Exact and Approximate Probabilistic Symbolic Execution for Nondeterministic Programs

Kasper Luckow
Aalborg University, Denmark

Corina S. Păsăreanu
CMU/NASA Ames Research Center, USA

Matthew B. Dwyer
University of Nebraska, USA

Antonio Filieri
University of Stuttgart, Germany

Willem Visser
University of Stellenbosch, South Africa

ABSTRACT

Probabilistic software analysis seeks to quantify the likelihood of reaching a target event under uncertain environments. Recent approaches compute probabilities of execution paths using symbolic execution, but do not support nondeterminism. Nondeterminism arises naturally when no suitable probabilistic model can capture a program behavior, e.g., for multithreading or distributed systems.

In this work, we propose a technique, based on symbolic execution, to synthesize schedulers that resolve nondeterminism to maximize the probability of reaching a target event. To scale to large systems, we also introduce approximate algorithms to search for good schedulers, speeding up established random sampling and reinforcement learning results through the quantification of path probabilities based on symbolic execution.

We implemented the techniques in Symbolic PathFinder and evaluated them on nondeterministic Java programs. We show that our algorithms significantly improve upon a state-of-the-art statistical model checking algorithm, originally developed for Markov Decision Processes.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking, Reliability, Statistical methods*

1. INTRODUCTION

Probabilistic software analysis aims to quantify the probability that a software system satisfies a required property, under given probabilistic usage profiles. Recent applications include cyber-physical systems, e.g., check that the probability of an unmanned aerial vehicle turning too fast is less than 10^{-6}, by analyzing the vehicle's control software, under suitable profiles built from the telemetry data of previous versions or similar systems. Such critical systems are usually checked through simulation only and probabilistic software analysis can complement that, for increased assurance. Other applications include program understanding and debugging [18], computing software reliability [15, 6], quantitative information flow analysis for security [32], etc.

Traditional formal approaches based on probabilistic model checking [24, 2] require a high-level design or architectural model of the software. However such models are difficult to maintain and may abstract important details that impact the chance of property satisfaction in the system. Our goal is to perform probabilistic analysis directly on implementations, not on high-level models. Recent promising approaches, developed by us and others, have proposed to use bounded symbolic execution [18, 15, 35, 6] to support probabilistic analysis on the source code. The analyses in [18, 15] address programs with integer domains and linear constraints, with [15] also treating complex data structures as inputs, while the analyses in [35, 6] address programs with linear and complex floating-point computations, respectively. However, none of these (with the exception of [15] discussed below) treat the orthogonal but important issue of nondeterminism. Nondeterminism arises naturally when no suitable probabilistic model can capture a program behavior, e.g., for multithreaded, distributed or component-based systems.

In this paper, we extend probabilistic symbolic execution of programs to deal with nondeterminism. We aim to compute a *scheduler* that resolves the nondeterminism to maximize the probability of property satisfaction, or conversely that maximizes the probability of non-satisfaction. Inspection of the computed scheduler will then provide insights for the design of the analyzed system, to debug or improve it. In [15] we proposed to compute probabilities along linear schedules (i.e. thread interleavings) and report the best/-worst cases to the user. In this paper we examine tree-like schedulers that can provide more precise information about the best/worst cases as compared to linear schedulers (see example in the next section).

We first describe a simple exact algorithm for computing a tree-like scheduler that resolves the nondeterminism to maximize the probability of property satisfaction (or failure). The algorithm takes a bottom-up approach to propagate computed values, and resembles the value-iteration for Markov Decision Processes (MDPs) [38], but it works directly on the code (not on MDP models) and is tailored to ef-

ASE'14, September 15-19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2643011.

ficiently process the symbolic tree generated with a bounded symbolic execution of the program. This algorithm forms the basis of an approximate algorithm for the synthesis of schedulers that we use for increased scalability.

The approximate algorithm uses Monte Carlo sampling over program paths as dictated by the conditional probabilities computed from the conditions in the code (using symbolic execution). One well-known shortcoming of sampling-based techniques [27] is that, unlike an exact probabilistic analysis, they cannot be directly applied to systems featuring nondeterminism, since it is not clear how to take meaningful decisions for nondeterministic choices during the Monte Carlo sampling. To address this problem, our algorithm starts by assuming a uniform distribution over the nondeterministic choices (i.e. assumes all nondeterministic choices are equally likely) and then iteratively uses *reinforcement learning* to provably improve the resolution of nondeterminism with respect to the target event. A key insight for our randomized algorithm is that the search for the best scheduler can be accelerated by exploiting the full probabilistic quantification of sampled symbolic paths. This also enables state *pruning* to reduce the sampling space and speed-up the technique.

To study the effectiveness of learning, we have also implemented a baseline algorithm that simply uses a uniform distribution for the nondeterministic choices (with no learning), but which can also benefit from the state pruning.

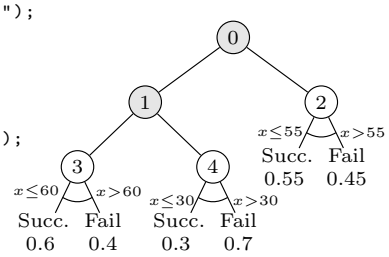
Both the learning-based and the baseline approximate algorithms significantly improve upon a state-of-the-art statistical model checking algorithm, originally developed for MDPs [20]. That algorithm also uses sampling and reinforcement learning, but it needs to sample multiple (possibly many) times along the same path to obtain a good estimate of the *quality function* used for reinforcement [37]. In our case, it is sufficient to sample a path only *once* to gather the full count of all the inputs associated with that path. Despite the potentially high cost of computing the full count, the benefit over pure statistical estimation, which works with counts incremented once per sample, leads to a significant improvement in performance for our algorithms. Furthermore, our algorithms enable aggressive state pruning which is not possible with classical statistical approaches.

Our approximate algorithms are *true biased* (meaning that true results can always be trusted), can be made arbitrarily correct (Theorem 1) and in the limit converge to the results of the exact analysis (Theorem 3 and Proposition 2), making them suitable for the analysis of critical software. In contrast, we show that the statistical approach from [20] does not always converge (see the example in the next section).

We make our presentation in terms of Java bytecode analysis and the Symbolic Pathfinder (SPF) symbolic execution tool [33]. However our algorithms are applicable in the context of other languages for which symbolic execution tools exists (e.g., Klee [8] for C). The contributions of our work are: (1) an exact algorithm for probabilistic bounded symbolic execution of nondeterministic programs; (2) approximate algorithms that exploit accelerated sampling of symbolic paths, reinforcement learning and state-space pruning, with theoretical guarantees; (3) the extension of SPF to implement probabilistic symbolic execution of nondeterministic programs, and (4) evidence from applying the implementation to a collection of multithreaded Java programs that our algorithms outperform existing methods.

```
public static void testMethod1(int x) {
 if(Verify.getBoolean()) {
  if(Verify.getBoolean()) {
   if(x <= 60)
    println("success");
   else
    assert false;
  } else {
   if(x <= 30)
    println("success");
   else
    assert false;
  }
 } else {
  if(x <= 55)
   println("success");
  else
   assert false;
 }
}
```

(a) Source snippet.



(b) Tree.

Figure 1: Example 1

2. EXAMPLES

In this section, we provide examples that illustrate our approach and facilitate comparison with [20, 15].

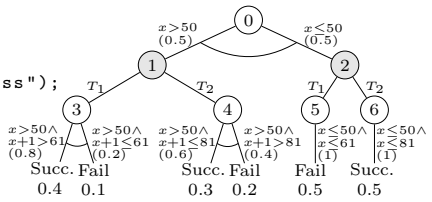
Example 1 Figure 1 shows an example Java program with nondeterministic code obtained by method `Verify.getBoolean()`, which nondeterministically returns true or false in SPF. Assume that the input `x` ranges over [1..100] – in practice, the input domain can be much larger.

The corresponding symbolic execution tree is also sketched in the figure; it encodes all the paths taken during the symbolic execution of the program. Shaded nodes represent nondeterministic choices; white nodes represent probabilistic choices. We also annotated tree edges with the conditions on the inputs to reach that edge, i.e. the path conditions computed with symbolic execution. The probabilities are computed from the path conditions, using a quantification procedure (as described in Section 4). For example, assuming a uniform usage profile, the probability of taking the *then* branch through the code corresponding to condition $x \leq 60$ is $60/100 = 0.6$, since there are 60 inputs that satisfy the condition, out of 100 possible inputs. Similarly, the probability of taking the *else* branch corresponding to condition $x > 60$ is 0.4 etc.

Our goal is to identify a *scheduler* that decides for each nondeterministic choice the best alternative to select to maximize the probability of success. The execution tree can be seen as an MDP and analyzed by probabilistic model checking [38, 17]. The result can also be computed with our *Exact* algorithm, yielding the scheduler that selects $0 \rightarrow 1 \rightarrow 3$ in the tree, with the maximum success probability 0.6.

We have also analyzed the example using our approximate algorithms where we fixed the number of samples to 100 (default greediness and history parameters 0.5; see Section 5). For the approximate algorithms, we pose a different verification query: instead of asking for the maximum probability, we ask if *there exists a scheduler for which the probability of success is greater or equal to an hypothesis* (see Section 5 for the existential/universal queries we can answer). In our case the hypothesis is 0.6 corresponding to the maximum probability for success. The solution is found easily (no learning

```
public static void testMethod2(int x) {
 if(x > 50)
  x++;
 if(Verify.getBoolean()) {//T1
   if(x > 61)
    println("success");
   else
    assert false;
 } else {//T2
   if(x <= 81)
    println("success");
   else
    assert false;
 }
}
```



(a) Source snippet. (b) Tree.

Figure 2: Example 2

necessary). Pruning further accelerates the search for an optimal scheduler, as it prevents resampling the same paths.

We analyzed the same example using the statistical model checking algorithm from [20]. For 100 samples (and same default parameters), the algorithm first computes a suboptimal scheduler $0 \to 2$, with maximum probability of success 0.55 and then it is not able to improve on it due to the poor information obtained from sampling. Furthermore, even assuming perfect information from sampling (e.g., by increasing the number of samples to 10000, or by replacing the statistical assessment with an exact computation) the algorithm is still not able to stabilize towards the best scheduler. The reason is that the approach reaches a point where the quality measures for nodes 1 and 2 are the same (0.55) and from that point on no progress can be made.

This example shows that the algorithm may fail to learn the scheduler in the limit, even assuming perfect information from sampling, thus contradicting the convergence results from [20]. These findings were graciously confirmed by the authors of [20]. In contrast, our algorithms are guaranteed to find the correct answer, in the worst case after all the paths have been explored at least once, though in practice they may converge earlier.

Example 2 This example illustrates that tree-like schedulers can obtain more precise information than linear schedulers [15]. Figure 2 shows another nondeterministic Java program. We have marked with T_1 and T_2 the tasks (i.e. the code fragments) that can be performed nondeterministically by the program, according to the choice prescribed by `Verify.getBoolean()`. Assume again that the input variable `x` ranges over [1..100]. The corresponding symbolic execution tree is also sketched in the figure. We also annotated tree edges with path conditions and the corresponding conditional probabilities (Section 4). Each path through the tree leads to either success or failure, with the corresponding path probabilities also depicted in the figure. For example, path $0 \to 1 \to 3$ leads to success with probability $0.5 \cdot 0.8 = 0.4$.

If we take the approach from [15], we compute the probability of success along each linear schedule and then report the maximum. For our simple example, we only have two linear schedules, corresponding to choosing to perform either task T_1 or task T_2. If the scheduler chooses T_1, then the probability of success is 0.4 (for path $0 \to 1 \to 3$) while if the scheduler chooses T_2, the probability of success is the sum of probabilities along paths $0 \to 1 \to 4$ and $0 \to 2 \to 6$, respectively, yielding 0.3 plus 0.5 for a total of 0.8, which can be deemed as the maximum value. However consider now a tree-like scheduler that in state 1 decides to take T_1 while in state 2 decides to take T_2. This yields probability of success 0.4 (path $0 \to 1 \to 3$) plus 0.5 (path $0 \to 2 \to 6$), yielding 0.9 which is larger than the probabilities computed along linear schedules. In the rest of the paper, we will describe exact and approximate algorithms for computing tree-like schedulers.

3. PRELIMINARIES

In this section, we give background information for symbolic execution and probabilistic analysis in the context of sequential programs. We will extend these notions to programs with nondeterminism in Section 4.

Symbolic Execution Symbolic Execution [22, 11] is a program analysis technique that executes programs on unspecified inputs, by using symbolic inputs instead of concrete data. The state of a symbolically executed program is defined by the (symbolic) values of the program variables, a *path condition* (pc), and a program counter. The path condition is a (quantifier-free) boolean formula over the symbolic inputs; it accumulates constraints that concrete on the inputs to follow that path. The program counter defines the next statement to be executed.

A *symbolic execution tree* characterizes the execution paths followed during symbolic execution. The tree nodes represent program states and the arcs the transitions between states due to the execution of program instructions. We built our approach upon the symbolic execution tool Symbolic Java PathFinder (SPF) [33], which has built-in support for preconditions (used for encoding usage profiles).

Probabilistic Analysis The goal of the analysis is: (1) to identify the symbolic constraints characterizing the inputs that make the execution satisfy a given property, and then (2) to quantify the probability of satisfying the constraints. For simplicity, we assume the satisfaction of the target property to be characterized by the occurrence of a target event (e.g., successful termination or failure), but our work generalizes to bounded LTL properties [40].

To deal with programs with loops, we perform a bounded symbolic execution of the program. The result is a finite set of symbolic paths, each with a path condition. Some of these paths lead to failure, some of them to success (termination without failure) and some of them lead neither to success nor failure (they were interrupted because of the bounded exploration) – the latter are called *grey* paths. The path conditions are therefore classified in three sets: $PC^s = \{pc_1^s, pc_2^s, \ldots, pc_m^s\}$, $PC^f = \{pc_1^f, pc_2^f, \ldots, pc_p^f\}$ and $PC^g = \{pc_1^g, pc_2^g, \ldots, pc_q^g\}$. The path conditions define disjoint input sets and they cover the whole input domain.

Usage Profiles The constraints generated with symbolic execution are analyzed to quantify the likelihood of an input to satisfy them, where the inputs are distributed according to given *usage profiles* [15]. The usage profile is a probabilistic characterization of the software interactions with the external world, e.g., the users or the physical execution environment. It assigns to each valid combination of inputs its probability to occur during execution. Usage profiles can

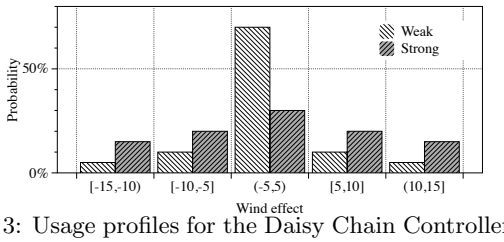


Figure 3: Usage profiles for the Daisy Chain Controller [15].

come from monitoring the usage of actual or similar systems or expert and domain knowledge (physical phenomena). In this paper, we assume that the usage profile is given and we direct the reader to the literature on usage and operational profiles for further details on their specification, automatic inference, and advanced applications (e.g., [29, 30, 19]).

We can handle arbitrarily complex probability distributions for usage profiles by *discretizing* them up to the required accuracy. The discretized distribution partitions the inputs in as many (non-empty) sets as needed and assigns to each of them a probability p, represented by a rational number with arbitrary precision. In [15] we provide an extensive treatment of usage profiles and show how they are used *after* the symbolic execution of the program is performed to compute the desired probabilities. Here we take a different (but equivalent) approach, and encode the constraints that define the usage profile as *preconditions* for the analyzed code. Handling the usage profiles in this way is necessary to support the Monte Carlo simulation, which requires a forward computation of the probability of each branch to drive the symbolic execution. More general usage profiles, given as e.g., Markov Chains, could be encoded similarly (as "stateful" assumptions); we leave this for future work.

Figure 3 illustrates two non-uniform wind-effect usage profiles. We will show in Section 6 how we encode them for the probabilistic analysis of a Daisy Chain Controller.

Probability of Target Event The probability of success is then defined as the probability of executing the program (extended with the preconditions) P with an input satisfying any of the successful path conditions: $Pr^s(P) = \sum_i Pr(pc_i^s)$.

An analogous definition is provided for the probability of failure, $Pr^f(P)$, and the probability of grey, $Pr^g(P)$. Note that $Pr^s(P) + Pr^f(P) + Pr^g(P) = 1$.

$Pr^g(P)$ can be used to define the *confidence* we can put on probability estimation, under current exploration bound [15].

Quantification Procedure To compute the probabilities of path conditions, we use a quantification procedure for the generated constraints. In [18, 15] we used model counting techniques, i.e. LattE [14], to estimate (algorithmically) the exact number of points of a bounded (possibly very large) discrete domain that satisfy linear constraints. The work in [15] was extended to handle arbitrary complex floating-point constraints in [6], using QCoral, an *approximate* quantification procedure.

For simplicity, we use LattE [14], but our work can also accommodate QCoral [6]. However, the approximate nature of QCoral would complicate the presentation of the approximate treatment of nondeterminism (we just note briefly that Proposition 2 would hold when using QCoral too).

For a finite (possibly very large, nonempty) integer domain D and a given constraint c, LattE computes the number of elements of D that satisfy c, denoted as $\sharp(c)$. $Pr(c)$ is then defined as $\sharp(c)/\sharp(D)$ (where $\sharp(D)$ is the size of D).

The success probability (or failure or grey probability) can then be computed as $Pr^s(P) = \sum_i Pr(pc_i^s) = \frac{\sum_i \sharp(pc_i^s)}{\sharp(D)}$

4. PROBABILISTIC ANALYSIS FOR NONDETERMINISTIC PROGRAMS

We consider now the problem of probabilistic analysis for nondeterministic programs. Without making any assumption on the way the nondeterministic choices are resolved (i.e. without assuming a-priori a next-choice distribution nor a specific scheduling policy) we want to identify the best possible choices from each state, i.e., the choices that lead to the highest probability of success; conversely, we may want to identify the worst possible choices which lead to the lowest probability of success.

Symbolic Execution First, we extend the definition of symbolic execution provided in Section 3 to account for nondeterministic choices. We extend the symbolic execution tree with a new kind of node corresponding to nondeterministic choices in the program. Thus, the *symbolic execution tree* of a nondeterministic program has three kinds of nodes (or states): i) *PC*: path condition choice; ii) *NC*: nondeterministic choice; iii) *other*: all the other nodes (i.e. assignments, method invocations, returns, etc.)

A PC choice is introduced whenever a conditional statement is executed in the program. The evaluation of the statement (on condition c) introduces two new transitions. The first one leads to the execution of the *then* block in the code and the path condition is updated as $pc_{then} = c \wedge pc$. The second leads to the execution of the *else* block and the path condition is updated with $pc_{else} = \neg c \wedge pc$. If the path condition for a branch is not satisfiable, symbolic execution will not follow the branch.

A NC choice is introduced whenever nondeterminism is present in the analyzed application; this may be due to handling of multithreading or to explicit nondeterministic instructions in the code (e.g., `Verify.getBoolean()`).

A symbolic execution tree is then denoted as $T = \langle S, s_0, \rightarrow, S_{NC}, S_{PC} \rangle$, where S is the set of nodes, s_0 is the initial state, $\rightarrow \subseteq S \times S$ is the transition relation, $S_{NC} \subseteq S$ is the set of NC nodes and $S_{PC} \subseteq S$ is the set of PC nodes (S_{NC} and S_{PC} are disjoint by construction). Let $child(s)$ denote the children nodes of s and let $parent(s)$ denote the parent of s. Note that both NC and PC nodes can have more than one child, while all the other nodes can have at most one.

Branch probabilities for PC nodes For a PC node, we define branch probabilities as the probability of taking the *then* or the *else* branch from the given PC node. These branch probabilities can be computed using model counting as was done in [18]. Let pc_s be the path condition at the current PC node s, and let c be the branching condition at that state. We can then compute the branch probabilities as follows.

$$p_{then} = Pr(c|pc_s) = \frac{\sharp(c \wedge pc_s)}{\sharp(pc_s)}$$

$$p_{else} = Pr(\neg c|pc_s) = \frac{\sharp(\neg c \wedge pc_s)}{\sharp(pc_s)}$$

Note that $\sharp(pc_{then}) + \sharp(pc_{else}) = \sharp(pc_s)$, thus $p_{then} + p_{else} = 1$.

Probabilistic Analysis In our setting, the symbolic execution trees computed with the probabilistic symbolic execution described above can be seen as a tree-shaped MDP [17]. MDPs are a popular choice to model discrete state transition systems that are both probabilistic and nondeterministic. Schedulers are functions used to resolve the nondeterminism in MDPs. An MDP in which nondeterminism has been resolved becomes a fully probabilistic system known as a Markov Chain. In our case, the NC nodes in the symbolic execution tree have only outgoing nondeterministic transitions, while the PC nodes only have probabilistic transitions.

Without going into much detail about MDPs, we can borrow from the literature on MDPs and define a memoryless scheduler σ for a symbolic execution tree which resolves the nondeterminism in each NC node. Note that in general memoryless schedulers are insufficient for achieving the maximal probability for bounded properties; schedulers that maintain historic information may be more powerful. However, similar to previous approaches [20] we study here memoryless schedulers, that are simpler and can be computed efficiently. We will study history-dependent schedulers in future work.

We first define a probabilistic (memoryless) scheduler, which provides a distribution over the set of children of that NC node [20] (we will use this later in our approximate algorithms).

Let $S_{NCchildren}$ denote all the children of NC nodes, i.e. $S_{NCchildren} = \{s \in S \mid parent(s) \in S_{NC}\}$.

Definition 1. *A memoryless scheduler σ for a symbolic execution tree T is a function $\sigma : S_{NCchildren} \rightarrow [0,1]$ s.t. $\forall s \in S_{NC} : \Sigma_{s' \in child(s)} \sigma(s') = 1$.*

A scheduler for which either $\sigma(s)$ is 0 or 1 for all $s \in S_{NCchildren}$ is called *deterministic*. Similar to [20], we consider here only memoryless schedulers.

The goal is to identify the best possible deterministic scheduler, that is the one that leads to the highest probability of success. Let us first note that a (nondeterministic) program P and a deterministic scheduler σ, induce what amounts to a sequential program P_σ, with all the nondeterminism resolved according to σ. The symbolic execution tree of this program is the same as P's but with transition relation $\rightarrow - \{(n,c) \mid n \in S_{NC} \wedge \sigma(c) = 0\}$ (i.e. we remove from $\rightarrow$ all transitions (n,c) for which $\sigma(c) = 0$). For P_σ one can then compute $Pr^s(P_\sigma)$ as described in Section 3. For a nondeterministic program one can then define the maximum probability of success as:

$$Pr^s(P) = \max_\sigma Pr^s(P_\sigma)$$

where σ is deterministic. Similar definitions apply for $Pr^f(P)$ and $Pr^g(P)$. Below we describe a procedure for computing $Pr^t(P)$, where $t \in \{s, f, g\}$; the procedure forms the basis of the approximate algorithms described in the next section.

Exact Analysis The procedure is depicted in Algorithm 1: it takes as input a nondeterministic program P and a target event t. The procedure performs a bounded symbolic execution of the program (in depth-first search order). For each explored path π, it checks whether it reaches the target event, in which case it computes the count associated with the path condition ($\sharp(pc_\pi)$). This count is then propagated up along the path, to record how many inputs reach the target event. For this purpose, the procedure maintains a count s^+ for each state s. For NC nodes, s^+ is updated with the maximum value among the children, while for all the other nodes s^+ is the sum of the counters for their children.

Algorithm 1 Exact analysis.

```
1: function EXACT(Program P, target event t)
2:     Perform bounded SE of P
3:     for each π = s_0 s_1 ... s_k do
4:         if π yielded event t then
5:             s_k^+ ← ♯(pc_π)
6:             for i = k − 1, ..., 0 do
7:                 if s_i ∈ S_NC then
8:                     s_i^+ ← max_{s ∈ child(s_i)} (s^+)
9:                 else
10:                    s_i^+ ← Σ_{s ∈ child(s_i)} (s^+)
11:                end if
12:                if s_i^+ unchanged then
13:                    Break
14:                end if
15:            end for
16:        end if
17:    end for
18:    return Pr^t(P) = s_0^+ / ♯(D)
19: end function
```

Algorithm 2 Optimal scheduler.

```
1: function OPTSCHEDULER(State s)
2:     if s has no children then
3:         return
4:     end if
5:     if s is PC then
6:         for ∀ s_c ∈ child(s) do
7:             OPTSCHEDULER(s_c)
8:         end for
9:     else if s is NC then
10:        s_c^* ← arg max_{s_c ∈ child(s)} s_c^+
11:        mark(s_c^*)
12:        OPTSCHEDULER(s_c^*)
13:    end if
14: end function
```

After exploring all paths, the maximum probability for the target event ($Pr^t(P)$, shorthand for $Pr^t(s_0)$) is given by $s_0^+ / \sharp(D)$, where s_0 is the root of the symbolic tree and D is the input domain. The optimal scheduler is simply defined by selecting for each NC node the child with the maximum value of s^+. See Algorithm 2, which recursively visits the children of a state, s, and marks the nodes belonging to the optimal scheduler. In case of a tie for maximum value, we pick the first choice.

The intuition for the exact analysis is captured by the following proposition; let $Pr^t(s)$ be the maximum probability that a path crossing state s leads to the target event.

Proposition 1. *For every state s, the maximum probability of reaching the target event is $Pr^t(s) = s^+ / \sharp(pc_s)$.*

Proof. By induction on the structure of the symbolic tree. For leaves and NC nodes it is straightforward. For PC nodes s: $Pr^t(s) = p_{then} \cdot Pr^t(s_{then}) + p_{else} \cdot Pr^t(s_{else})$. From induction hypothesis this is equal to $\sharp(c \wedge pc)/\sharp(pc) \cdot s_{then}^+ / \sharp(c \wedge$

$pc)+\sharp(\neg c \wedge pc)/\sharp(pc) \cdot s^+_{else}/\sharp(\neg c \wedge pc) = (s^+_{then}+s^+_{else})/\sharp(pc) = s^+/\sharp(pc)$. □

It follows that the value returned at Line 18 of Algorithm 1: $s^+_0/\sharp(D) = s^+_0/\sharp(pc_{s_0})$ is indeed the probability of reaching the target event in the program. The procedure terminates in $k \cdot n$ steps, where k is the bound of symbolic execution and n is the number of symbolic paths.

Discussion In practice, we work with an abstraction of the symbolic execution tree, that only keeps the NC and PC nodes, and merges together all the other nodes. A node in the tree is uniquely characterized by the sequence of choices that lead to that node. We use this sequence as an efficient encoding of a state.

Note also that the grey case can also be interpreted *pessimistically* or *optimistically*, meaning that grey will be regarded as failure or success, respectively.

Finally, we mention that in practice we perform an optimized computation for the *Exact* procedure. Instead of recomputing the maximum count for the target event, s^+, for state s by performing the max operation of the counts of the children i.e. $s^+ = \max_{s_c \in child(s)}(s^+_c)$, we perform an efficient algorithm that based on the current count for the target event, c^+, and a count update, Δc^+, updates the count for state s incrementally. We presented the un-optimized version here for clarity.

5. APPROXIMATE ANALYSIS

We describe here two approximate algorithms, *Max* and *Random*, which use randomized sampling of symbolic paths to compute approximate solutions to the scheduler synthesis problem. Random uses a uniform distribution for the nondeterministic choices while Max uses Reinforcement Learning to iteratively improve resolutions of nondeterminism. We start with a statement of the verification queries that can be answered with our algorithms. We build upon and use the terminology from [20].

Verification Query Instead of computing the maximum probability of reaching t ($t \in \{s, f, g\}$), as in Exact, we pose the following query: given t and a *hypothesis* $\theta \in [0, 1]$, we try to decide whether $\exists \sigma : Pr^t(P)$ *op* θ, where $op \in \{>, \geq\}$. Such queries can be used both for verification and scheduler synthesis. For example, for verification, assume we want to check that $\forall \sigma : Pr^s(P) \geq 90\%$. This can be decided by the query $\exists \sigma : Pr^f(P) > 10\%$. On the other hand, for scheduler synthesis, we check directly the existential query: $\exists \sigma : Pr^s(P) \geq 90\%$. In both cases, if the existential formulation of the query is true, a scheduler is produced. Throughout this section, we assume for simplicity that grey paths are treated pessimistically.

Approximate Algorithms Both Max and Random follow the overall algorithm depicted in Algorithm 3, with the difference that Random does not perform scheduler improvement (Line 10). At a high level, the goal of each run of the algorithm (Lines 3-11) is to compute information about the best choices with respect to the target event. The algorithm maintains a *probabilistic* scheduler σ, initialized with a uniform candidate (Line 4). Each run iterates over two procedures (*for-loop* at Line 5 with parameter L): *scheduler evaluation* (Line 6), which uses sampling to compute the counts s^+ and *scheduler improvement* (Line 10), which uses the computed information to improve on σ.

Algorithm 3 Approximate analysis.

```
1: function APPROXIMATE(T restarts, L optimizations, N samples, History parameter 0 < h < 1, Greediness parameter 0 < ϵ < 1, operator op, hypothesis θ, target event t)
2:     for i = 1, ..., T do
3:         ∀_s, s^+ ← 0
4:         ∀_{s∈S_NC} ∀_{s'∈child(s)}, σ(s') ←1/|child(s)|
5:         for i = 1, ..., L do
6:             Q ← SchEvaluation(σ,N,t)
7:             if s_0^+/♯(D) op θ then
8:                 return True
9:             end if
10:            σ ← SchImprovement(σ,h,ϵ,Q)
11:        end for
12:    end for
13:    return Probably False
14: end function
```

Algorithm 4 Scheduler evaluation.

```
1: function SCHEVALUATION(Scheduler σ, N samples, target event t)
2:     ∀s ∈ S_NCchildren: Q(s) ← σ(s)
3:     for i = 1, ..., N do
4:         Sample π = s_0 s_1 ... s_k
5:         if π yielded event t then
6:             s_k^+ ← ♯(pc_π)
7:             for i = k − 1, ..., 0 do
8:                 if s_i ∈ S_NC then
9:                     s_i^+ ← max_{s∈child(s_i)} (s^+)
10:                else
11:                    s_i^+ ← Σ_{s∈child(s_i)} (s^+)
12:                end if
13:                if s_i^+ unchanged then
14:                    Break
15:                end if
16:            end for
17:        end if
18:    end for
19:    for s ∈ S_NCchildren, s.t. s^+ was updated above do
20:        Q(s) ← s^+/♯(pc_s)
21:    end for
22:    return Q
23: end function
```

After each scheduler evaluation, we check if the verification query is true (Line 7). Note that the scheduler evaluation has the side effect that the values for each count s^+ are updated. Note also that while the probabilistic scheduler σ is used to guide the sampling (in scheduler evaluation), it does not participate in the query checking. If the query is true, the answer is returned to the user; the deterministic scheduler that confirms it is built similarly to the exact analysis. Our algorithm is a true-biased Monte Carlo algorithm [7], meaning that it is guaranteed to be correct when it confirms the hypothesis. If it can not, we restart the search (*for-loop* at Line 2 with parameter T); if it fails again, then

the confidence about the unsatisfiability of the hypothesis becomes higher.

Scheduler Evaluation Scheduler evaluation (Algorithm 4) performs N Monte Carlo samplings of symbolic paths, according to the branch probabilities for PC nodes and the probabilistic scheduler σ for the NC nodes. From each sample, it collects information s^+ for each state s, in a manner similar to the exact algorithm (Lines 5–17 are identical to Exact). In addition, s^+ is used to compute the *quality* Q for each choice (child of an NC node) that occurs due to nondeterminism (Line 20). The quality is used for reinforcement, i.e. scheduler improvement, and is ignored by Random. The quality is defined as $Q(s) = s^+/\sharp(pc_s)$ and is an estimate for the maximum probability of reaching t from state s, $Pr^t(s)$ (see Proposition 1).

In the absence of new information from sampling (i.e. if it happens to re-sample the same paths), the counts s^+ remain unchanged and consequently also the values for Q. Note also that we do not reset the counts s^+ at the beginning of each scheduler evaluation. This explains why even Random (with no learning) can be very effective in finding an optimal scheduler, because it keeps accumulating information about the counts the more it samples.

Scheduler Improvement Scheduler improvement (Algorithm 5) uses quality Q to compute how likely it is for each choice to lead to success (Line 4); it also updates σ by reinforcing the actions that are more promising (Lines 5-6). The procedure is identical to the scheduler improvement in [20] (we show it here for completeness). As in [20], we use a greediness parameter $(1-\epsilon)$ that controls the probability we assign to the most promising choice. Combining the new greedy choices with the previous scheduler, according to history parameter h, ensures that no choice is ever blocked as long as the initial scheduler does not block any actions.

Algorithm 5 Scheduler improvement [20].

1: **function** SchImprovement(Scheduler σ, History parameter $0 < h < 1$, Greediness parameter $0 < \epsilon < 1$, Quality function Q)
2: $\sigma' \leftarrow \sigma$
3: **for** $\forall s$ of type NC **do**
4: $s^* \leftarrow \arg\max_{s' \in child(s)}\{Q(s')\}$
5: $\forall_{s' \in child(s)}, p(s') \leftarrow I\{s' = s^*\}(1-\epsilon) + \epsilon(Q(s')/\Sigma_{s'' \in child(s)} Q(s''))$
6: $\forall_{s' \in child(s)}, \sigma'(s') \leftarrow h\sigma(s') + (1-h)p(s')$
7: **end for**
8: **return** σ'
9: **end function**

Correctness and Convergence As mentioned, our approximate algorithms are true-biased algorithms, for which the following result holds (See [7, p. 266]). This general result refers to true-biased p-correct algorithms (i.e. algorithms for which the probability that it outputs a correct solution is at least p) and it is only of theoretical importance, as in practice it is difficult to quantify p.

Theorem 1 (Bounding Theorem)**.** *For a true-biased p-correct Monte Carlo algorithm (with $0 < p < 1$) to achieve a correctness level of $(1-\eta)$ it is sufficient to run the algorithm a number of times: $T = log_2\eta / log_2(1-p)$. Random and Max are true-biased and p-correct.*

We also state here the correctness of our approximate algorithms, meaning that the probabilities computed with our approximate algorithms converge to the maximum one, and the deterministic schedulers converge to the optimal one, with respect to the target event.

Theorem 2 (Scheduler Improvement)**.** *Let c and c' be the counters computed in s^+ for a state s in consecutive* Scheduler Evaluation *phases, then $c/\sharp(pc_s) \leq c'/\sharp(pc_s) \leq Pr^t(s)$.*

Proof. s^+ for leaves is constant and ≥ 0. For a newly sampled path π, the counters s^+ are updated with values ≥ 0. Since both max and $\sum$ are used in computing s^+, it can only increase when considering additional positive elements, and since $\sharp(pc_s) > 0$ for each state, $c/\sharp(pc_s) \leq c'/\sharp(pc_s)$ follows. When all the execution paths have been sampled, the values of s^+ cannot be further increased and their value corresponds to $Pr^t(s)$ as a consequence of Proposition 1. □

We also need to make sure that Random and Max will not get stuck in a local optimum.

Theorem 3 (Asymptotic convergence)**.** *In the limit (for large N or L) the probability of sampling the optimal alternative converges to* 1.

Proof. Consider Max. For each NC node s, the probability $p(s')$ of taking a transition to $s' \in child(s)$ is initialized to $1/|child(s)| > 0$. When a successful path is sampled, $p(s')$ becomes $\epsilon(Q(s')/\Sigma_{s'' \in child(s)} Q(s''))$, where $Q(s') = s'^+/\sharp(pc_{s'}) \geq 1/\sharp(pc_{s'}) \geq 1/\sharp(D)$. Furthermore $\sum_{s'' \in child(s)} Q(s'') \leq |child(s)|$ (since $\forall s'' : 0 < Q(s'') \leq 1$). It follows that $p(s') \geq \epsilon/(\sharp(D) \cdot |child(s)|) > 0$ – in other words, $p(s')$ is guaranteed to be greater than a positive constant (h does not change the lower bound). Hence, by Borel's Law of Large Numbers *[31, p. 304]*, it follows that when the number of samples tends to infinity, each possible transition, including the optimal one, will almost surely be eventually sampled. For Random, $p(s') \geq 1/|child(s)| > 0$, and similar considerations apply. □

Pruning Our techniques collect the full count ($\sharp(pc_\pi)$) for each explored symbolic path (π). Therefore subsequent explorations of those paths do not yield more information and we can remove those paths from being explored again to speed up the analyses and achieve memory savings. After sampling a path, we mark the leaf in the symbolic execution tree as "explored" and then go up in the tree along the path to mark as "explored" all the nodes for which all of their children have been marked "explored". Sampling is then performed only from nodes that were not marked as "explored". When the root is marked "explored", we are guaranteed that the tree has been fully explored.

Proposition 2 (Termination of Max and Random with pruning)**.** *If pruning is applied, the optimal alternative for each nondeterministic choice will be sampled within n iterations, where n is the total number of symbolic paths.*

Comparison with [20] We provide here a brief comparison with the statistical algorithm for MDPs from [20]; let us denote it as Sum_1. Max uses as quality Q the *expected maximum probability* of reaching the target from the current state (irrespective of current σ, which is only used to drive sampling). In contrast, Sum_1 uses as quality Q the *expected*

Listing 1: "Rare" example.

```
void testMethod(int x) { // domain of x is [0..100]
 if (Verify.getBoolean()) {
  if (x < 2) {
   ...println("success"); return;
  } else {
   if (Verify.getBoolean())
   if (Verify.getBoolean())
    ... // repeat 500 times
    if (x > 5) {
     ...println("success"); return;
    } } }
 assert false;
}
```

probability of reaching the target from current the state, under the current probabilistic scheduler σ (i.e. the probabilities from σ contribute to Q). Therefore Max does not need to reset the computed s^+ with each new σ and keeps improving while Sum_1 needs to reset its estimates before each scheduler evaluation. Max and Random consider the full count of the sampled paths, instead of counting sample by sample as done in Sum_1. Furthermore, Sum_1 needs to sample many times along the same paths to obtain good quality estimates; this makes pruning inapplicable to Sum_1. Finally, Sum_1 needs a determinization step and another round of evaluation for the induced Markov Chain, which are not needed in Max and Random, because they directly estimate the maximum probabilities.

6. IMPLEMENTATION AND EXPERIMENTS

We have implemented Exact, Max and Random (with and without pruning) together with the statistical procedure from [20], denoted Sum_1, within a generic framework on top of SPF. The framework can be easily extended with other algorithms for approximate analysis; we plan to make the tool available as open-source. Notable in the tool is the implementation for Monte Carlo sampling. Each sample is performed by one symbolic execution run, as guided by a JPF listener. The listener monitors for choices made during execution. Whenever a path-condition choice is encountered, the decision of exploring the `then` or the `else` branch is determined by generating a random number, $x \in [0, 1]$, which is then compared with the computed conditional probabilities for the branches. A similar approach is taken for non-deterministic choices; for Random, the likelihood of selecting the choices is uniformly distributed whereas for Max, the probabilities are set according to the learning.

Case Studies We evaluated our implementation on the following multithreaded Java programs. **Windy:** An example from the reinforcement learning literature; a robot, affected by wind, moves in a grid with start and target positions. We use two versions: *simple* (5×4 grid) and *complex* (9×6 grid). **Daisy Chain Controller:** An example from previous work [15]: two threads run the actuation procedures for the flap controllers of an aircraft; it also includes a safety check. A wind effect hampers the operation by pushing on the flap's head or tail. **MER Arbiter:** An example derived from a flight component for the Mars Exploration Rover developed at NASA JPL; it contains an arbiter and two user threads competing for shared resources. **Parallel Quick Sort (PQS):** Three threads sort an array with six elements. It uses complex facilities from `java.util.concurrent` (e.g., `Semaphore` and `ThreadPoolExecutor`). We analyzed two versions based on the granularity with which data is bundled up and passed to the threads (*complex* and *simple*). **Airline:** Reservation system controlled by five threads with a bug based on data and thread choices. **Rare:** This is a "pathological" case for approximate analysis (see the code in Listing 1). We provide the source code at: `http://people.cs.aau.dk/~luckow/probabilistic/`. The experiments were run on a machine with an Intel Xeon E5-2670 2.60GHz and 64GB of memory.

Results Table 1 shows the results of a first set of experiments, where we compared all the techniques, for a fixed *budget* of scheduler samples. The best results are marked with bold. We have set the hypothesis θ according to the best probability obtained with Exact. We used default greediness $\epsilon = 0.5$ and history $h = 0.5$ as these were the best values suggested in [20]. We set restarts $T = 1$, we used a uniform usage profile, and grey paths were treated pessimistically. For each configuration, we conducted five trials and we picked the best result, i.e. the result with the lowest number of scheduler evaluations for verifying the hypothesis, or, if the hypothesis was not verified, the result with the probability closest to θ.

The results indicate that Sum_1 performs poorly both in terms of analysis result and performance: the former is a result of each sample not carrying the full count information as is the case for the other techniques. Performance is a consequence of the required determinization step. While exact analysis is tractable for this set of examples, the sampling-based techniques are consistently faster while still finding the optimal scheduler when the state space becomes sufficiently large (Daisy Depth 18 and PQS Simple). For the smaller examples, Random is slightly better than Max. From our results, it is difficult to conclude on good values of N and L. Analysis of the larger examples indicates that $N < L$ seem to both verify the hypothesis in less scheduler evaluations or yield a better result regardless of whether pruning is used or not. The effect of pruning is evident; it is consistently better to use pruning for Random and Max.

For Rare, the maximum probability of reaching success is easily computed with Exact but very difficult with Max and Random. This is not surprising since it is known that purely statistical methods are typically ill-suited for "rare" events [39]. Our pruning techniques partially address the problem: in worst case both Max_p and Random_p explore all program paths (but not more – Proposition 2) and in general may finish much earlier. For Rare, both Max_p and Random_p confirmed the hypothesis (close to worst case), with Max_p slightly better.

Table 2 shows the results for a second set of experiments, where we run all the techniques to determine the budget required to verify a hypothesis with fixed θ. Here we use larger examples for which the exact analysis is intractable and only show results for the best techniques, namely Max and Random with pruning. To determine the θ values we first ran experiments with approx 40K samples and $\theta = 1.0$. The best probability obtained was used as θ in the table.

Both Max and Random enable increasing the bound of symbolic execution far beyond what can be analyzed with Exact; increasing the bound naturally reveals more information about the paths. For example, for Daisy Chain Controller at depth limit 20, we can find a scheduler with a

Table 1 Exact vs. Max, Random and Sum_1; "P" denotes pruning. If θ was not verified, values in parentheses after Samples show number of scheduler evaluations to establish the best result. Percentage of experiments where hypothesis was verified is shown next to Results.

Example	Exact Analysis Result, Time, $[ms]$, # of paths	Approx. Analysis	N	L	Result	Samples	Time, $[ms]$
MER	$Pr_s = 0.5$ $Pr_f = 0.5$ 4,593 28	Random	1,000	–	0.5 (100%)	13	26,085
		Random$_P$	**1,000**	–	**0.5 (100%)**	**10**	**22,324**
		Max	10	100	0.5 (100%)	17	30,212
		Max$_P$	10	100	0.5 (100%)	13	23,684
		Sum_1	10	100	0.5 (100%)	1,300	1,440,643
		Max	100	10	0.5 (100%)	21	33,369
		Max$_P$	100	10	0.5 (100%)	13	23,921
		Sum_1	100	10	0.5 (100%)	1,300	1,428,632
Windy Simple	$Pr_s = 0.71$ $Pr_f = 1.0$ 3,807 614	Random	1,000	–	0.71 (100%)	31	10,382
		Random$_P$	**1,000**	–	**0.71 (100%)**	**14**	**6,209**
		Max	10	100	0.0 (0%)	1,000	147,040
		Max$_P$	10	100	0.71 (100%)	100	22,283
		Sum_1	10	100	0.71 (80%)	1,300	187,631
		Max	100	10	0.71 (100%)	15	6,433
		Max$_P$	100	10	0.71 (100%)	35	11,048
		Sum_1	100	10	0.71 (100%)	1,300	186,787
Daisy Depth 13	$Pr_s = 0.026860$ $Pr_f = 1.0$ 48,886 20,248	Random	1,000	–	0.023919 (0%)	1,000(103)	154,864
		Random$_P$	1,000	–	0.026860 (100%)	141	35,616
		Max	10	100	0.023919 (0%)	1,000(21)	163,387
		Max$_P$	**10**	**100**	**0.026860 (100%)**	**97**	**27,629**
		Sum_1	10	100	0.026860 (20%)	1,500	230,914
		Max	100	10	0.023919 (0%)	1,000(125)	166,545
		Max$_P$	100	10	0.026860 (100%)	143	36,621
		Sum_1	100	10	0.025684 (0%)	1,500	228,734
Daisy Depth 18	$Pr_s = 0.028625$ $Pr_f = 1.0$ 1,971,108 755,244	Random	1,000	–	0.024507 (0%)	1,000(77)	160,669
		Random$_P$	1,000	–	0.028625 (100%)	190	47,322
		Max	10	100	0.027448 (0%)	1,000(583)	165,903
		Max$_P$	**10**	**100**	**0.028625 (60%)**	**97**	**30,970**
		Sum_1	10	100	0.025978 (0%)	1,500	240,876
		Max	100	10	0.025684 (0%)	1,000(441)	166,915
		Max$_P$	100	10	0.028625 (100%)	125	36,278
		Sum_1	100	10	0.027448 (0%)	1,500	241,522
Rare	$Pr_s = 0.96$ $Pr_f = 1.0$ 4,800 504	Random	1,000	–	0.01 (0%)	1,000(22)	214,225
		Random$_P$	1,000	–	0.96 (100%)	501	117,696
		Max	10	100	0.01 (0%)	1,000(27)	153,853
		Max$_P$	10	100	0.96 (100%)	500	102,462
		Sum_1	10	100	0.01 (0%)	1,500	227,018
		Max	100	10	0.01 (0%)	1,000(47)	155,425
		Max$_P$	**100**	**10**	**0.96 (100%)**	**496**	**145,140**
		Sum_1	100	10	0.01 (0%)	1,500	224,441
PQS Simple	$Pr_s = 1.0$ $Pr_f = 0.0$ 1,360,578 391,536	Random	1,000	–	0.59179 (0%)	1,000(999)	374,351
		Random$_P$	1,000	–	0.89498 (0%)	1,000(1,000)	418,761
		Max	10	100	0.64467 (0%)	1,000(1,000)	426,958
		Max$_P$	**10**	**100**	**0.99476 (0%)**	**1,000(994)**	**410,824**
		Sum_1	10	100	0.43527 (0%)	1,500	638,972
		Max	100	10	0.60508 (0%)	1,000(1000)	451,734
		Max$_P$	100	10	0.97803 (0%)	1,000(999)	436,525
		Sum_1	100	10	0.43945 (0%)	1,500	625,670
		Random	10,000	–	0.97179 (0%)	10,000(9,878)	3,445,703
		Random$_P$	10,000	–	1.0 (100%)	1,421	620,681
		Max	10	1,000	0.98888 (0%)	10,000(9,755)	3,505,138
		Max$_P$	**10**	**1,000**	**1.0 (100%)**	**989**	**417,425**
		Sum_1	10	1,000	0.42793 (0%)	10,500	3,331,941
		Max	1,000	10	0.98181 (0%)	10,000(9,922)	3,020,768
		Max$_P$	1,000	10	1.0 (100%)	1,331	533,125
		Sum_1	1,000	10	0.45450 (0%)	10,500	3,098,937

Table 2 Random vs. Max (w/ pruning); Exact runs out of memory.

Example	Hypothesis	Approx. Analysis	Samples	Time $[ms]$
Windy Complex	$Pr^s(P) \geq$ 0.71	Max_P	214	51,711
		$\mathbf{Random}_P$	**23**	**8,688**
Daisy Depth 20	$Pr^s(P) \geq$ 0.028723	$\mathbf{Max}_P$	**136**	**38,261**
		Random_P	224	56,948
Daisy Depth 30	$Pr^s(P) \geq$ 0.029409	$\mathbf{Max}_P$	**129**	**43,347**
		Random_P	349	84,694
PQS Complex	$Pr^s(P) \geq$ 1.0	$\mathbf{Max}_P$	**3,675**	**1,273,642**
		Random_P	12,047	3,914,474
Airline	$Pr^s(P) \geq$ 1.0	$\mathbf{Max}_P$	**169**	**40,851**
		Random_P	1,843	287,456

better probability for success than what the exact analysis found at depth limit 18 as shown in Table 1. These results furthermore demonstrate the benefits of reinforcement learning as compared to Random when the state space is large (see all cases except Windy Complex).

Our approximate algorithms are well suited for exploring systems with large state spaces but that are well structured, i.e. they may have multiple components running the same or similar algorithms and have few interactions points between components (e.g., MER, Daisy, Windy). Common examples include planning and scheduling for robots, control software for aircrafts and many critical applications of interest. However, for unstructured systems (e.g., Rare) the approximate algorithms require scheduling decisions to be made on *all* states, thus defeating their purpose. This is confirmed by the related literature [39, 20] and more research is needed to address the issue.

Non-Uniform Usage Profiles To see how our approach applies for non-uniform usage profiles, let us revisit the Daisy Chain Controller and consider two different scenarios where the wind effect is weak (UP_w) and strong (UP_s), respectively (see Figure 3). In particular, the weak and strong wind usage profiles are defined as the case where respectively 5% and 15% of the input values yield $wind > 10$. We would expect that under the conditions of UP_w, the flap controller is more likely to operate successfully because the flap is less likely to exceed the goal position. We use a symbolic variable, *up*, constrained such that $1 \leq up \leq 100$, for controlling the distribution of the input values for the *wind* variable.

Listing 2: UP_w

```
if(up<=5){
 assume(wind<-10);
}else if(up<=15){
 assume(wind>=-10 &&
        wind<=-5);
}else if(up<=85){
 assume(wind>-5 &&
        wind<5);
}else if(up<=95){
 assume(wind>=5 &&
        wind<=10);
}else{
 assume(wind>10);
}
// rest of the code
```

Listing 3: UP_s

```
if(up<=15){
 assume(wind<-10);
}else if(up<=35){
 assume(wind>=-10 &&
        wind<=-5);
}else if(up<=65){
 assume(wind>-5 &&
        wind<5);
}else if(up<=85){
 assume(wind>=5 &&
        wind<=10);
}else{
 assume(wind>10);
}
// rest of the code
```

Listing 2 and Listing 3 show how UP_w and UP_s are encoded as preconditions, i.e. *assume* statements in the code. The assume statements are implemented using the built-in `Debug.assume()` method from SPF. With the usage profiles, we ran Exact and obtained $Pr^s_{UP_w} = 0.048387$ which is indeed better than $Pr^s_{UP_s} = 0.024037$.

7. RELATED WORK

In previous work [15] we defined the probabilistic symbolic analysis for Java programs that forms the basis of our work here. However in [15] we only discuss exact algorithms and multithreading is treated by computing probabilities along linear schedules. We study here more general tree-like schedules. In recent work we have investigated approximate procedures for the probabilistic analysis of floating-point programs [6] and for the scalable probabilistic symbolic execution of sequential code [16]. However none of these approaches address sampling for nondeterministic programs and hence the challenge of computing (near)optimal schedulers.

Other related work includes probabilistic abstract interpretation [28, 12], probabilistic static analysis [1, 10, 9] and probabilistic model checking [3, 17, 2]. In particular the program analysis from [9] is relevant here as it performs aggressive pruning, but not for symbolic execution. Again none of these works address sampling in the presence of nondeterminism which is one of the main contributions here.

Statistical verification techniques [27, 25, 36, 13] perform sampling over the analyzed state spaces, but aside from the work in [20], there are very few other approaches that study nondeterminism, e.g., [26, 5]. In [26] random sampling is exploited to search for a near-optimal scheduler, whose quality is again evaluated by a statistical approach. However, the iterated use of the conservative Chernoff-Hoeffding bound [21] to determine the necessary number of samples might require an impractically large number of them. The work in [4] studies partial order reductions for MDPs to reduce nondeterminism, and thus it is orthogonal to ours. Our work is also generally related to planning for MDPs [23, 34, 37]. In the future we plan to investigate whether our techniques are applicable to planning as well.

8. CONCLUSIONS AND FUTURE WORK

We presented exact and approximate symbolic execution techniques for the probabilistic analysis of nondeterministic programs. We implemented and evaluated them showing improvement over established techniques.

In the future we plan to investigate replacing the exact model counting with approximate quantification (e.g., QCoral [6] for floating-point constraints). We would need to revise our theoretical results but we note that one of our main results (Proposition 2) would (trivially) hold in that case too. We further plan to study schedulers that use more information from the program execution (history and/or current path condition) to compute more accurate information about the maximum probability.

The sampling process is highly parallelizable. We implemented a parallel prototype and results show improvement in performance, even though some overhead due to thread contention is inevitable; e.g., distributing the workload to two clients for Example 1, reduces the analysis runtime by 30%. More experimentation is planned for the future.

9. REFERENCES

[1] A. Adje, O. Bouissou, J. Goubault-Larrecq, E. Goubault, and S. Putot. Static analysis of programs with imprecise probabilistic inputs. In E. Cohen and A. Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments*, volume 8164 of *Lecture Notes in Computer Science*, pages 22–47. 2014.

[2] C. Baier, J.-P. Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[3] A. Bianco and L. Alfaro. Model checking of probabilistic and nondeterministic systems. In P. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *Lecture Notes in Computer Science*, pages 499–513. 1995.

[4] J. Bogdoll, L. Ferrer Fioriti, A. Hartmanns, and H. Hermanns. Partial order methods for statistical model checking and simulation. In R. Bruni and J. Dingel, editors, *Formal Techniques for Distributed Systems*, volume 6722 of *Lecture Notes in Computer Science*, pages 59–74. 2011.

[5] J. Bogdoll, A. Hartmanns, and H. Hermanns. Simulation and statistical model checking for modestly nondeterministic models. In J. Schmitt, editor, *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*, volume 7201 of *Lecture Notes in Computer Science*, pages 249–252. 2012.

[6] M. Borges, A. Filieri, M. D'Amorim, C. S. Păsăreanu, and W. Visser. Compositional solution space quantification for probabilistic software analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 123–132, New York, NY, USA, 2014. ACM.

[7] G. Brassard and P. Bratley. *Algorithmics: theory and practice*. Prentice Hall, 1988.

[8] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008.

[9] A. T. Chaganty, A. V. Nori, and S. K. Rajamani. Efficiently sampling probabilistic programs via program analysis. In *AISTATS'13: Artificial Intelligence and Statistics*, 2013.

[10] G. Claret, S. K. Rajamani, A. V. Nori, A. D. Gordon, and J. Borgström. Bayesian inference using data flow analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 92–102, New York, NY, USA, 2013. ACM.

[11] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, Sept 1976.

[12] P. Cousot and M. Monerau. Probabilistic abstract interpretation. In H. Seidl, editor, *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*, pages 169–193. Springer, 2012.

[13] A. David, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, J. Van Vliet, and Z. Wang. Statistical model checking for networks of priced timed automata. In *Proceedings of the 9th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS'11, pages 80–96. Springer-Verlag, 2011.

[14] J. A. De Loera, B. Dutra, M. Köppe, S. Moreinis, G. Pinto, and J. Wu. Software for exact integration of polynomials over polyhedra. *Comput. Geom. Theory Appl.*, 46(3):232–252, Apr. 2013.

[15] A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 622–631. IEEE Press, 2013.

[16] A. Filieri, C. S. Păsăreanu, W. Visser, and J. Geldenhuys. Statistical symbolic execution with informed sampling. FSE, 2014. To appear.

[17] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker. Automated verification techniques for probabilistic systems. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 53–113. 2011.

[18] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *ISSTA*, pages 166–176, 2012.

[19] M. Gittens, H. Lutfiyya, and M. Bauer. An extended operational profile model. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 314–325, Nov 2004.

[20] D. Henriques, J. Martins, P. Zuliani, A. Platzer, and E. M. Clarke. Statistical model checking for markov decision processes. In *QEST*, pages 84–93, 2012.

[21] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[22] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385—394, July 1976.

[23] A. Kolobov. *Planning with Markov Decision Processes: An AI Perspective*. Synthesis digital library of engineering and computer science. Morgan & Claypool, 2012.

[24] M. Kwiatkowska, G. Norman, and D. Parker. A framework for verification of software with time and probabilities. In *Formal Modeling and Analysis of Timed Systems*, pages 25–45. Springer, 2010.

[25] K. G. Larsen. Statistical model checking, refinement checking, optimization, ... for stochastic hybrid systems. In *Formal Modeling and Analysis of Timed Systems*, volume 7595 of *Lecture Notes in Computer Science*, pages 7–10. 2012.

[26] R. Lassaigne and S. Peyronnet. Approximate planning and verification for large markov decision processes. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1314–1319. ACM, 2012.

[27] A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: An overview. In H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, and N. Tillmann, editors, *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 122–135. 2010.

[28] D. Monniaux. Abstract interpretation of probabilistic semantics. In J. Palsberg, editor, *Static Analysis*, volume 1824 of *Lecture Notes in Computer Science*, pages 322–339. 2000.

[29] J. Musa. The operational profile. In S. Özekici, editor, *Reliability and Maintenance of Complex Systems*, volume 154 of *NATO ASI Series*, pages 333–344. 1996.

[30] M. Nagappan, K. Wu, and M. A. Vouk. Efficiently extracting operational profiles from execution logs using suffix arrays. In *Proceedings of the 2009 20th International Symposium on Software Reliability Engineering*, ISSRE '09, pages 41–50. IEEE Computer Society, 2009.

[31] M. Neuts. *Probability*. Allyn and Bacon, 1973.

[32] Q.-S. Phan, P. Malacaria, C. S. Păsăreanu, and M. D'Amorim. Quantifying information leaks using reliability analysis. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, pages 105–108, New York, NY, USA, 2014. ACM.

[33] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[34] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Prentice Hall, 2010.

[35] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *PLDI*, pages 447–458, 2013.

[36] K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In R. Alur and D. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 202–215. 2004.

[37] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. A Bradford book. Bradford Book, 1998.

[38] D. White. *Markov Decision Processes*. Wiley, 1993.

[39] P. Zuliani, C. Baier, and E. M. Clarke. Rare-event verification for stochastic hybrid systems. In *HSCC*, pages 217–226, Apr 2012.

[40] P. Zuliani, A. Platzer, and E. M. Clarke. Bayesian statistical model checking with application to simulink/stateflow verification. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '10, pages 243–252. ACM, 2010.