)
`distance1`	7.1	[3.2–36]	0.003	[12–39]
`fst`	6.8	[3.4–36]	0.005	[12–36]
`knapsackDP`	7.5	[4.2–37.8]	0.01	[12–37]
`devPerf`	6.7	[2.3–36]	0.004	[11–36]
`minPathSum`	7.8	[3.7–37.4]	0.012	[12–36]
`binarySearch`	7.1	[2.6–36.2]	0.002	[12–37]

a MATLAB-based optimisation package within FACT [21] to establish the confidence interval for the property. In terms of memory, PROPER independent of the CUA and type of analysis consumes on average the same amount of memory. Accordingly, we have sufficient evidence that PROPER can be used in practice with modest execution time and memory overheads.

Discussion. Our experimental evaluation provides empirical evidence for the accuracy and efficiency of our PROPER method. The evaluation also illustrates the capabilities of PROPER in supporting software engineers and practitioners to predict the performance (i.e., cost, energy, execution time) of their code when making internal changes (e.g., using a different software library to execute specific functions) or when deploying the code on a new hardware platform. Through PROPER, this performance assessment is achieved without updating the code or deploying it in the target hardware platform. Software refactoring and modernisation activities are costly and time-consuming [45,47], and PROPER helps software engineers to perform this assessment by spending less effort and cost. If the engineers are satisfied with the predicted impact of the modification, they can then proceed with its implementation.

As described in Section 4.1, applying PROPER in practice entails that the Java statements of interest of the CUA are annotated using the construct from (1). Software engineers are familiar with adding annotations within a code base [48], so we consider this task relatively easy to achieve. Determining which statements should be annotated entails an overall understanding of the code. Software engineers responsible for developing and maintaining the code clearly have this understanding, and information about the cost, energy and execution time of the statements of interest can be extracted from the documentation of employed software libraries. Finally, inferring usage profiles for the CUA involves collecting representative logs during the system's execution through code instrumentation [1], an activity incurring modest setup overheads as described in related research [14,16–18].

6.4. Threats to validity

Construct validity threats may arise from the construction of the models' representations 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, e.g., to obtain the transition probabilities in the case of calculating point estimates. To mitigate these threats, we performed simulation up to 10^6 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 pDTMC 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 establish the applicability, feasibility and scalability of PROPER in domains and applications with characteristics different from those used in our evaluation (e.g., databases, distributed systems).

7. Related work

Probabilistic software analysis (PSA) [49] has been used successfully in domains including testing, cryptographic protocols, cyber–physical systems, and reliability analysis [50]. 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* [51] and *software reliability analysis* [52]. Unlike our approach, research in these areas uses mostly techniques such as symbolic execution [53,54] and simulation [55,56], rather than probabilistic model checking.

Probabilistic symbolic execution [53] is an extension of symbolic execution that allows probabilistic reasoning. A probabilistic environment for Java based on symbolic execution is proposed in [54]. This framework can handle probabilistic programming features, and be used for the encoding and analysis of DTMCs, Bayesian Networks, etc. Additionally, research proposed in [19] 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. The work in [20] 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 [57] 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, the work described in [58] introduces reduction methods for probabilistic programs that operate purely on a syntactic level, while the research in [56] proposes a framework of incorporating path testing into reliability estimation for modular software systems. Also, [55] 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.

Techniques for the analysis of nonfunctional properties are not limited on source code-level. On the contrary, many approaches are aimed towards the system architecture-level of the targeted software system. The following paragraph describes research work falling within this category.

The approach proposed in [59] combines synthesis of spaces of system design alternatives from formal specifications of architectural styles with probabilistic formal verification. Additionally, the work presented in [10] introduces an improvement to the planning stage of self-adaptive systems by predicting the outcome of each adaptation strategy. A stochastic model is derived from a formal architecture description of the managed system with the changes imposed by each strategy. This information is then used to optimise the self-adaptation decisions to fulfil the desired quality goals.

In a similar track, the approach introduced in [60] automates the traceability between software architectural models and extra-functional results by investigating the uncertainty while bridging these two domains. This approach makes use of extra-functional patterns and antipatterns, to deduce the logical consequences between the architectural elements and analysis results. By building a graph of traces, it becomes possible to identify the most critical causes of extra-functional flaws. The model-based approach described in [11] enables software engineers to assess their design solutions for software product lines in the early stages of development. A nonfunctional MDA framework (NFMDA) is considered in [9] that embeds new types of model transformations that allow the generation of quantitative models for nonfunctional analysis. By using the framework with two methodologies, one for performance analysis and one for reliability assessment, an illustration of the relationships between nonfunctional models and software models is achieved. The above presented approaches are complementary to PROPER, as they do not support the code-level analysis of software performance and other quality properties.

We have categorised the related work in this area based on the type of analysis for the targeted system (i.e., code or system architecture-level). Our PROPER method belongs to the former type of analysis; however, in [61] we demonstrate how it can also be used in a combined approach to support software engineers in assessing the software system at both levels. Unlike other code-level analysis techniques, PROPER focuses on performance rather than reliability, synthesises probabilistic models directly from source code (while the related techniques rely on symbolic execution and simulation), and can compute confidence intervals for the analysed nonfunctional properties.

8. Conclusion

We presented PROPER, a tool-supported method for the automated performance analysis of probabilistic programs, that operates in three stages. In the first stage, PROPER automatically synthesises a pDTMC model representation of the CUA, that is parameterised by the execution probabilities of the conditional statements and loops appearing in the CUA. In the second stage, PROPER offers the flexibility of either computing point estimates or confidence intervals for the unknown pDTMC parameters. This process is carried out using information from program logs that capture the operational profile of the CUA. In the third, and final stage, PROPER enables the analysis of CUA performance properties of interest (e.g., timing, resource use, cost), and obtains point estimates or confidence intervals for these properties.

We evaluated PROPER on six applications and demonstrated how it can support the performance analysis in scenarios involving changes in hardware platforms, function libraries or usage profile. As such, PROPER can help practitioners to predict the performance that their code will achieve if deployed on a new platform, run in a scenario with a different usage profile, or modified to use a different library for some of its functions.

We plan to continue to develop our program performance analysis method by: (1) extending PROPER to support analysis of reliability properties; (2) extending the pDTMC model synthesis stage of PROPER with the ability to transform multiple Java classes into a single pDTMC model; (3) assessing its scalability to larger programs; and (4) validating PROPER using code from other systems and domains (e.g., databases, distributed systems).

CRediT authorship contribution statement

Ioannis Stefanakos: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft. **Radu Calinescu:** Conceptualization, Methodology, Software, Writing – original draft, Supervision, Funding acquisition. **Simos Gerasimou:** Conceptualization, Methodology, Software, Formal analysis, Supervision, Writing – original draft.

Declaration of competing interest

One or more of the authors of this paper have disclosed potential or pertinent conflicts of interest, which may include receipt of payment, either direct or indirect, institutional support, or association with an entity in the biomedical field which may be perceived to have potential conflict of interest with this work. For full disclosure statements refer to https://doi.org/10.1016/j.infsof.2022.107143. Ioannis Stefanakos reports financial support was provided by Assuring Autonomy International Programme. Ioannis Stefanakos reports a relationship with Assuring Autonomy International Programme that includes: funding grants. Ioannis Stefanakos reports a relationship with Microsoft Research that includes: funding grants.

Data availability

All data/code used in this work can be found in our GitHub repository for which a link is provided in the paper.

Acknowledgements

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

References

[1] S. Balsamo, A. Di Marco, P. Inverardi, M. Simeoni, Model-based performance prediction in software development: A survey, IEEE Trans. Softw. Eng. 30 (5) (2004) 295–310, http://dx.doi.org/10.1109/TSE.2004.9.

[2] L. Traini, D. Di Pompeo, M. Tucci, B. Lin, S. Scalabrino, G. Bavota, M. Lanza, R. Oliveto, V. Cortellessa, How software refactoring impacts execution time, ACM Trans. Softw. Eng. Methodol. (TOSEM) 31 (2) (2021) 1–23, http://dx.doi.org/10.1145/3485136.

[3] D. Ameller, X. Franch, C. Gómez, S. Martínez-Fernández, J. Araújo, S. Biffl, J. Cabot, V. Cortellessa, D.M. Fernández, A. Moreira, et al., Dealing with non-functional requirements in model-driven development: A survey, IEEE Trans. Softw. Eng. 47 (4) (2019) 818–835, http://dx.doi.org/10.1109/TSE.2019.2904476.

[4] L. Chung, B.A. Nixon, E. Yu, J. Mylopoulos, Non-Functional Requirements in Software Engineering, in: International Series in Software Engineering, vol. 5, Springer Science & Business Media, 2012, http://dx.doi.org/10.1007/978-1-4615-5269-7.

[5] R. Calinescu, Emerging techniques for the engineering of self-adaptive high-integrity software, in: Assurances for Self-Adaptive Systems, Springer, 2013, pp. 297–310, http://dx.doi.org/10.1007/978-3-642-36249-1_11.

[6] H. Koziolek, Performance evaluation of component-based software systems: A survey, Perform. Eval. 67 (8) (2010) 634–658, http://dx.doi.org/10.1016/j.peva.2009.07.007.

[7] M. Plauth, L. Feinbube, A. Polze, A performance survey of lightweight virtualization techniques, in: European Conference on Service-Oriented and Cloud Computing, 2017, pp. 34–48, http://dx.doi.org/10.1007/978-3-319-67262-5_3.

[8] D. Arcelli, Exploiting queuing networks to model and assess the performance of self-adaptive software systems: A survey, Procedia Comput. Sci. 170 (2020) 498–505, http://dx.doi.org/10.1016/j.procs.2020.03.108.

[9] V. Cortellessa, A. Di Marco, P. Inverardi, Integrating performance and reliability analysis in a non-functional MDA framework, in: Fundamental Approaches to Software Engineering, 2007, pp. 57–71, http://dx.doi.org/10.1007/978-3-540-71289-3_6.

[10] J.M. Franco, F. Correia, R. Barbosa, M. Zenha-Rela, B. Schmerl, D. Garlan, Improving self-adaptation planning through software architecture-based stochastic modeling, J. Syst. Softw. 115 (2016) 42–60, http://dx.doi.org/10.1016/j.jss.2016.01.026.

[11] C. Ghezzi, A. Molzam Sharifloo, Model-based verification of quantitative non-functional properties for software product lines, Inf. Softw. Technol. 55 (3) (2013) 508–524, http://dx.doi.org/10.1016/j.infsof.2012.07.017.

[12] K. Cooper, L. Dai, Y. Deng, Performance modeling and analysis of software architectures: An aspect-oriented UML based approach, Sci. Comput. Program. 57 (1) (2005) 89–108, http://dx.doi.org/10.1016/j.scico.2004.10.007.

[13] S. Becker, H. Koziolek, R. Reussner, The Palladio component model for model-driven performance prediction, J. Syst. Softw. 82 (1) (2009) 3–22, http://dx.doi.org/10.1016/j.jss.2008.03.066.

[14] N. Kumar, B.R. Childers, M.L. Soffa, Low overhead program monitoring and profiling, SIGSOFT 31 (1) (2005) 28–34, http://dx.doi.org/10.1145/1108792.1108801.

[15] T. Ball, J.R. Larus, Optimally profiling and tracing programs, TOPLAS 16 (4) (1994) 1319–1360, http://dx.doi.org/10.1145/183432.183527.

[16] P. Arafa, G.M. Tchamgoue, H. Kashif, S. Fischmeister, QDIME: QoS-aware dynamic binary instrumentation, in: MASCOTS, 2017, pp. 132–142, http://dx.doi.org/10.1109/MASCOTS.2017.19.

[17] A. Van Hoorn, J. Waller, W. Hasselbring, Kieker: A framework for application performance monitoring and dynamic software analysis, in: ICPE, 2012, pp. 247–248, http://dx.doi.org/10.1145/2188286.2188326.

[18] S. Schubert, D. Kostic, W. Zwaenepoel, K.G. Shin, Profiling software for energy consumption, in: GreenCom, 2012, pp. 515–522, http://dx.doi.org/10.1109/GreenCom.2012.86.

[19] A. Filieri, C. Pasareanu, W. Visser, Reliability analysis in symbolic pathfinder, in: ICSE, 2013, pp. 622–631, http://dx.doi.org/10.1109/ICSE.2013.6606608.

[20] A. Filieri, C.S. Pasareanu, G. Yang, Quantification of software changes through probabilistic symbolic execution, in: ASE, 2015, pp. 703–708, http://dx.doi.org/10.1109/ASE.2015.78.

[21] R. Calinescu, K. Johnson, C. Paterson, FACT: A probabilistic model checker for formal verification with confidence intervals, in: TACAS, 2016, pp. 540–546, http://dx.doi.org/10.1007/978-3-662-49674-9_32.

[22] I. Stefanakos, R. Calinescu, S. Gerasimou, Probabilistic program performance analysis, in: 47th Euromicro Conference on Software Engineering and Advanced Applications, SEAA, 2021, pp. 148–157, http://dx.doi.org/10.1109/SEAA53835.2021.00027.

[23] J.-P. Katoen, The probabilistic model checking landscape, in: LICS, 2016, pp. 31–45, http://dx.doi.org/10.1145/2933575.2934574.

[24] S. Andova, H. Hermanns, J.-P. Katoen, Discrete-time rewards model-checked, in: International Conference on Formal Modeling and Analysis of Timed Systems, 2004, pp. 88–104, http://dx.doi.org/10.1007/978-3-540-40903-8_8.

[25] C. Daws, Symbolic and parametric model checking of discrete-time Markov chains, in: International Colloquium on Theoretical Aspects of Computing, Springer, 2004, pp. 280–294, http://dx.doi.org/10.1007/978-3-540-31862-0_21.

[26] H. Hansson, B. Jonsson, A logic for reasoning about time and reliability, Form. Asp. Comput. 6 (1994) 512–535, http://dx.doi.org/10.1007/BF01211866.

[27] R. Calinescu, C. Ghezzi, K. Johnson, M. Pezzé, Y. Rafiq, G. Tamburrelli, Formal verification with confidence intervals to establish quality of service properties of software systems, IEEE Trans. Reliab. 65 (1) (2015) 107–125, http://dx.doi.org/10.1109/TR.2015.2452931.

[28] E.M. Hahn, H. Hermanns, B. Wachter, L. Zhang, PARAM: A model checker for parametric Markov models, in: Computer Aided Verification, Springer, 2010, pp. 660–664, http://dx.doi.org/10.1007/978-3-642-14295-6_56.

[29] M. Kwiatkowska, G. Norman, D. Parker, PRISM 4.0: Verification of probabilistic real-time systems, in: CAV, 2011, pp. 585–591, http://dx.doi.org/10.1007/978-3-642-22110-1_47.

[30] C. Dehnert, S. Junges, J. Katoen, M. Volk, A storm is coming: A modern probabilistic model checker, in: Computer Aided Verification, 2017, pp. 592–600, http://dx.doi.org/10.1007/978-3-319-63390-9_31.

[31] R. Calinescu, C.A. Paterson, K. Johnson, Efficient parametric model checking using domain knowledge, IEEE Trans. Softw. Eng. 47 (6) (2021) 1114–1133, http://dx.doi.org/10.1109/TSE.2019.2912958.

[32] X. Fang, R. Calinescu, S. Gerasimou, F. Alhwikem, Fast parametric model checking through model fragmentation, in: 43rd IEEE/ACM International Conference on Software Engineering, ICSE, 2021, pp. 835–846, http://dx.doi.org/10.1109/ICSE43902.2021.00081.

[33] M. Hort, M. Kechagia, F. Sarro, M. Harman, A survey of performance optimization for mobile applications, IEEE Trans. Softw. Eng. 48 (8) (2022) 2879–2904, http://dx.doi.org/10.1109/TSE.2021.3071193.

[34] M. Borges, Q.-S. Phan, A. Filieri, C.S. Păsăreanu, Model-counting approaches for nonlinear numerical constraints, in: NASA Formal Methods, 2017, pp. 131–138, http://dx.doi.org/10.1007/978-3-319-57288-8_9.

[35] J.A. De Loera, R. Hemmecke, J. Tauzer, R. Yoshida, Effective lattice point counting in rational convex polytopes, J. Symbolic Comput. 38 (4) (2004) 1273–1302, http://dx.doi.org/10.1016/j.jsc.2003.04.003.

[36] W. Visser, N. Bjørner, N. Shankar, Software engineering and automated deduction, in: Proceedings of the on Future of Software Engineering, Association for Computing Machinery, 2014, pp. 155–166, http://dx.doi.org/10.1145/2593882.2593899.

[37] A. Filieri, M.F. Frias, C.S. Păsăreanu, W. Visser, Model counting for complex data structures, in: Model Checking Software, Springer International Publishing, 2015, pp. 222–241, http://dx.doi.org/10.1007/978-3-319-23404-5_15.

[38] W. Visser, C.S. Păsăreanu, Probabilistic programming for Java using symbolic execution and model counting, in: Proceedings of the South African Institute of Computer Scientists and Information Technologists, Association for Computing Machinery, 2017, pp. 1–10, http://dx.doi.org/10.1145/3129416.3129433.

[39] S. Teuber, A. Weigl, Quantifying software reliability via model-counting, in: Quantitative Evaluation of Systems, Springer International Publishing, 2021, pp. 59–79, http://dx.doi.org/10.1007/978-3-030-85172-9_4.

[40] D.P. Bertsekas, J.N. Tsitsiklis, Introduction to Probability, Vol. 1, Athena Scientific, 2008.

[41] G. Su, D.S. Rosenblum, Asymptotic bounds for quantitative verification of perturbed probabilistic systems, in: International Conference on Formal Engineering Methods, Springer, 2013, pp. 297–312, http://dx.doi.org/10.1007/978-3-642-41202-8_20.

[42] B. Bartels, U. Ermel, P. Sandborn, M.G. Pecht, Strategies to the Prediction, Mitigation and Management of Product Obsolescence, Vol. 87, John Wiley & Sons, 2012, http://dx.doi.org/10.1002/9781118275474.

[43] S. Gerasimou, D. Kolovos, R. Paige, M. Standish, Technical obsolescence management strategies for safety-related software for airborne systems, in: Federation of International Conferences on Software Technologies: Applications and Foundations, Springer, 2017, pp. 385–393, http://dx.doi.org/10.1007/978-3-319-74730-9_34.

[44] B.E. Cossette, R.J. Walker, Seeking the ground truth: A retroactive study on the evolution and migration of software libraries, in: FSE, 2012, pp. 1–11, http://dx.doi.org/10.1145/2393596.2393661.

[45] S. Gerasimou, M. Kechagia, D. Kolovos, R. Paige, G. Gousios, On software modernisation due to library obsolescence, in: 2018 IEEE/ACM 2nd International Workshop on API Usage and Evolution, WAPI, IEEE, 2018, pp. 6–9, http://dx.doi.org/10.1145/3194793.3194798.

[46] S. Wood, N. Matragkas, D. Kolovos, R. Paige, S. Gerasimou, Supporting robotic software migration using static analysis and model-driven engineering, in: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, 2020, pp. 154–164, http://dx.doi.org/10.1145/3365438.3410965.

[47] M. Kim, T. Zimmermann, N. Nagappan, A field study of refactoring challenges and benefits, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012, pp. 1–11, http://dx.doi.org/10.1145/2393596.2393655.

[48] Z. Yu, C. Bai, L. Seinturier, M. Monperrus, Characterizing the usage, evolution and impact of Java annotations in practice, IEEE Trans. Softw. Eng. 47 (5) (2019) 969–986, http://dx.doi.org/10.48550/arXiv.1805.01965.

[49] M.B. Dwyer, A. Filieri, J. Geldenhuys, M.J. Gerrard, C.S. Pasareanu, W. Visser, Probabilistic program analysis, in: Grand Timely Topics in Software Engineering, GTTSE, in: Lecture Notes in Computer Science, vol. 10223, Springer, 2015, pp. 1–25, http://dx.doi.org/10.1007/978-3-319-60074-1_1.

[50] A.D. Gordon, T.A. Henzinger, A.V. Nori, S.K. Rajamani, Probabilistic programming, in: Proceedings of the on Future of Software Engineering, 2014, pp. 167–181, http://dx.doi.org/10.1145/2593882.2593900.

[51] K.H. Bennett, V. Rajlich, Software maintenance and evolution: A roadmap, in: ICSE, 2000, pp. 73–87, http://dx.doi.org/10.1145/336512.336534.

[52] P.N. Misra, Software reliability analysis, IBM Syst. J. 22 (3) (1983) 262–270, http://dx.doi.org/10.1147/sj.223.0262.

[53] J. Geldenhuys, M.B. Dwyer, W. Visser, Probabilistic symbolic execution, in: ISSTA, 2012, pp. 166–176, http://dx.doi.org/10.1145/2338965.2336773.

[54] W. Visser, C. Pasareanu, Probabilistic programming for Java using symbolic execution and model counting, in: SAICSIT, 2017, pp. 1–10, http://dx.doi.org/10.1145/3129416.3129433.

[55] S.S. Gokhale, Michael Rung-Tsong Lyu, A simulation approach to structure-based software reliability analysis, IEEE Trans. Softw. Eng. 31 (8) (2005) 643–656, http://dx.doi.org/10.1109/TSE.2005.86.

[56] C. Hsu, C. Huang, An adaptive reliability analysis using path testing for complex component-based software systems, IEEE Trans. Reliab. 60 (1) (2011) 158–170, http://dx.doi.org/10.1109/TR.2011.2104490.

[57] J. Zhang, Y. Lu, K. Shi, C. Xu, Empirical research on the application of a structure-based software reliability model, IEEE/CAA J. Autom. Sin. (2020) 1–10, http://dx.doi.org/10.1109/JAS.2020.1003309.

[58] C. Dubslaff, A. Morozov, C. Baier, K. Janschek, Reduction methods on probabilistic control-flow programs for reliability analysis, 2020, http://dx.doi.org/10.48550/arXiv.2004.06637, CoRR.

[59] J. Camara, D. Garlan, B. Schmerl, Synthesis and quantitative verification of tradeoff spaces for families of software systems, in: Software Architecture, 2017, pp. 3–21, http://dx.doi.org/10.1007/978-3-319-65831-5_1.

[60] C. Trubiani, A. Ghabi, A. Egyed, Exploiting traceability uncertainty between software architectural models and extra-functional results, J. Syst. Softw. 125 (2017) 15–34, http://dx.doi.org/10.1016/j.jss.2016.11.032.

[61] I. Stefanakos, S. Gerasimou, R. Calinescu, Software performance engineering with performance antipatterns and code-level probabilistic analysis, in: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, IEEE, 2021, pp. 249–253, http://dx.doi.org/10.1109/MODELS-C53483.2021.00045.