Mining Statistically Likely k-Reachable States in Probabilistic Programs*

Arnab Ray[1], **Nitesh Trivedi**[2], **Ansuman Banerjee**[1], **Sourav Chakraborty**[1], **Arijit Ghosh**[1] and **Subhajit Roy**[2]

[1]Indian Statistical Institute

[2]Indian Institute of Technology Kanpur

rayarnab015@gmail.com, {ansuman, sourav, arijitghosh}@isical.ac.in, {subhajit, nitesht}@iitk.ac.in

Abstract

We propose the notion of *statistically likely k-step reachable set* in probabilistic programs, a statistically robust notion for high-probability k-step reachable program states. We design an inductive algorithm to capture this set as a symbolic representation in propositional logic for Boolean probabilistic programs. Our methodology iteratively learns a symbolic formula for *statistically likely k-step reachable set* that involves (a) learning an initial symbolic candidate via decision tree learning, (b) collecting positive and negative counterexamples via forward and backward verification checks and (c) refining the current candidate via a sequence of *prune* and *split* moves on the decision tree. We demonstrate that *statistically likely k-step reachable set* can reveal some interesting properties about programs by studying some probabilistic programs from literature.

1 Introduction

Symbolic analysis has significantly advanced the state-of-the-art for program verification. Such analyzers compute a symbolic representation of a target program, commonly in some fragment of logic, and use satisfiability engines (SAT solvers) to search for proofs and counterexamples. However, verification of real-world programs remains challenging. The primary difficulty is the challenge in modeling the uncertainties presented by the environment within which a program is executing. The environment can, though sparingly, provide unexpected values that deviate the program from its "regular" set of reachable states. For example, a memory allocation using a `malloc()` may infrequently fail (return NULL), that may require executing an exception handling routine.

A possible strategy is to model the environment as a stochastic process, where the probability distribution over different kinds of failures can be based on recommendation of domain experts or estimated from empirical studies. This allows us to ignore rare events and understand the *common-case* behaviors of the program. We refer to such a set of reachable states as *statistically likely k-step reachable set*.

*For full version of the paper and the artifact, see https://github.com/arnab-007/PRORP_

In this work, we attempt to synthesize the *statistically likely k-step reachable set* for such probabilistic programs. Such reachable sets have applications in program understanding as the computed reachable sets do not include corner-case exception states. Further, along the lines of prior work like angelic verification [Das *et al.*, 2015; Das and Lal, 2017; Joshi *et al.*, 2012; Lahiri and Roy, 2022], the attempt to identify potential false positives that are signaled due to unduly demonic assumptions about the environment, *statistically likely k-step reachable set* can also have applications in sieving through false positives for real bugs. Our approach uses a combination of formal and learning based approaches to tackle this problem. Symbolically characterizing the common-case behaviors of probabilistic programs is a core problem in AI, with applications spanning Bayesian inference, AI safety, and bug prioritization. We address this by learning a *symbolic posterior summary* that distinguishes high-probability program behaviors from rare corner cases.

Contributions

The contributions of this work are as follows:

- We propose the notion of *statistically likely k-step reachable set* ϵ-$\mathtt{Cl}_k(\varphi)$, a symbolic characterization of states that probabilistic programs reach with non-negligible probability;
- We prove that even the problem of deciding the emptiness of ϵ-$\mathtt{Cl}_k(\varphi)$ is itself computationally hard, which motivates approximate solutions;
- We design an algorithm to compute this set with statistical guarantees, which to our knowledge is the first framework to learn symbolic approximation of high-probability reachable sets;
- We build our algorithm into a tool, PRORP, and use it to study the *statistically likely k-step reachable set* on probabilistic programs from literature.

2 Preliminaries

2.1 Boolean Formulas, Satisfiability and Samplers

For a Boolean formula φ, we use $Supp(\varphi)$ to denote the set of variables appearing in φ. An assignment $\sigma \in \{0,1\}^{|Supp(\varphi)|}$ to $Supp(\varphi)$ is a *satisfying assignment* or *witness* if it makes φ

evaluate to 1. We denote the set of all satisfying assignments of φ as R_φ and say a formula φ represents the set R_φ.

A formula is in **conjunctive normal form (CNF)** (resp. **disjunctive normal form (DNF)**) if it is a conjunction (resp. disjunction) of one or more clauses (resp. terms), where each clause (resp. term) is a disjunction (resp. conjunction) of literals. A **literal** is a variable or its negation.

Definition 1 (Uniform samplers)**.** *A uniform-**CNF**-sampler (resp. uniform-**DNF**-sampler) is a randomized algorithm $\mathcal{A}$ that takes as input a **CNF**-formula (resp. **DNF**-formula) φ and outputs a satisfying assignment of φ uniformly at random from the set of all satisfying assignments. That is, for any $\sigma \in R_\varphi$, $\Pr[\mathcal{A}$ outputs $\sigma] = \frac{1}{|R_\varphi|}$, where the probability is over the internal randomness used in $\mathcal{A}$.*

For a CNF or DNF formula φ, by drawing m iid samples from $\mathcal{D}(R_\varphi)$ we mean using the relevant samplers to draw m independent and identically distributed samples from the distribution $\mathcal{D}$ over the set R_φ.

2.2 Probabilistic Programs and k-Step Reachability

Definition 2 (Probabilistic programs)**.** *A* probabilistic program *is a tuple $(\mathcal{X}, \texttt{loc}, \texttt{trans})$ where (i) $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a finite set of Boolean program variables, (ii) $\texttt{loc}$ is a finite set of control locations and (iii) $\texttt{trans}$ is a finite set of transitions, each of the form $(\ell, \texttt{guard}) \Rightarrow \{(p_1, u_1, \ell'_1), \ldots, (p_m, u_m, \ell'_m)\}$, where $\ell, \ell'_i \in \texttt{loc}$, $\sum_i p_i = 1$, each u_i is a deterministic update over $\mathcal{X}$, and when the Boolean predicate $\texttt{guard}$ holds at ℓ, the program moves to ℓ'_i applying u_i with probability p_i.*

Programs defined over variables taking values in a bounded integer domain can be encoded as a finite propositional formula via bit-blasting, a technique formalized in [Biere *et al.*, 1999]. The encoding extends naturally to probabilistic programs over finite domains [Kwiatkowska *et al.*, 2011].

Definition 3 (Program states)**.** *Given a probabilistic program P defined over a set of Boolean program variables $\mathcal{X}$, a **program state** σ is a mapping from $\mathcal{X}$ to $\{0,1\}^{|\mathcal{X}|}$, i.e., it is a valuation over the program variables. Thus, the entire program state space can be represented as $\Omega_P := \{0,1\}^{|\mathcal{X}|}$.*

Example: Consider a boolean probabilistic program P defined over a non-negative integer variable $\texttt{z}$ and a Boolean flag $\texttt{flip}$. For 3-bit encoding of $\texttt{z} = z_2 z_1 z_0$ with $\mathcal{X} = \{z_2, z_1, z_0, \texttt{flip}\}$, the state space is given by $\Omega_P = \{0,1\}^4$. An example of a program state is $\{z_2 = z_0 = 1, z_1 = 0, \texttt{flip} = 0\}$ which corresponds to $(\texttt{z} = 5, \texttt{flip} = 0)$.

Definition 4 (k-step transition and terminal states)**.** *For a probabilistic program P defined over program variables $\mathcal{X}$ with Ω_P being the entire state space and for any $k \in \mathbb{N}$, the **k-step program transition probability function** is a function $\Gamma_P^k : \Omega_P \times \Omega_P \to [0,1]$ such that $\Gamma_P^k(\sigma, \sigma')$ gives the probability of reaching state σ' from the state σ in exactly k steps. So clearly, for all $\sigma \in \Omega_P$, $\sum_{\sigma'} \Gamma_P^k(\sigma, \sigma') = 1$.*

*Let P be a probabilistic program over program variables $\mathcal{X}$, and let Ω_P denote its state space. A state $\sigma \in \Omega_P$ is called a **terminal state** or sink state of P if no further program transitions are possible from σ, that is,*

$$\forall \sigma' \in \Omega_P, \quad \Gamma_P^1(\sigma, \sigma') = \begin{cases} 1 & \text{if } \sigma' = \sigma, \\ 0 & \text{otherwise.} \end{cases}$$

We denote the set of all terminal states of P by $\mathcal{T}_P \subseteq \Omega_P$. Specifically, terminal states correspond to program configurations in which the program execution has halted.

Given $\sigma' \in \mathcal{T}_P$, $\Gamma_P^{\leq k}(\sigma, \sigma')$ denotes the probability of reaching σ' starting from σ in at most k steps, that is, $\Gamma_P^{\leq k}(\sigma, \sigma') := \Gamma_P^i(\sigma, \sigma')$, where $i = \min\{j \in [1,k] \mid \Gamma_P^j(\sigma, \sigma') > 0\}$, otherwise 0.

*The **k-step program formula** F_P^k is a Boolean formula over $2|\mathcal{X}|$ number of variables representing a k-step single execution path of the program. In other words, or any $\sigma, \sigma' \in \Omega_P$, $F_P^k(\sigma, \sigma') = 1$ if $\Gamma_P^k(\sigma, \sigma') > 0$*

Definition 5 (Precondition)**.** *For a probabilistic program P defined over program variables $\mathcal{X}$, with Ω_P being the entire state space, a **precondition set** is a subset R_φ of Ω_P describing a possible set of valuations at the beginning of P. The subset is described implicitly by the Boolean formula φ, known as the **precondition**.*

2.3 Compiling Decision Trees to Boolean Formula

A *decision tree* for a Boolean function $f : \{0,1\}^n \to \{0,1\}$ is a rooted binary tree whose internal nodes are labeled by variables $x_i \in \{x_1, \ldots, x_n\}$, whose outgoing edges are labeled by the Boolean outcomes 0 and 1, and whose leaves are labeled by output values of f. Each root–to–leaf path encodes a conjunction of literals specifying an input $\mathbf{x} \in \{0,1\}^n$, and the label of the leaf is the value $f(\mathbf{x})$. Equivalently, a decision tree represents the function f as the disjunction of the conjunctions of the decisions along all root-to-leaf paths labeled 1 at the leaves.

For example, the tree in Figure 1c corresponds to the DNF $(\texttt{z}_1 \vee (\neg \texttt{z}_1 \wedge \texttt{flip}))$. For a given decision tree $\mathfrak{T}$, we assume that $\texttt{ExtractLeaves}(\mathfrak{T})$ returns the list of all leaves in $\mathfrak{T}$, and $\texttt{TreeDNF}(\mathfrak{T})$ outputs the DNF represented by $\mathfrak{T}$.

3 Statistically Likely k-Step Reachable Set

To formalize the notion of statistically likely k-step reachable set, we first define the concept of k-step reachability.

Definition 6 ((At most k)-step reachability)**.** *For a precondition φ of a probabilistic program P, a **k-step reachable formula with respect to precondition** φ is a Boolean formula ζ that satisfies the following condition: $\sigma' \in R_\zeta \iff \exists \sigma \in R_\varphi$ with $\left(\bigvee_{i=0}^k F_P^i(\sigma, \sigma')\right) = 1$. Thus, R_ζ is the set of all terminal states that reachable in atmost k steps if the program P is executed with a starting state in R_φ. We call this set the **k-step reachable set** and denote it by $\texttt{Cl}_k(\varphi)$.*

Definition 7 (Atmost k-step transition probability)**.** *For a precondition φ consider the k-step reachable set $\texttt{Cl}_k(\varphi)$. Then for any $\sigma' \in \texttt{Cl}_k(\varphi)$ and any given probability distribution $\mathcal{D}$ over the set R_φ, we define the **at most k-step program transition function w.r.t. to precondition** φ as $\Gamma_{P,\varphi}(\sigma') := \mathbb{E}_{\sigma \sim \mathcal{D}(R_\varphi)}\left[\Gamma_P^{\leq k}(\sigma, \sigma')\right]$. This quantity gives the probability of*

reaching the terminal state σ' in atmost k steps from an initial state $\sigma \in R_\varphi$ chosen according to $\mathcal{D}$.

However, $\texttt{Cl}_k(\varphi)$ is the set of all reachable states when P starts from R_φ. To only capture the states reached with "non-negligible" probability, we introduce the concept of *statistically likely k-step reachable set*.

Definition 8 (Statistically likely k-step reachable set)**.** *For a probabilistic program P, a precondition φ and a threshold $\epsilon \in (0,1]$, we consider the set ϵ-$\texttt{Cl}_k(\varphi) \subseteq \texttt{Cl}_k(\varphi)$, which we call the ϵ-statistically likely k-step reachable set for P* ***w.r.t. the precondition*** *φ and is defined as ϵ-$\texttt{Cl}_k(\varphi) := \texttt{Cl}_k(\varphi) \cap \{\sigma' \mid \Gamma_{P,\varphi}(\sigma') \geq \epsilon\}$.*

3.1 Exact Synthesis of ϵ-$\texttt{Cl}_k(\varphi)$

Given a probabilistic Boolean program P with input variables $\mathcal{X}$, a distribution $\mathcal{D}$ from which $\mathcal{X}$ is sampled, precondition φ, $k \in \mathbb{N}$ representing the number of steps, and the parameter $\epsilon \in (0,1]$, we refer to the task of *exactly* synthesizing the set ϵ-$\texttt{Cl}_k(\varphi)$ as ϵ-$\texttt{Cl}_k(\varphi)$ **SYNTHESIS**. We analyze the ϵ-$\texttt{Cl}_k(\varphi)$ **EMPTINESS** problem (asking whether ϵ-$\texttt{Cl}_k(\varphi)$ is empty), whose hardness lower-bounds that of ϵ-$\texttt{Cl}_k(\varphi)$ SYNTHESIS. We prove PSPACE-hardness of ϵ-$\texttt{Cl}_k(\varphi)$ EMPTINESS by a Karp reduction from the known PSPACE-complete problem **DFA-INTERSECTION** [Kozen, 1977]. We state the theorem below.

Theorem 1. *ϵ-$\texttt{Cl}_k(\varphi)$* ***EMPTINESS*** *is* PSPACE*-complete.*

This result motivates approximate synthesis of ϵ-$\texttt{Cl}_k(\varphi)$. We provide the formal proof for PSPACE-completeness of the ϵ-$\texttt{Cl}_k(\varphi)$ **EMPTINESS** problem in the full version [Ray *et al.*, 2026] of this paper. We now formalize the problem of approximate synthesis of ϵ-$\texttt{Cl}_k(\varphi)$.

3.2 Approximate Synthesis of ϵ-$\texttt{Cl}_k(\varphi)$

For a probabilistic program P and a precondition φ, a set $S \subseteq \texttt{Cl}_k(\varphi)$ is called a ***ϵ-approximate k-step reachable set*** if $\sum_{\sigma \in \texttt{Cl}_k(\varphi) \setminus S} \Gamma_{P,\varphi}(\sigma) \leq \epsilon$. A formula ψ representing an *ϵ-approximate k-step reachable set* R_ψ is called an ***ϵ-approximate k-step reachable formula***. Note that, if S is any ϵ-approximate k-step reachable set, ϵ-$\texttt{Cl}_k(\varphi) \subseteq S$. Trivially, the entire state space is an ϵ-approximate k-step reachable set. So our goal is to learn an ϵ-approximate k-step reachable formula ψ whose set R_ψ is *close* to ϵ-$\texttt{Cl}_k(\varphi)$. To quantify closeness, we define two error measures between ψ and $\texttt{Cl}_k(\varphi)$.

Definition 9 (Error distances for candidate formula)**.** *Given a probabilistic program P, a precondition φ and a distribution $\mathcal{D}$ over R_φ, the* Fo*-errors and* Ba*-errors between a candidate Boolean formula ψ and the k-step reachable set $\texttt{Cl}_k(\varphi)$ are quantified as follows*

- *The* Fo*-error distance $\texttt{Error}_{\texttt{Fo}}(\psi, \texttt{Cl}_k(\varphi))$ is the probability that the terminal state reached after execution of atmost k steps of P on an initial state randomly picked from R_φ according to $\mathcal{D}$ is not satisfied by ψ. It is defined as $\texttt{Error}_{\texttt{Fo}}(\psi, \texttt{Cl}_k(\varphi)) := \sum_{\sigma \in (\texttt{Cl}_k(\varphi) \setminus R_\psi)} \Gamma_{P,\varphi}(\sigma)$.*
- *The* Ba*-error distance $\texttt{Error}_{\texttt{Ba}}(\psi, \texttt{Cl}_k(\varphi))$ is a measure of the fraction of program states in R_ψ that are not satisfied by $\texttt{Cl}_k(\varphi)$. It is defined as $\texttt{Error}_{\texttt{Ba}}(\psi, \texttt{Cl}_k(\varphi)) := \Pr_{\sigma \sim_U R_\psi}[\sigma \notin \texttt{Cl}_k(\varphi)]$*

φ	R_φ	ψ
z_{14}	$\bigcup_{j=0}^{1}[(2j+1)2^{14}, (j+1)2^{15}-1]$	z_{14}
z_{12}	$\bigcup_{j=0}^{7}[(2j+1)2^{12}, (j+1)2^{13}-1]$	z_{12}
z_{10}	$\bigcup_{j=0}^{31}[(2j+1)2^{10}, (j+1)2^{11}-1]$	z_{10}
z_8	$\bigcup_{j=0}^{127}[(2j+1)2^{8}, (j+1)2^{9}-1]$	$z_8 \vee (\bar{z}_7 \wedge \bar{z}_6 \wedge \bar{z}_8)$
z_6	$\bigcup_{j=0}^{511}[(2j+1)2^{6}, (j+1)2^{7}-1]$	True

Table 1: Output of PRORP on Geo0 (ϵ_1=ϵ_2=ϵ=0.05)

Definition 10 ((ϵ_1, ϵ_2)-approximate statistically likely k-step reachability)**.** *For any probabilistic program P, precondition φ, input distribution $\mathcal{D}$ over R_φ, $k \in \mathbb{N}$ and any $0 \leq \epsilon_1, \epsilon_2 \leq 1$, a formula ψ is called a (ϵ_1, ϵ_2)-* ***approximate statistically likely k-step reachable formula*** *of P with respect to φ if $\texttt{Error}_{\texttt{Fo}}(\psi, \texttt{Cl}_k(\varphi)) < \epsilon_1$ and $\texttt{Error}_{\texttt{Ba}}(\psi, \texttt{Cl}_k(\varphi)) < \epsilon_2$.*

Given P, precondition φ, $0 \leq \epsilon_1, \epsilon_2, \delta \leq 1$, and $k \in \mathbb{N}$ an algorithm is said to synthesize an $(\epsilon_1, \epsilon_2, \delta)$- ***approximate statistically likely k-step reachable set*** *of P with respect to φ if it outputs a Boolean formula ψ such that with probability at least $(1-\delta)$, R_ψ is a (ϵ_1, ϵ_2)-approximate statistically likely k-step reachable formula of P with respect to φ.*

Let us define the problem of *approximating* ϵ-$\texttt{Cl}_k(\varphi)$.

Problem statement: Given a probabilistic Boolean program P with input variables $\mathcal{X}$, precondition φ, input distribution $\mathcal{D}$ over R_φ, $k \in \mathbb{N}$ representing the number of steps, and error-bounds $\epsilon_1, \epsilon_2, \delta \in (0,1]$, we want to synthesize an $(\epsilon_1, \epsilon_2, \delta)$-approximate *statistically likely k-step reachable set* of the probabilistic program P with respect to φ.

Clearly, for the given problem setup, the set ϵ_1-$\texttt{Cl}_k(\varphi) \subseteq S$, where S is some $(\epsilon_1, \epsilon_2, \delta)$-approximate *statistically likely k-step reachable set* of P with respect to φ. To solve the above problem, we introduce an algorithm PRORP. We now study the output of PRORP using an illustrative example.

Illustrative Example. We illustrate the notion of approximate *statistically likely k-step reachable set* using the probabilistic program Geo0 (Figure 1a). Starting from $z = 0$, the program returns geometrically distributed values, so small outputs are exponentially more likely than large ones, creating a structural asymmetryhigh-order bits rarely change whereas low-order bits vary often. For a precondition ϕ, let R_ϕ denote the initial states, and let ϵ-$\texttt{Cl}_k(\varphi)$ be the states reachable within k iterations with probability at least ϵ. For Geo0, while $\texttt{Cl}_k(\varphi)$ contains many low-probability states, ϵ-$\texttt{Cl}_k(\varphi)$ remains small and captures the common-case behavior. The results (z being a 16-bit unsigned integer) in Table 1 show that PRORP closely tracks ϵ-$\texttt{Cl}_k(\varphi)$. For preconditions involving higher-order bits (z_{14}, z_{12}, z_{10}), the learned candidate exactly characterizes the likely reachable states. As the precondition moves to lower-order bits, chances of leaving the initial region increases, so the formulas admit more states. In the extreme case z_6, almost the entire state space becomes likely reachable, and PRORP returns True. These observations highlight the common-case behaviour of Geo0.

4 Algorithm

We use an iterative refinement strategy which "guesses" a potential candidate, and then, our verification subroutines validate the solution. Any counterexample augment the dataset and the process is repeated till a candidate is verified.

4.1 The Verification Subroutines

A candidate formula ψ for the $(\epsilon_1, \epsilon_2, \delta)$-*statistically likely k-step reachable set* problem can have errors in two directions:

- The formula ψ excludes program states reachable from the inputs; we call such examples Fo-*counterexamples.*
- The formula ψ includes program states not reachable from the inputs; such examples are Ba-*counterexamples.*

We design two verification subroutines, FoRT (Forward-Reachability-Tester) to test for Fo-errors and BaRT (Backward-Reachability-Tester) to test for Ba-errors. Our algorithm alternates between these two subroutines to obtain the two types of counterexamples.

FoRT

This subroutine (Algorithm 1) takes as input a probabilistic program P, a precondition φ (in CNF), an input distribution $\mathcal{D}$ over R_φ, a candidate formula ψ (in DNF), a closeness parameter ϵ, a confidence parameter δ and the number of steps k. It returns an estimate $\texttt{Error}_1$ of the Fo-error and a multiset $\texttt{CE}_{\texttt{Fo}}$ of Fo-counterexamples. To capture "high-probability" counterexamples, FoRT samples a set U of initial states from $\mathcal{D}(R_\varphi)$ (Line 2), executes P for at most k steps (represented by $P^{\leq k}(u)$ for $u \in U$ in Line 5), and records the reached terminal states $y_1, \ldots, y_m$ (possibly with repetitions). Each violating y_i is added to $\texttt{CE}_{\texttt{Fo}}$ as a Fo-counterexample (Line 7); $\texttt{Error}_1$ is the fraction of samples failing ψ (Line 8).

Sample size m is chosen so that, with probability at least $(1-\delta)$, $\texttt{Error}_1$ is a ϵ-additive approximation of the true Fo-error. FoRT only requires a black-box sampling access to $\mathcal{D}$.

BaRT

This subroutine (Algorithm 2) takes as input a probabilistic program P, a precondition φ (in CNF), a candidate formula ψ (in DNF), a closeness parameter ϵ, a confidence parameter δ and the number of steps k. It returns an estimate $\texttt{Error}_2$ of the Ba-error and a multiset $\texttt{CE}_{\texttt{Ba}}$ of Ba-counterexamples. Recall that Ba-counterexamples are states in $(R_\psi \setminus \texttt{Cl}_k(\varphi))$, and true Ba-error is $\frac{|R_\psi \setminus \texttt{Cl}_k(\varphi)|}{|R_\psi|}$. To generate Ba-counterexamples, BaRT samples m states from R_ψ using a uniform-like DNF-sampler (Line 2) and checks if the samples are in $\texttt{Cl}_k(\varphi)$ via a SAT-solver (Line 6). The formula Υ_i (in line 5) encodes "is the sampled state v in $\texttt{Cl}_k(\varphi)$." If v is not in $\texttt{Cl}_k(\varphi)$ then it is a Ba-counterexample and is added to the multi-set $\texttt{CE}_{\texttt{Ba}}$ (in line 7) and the $\texttt{Error}_2$ (updated in line 8) records what fraction of the samples that are drawn from R_ψ are not is $\texttt{Cl}_k(\varphi)$. Since our objective is to track the unreachable states in R_ψ, we give equal importance to these states by sampling from the uniform distribution $\mathcal{U}$ over R_ψ.

We choose sample size m so that with probability at least $(1-\delta)$, $\texttt{Error}_2$ is a ϵ-additive approximation of true Ba-error.

Algorithm 1 FoRT $(P, \varphi, \mathcal{D}, \psi, \epsilon, \delta, k)$

1: $m \leftarrow \lceil \frac{2}{\epsilon^2} \log \frac{4}{\delta} \rceil$
2: $U \leftarrow m$ iid samples from $\mathcal{D}(R_\varphi)$
3: $\texttt{CE}_{\texttt{Fo}} \leftarrow \emptyset$, $\texttt{Error}_1 \leftarrow 0$
4: **for** $u \in U$ **do**
5: $\quad y \leftarrow P^{\leq k}(u)$
6: $\quad$ **if** $y \notin R_\psi$ **then**
7: $\quad\quad \texttt{CE}_{\texttt{Fo}} \leftarrow \texttt{CE}_{\texttt{Fo}} \cup \{y\}$
8: $\quad\quad \texttt{Error}_1 \leftarrow \texttt{Error}_1 + \frac{1}{m}$
9: **return** $(\texttt{Error}_1, \texttt{CE}_{\texttt{Fo}})$

Algorithm 2 BaRT$(P, \varphi, \psi, \epsilon, \delta, k)$

1: $m \leftarrow \lceil \frac{2}{\epsilon^2} \log \frac{4}{\delta} \rceil$
2: $V \leftarrow m$ iid samples from $\mathcal{U}(R_\psi)$
3: $\texttt{CE}_{\texttt{Ba}} \leftarrow \emptyset$, $\texttt{Error}_2 \leftarrow 0$
4: **for** $v \in V$ **do**
5: $\quad \Upsilon \leftarrow \varphi(X_0) \wedge \bigwedge_{j<k} F_P(X_j, X_{j+1}) \wedge \bigvee_{j \leq k}(X_j = v)$
6: $\quad$ **if** Υ is UNSAT **then**
7: $\quad\quad \texttt{CE}_{\texttt{Ba}} \leftarrow \texttt{CE}_{\texttt{Ba}} \cup \{v\}$
8: $\quad\quad \texttt{Error}_2 \leftarrow \texttt{Error}_2 + \frac{1}{m}$
9: **return** $(\texttt{Error}_2, \texttt{CE}_{\texttt{Ba}})$

4.2 The Refinement Subroutine

TreeRepair (Algorithm 3) repairs the decision-tree representation of a candidate formula ψ using Fo and Ba counterexamples. It takes as input a decision tree $\mathfrak{T}$, counterexample sets $\texttt{CE}_{\texttt{Fo}}$ and $\texttt{CE}_{\texttt{Ba}}$, thresholds $\texttt{Th}_1, \texttt{Th}_2$, and the variable set $\mathcal{X}$. It aims to modify $\mathfrak{T}$ so that no more than $\texttt{Th}_1$ members of $\texttt{CE}_{\texttt{Fo}}$ and $\texttt{Th}_2$ members of $\texttt{CE}_{\texttt{Ba}}$ remain misclassified.

The algorithm operates in rounds. At each round it first identifies the set L of current leaves using ExtractLeaves (line 4) and considers every potential refinement $(\ell, \mu) \in L \times (\{\texttt{Prune}\} \cup (\{\texttt{Split}\} \times \mathcal{X}))$. Each refinement is one of:

- **Prune**: delete a leaf node ℓ and then reclassifying all states assigned to that subtree. E.g., pruning node id 3 reduces both $\texttt{Error}_1$ and $\texttt{Error}_2$ to 0 (Figure 1e). The cost of the prune operation is 1.
- **Split**: replace a leaf ℓ with an internal decision on some variable in $\mathcal{X}$, creating two new leaves. For example, splitting node id 3 on predicate flip also eliminates all errors (Figure 1d). Splitting incurs cost 2.

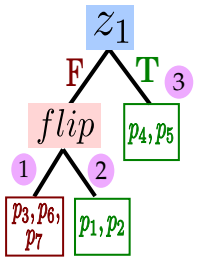
For each potential refinement (ℓ, μ), GetGain (line 6) computes the number of counterexamples that would be correctly classified if the refinement were applied, and GetCost (line 7) returns the "cost" for making the refinement corresponding to (ℓ, μ). Finally, in line 8, a ratio $\text{Impact}_{(\ell,\mu)}$ of the gain to cost is calculated for each potential refinement. A refinement is sampled randomly with probability proportional to Impact (line 9) and applied via MakeTheMove (line 10). The set of counterexamples is appropriately refined by removing those that have already been addressed. Refinements occur in a top–down manner: shallow splits result in coarse approximations of the reachable region, whereas deeper splits

```
def Geo0(int z, Bool flip):
  while (flip == 0):
    d = bernoulli(0.5)
    if d:
      flip = 1
    else:
      z = z + 1
  return {z,flip}
```

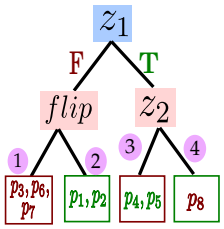
(a) Program Geo0

Id	z2	z1	z0	flip	Label
p_1	0	0	0	1	+
p_2	0	0	1	1	+
p_3	0	0	1	0	-
p_4	0	1	0	1	+
p_5	0	1	1	1	+
p_6	1	0	0	0	-
p_7	1	0	1	0	-

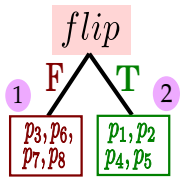
(b) Initial Samples



(c) Initial Tree



(d) Split



(e) Prune

Figure 1: Tree repair on program Geo0

Algorithm 3 $\texttt{TreeRepair}(\mathfrak{T}_{\text{init}}, \texttt{CE}_{\texttt{Fo}}, \texttt{CE}_{\texttt{Ba}}, \texttt{Th}_1, \texttt{Th}_2, \mathcal{X})$

Initialize $\mathfrak{T} \leftarrow \mathfrak{T}_{\text{init}}$
Initialize $E_1 \leftarrow \texttt{CE}_{\texttt{Fo}}$, $E_2 \leftarrow \texttt{CE}_{\texttt{Ba}}$
while NOT ($|E_1| \leq \texttt{Th}_1$ and $|E_2| \leq \texttt{Th}_2$) **do**
  $L \leftarrow \texttt{ExtractLeaves}(\mathfrak{T})$
  for $(\ell, \mu) \in L \times (\{\texttt{Prune}\} \cup (\{\texttt{Split}\} \times \mathcal{X}))$ **do**
    $\text{Gain}_{(\ell,\mu)} \leftarrow \texttt{GetGain}(\mathfrak{T}, E_1, E_2, \ell, \mu)$
    $\text{Cost}_{(\ell,\mu)} \leftarrow \texttt{GetCost}(\mathfrak{T}, E_1, E_2, \ell, \mu)$
    $\text{Impact}_{(\ell,\mu)} \leftarrow \frac{\text{Gain}_{(\ell,\mu)}}{\text{Cost}_{(\ell,\mu)}}$
  Sample (ℓ, μ) with probability proportional to $\text{Impact}_{(\ell,\mu)}$
  $\mathfrak{T}, E_1, E_2 \leftarrow \texttt{MakeTheMove}(\mathfrak{T}, E_1, E_2, \ell, \mu)$
Output $\mathfrak{T}$

Algorithm 4 $\texttt{PRORP}(P(\mathcal{X}), \varphi, \mathcal{D}, \epsilon_1, \epsilon_2, \delta, k)$

$\texttt{Error}_1 \leftarrow 1, \texttt{Error}_2 \leftarrow 1$
$\texttt{CE}_{\texttt{Fo}} \leftarrow \emptyset, \texttt{CE}_{\texttt{Ba}} \leftarrow \emptyset$
$\texttt{Th}_1 \leftarrow \lceil \frac{2}{\epsilon_1} \log(\frac{4}{\delta}) \rceil, \texttt{Th}_2 \leftarrow \lceil \frac{2}{\epsilon_2} \log(\frac{4}{\delta}) \rceil$
$\texttt{PosExamples} \leftarrow \texttt{FoRT}(P, \varphi, \mathcal{D}, \texttt{False}, \frac{0.1}{\sqrt{|\mathcal{X}|}}, 0.1, k)$
$\texttt{NegExamples} \leftarrow \texttt{BaRT}(P, \varphi, \texttt{True}, \frac{0.1}{\sqrt{|\mathcal{X}|}}, 0.1, k)$ PositiveExamples
$\mathfrak{T} \leftarrow \texttt{CreateTree}(\texttt{PosExamples}, \texttt{NegExamples})$
while NOT ($\texttt{Error}_1 \leq \epsilon_1$ and $\texttt{Error}_2 \leq \epsilon_2$) **do**
  $\psi \leftarrow \texttt{TreeDNF}(\mathfrak{T})$
  $\mathfrak{T} \leftarrow \texttt{TreeRepair}(\mathfrak{T}, \texttt{CE}_{\texttt{Fo}}, \texttt{CE}_{\texttt{Ba}}, \texttt{Th}_1, \texttt{Th}_2, \mathcal{X})$
  $\texttt{Error}_1, \texttt{CE}_{\texttt{Fo}} \leftarrow \texttt{FoRT}(P, \varphi, \mathcal{D}, \psi, \epsilon_1, \delta, k)$
  $\texttt{Error}_2, \texttt{CE}_{\texttt{Ba}} \leftarrow \texttt{BaRT}(P, \varphi, \psi, \epsilon_2, \delta, k)$
Output $\texttt{TreeDNF}(\mathfrak{T})$

produce fine-grained corrections.

These rounds continue till remaining counterexamples in $\texttt{CE}_{\texttt{Fo}}$ and $\texttt{CE}_{\texttt{Ba}}$ fall below their thresholds.

4.3 The Learning Algorithm

Our main learning algorithm, PRORP, takes as input a probabilistic program P (over a set of variables $\mathcal{X}$), a precondition given as a CNF-formula φ, the input distribution $\mathcal{D}$ over R_φ, two error parameters (a Fo-error parameter ϵ_1 and a Ba-error parameter ϵ_2), a confidence parameter δ, and the number of steps k. The subroutine returns a DNF-formula that with probability at least $(1-\delta)$ represents a (ϵ_1, ϵ_2)-approximate *statistically likely k-step reachable set* of P with respect to φ. PRORP does so by learning the corresponding decision tree.

PRORP starts by obtaining states from $\texttt{Cl}_k(\varphi)$ (positive examples; in line 4) and states from outside of $\texttt{Cl}_k(\varphi)$ (negative examples; in line 5) by calling the subroutines FoRT with the candidate formula set as False and BaRT with the candidate formula set as True respectively. For the initial sampling phase, the value of ϵ is chosen heuristically as $\frac{0.1}{sqrt(|\mathcal{X}|)}$ to account for programs with large state space as well as to collect "enough" labeled examples. Using the subroutine CreateTree a decision tree is learned that is consistent with both the positive and negative examples.

Once the initial decision tree is obtained, PRORP computes the corresponding DNF-formula using the subroutine TreeDNF (in line 8). Then it uses subroutines FoRT and BaRT (in lines 10–11) to estimate the Fo/Ba errors and construct the Fo/Ba counterexamples. If the errors are not within the acceptable limits the set of counterexamples are given to TreeRepair (in line 9) that appropriately modifies the tree and returns a new tree. This process is repeated until the errors fall within acceptable bounds and in practice for our experiments, we set a timeout. Finally, the DNF corresponding to the tree is returned.

Termination and Convergence. Termination is ensured by memorizing visited trees to prevent cycling; the finite state space $\{0,1\}^{|\mathcal{X}|}$ bounds the search. The procedure adopts a global guess-and-check strategy rather than greedy ascent, since the non-convex search space admits local minima.

4.4 Illustrative Example

We illustrate PRORP on program Geo0 (Figure 1a) with $\varphi : (\texttt{z} = 0) \wedge (\texttt{flip} = 0)$, unsigned 3-bit encoding for z, $k = 5$, and $\epsilon_1 = \epsilon_2 = 0.2$. To generate labeled positive and negative samples, we call FoRT and BaRT with the candidate formula set to False and True respectively, yielding the sample multiset in Figure 1b (every element distinct here), from which we learn a decision tree (Figure 1c). The green (positive) and red (negative) states are compiled into the formula $\psi : (z_1 \vee \texttt{flip})$ as described in Section 2.3.

The verification–repair loop continues until both checks succeed. In this instance, FoRT produces no counterexamples, while BaRT reports 1100 (denoted by p_8) as a Ba-counterexample, raising $\texttt{Error}_2$ from 0.0 to 0.4. TreeRepair considers two repairs of leaf 3: (i) splitting on flip (Figure 1d) or (ii) pruning (Figure 1e), both driving $(\texttt{Error}_1, \texttt{Error}_2)$ to 0. Assuming the prune mutation is chosen, the resulting tree (Figure 1e) represents flip. Re-verification yields a new Ba-counterexample 1101, prompt-

ing a refinement (e.g., splitting leaf 2 on z_2) that produces the final tree for $\psi := (\neg z_2 \wedge \texttt{flip})$. The verification errors now lie within the thresholds $(\epsilon_1, \epsilon_2) = (0.2, 0.2)$, and the algorithm terminates.

4.5 Theoretical Guarantee

Theorem 2 (**Correctness of** PRORP**).** *If* PRORP *halts, the returned formula ψ represents a $(\epsilon_1, \epsilon_2, \delta)$-approximate statistically likely k-step reachable set.*

Corollary 1. *If* PRORP *halts and returns the formula ψ, then with probability at least $(1-\delta)$, ϵ_1-$\texttt{Cl}_k(\varphi) \subseteq R_\psi$.*

The formal proof of Theorem 2 is given in the full version [Ray *et al.*, 2026] of this paper.

5 Evaluation

PRORP is implemented in Python using scikit-learn for decision trees, CMSGen [Golia *et al.*, 2021] and Pepin [Soos *et al.*, 2023] for uniform-like CNF/DNF sampling, CadiCaL [Biere *et al.*, 2024] for SAT solving, and `z3` [Moura and Bjørner, 2008] to generate program formula. We evaluate 16 benchmarks from prior work [Bao *et al.*, 2024; Chen *et al.*, 2015; Kaminski and Katoen, 2017] by varying the precondition φ, unroll depth $k \in \{16, 32, 64, 128\}$, bit-width (16/32-bit), using $\epsilon_1 = \epsilon_2 = 0.05$, $\delta = 0.1$ and uniform input distribution. We set a `timeout` of 2 hours. Our experiments address four research questions:

- **RQ1:** Can PRORP synthesize an $(\epsilon_1, \epsilon_2, \delta)$-approximate statistically likely k-step reachable set for a range of programs and preconditions?
- **RQ2:** Does PRORP scale for many variables and large k?
- **RQ3:** Do candidate sets synthesized by PRORP correspond to meaningful interpretations?
- **RQ4:** Is PRORP better than naïve Monte Carlo sampling?

5.1 RQ1: Efficiency of Synthesizing Approximate Statistically Likely k-Step Reachable Set

PRORP successfully produced a candidate formula for the $(\epsilon_1, \epsilon_2, \delta)$-approximate k-reachable set on all 218 instances, with runtimes ranging from 6 to 755 seconds for the 16-bit encodings and 6 to 1885 seconds for the 32-bit encodings. We manually verified that, for each problem instance, the candidate set indeed contains the entire ϵ_1-$\texttt{Cl}_k(\varphi)$ set. Table 2 reports the number of learning phases (#PH), total time (TT) and candidate set obtained for seven illustrative benchmarks with a chosen precondition for 16-bit encoding and unrolling depth k set to 64. The complete and detailed set of experimental results is given in the full version [Ray *et al.*, 2026].

5.2 RQ2: Scalability of PRORP with Respect to k and Bit-Width of Program Variables

To address this question, we selected two representative benchmarks, `LinExp` and `RevBin`, each with a single precondition under a uniform input distribution, and measured runtime across varying bit-widths and unrolling depths. Figure 2 reports the resulting performance curves. As expected, runtime increases with both program bit-width and unrolling depth k: larger bit-width blows up the state space exponentially, while higher value of k expands the reachable portion of the state space. From the evaluation results, `LinExp` incurs the highest runtime cost, exceeding 1000 seconds for large k and bit-width, whereas `RevBin` exhibits a similar upward trend in runtime but remains under 500 seconds. Overall, these results indicate that PRORP scales predictably across varying bit-widths and unrolling depths for the evaluated programs, within the parameter ranges studied.

Program	Precondition	k	#PH	TT (secs)	Candidate
Geo0	(z_{14})	64	1	6.96	(z_{14})
LinExp	(count<64)	64	3	44.89	0≤count<256
Sum0	(x=64, n<$\frac{k}{2}$)	64	4	63.80	64≤x<512
Fair	(count<64)	64	2	21.90	0≤count<64
DepRV	(x=0, y=0, n=k)	64	7	104.52	20≤x<44, y=64−x
RevBin	(x=$\frac{k}{2}$, z<16)	64	4	40.58	48≤z<80
Mart	(c=0, b=64, r=0)	64	7	65.79	c=64, b=0, 0<r<8

Table 2: Results on programs modelled with 16-bit integers

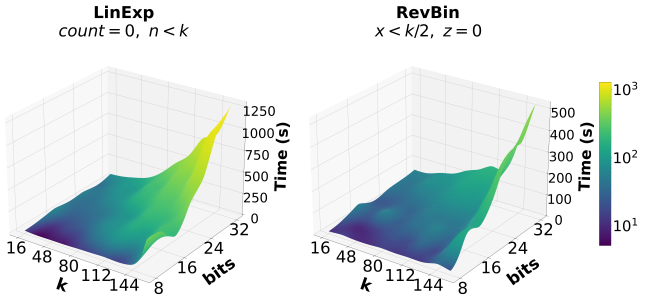


Figure 2: Scalability plots for `LinExp` and `RevBin` with preconditions ($\texttt{count} = 0, \texttt{n} < k$) and ($\texttt{x} < \frac{k}{2}, \texttt{z} = 0$) respectively

5.3 RQ3: Interpretation of the Candidate Sets to Interesting Problems (Case Study)

Let us discuss and interpret the output of PRORP on the program `Sum0` [Chen *et al.*, 2015]. The program (Figure 4) has a loop whose number of iterations is fixed by the input variable `n`. In the i-th iteration the value of `x` is incremented by $(\texttt{n} - i)$ with probability $\frac{1}{2}$ and not incremented with remaining probability $\frac{1}{2}$. Here, we consider only preconditions with exactly one satisfying assignment, i.e., $(\texttt{x} = x_0, \texttt{n} = n_0)$. Note that for `Sum0`, all terminal states are characterized by $\texttt{n} = 0$. Let `X` be the random variable denoting the value of `x` for terminal states. Thus, $\mathbb{E}[\texttt{X}] = x_0 + \frac{n_0(n_0+1)}{4}$ and `X` takes values in the interval $[x_0, x_0 + \frac{n_0(n_0+1)}{2}]$. That is, if $R_\varphi = \{(x_0, n_0)\}$, then $\texttt{Cl}_k(\varphi) := x_0 \leq \texttt{x} \leq x_0 + \frac{n_0(n_0+1)}{2}$, assuming $k \geq n_0$. We also note that the random variable `X` follows a symmetric distribution, with $\Pr[\texttt{X} = x_0 + t] = \Pr[\texttt{X} = x_0 + \frac{n_0(n_0+1)}{2} - t]$. In Figure 3 the distribution is plotted for $(x_0, n_0) = (28, 8)$ which is clearly a bell-shaped curve. Thus, for any $\epsilon_1 \in (0, 1]$, ϵ_1-$\texttt{Cl}_k(\varphi)$ is of the form $(x_0 + \frac{n_0(n_0+1)}{4} - t \leq \texttt{x} \leq x_0 + \frac{n_0(n_0+1)}{4} + t)$ which is

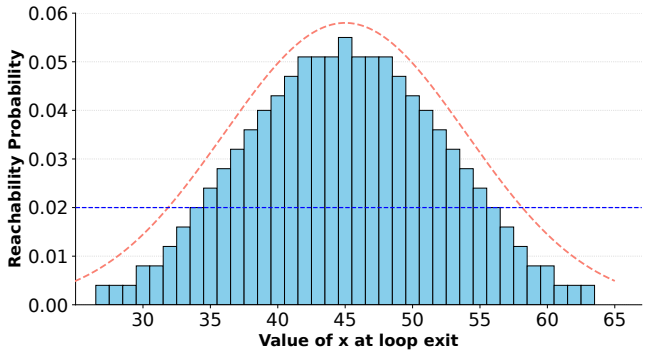


Figure 3: Output distribution of x in Sum0 for $R_\varphi : (\mathtt{x} = 28, \mathtt{n} = 8)$

```
def Sum0(int x, int n):
 while (n > 0):
  d = bernoulli(0.5)
  if (d):
   x = x + n
  n = n - 1
 return {x,n}
```

Figure 4: Program Sum0

```
def Unif2(int x, int l):
 y = 0; n = 32-l;
 while (n > 0):
  d = bernoulli(0.5)
  if d:
   y = 2y
  else:
   y = 2y + 1
  n = n - 1
 x = x + y
 return x
```

Figure 5: Program Unif2

obtained by chopping off the tails of the distribution.

We observe that the set represented by the output formula is close to ϵ_1-$\mathtt{Cl}_k(\varphi)$. We run PRORP on Sum0 for 4 preconditions with $k = 16$, each with a single satisfying assignment and parameters $\epsilon_1 = \epsilon_2 = 0.02$ and $\delta = 0.1$. The inputs x and n are 8-bit unsigned integers. The results of our experiment are compiled in Table 3. For each of the 4 preconditions, the right-hand tail of $\mathtt{Cl}_k(\varphi)$ gets chopped off in R_ψ. Also, for two preconditions: $\{(\mathtt{x} = 28, \mathtt{n} = 8)$ and $(\mathtt{x} = 31, \mathtt{n} = 16)\}$, both the tails of $\mathtt{Cl}_k(\varphi)$ are trimmed in R_ψ and includes the set ϵ_1-$\mathtt{Cl}_k(\varphi)$ fully.

5.4 RQ4: Performance Comparison with Baseline

We evaluate the precision of PRORP against a baseline of naïve Monte Carlo (MC) sampling on a family of 32-bit instances of the benchmark program Unif2 (Figure 5), which returns uniformly distributed values over a configurable interval.

In this family, the reachable set consists of offsets $x \in [x_0, (x_0 + 2^{32-\ell}) \bmod 2^{32})$ for $\ell = 0, \ldots, 32$, with precondition $\varphi := (x = x_0)$. We take parameters $x_0 = 251$ and $\epsilon_1 = \epsilon_2 = 0.05$, $\delta = 0.1$, $k = 32$, and report *Fo-error* and *Ba-error* (as percentages) for each ℓ. Figure 6 plots three curves: MC Fo-error, PRORP Fo-error, and PRORP Ba-error (Ba-error is always zero for MC).

In the *large-support regime* ($\ell \leq 18$), MC yields essentially 100% Fo-error as the sampling budget is overwhelmed by the support size, whereas PRORP attains near-zero Fo and Ba errors. In the *transition regime* ($18 \leq \ell \leq 22$), MC Fo-error drops sharply while PRORP incurs minor errors due to tree generalization yet remains substantially more accurate. In the *small-support regime* ($\ell \geq 23$), both methods achieve negligible error. Overall, PRORP maintains low error across all support sizes, whereas MC becomes competitive only once the reachable support is small enough for exhaustive sampling.

R_φ	$\mathtt{Cl}_k(\varphi)$	ϵ_1-$\mathtt{Cl}_k(\varphi)$	R_ψ
$(\mathtt{x} = 28,\ \mathtt{n} = 8)$	$28 \leq \mathtt{x} \leq 64$	$36 \leq \mathtt{x} \leq 56$	$28 \leq \mathtt{x} \leq 63$
$(\mathtt{x} = 48,\ \mathtt{n} = 16)$	$48 \leq \mathtt{x} \leq 184$	$96 \leq \mathtt{x} \leq 136$	$48 \leq \mathtt{x} \leq 175$
$(\mathtt{x} = 31,\ \mathtt{n} = 16)$	$31 \leq \mathtt{x} \leq 167$	$79 \leq \mathtt{x} \leq 119$	$32 \leq \mathtt{x} \leq 159$
$(\mathtt{x} = 15,\ \mathtt{n} = 16)$	$15 \leq \mathtt{x} \leq 151$	$63 \leq \mathtt{x} \leq 103$	$32 \leq \mathtt{x} \leq 127$

Table 3: Output of PRORP on Sum0 ($\epsilon_1 = \epsilon_2 = 0.02, k = 16$)

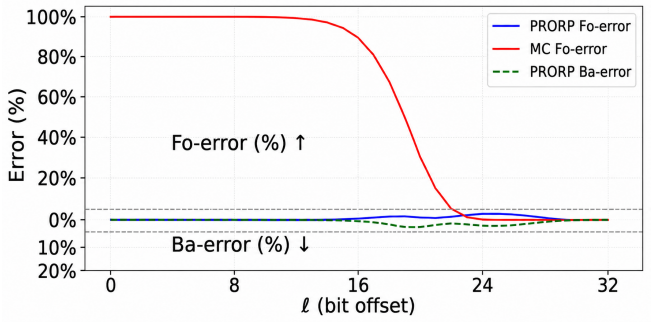


Figure 6: Fo and Ba error % for PRORP versus Monte Carlo

6 Related Work

Probabilistic program analysis is well studied. Exist [Bao *et al.*, 2024], Prinsys [Gretz *et al.*, 2013], Chen et al. [Chen *et al.*, 2015], Feng et al. [Feng *et al.*, 2017], Wang et al. [Wang *et al.*, 2018], MORA [Bartocci *et al.*, 2020], and CegisPro2 [Batz *et al.*, 2023] synthesize quantitative invariants using techniques such as template-based methods, abstract interpretation, and counterexample-guided or data-driven learning. PSI [Gehr *et al.*, 2016] and λPSI [Gehr *et al.*, 2020] compute posterior distributions via computer algebra, while PSE [Geldenhuys *et al.*, 2012] and Plinko [Susag *et al.*, 2022] support symbolic execution of probabilistic programs.

Our work is also related to angelic verification [Das *et al.*, 2015; Das and Lal, 2017; Joshi *et al.*, 2012], which mitigates the pessimism of traditional verification by favoring benign environment behaviors. In contrast, we model the environment probabilistically and learn symbolic summaries of statistically likely reachable states, while guaranteeing that long failure sequences remain improbable.

Methodologically, our approach follows the guess-and-check paradigm for invariant synthesis. Prior work has used decision trees [Aiken *et al.*, 2016; Ezudheen *et al.*, 2018; Garg *et al.*, 2014; Garg *et al.*, 2016; Riley and Fedyukovich, 2022], PAC learning [Sharma *et al.*, 2013], reinforcement learning [Si *et al.*, 2018; Yu *et al.*, 2023], continuous logic networks [Ryan *et al.*, 2019; Yao *et al.*, 2020], and large language models [Chakraborty *et al.*, 2023; Wu *et al.*, 2024]. Unlike these methods, which synthesize inductive invariants for all reachable states, PRORP learns symbolic summaries of statistically likely reachable states with finite-sample approximation guarantees.

Ethical Statement

This work presents PRORP, a framework for mining statistically likely reachable states in probabilistic programs. The research is purely theoretical and algorithmic in nature, with applications in program verification, AI safety, and bug prioritization. We do not foresee direct negative societal impacts from this work. The benchmarks used in our evaluation are sourced from prior published literature, and no human subjects, private data, or sensitive information were involved in any part of this research.

Acknowledgements

Arijit Ghosh acknowledges partial support from the Science and Engineering Research Board (SERB), Government of India, through the MATRICS grant MTR/2023/001527, and from the Department of Science and Technology (DST), Government of India, through grant TPN-104427. Sourav Chakraborty acknowledges partial support from the TAC-DCSW/VPP project. This research work was partially supported by Research-I Foundation of IIT Kanpur. Subhajit Roy acknowledges the support of the Jainendra Navlakha (1974) Chair Professorship at IIT Kanpur. Nitesh Trivedi acknowledges the support of FARE Fellowship of IIT Kanpur.

Contribution Statement

Arnab Ray and Nitesh Trivedi contributed equally.

References

[Aiken *et al.*, 2016] Andrew Aiken, Marco Corradini, and Daniel Strubell. From Invariant Checking to Invariant Inference Using Randomized Search. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV)*, pages 388–402, 2016.

[Bao *et al.*, 2024] Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. Data-driven invariant learning for probabilistic programs. *Formal Methods in System Design*, 66:278–306, 2024.

[Bartocci *et al.*, 2020] Ezio Bartocci, Laura Kovács, and Miroslav Stankovič. Mora: Automatic Generation of Moment-Based Invariants. In *Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 492–498, 2020.

[Batz *et al.*, 2023] Kevin Batz, Mingshuai Chen, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. Probabilistic Program Verification via Inductive Synthesis of Inductive Invariants. In *Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 410–429, 2023.

[Biere *et al.*, 1999] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 193–207, 1999.

[Biere *et al.*, 2024] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In *Proceedings of the 36th International Conference on Computer Aided Verification (CAV)*, pages 133–152, 2024.

[Chakraborty *et al.*, 2023] Saikat Chakraborty, Shuvendu Lahiri, Sarah Fakhoury, Akash Lal, Madanlal Musuvathi, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. Ranking LLM-Generated Loop Invariants for Program Verification. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9164–9175, 2023.

[Chen *et al.*, 2015] Yu-Fang Chen, Chih-Duo Hong, Bow-Yaw Wang, and Lijun Zhang. Counterexample-Guided Polynomial Loop Invariant Generation by Lagrange Interpolation. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*, pages 658–674, 2015.

[Das and Lal, 2017] Ankush Das and Akash Lal. Precise null pointer analysis through global value numbering. In *Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 25–41, 2017.

[Das *et al.*, 2015] Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. Angelic Verification: Precise Verification Modulo Unknowns. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*, pages 324–342, 2015.

[Ezudheen *et al.*, 2018] P. Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P. Madhusudan. Horn-ICE Learning for Synthesizing Invariants and Contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):131:1–131:25, 2018.

[Feng *et al.*, 2017] Yijun Feng, Lijun Zhang, David N. Jansen, Naijun Zhan, and Bican Xia. Finding Polynomial Loop Invariants for Probabilistic Programs. In *Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 400–416, 2017.

[Garg *et al.*, 2014] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, volume 8663 of *Lecture Notes in Computer Science*, pages 69–87, 2014.

[Garg *et al.*, 2016] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning Invariants using Decision Trees and Implication Counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 499–512, 2016.

[Gehr *et al.*, 2016] Timon Gehr, Sasa Misailovic, and Martin Vechev. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Proceedings of the 28th International Conference on Computer-Aided Verification (CAV)*, pages 62–83, 2016.

[Gehr *et al.*, 2020] Timon Gehr, Samuel Steffen, and Martin Vechev. λPSI: Exact Inference for Higher-Order Probabilistic Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 883–897, 2020.

[Geldenhuys *et al.*, 2012] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. Probabilistic Symbolic Execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*, pages 166–176, 2012.

[Golia *et al.*, 2021] Priyanka Golia, Mate Soos, Sourav Chakraborty, and Kuldeep S. Meel. Designing Samplers is Easy: The Boon of Testers. In *Proceedings of the 21st Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 222–230, 2021.

[Gretz *et al.*, 2013] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Prinsys: On a Quest for Probabilistic Loop Invariants. In *Proceedings of the 10th international conference on Quantitative Evaluation of Systems (QEST)*, pages 193–208, 2013.

[Joshi *et al.*, 2012] Saurabh Joshi, Shuvendu K. Lahiri, and Akash Lal. Underspecified Harnesses and Interleaved Bugs. *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 19–30, 2012.

[Kaminski and Katoen, 2017] Benjamin Lucien Kaminski and Joost-Pieter Katoen. A Weakest Pre-Expectation Semantics for Mixed-Sign Expectations. In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, 2017.

[Kozen, 1977] Dexter Kozen. Lower Bounds for Natural Proof Systems. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 254–266, 1977.

[Kwiatkowska *et al.*, 2011] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems . In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, pages 585–591, 2011.

[Lahiri and Roy, 2022] Sumit Lahiri and Subhajit Roy. Almost Correct Invariants: Synthesizing Inductive Invariants by Fuzzing Proofs. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 352–364, 2022.

[Moura and Bjørner, 2008] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[Ray *et al.*, 2026] Arnab Ray, Nitesh Trivedi, Ansuman Banerjee, Sourav Chakraborty, Arijit Ghosh, and Subhajit Roy. Mining Statistically Likely k-Reachable States in Probabilistic Programs, 2026. An extended abstract of this paper will appear in IJCAI 2026. Full version available at https://github.com/arnab-007/PRORP_.

[Riley and Fedyukovich, 2022] Daniel Riley and Grigory Fedyukovich. Multi-Phase Invariant Synthesis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 607–619, 2022.

[Ryan *et al.*, 2019] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. Cln2inv: learning loop invariants with continuous logic networks. *arXiv preprint arXiv:1909.11542*, 2019.

[Sharma *et al.*, 2013] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. Verification as Learning Geometric Concepts. In *Proceedings of the 20th International Symposium on Static Analysis (SAS)*, volume 7935 of *Lecture Notes in Computer Science*, pages 388–411, 2013.

[Si *et al.*, 2018] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning Loop Invariants for Program Verification. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NeurIPS)*, pages 7762–7773, 2018.

[Soos *et al.*, 2023] Mate Soos, Divesh Aggarwal, Sourav Chakraborty, Kuldeep S. Meel, and Maciej Obremski. Engineering an Efficient Approximate DNF-counter. In *Proceedings of the 32nd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2031–2038, 2023.

[Susag *et al.*, 2022] Zachary Susag, Sumit Lahiri, Justin Hsu, and Subhajit Roy. Symbolic execution for randomized programs. *Proceedings of the ACM on Programming Languages 2*, 6(OOPSLA):1583–1612, 2022.

[Wang *et al.*, 2018] Di Wang, Jan Hoffmann, and Thomas Reps. PMAF: An Algebraic Framework for Static Analysis of Probabilistic Programs. *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementatio (PLDI)*, pages 513–528, 2018.

[Wu *et al.*, 2024] Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. LLM Meets Bounded Model Checking: Neuro-Symbolic Loop Invariant Inference. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 406–417, 2024.

[Yao *et al.*, 2020] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. Learning Nonlinear Loop Invariants with Gated Continuous Logic Networks. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 106–120, 2020.

[Yu *et al.*, 2023] Shiwen Yu, Ting Wang, and Ji Wang. Loop Invariant Inference through SMT Solving Enhanced Reinforcement Learning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 175–187, 2023.