Structural Abstraction and Refinement for Probabilistic Programs

GUANYAN LI*, Tsinghua University, China and University of Oxford, United Kingdom
JUANEN LI, Beijing Normal University, China
ZHILEI HAN, Tsinghua University, China
PEIXIN WANG†, East China Normal University, China
HONGFEI FU†, Shanghai Jiao Tong University, China
FEI HE†, Tsinghua University, China

In this paper, we present structural abstraction refinement, a novel framework for verifying the threshold problem of probabilistic programs. Our approach represents the structure of a Probabilistic Control-Flow Automaton (PCFA) as a Markov Decision Process (MDP) by abstracting away statement semantics. The *maximum reachability* of the MDP naturally provides a proper upper bound of the violation probability, termed the *structural upper bound*. This introduces a fresh "structural" characterization of the relationship between PCFA and MDP, contrasting with the traditional "semantical" view, where the MDP reflects semantics. The method uniquely features a clean separation of concerns between probability and computational semantics that the abstraction focuses solely on probabilistic computation and the refinement handles only the semantics aspect, where the latter allows non-random program verification techniques to be employed without modification.

Building upon this feature, we propose a general counterexample-guided abstraction refinement (CEGAR) framework, capable of leveraging established non-probabilistic techniques for probabilistic verification. We explore its instantiations using trace abstraction. Our method was evaluated on a diverse set of examples against state-of-the-art tools, and the experimental results highlight its versatility and ability to handle more flexible structures swiftly.

CCS Concepts: • **Theory of computation** → **Program analysis**; **Assertions**.

Additional Key Words and Phrases: probabilistic programs, abstraction refinement, CEGAR, trace abstraction

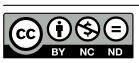
ACM Reference Format:
Guanyan Li*, Juanen Li, Zhilei Han, Peixin Wang†, Hongfei Fu†, and Fei He†. 2025. Structural Abstraction and Refinement for Probabilistic Programs. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 337 (October 2025), 28 pages. https://doi.org/10.1145/3763115

1 Introduction

Since the early days of computer science, formalisms for reasoning about probability have been widely studied. As an extension to classical imperative programs, probabilistic programs enable

† Corresponding authors, no ordering among the corresponding authors
∗ This paper describes the work whose majority was performed while Guanyan Li was at Tsinghua University.

Authors' Contact Information: Guanyan Li*, Tsinghua University, Beijing, China and University of Oxford, Oxford, United Kingdom, guanyan.li.cs@gmail.com; Juanen Li, Beijing Normal University, Beijing, China, juanen.li.se@mail.bnu.edu.cn; Zhilei Han, Tsinghua University, Beijing, China, hzl21@mails.tsinghua.edu.cn; Peixin Wang†, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China, pxwang@sei.ecnu.edu.cn; Hongfei Fu†, Shanghai Jiao Tong University, Shanghai, China, jt002845@sjtu.edu.cn; Fei He†, Tsinghua University, Beijing, China, hefei@tsinghua.edu.cn.



ACM 2475-1421/2025/10-ART337
https://doi.org/10.1145/3763115

efficient solutions to many algorithmic problems [42, 44, 48]. It has also led to new models which play key roles in cryptography [21], linguistics [36], and especially machine learning where generic models [9, 22, 53] are expressed by them.

In this paper, we revisit the probabilistic threshold problem. Specifically, given a probabilistic program P, a pre-condition φ_e, and a post-condition φ_f, the *violation probability* is defined as the probability that an input satisfying φ_e leads to a final result violating φ_f after the execution of P. This probability, represented as $\mathbb{P}\left[\lightning \{\varphi_e\}\ P\ \{\varphi_f\}\right]$ in the form of a Hoare triple [34, 47], raises the question of whether it is bounded by a given threshold. Such a problem is commonly encountered in practical scenarios. For example, the accuracy of differential privacy mechanisms [18] can be analysed as threshold problems where we bound the probability that the randomly-perturbed output deviates largely from the actual value. The target of approximate computing [52, 57] can be described as the assertion violation that the faulty hardware runs into erroneous status, and the tail bounds of randomized algorithms [54] can be described as the assertion violation that the runtime exceeds a given threshold. In fact, the threshold problem has been extensively researched, with numerous approaches proposed to address the issue. These include predicate abstraction [33], fixed-point computation [5, 54], and concentration inequalities [10], etc.

We tackle the problem by proposing a novel *abstraction refinement* [14, 28] framework tailored specifically for probabilistic programs, named *structural abstraction refinement*. A distinctive feature of our approach is the separation of concerns between probability and semantics, thus enabling the direct application of existing non-probabilistic program analysis and verification techniques to probabilistic programs.

As an overview, our method exploits a distinctive yet intuitive connection between the *Probabilistic Control-Flow Automaton* (PCFA) and the *Markov Decision Process* (MDP). The formalisms of MDP and *Markov Chain* (MC) are widely adopted mathematical tools in the analysis of probabilistic programs, and their relationship with program verification has also been extensively researched [1, 33]. However, in existing literature, the relation between the automaton and MDP is typically *semantical*, where the latter functions as the semantic representation of the PCFA by (potentially infinitely) unrolling the concrete state space of the automaton [25, 33, 35, 39]. In contrast, our work seeks to establish a connection between these two formalisms in a syntactical, or rather a *structural*, manner.

Unlike the semantical connection, the main insight of our work is that the structure of a PCFA A can be viewed as an MDP by *abstracting away the computational semantics* of statements in the automaton. Following this abstraction, one obtains precisely the structure of an MDP, where the program statements serve as the *action* labels. This *underlying MDP* of the PCFA A provides an appropriate abstraction of the program in the sense that the violation probability is bounded by the *maximum reachability* probability of the MDP from its initial to ending location, which is termed the *structural upper bound* of A. In summary, we have:

$$\text{Structural Upper Bound} := \text{Maximum MDP Probability} \geq \text{Violation Probability} \tag{1}$$

This intuition is visualised in Section 2 and formalised in Section 4.1.

As the structural abstraction disregards semantics, enabling us to focus solely on the probabilistic aspect, the refinement process focuses solely on the semantic aspect, independent of probabilities. Specifically, we establish Theorem 4.7, a key theoretical result allowing refinement by constructing a *refinement automaton* V that over-approximates the *violating traces* of the original PCFA A. Here, a *trace* refers to a *sequence of statements* accepted by an automaton, indicating that the construction focuses on execution sequences, as in the non-probabilistic case, without handling probabilities. A rough visualisation of this procedure is provided in Section 2 with principles detailed in Section 4.3.

These theoretical foundations thus offer a clean separation between probabilities and semantics. Aligning with this feature, we employ a modular way of introducing the automation algorithms.

Firstly, building upon this feature, we propose a generalised probabilistic *counterexample-guided abstraction refinement* (CEGAR) framework. This framework is uniquely capable of directly automating various *non-random* analysis and verification methodologies for probabilistic program verification. Second, we instantiate the framework with *trace abstraction* [28, 49]. Thanks to the clean separation between probabilities and semantics — with the probabilistic aspect handled entirely by our framework — the non-probabilistic refinement technique can be applied *directly* to the semantic component exactly as in the deterministic case, *without requiring any modifications*. This serves as an example of how non-random refinement techniques can be utilised to verify probabilistic threshold problems, empowered by our framework. Further, beyond such a direct instantiation, we observed that the integration could be further optimised, allowing it to retain the *refutational completeness* characteristic of the original trace abstraction technique, which is a property rarely seen in probabilistic verification beyond the finite-state (incl. bounded) techniques [30, 35, 37].

Finally, we implemented our framework and conducted a comparative evaluation with state-of-the-art verification tools on a diverse and arguably fair set of benchmark examples collected from a wide range of literature. Our method shows evident advantages in better handling a broader range of examples as compared to the existing tools, including those with a large or even unbounded state space and more tricky control-flow structures. It successfully handles over 2× more examples than the state-of-the-art tools for the benchmarks, and demonstrates better efficiency in more than 73% of the examples.

Contributions. In summary, our contributions are as follows:

(1) We introduced structural abstraction and its refinement principle for probabilistic program verification, leveraging a novel structural link between PCFA and MDP. This method distinctly separates probability from semantics.
(2) To automate the theories developed in (1), we developed a generalised CEGAR framework that can integrate directly with non-random techniques. We instantiated this framework with trace abstraction and optimise it to retain refutational completeness.
(3) We implemented the optimised algorithm in (2) and benchmarked it against state-of-the-art tools across a diverse set of examples from the literature, demonstrating the method's versatility. The experimental result strongly backs our claim on the strength of our method.

Outline. The remainder of this paper is organised as follows. We begin with a motivating example to illustrate our approach in more detail in Section 2. We then formally define the relevant concepts and the problem we aim to address in Section 3. Next, we explore the theoretical foundations of our framework in Section 4, establishing the validity of structural abstraction and principles of refinement (i.e., Theorem 4.7). To automate this theoretical principle, we discuss a general CEGAR framework utilising structural abstraction and refinement, and the instantiations of the framework in Section 5. Finally, we present our experimental results in Section 6 and conclude with a discussion of related work in Section 7.

2 Motivating Example

To better present the key idea of our approach from a high-level perspective, we illustrate it through a motivating example.

Example 2.1 (Limit). Consider the Hoare-style program [34, 47] shown in Fig. 1, where X and C are of integer value and the operator $\oplus$ is a fair binary probabilistic choice.

The program can be interpreted as a coin-flipping game between two players, say, Alice and Bob. They make an initial toss (line 3); if heads-down, Alice wins immediately. Otherwise, Bob chooses a number of rounds (variable C) to continue. Alice then guesses that no heads-up will occur during

1 {True}
2 $X ::= 0$;
3 $C ::= 0 \oplus skip$;
4 **while** $C > 0$ **do**
5 $\quad X ::= X + 1 \oplus skip$;
6 $\quad C ::= C - 1$
7 **done**
8 $\{X = 0\}$

Fig. 1. Program of Example 2.1

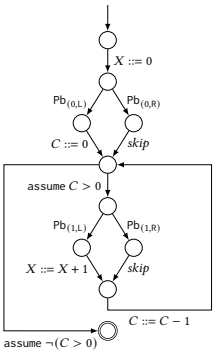


Fig. 2. PCFA P of Example 2.1

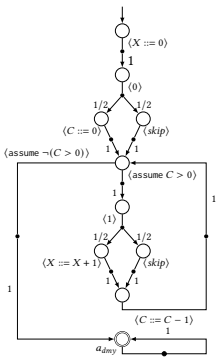


Fig. 3. The MDP $\mathcal{D}(P)$ Underlying P

these rounds. The variable X (line 5) tracks heads-up events, and Alice wins if the postcondition $X = 0$ is met. Notably, the probability of Bob winning, i.e., violating the postcondition, increases as C grows. However, this probability is capped at 0.5 as $C \longrightarrow \infty$.

We proceed to demonstrate how our approach can establish the *exact* bound of 0.5 on the violation probability. First, the program is transformed into a Probabilistic Control-Flow Automaton (PCFA), shown in Fig. 2. In this transformation, each binary probabilistic choice is assigned a unique *distribution tag* (0 and 1 here). Additionally, each choice is tagged as left (L) or right (R) for clarity. Thus, probabilistic statements appear in the form $\mathsf{Pb}_{(i,d)}$, where i is the distribution tag, and d is either L or R. The resulting PCFA P will serve as the primary focus of our analysis.

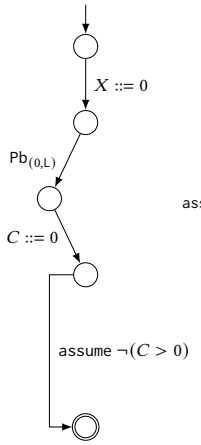
Structural Abstraction. Given the PCFA, one may observe that the formalism's structure resembles a Markov decision process (MDP), where Fig. 3 visualised this intuition, in which distributions are marked by hyper-edges with a central • to indicate branching points. This MDP is called the underlying MDP of P, denoted by $\mathcal{D}(P)$. In the MDP, statements are treated purely as labels without semantic functions, noted in $\langle - \rangle$. Tags 0 and 1 appear as action names, $\langle 0 \rangle$ and $\langle 1 \rangle$, representing fair Bernoulli distributions, while other actions denote Dirac distributions. Finally, to match the definition of MDP, we add a dummy action a_{dmy} inducing a self-looping Dirac distribution at the ending location.

Notably, unlike the usual semantic approach seen in the literature [33, 35], the MDP is *not* an unrolling of the state space of the program. In our method, *structural abstraction*, the semantics of the statements are completely omitted, so that disjoint assumptions like $\langle \mathtt{assume}\ \neg(C > 0) \rangle$ and $\langle \mathtt{assume}\ (C > 0) \rangle$ now become simply the usual *non-deterministic* choices (actions) in the MDP.

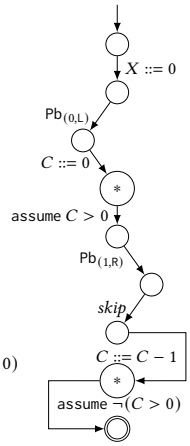
With this MDP, the probability of violating the property (i.e., Bob winning) is surely bounded by the maximum reachability probability from the start to the end location. Namely, the violation probability $\mathbb{P}\left[\nvdash \{\mathtt{True}\}\ P\ \{X = 0\}\right]$ is bounded by the *maximum reachability probability* in $\mathcal{D}(P)$, termed the *structural upper bound*, $\mathbb{P}_{\mathcal{U}}[P]$.

However, $\mathcal{D}(P)$ over-approximates P, yielding $\mathbb{P}_{\mathcal{U}}[P] = 1$, a trivial upper bound. Thus, this abstraction necessitates *refinement*. Examining the trivial bound reveals that both violating and non-violating traces contribute to this over-estimation. For example, trace 1 shows a safe trace that is not violating, while trace 2 illustrates an *infeasible* trace that is impossible to occur in actual execution. In trace 2, the small $*$ marks the starting location of the loop in Fig. 2.

Structural Refinement. The refinement process reduces the structural probability of the MDP derived from the PCFA by *eliminating* non-violating traces. To make this process operational, we show (in Theorem 4.7) that it suffices to construct a *refinement automaton* V that over-approximates the *violating traces* of P. With this condition, the following equality holds: $\mathbb{P}\left[\nvdash \{\varphi_e\}\ P\ \{\varphi_f\}\right] =$



Trace 1. Safe



Trace 2. Infeasible

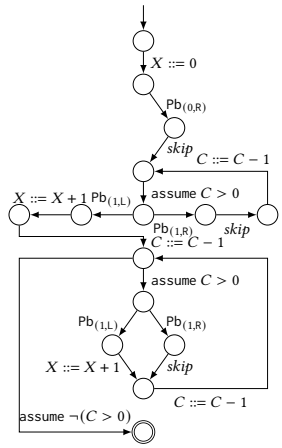


Fig. 4. The Refined PCFA P'

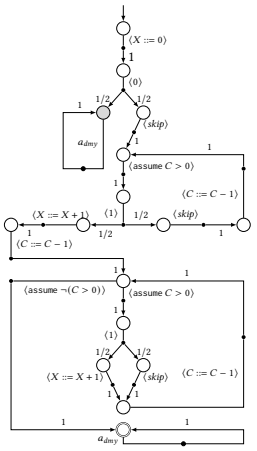


Fig. 5. The Refined MDP P'

$\mathbb{P}\left[\lightning \{\varphi_e\}\ \mathsf{min}(P \cap V)\ \{\varphi_f\}\right]$, where min and $\cap$ represent the standard deterministic minimisation and intersection operations [1] for finite automata. This implies that $\mathbb{P}_{\mathcal{U}}\left[\mathsf{min}(P \cap V)\right]$ remains an upper bound for $\mathbb{P}\left[\lightning \{\varphi_e\}\ P\ \{\varphi_f\}\right]$, referred to as the *refined (structural) upper bound*.

Notably, constructing V depends solely on the computational semantics of statements, without involving any probabilistic considerations. Specifically, as shown in traces like traces 1 and 2, the probabilistic statements ($\mathsf{Pb}_{(i,d)}$) do not affect whether a trace is violating. In essence, these statements act as neutral *skip* statements in computation. This enables the use of traditional non-probabilistic program verification and analysis techniques for constructing V. We illustrate this approach through *trace abstraction* [28, 49], the renowned technique in non-probabilistic verification, for this example.

Refinement with Trace Abstraction. Using trace abstraction, we first *generalise* the two identified non-violating traces into automata Q_1 and Q_2, shown in Fig. 6 and Fig. 7. The generalisation intuitively yields automata that encompass non-violating traces for the "same reasons" [1] as the provided traces. For instance, Figure 6 generalises Trace 1 by including all traces where X does not increase after being assigned to 0, which must be safe as they always satisfy $X = 0$. The complete rationale for this construction and more details on the techniques of trace abstraction are elaborated subsequently in Section 5.2. Following the principles of trace abstraction, the refinement automaton V is defined as $V := \overline{Q_1 \cup Q_2}$, where the overline denotes the standard complement operation for finite automata; since Q_1 and Q_2 contain only non-violating traces, the complement of their union, V, over-approximates the violating traces in P.

The refined PCFA P' is then constructed as: $P' := \mathsf{min}(P \cap V)$, as illustrated in Fig. 4, with its underlying MDP depicted in Fig. 5. Intuitively, this PCFA includes *only* traces that could lead to violation. Specifically, it avoids the $\mathsf{Pb}_{(0,\mathsf{L})}$ branch, which resets C and prevents loop entry. To align with the definition of MDP, when reflected in the underlying MDP in Figure 5, this becomes a transition with probability 1/2 to go to a dummy, self-absorbing MDP node, as to be detailed in Section 4.1. Further, for a trace to be accepted, the loop must iterate at least once through the $\mathsf{Pb}_{(1,\mathsf{L})}$ branch before termination. The refined upper bound $\mathbb{P}_{\mathcal{U}}\left[\mathsf{min}(P \cap V)\right]$ is then 0.5, implying that the probability of Bob winning is also bounded by 0.5, as expected. It is straightforward to observe that the refinement process, including the generation of V and the computation of the refined PCFA $\mathsf{min}(P \cap V)$, does not entail any probabilistic computation. Further, as will be further

[1] Usually captured by the technique of "interpolation" [28, 41, 52], to be detailed in Section 5.2.

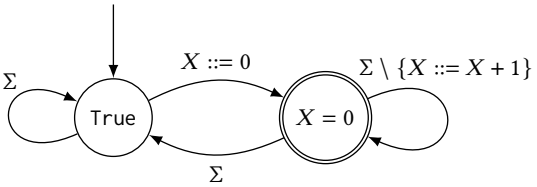


Fig. 6. Generalised Automaton Q_1 of trace 1

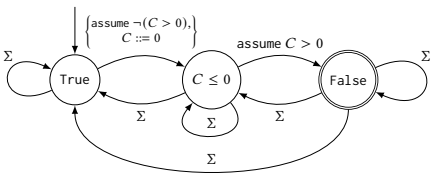


Fig. 7. Generalised Automaton Q_2 of trace 2

elaborated below, the refinement above amounts to applying well-established non-probabilistic techniques. This demonstrates the clear separation between probability and semantics in our approach.

3 Preliminaries

Following functional conventions, we assume the data structure of lists to be linked lists with constructor "::". We use the function init to take the whole list except the last element. We denote the update of a function f with a new mapping $X \mapsto V$ by $f[X \mapsto V]$. By slightly abusing symbols, we will use $\in$ and $\notin$ for lists.

Probabilistic Theory. A distribution over a countable set S, $\mathbb{P} \in \mathit{Dist}(S)$, is a function that assigns to each subset [2] of S a number between 0 and 1 such that: $\mathbb{P}[\emptyset] = 0$, $\mathbb{P}[S] = 1$ and for any countable sequence of pairwise disjoint subsets of S, $S_1, S_2, \ldots$, $\mathbb{P}\left[\uplus_{i=1}^{\infty} S_i\right] = \sum_{i=1}^{\infty} \mathbb{P}[S_i]$. For single element s, if it is clear from the context, we denote $\mathbb{P}[\{s\}]$ by just $\mathbb{P}[s]$. We use $\mathit{Supp}(\mathbb{P})$ to denote exactly all the non-0 elements of S, that: $\mathit{Supp}(\mathbb{P}) = \{s \in S \mid \mathbb{P}[s] \neq 0\}$. A sub-distribution $\mathbb{P}$ over set S is almost the same as a distribution except that $0 \leq \mathbb{P}[S] \leq 1$.

A *Markov chain* (MC) $\mathcal{M}$ is a pair $(Q, \mathcal{P})$, where Q is a set of nodes [3], and $\mathcal{P} : Q \rightharpoonup \mathit{Dist}(Q)$ is a partial function that assigns nodes a distribution. A trajectory ζ of an MC $\mathcal{M}(Q, \mathcal{P})$ is a finite sequence of nodes $q_1, \ldots, q_n$, where for each $i \in \{1, \ldots, n-1\}$, $q_{i+1} \in \mathit{Supp}(\mathcal{P}(q_i))$. The set of trajectories starting from a nodes q is denoted by $\mathbf{Traj}_{\mathcal{M}}(q)$. Then, the *reachability probability* from a node q to a node q' is $\mathsf{Reach}_{\mathcal{M}}(q, q') := \mathbb{P}\left[\{\zeta \in \mathbf{Traj}_{\mathcal{M}}(q) \mid \mathsf{last}(\zeta) = q', q' \notin \mathsf{init}(\zeta)\}\right]$, where last denotes the final element of the sequence and init denotes the sequence with the last element removed.

A *Markov decision process* (MDP) $\mathcal{D}$ is a tuple $(Q, \mathbf{Act}, \mathcal{P})$, where Q is a set of nodes; $\mathbf{Act}$ is a set of actions; $\mathcal{P} : Q \times \mathbf{Act} \rightharpoonup \mathit{Dist}(Q)$ is a partial function that assigns a node-action pair a distribution on nodes, where for a given node q, an action a is said to be *enabled* by the node, if $(q, a) \in \mathrm{dom}(\mathcal{P})$. Here dom means the domain of a (partial) function. A (deterministic) policy ψ of an MDP is a tuple (S, δ, s_0), where S is a (possibly infinite) set of policy states; $\delta : Q \times S \rightharpoonup \mathbf{Act} \times S$ selects an enabled action and returns the next policy state given a policy state and MDP node; s_0 is the initial policy state. The policy set of an MDP $\mathcal{D}$ is denoted by $\mathcal{S}(\mathcal{D})$. A policy with $|S| = 1$ is a *simple policy*. Applying a policy $\psi(S, \delta, s_0)$ to an MDP $\mathcal{D}(Q, \mathbf{Act}, \mathcal{P})$, denoted by $\mathcal{D}^{\psi}$, produces an MC, $(Q \times S, \mathcal{P}^{\psi})$, where $\delta(q, s) = (a, s')$ implies $\mathcal{P}^{\psi}(q, s)(q', s') = \mathcal{P}(q, a)(q')$, with $\mathcal{P}^{\psi}(q, s)(q', s'') = 0$ for $s'' \neq s'$. The maximum reachability probability from q to q' in $\mathcal{D}$ is: $\mathsf{MaxReach}_{\mathcal{D}}(q, q') := \sup_{\psi \in \mathcal{S}(\mathcal{D})} \mathsf{Reach}_{\mathcal{D}^{\psi}}(q, q')$.

Statements and PCFA. Throughout this paper, we fix a finite set of variables $\mathcal{V}$ and define $\mathcal{E}$ and $\mathcal{B}$ as the sets of *arithmetic* and *Boolean* expressions over $\mathcal{V}$, respectively. We denote variables by upper case letters $X, Y, \ldots$. For simplicity, we assume each variable in $\mathcal{V}$ takes values in $\mathbb{N}$, the natural numbers. A *valuation* v is a function $\mathcal{V} \to \mathbb{N} \uplus \{\bot\}$, where $\bot$ intuitively represents a

[2] A simplified version of σ-algebra from [40].

[3] We use the name *MDP nodes* to distinguish from *program states* below.

"non-sense" value; the set of all valuations is denoted $\mathbb{V}$. We write $[\![E]\!]_v$ and $[\![B]\!]_v$ for the *evaluation* of expressions E and B under valuation v, yielding values other than $\bot$. A predicate φ is simply a Boolean expression, and we use Φ to denote the set of all predicates. We write $v \models \varphi$ to represent $[\![\varphi]\!]_v = \texttt{True}$. Notably, $[\![\varphi]\!]_\bot = \texttt{False}$ for any predicate φ. The satisfiability and unsatisfiability of predicates follow standard definitions. The program P studied in our paper is essentially standard, which is recursively given by the following production rules:

$$P := X ::= E \mid P_1 \oplus P_2 \mid P_1 \circledast P_2 \mid \mathit{skip} \mid P_1; P_2 \mid \texttt{if } B \texttt{ then } P_1 \texttt{ else } P_2 \mid \texttt{while } B \texttt{ do } P' \texttt{ done}$$

Here, $E \in \mathcal{E}$ and $B \in \mathcal{B}$, the operator $\oplus$ denotes the fair binary probabilistic choice [4] and $\circledast$ represents the non-deterministic choice. As in the rest of this work, programs are represented as *probabilistic control-flow automata* (PCFA) in the standard way [54, 55], for which, the details are presented in the extended version [5].

We then delve into the introduction of PCFA and its related concepts. A *statement* (to appear in a PCFA) σ over $\mathcal{V}$ is given by:

$$\sigma := \mathit{skip} \mid X ::= E \mid \texttt{assume } B \mid \mathsf{Pb}_{(i,\mathsf{L})} \mid \mathsf{Pb}_{(i,\mathsf{R})} \mid *_i$$

where, again, $E \in \mathcal{E}$ and $B \in \mathcal{B}$ range over arithmetic and Boolean expressions respectively; the statements $\mathsf{Pb}_{(i,\mathsf{L})}$ and $\mathsf{Pb}_{(i,\mathsf{R})}$ are the (fair) probabilistic statements, and $*_i$ is the non-deterministic statement, where i ranges over identifiers [6], which is called a *distribution tag* when appearing in a probabilistic statement and non-deterministic tag when in a non-deterministic statement. The set of all statements is denoted $\mathbf{ST}$. The *evaluation* of a statement σ over a valuation v, denoted by $[\![\sigma]\!]_v$, is defined in the standard way, that $[\![\texttt{assume } B]\!]_v := \bot$ if $[\![B]\!]_v = \texttt{False}$, $[\![\sigma]\!]_\bot := \bot$, and $[\![X ::= E]\!]_v := v[X \mapsto [\![E]\!]_v]$, with all other cases having $[\![\sigma]\!]_v := v$.

Definition 3.1 (Probabilistic Control-Flow Automaton). A probabilistic control-flow automaton (PCFA) A is a tuple $(L, \Sigma, \Delta, \ell_0, \ell_e)$, where L is a *finite* set of locations; $\Sigma \subseteq \mathbf{ST}$ is a *finite* set of statements; $\Delta : L \times \Sigma \rightharpoonup L$ is a *partial* transition function, and we write $\ell \xrightarrow{\sigma} \ell'$ to denote that $(\ell, \sigma) \in \mathrm{dom}(\Delta)$ and $\Delta(\ell, \sigma) = \ell'$; $\ell_0 \in L$ is the initial; and, $\ell_e \in L$ is the ending location. Specifically, no transition starts from the ending location ℓ_e, that is: $\forall \sigma.(\ell_e, \sigma) \notin \mathrm{dom}(\Delta)$.

Notably, from an automata-theoretic perspective, PCFA is inherently *restricted*: it is essentially a *Deterministic Finite Automaton* (DFA) [1] with a *single* ending location, which has *no out-transitions*. These restrictions are removed later in the concept of *general PCFA* in Section 4.3.

A *program state* C of a PCFA A is a pair (ℓ, v), where ℓ is a location of A and v is a valuation. A transition $\ell \xrightarrow{\sigma} \ell'$ is said to be enabled by program state (ℓ, v) if either: $\sigma = \texttt{assume } B$ is an assumption statement with $[\![B]\!]_v = \texttt{True}$, or σ is another type of statement. A program state (ℓ', v') is the *successor program state* of $C = (\ell, v)$ under transition $r := \ell \xrightarrow{\sigma} \ell'$ if: r is an enabled transition by C, and either $v' = v[X \mapsto [\![E]\!]_v]$ for an assignment $\sigma = X ::= E$; or $v' = v$ otherwise. A *computation* π is a finite sequence of program states with transitions: $(\ell_0, v_0) \xrightarrow{\sigma_1} (\ell_1, v_1) \cdots \xrightarrow{\sigma_n} (\ell_n, v_n)$, where: ℓ_0 is the initial location of A; each ℓ_i for $i \in \{0, \ldots, n\}$ is a location of A; and, for each $(\ell_i, v_i) \xrightarrow{\sigma_{i+1}} (\ell_{i+1}, v_{i+1})$, the program state (ℓ_{i+1}, v_{i+1}) is a successor of (ℓ_i, v_i) under transition $\ell_i \xrightarrow{\sigma_{i+1}} \ell_{i+1}$. A computation with the final program state at location ℓ_e is termed a *terminating computation*. The weight of a computation π is given by $wt(\pi) = (1/2)^n$, where n is the count of probabilistic transitions in π. Given a finite-length computation π starting from the initial location,

[4] For simplicity, we introduce only the fair binary choice, but it can be easily generalised to work with any number. And fair binary choice itself does not hinder theoretical generality [51].

[5] See https://arxiv.org/abs/2508.12344 for the extended version of this paper.

[6] The identifiers can be any set of symbols, e.g., natural numbers in the motivating example.

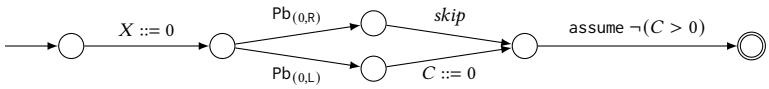


Fig. 8. Example of a Run in Example 2.1

a *trace* τ *induced* by it, denoted $\mathsf{ind}(\pi)$, refers to the sequence of labels involved. It is termed *accepting* if π ends in ℓ_e (i.e., is terminating). Additionally, we require traces to be non-empty. The set of accepting traces of a PCFA A is denoted $\mathcal{L}(A)$. Notably, a trace τ uniquely determines a computation and the definition of the *weight* of τ, follows naturally.

Multiple non-probabilistic transitions may be enabled by a given program state simultaneously, introducing non-determinism. To address this, we define an *automaton scheduler*. An automaton scheduler ξ for a PCFA A is a function that assigns each non-terminating computation π of A a unique (sub-)probabilistic distribution over the transitions enabled by the final program state of π. Specifically, $\xi(\pi)$ is: (a) a Dirac distribution on a transition with a non-probabilistic statement; (b) a fair Bernoulli distribution over two probabilistic transitions, $\mathsf{Pb}_{(i,\mathsf{L})}$ and $\mathsf{Pb}_{(i,\mathsf{R})}$, sharing a distribution tag i; or, (c) a sub-distribution on a single probabilistic transition, $\mathsf{Pb}_{(i,\mathsf{L})}$ or $\mathsf{Pb}_{(i,\mathsf{R})}$, tagged by i, with probability $1/2$. The set of schedulers for A is denoted by Ξ_A. A computation $\pi := C_0 \xrightarrow{\sigma_1} C_1 \cdots \xrightarrow{\sigma_n} C_n$ is *guided* by an automaton scheduler ξ if, for every prefix $C_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_i} (\ell_i, v_i) \xrightarrow{\sigma_{i+1}} (\ell_{i+1}, v_{i+1})$ of π, the transition $\ell_i \xrightarrow{\sigma_{i+1}} \ell_{i+1}$ belongs to $\mathit{Supp}(\xi(C_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_i} C_i))$.

Given an initial valuation v_{init} and a scheduler ξ, a PCFA A defines a probabilistic distribution over the final valuation at the ending location. Formally, let the set of terminating computations starting from (ℓ_0, v_{init}) and being guided by ξ to be $\Pi^{\xi}_{v_{init}}$, which is also called a *run* of A, the (sub)distribution [7] over a set of valuations at the ending location is:

$$\mathbb{P}_A^{(v_{init},\xi)}[V] := \sum_{\left\{\pi \in \Pi^{\xi}_{v_{init}} \,\middle|\, \mathsf{last}(\pi) = (\ell_e, v), v \in V\right\}} wt(\pi)$$

Intuitively, given a fixed initial value and a scheduler, a run of a PCFA represents the Markov chain that encompasses all the computations that may be executed due to probabilistic choices. See, e.g., Figure 8 for an example.

Problem Statement. Hence, quantifying both the initial valuation and the scheduler, ***violation probability*** of a PCFA A against a pre-condition φ_e and a post-condition φ_f is:

$$\mathbb{P}\left[\lightning \{\varphi_e\}\ A\ \{\varphi_f\}\right] := \sup_{v_{init} \models \varphi_e} \sup_{\xi \in \Xi_A} \mathbb{P}_A^{(v_{init},\xi)}\left[\{v \in \mathbb{V} \mid v \models \neg\varphi_f\}\right] \tag{2}$$

The pair of pre- and post-conditions (φ_e, φ_f) is called the specification or property of the program. Further, given a threshold β, the verification problem is then to answer whether $\mathbb{P}\left[\lightning \{\varphi_e\}\ A\ \{\varphi_f\}\right] \leq \beta$. In the rest of the paper, we fix a PCFA A and a specification (φ_e, φ_f).

4 Structural Abstraction and Refinement

In this section, we present the theoretical foundations of our method, namely, structural abstraction and its refinement. These established principles form the theoretical basis for the algorithms to be introduced in the subsequent section. In Section 4.1, we present a formal treatment of PCFA as an MDP, which lays down a solid foundation for the central concept of this paper: *structural abstraction*, introduced in Section 4.2. We complement the abstraction with its corresponding refinement principle in Section 5.

[7] Depending on whether the program is almost surely terminating (AST).

4.1 PCFA as MDP

This part delves into the key insights of our method – "*structurally* viewing PCFA as MDP". Formalising this insight necessitates a *new definition* of the violation probability. As a result, an equality between it and the standard definition will be established in Theorem 4.2. As briefly overviewed in Figure 2 and Figure 3, a PCFA A can be viewed as an MDP. This is achieved by mapping automata locations to MDP nodes, statements (or distribution tags) to actions, and transitions to probabilistic distributions. Notably, our method differs significantly from the *semantical* approaches established in the literature [14, 33]. In our approach, the nodes in the MDP correspond to *locations*, rather than to *program states* of the original PCFA. More specifically:

Definition 4.1 (Underlying MDP). For PCFA $A(L, \Sigma, \Delta, \ell_0, \ell_e)$, its underlying MDP $\mathcal{D}(A)$ is $(L \cup \{q_{dmy}\}, \mathbf{Act}(A), \mathcal{P})$ where:

- **Nodes**: A's locations plus dummy state q_{dmy}
- **Actions** $\mathbf{Act}(A)$ contains: (1) $\langle\sigma\rangle$ for non-probabilistic $\sigma \in \Sigma$; (2) $\langle i\rangle$ for each $\mathsf{Pb}_{(i,d)} \in \Sigma$; and, (3) a_{dmy} the dummy action for q_{dmy} and the ending location ℓ_e.
- **MDP Transitions** $\mathcal{P}$:

(1) Every transition $\ell \xrightarrow{\sigma} \ell' \in \Delta$ with non-probabilistic σ triggers a Dirac distribution on ℓ', namely, $\mathcal{P}(\ell, \langle\sigma\rangle)(\ell') = 1$.

(2) For probabilistic transitions $\ell \xrightarrow{\mathsf{Pb}_{(i,d)}} \ell_1$, $P := \mathcal{P}(\ell, \langle i\rangle)$ depends on whether the complementary transition $\ell \xrightarrow{\mathsf{Pb}_{(i,\overline{d})}} \ell_2$ exists (where $\overline{\mathsf{L}} = \mathsf{R}$ and $\overline{\mathsf{R}} = \mathsf{L}$): (1) if it exists, P is a fair Bernoulli distribution on ℓ_1 and ℓ_2 if $\ell_1 \neq \ell_2$; (2) if it exists with $\ell_1 = \ell_2$ then $P(\ell_1) = 1$ is Dirac again; and, (3) if it does not exist, P is a fair Bernoulli distribution on ℓ_1 and q_{dmy}.

(3) To match the definition of MDP, q_{dmy} and the ending location ℓ_e have a unique dummy action a_{dmy}, which yield Dirac self-loop for both of them.

Similar to MDP and Markov Chains (MC), a PCFA A where each location has at most one enabled action is effectively an MC. Thus, we define *policies for PCFA*, also denoted ψ, analogous to MDP [1], as a triple (S, δ, s_0), where S is a set of policy nodes; $\delta : L \times S \rightharpoonup \mathbf{Act}(A) \times S$ selects an action from the current node and PCFA location; s_0 is the initial policy node.

Inheriting from MDP, a policy with only one node, $|S| = 1$, is called a *simple policy*. The set of PCFA policies is denoted by $\mathcal{S}(A)$. Clearly, each policy $\psi \in \mathcal{S}(A)$ corresponds uniquely to a policy in $\mathcal{S}(\mathcal{D}(A))$. Applying a policy $\psi = (S, \delta, s_0)$ to a PCFA $A = (L, \Sigma, \Delta, \ell_0, \ell_e)$ induces another PCFA, denoted $A^\psi = (L^\psi, \Sigma, \Delta^\psi, (\ell_0, s_0), \ell_e)$, where: the new location set $L^\psi := ((L \setminus \{\ell_e\}) \times S) \uplus \{\ell_e\}$ isolates the ending location ℓ_e, making other locations the product with S; the new transition function Δ^ψ selects transitions per the policy, so $(\ell, s) \xrightarrow{\sigma} (\ell', s')$ in Δ^ψ iff: there exists $\ell \xrightarrow{\sigma} \ell'$ in Δ, and either (1) σ is not probabilistic and $\delta(\ell, s) = (\sigma, s')$, or (2) σ is probabilistic with distribution tag i, and $\delta(\ell, s) = (i, s')$; also, $(\ell, s) \xrightarrow{\sigma} \ell_e$ exists in Δ^ψ iff the above holds for some s'. This application induces a PCFA interpretable as an MC.

Each run Π of a PCFA corresponds to a specific policy. Intuitively, policy states correspond to program states, and actions to the scheduler's resolution of non-determinism. Applying a policy produces a PCFA where the set of accepting traces matches the run's induced traces $\{\mathsf{ind}(\pi) \mid \pi \in \Pi\}$. Conversely, not every policy represents a PCFA run. Some policies produce "non-sense" traces not induced by computations. For instance, in trace 2, assigning $C = 0$ followed by `assume` $C > 0$ creates a contradiction. Such traces are *infeasible*; others are *feasible*.

We define the *evaluation of a trace*, $[\![\tau]\!]$, as chaining its elements' computations: $[\![[\sigma]]\!](v) := [\![\sigma]\!]_v$ for a singleton trace, and $[\![\sigma :: \tau]\!](v) := [\![\tau]\!]([\![\sigma]\!]_v)$. Given an initial valuation v_{init} and a post-condition φ_f, a trace τ is a *violating trace* if $[\![\tau]\!](v_{init}) \models \neg\varphi_f$; otherwise, it is *non-violating*. These concepts

generalise to a pre-condition φ_e by existential quantification: $\exists v_{init} \models \varphi_e . [\![\tau]\!](v_{init}) \models \neg\varphi_f$. The *path condition* of a trace τ, denoted $\mathsf{PathCond}(\tau)$, is the *weakest* predicate indicating φ_e and satisfying $\forall v \models \mathsf{PathCond}(\tau).[\![\tau]\!](v) \models \neg\varphi_f$. For a trace set Θ, $\mathsf{PathCond}(\Theta) := \bigwedge_{\tau\in\Theta} \mathsf{PathCond}(\tau)$.

Thus, we observe: the probability of a PCFA A violating the pre- and post-condition φ_e and φ_f is derived by traversing all possible policies and, for each, accumulating all violating traces, formally expressed as:

$$\mathbb{P}^{\mathcal{D}}\left[\vDash \{\varphi_e\}\ A\ \{\varphi_f\}\right] := \sup_{v\models\varphi_e} \sup_{\psi\in\mathcal{S}(A)} \sum_{\tau\in\mathcal{L}(A^\psi)} wt(\tau)\cdot \mathbf{1}_{[[\tau]](v)\models\neg\varphi_f]} \tag{3}$$

where $\mathbf{1}_{[-]}$ is the indicator function that returns 1 if the given value equals to `True` and 0 otherwise. We call the formula $\mathbf{1}_{[[\tau]](v)\models\neg\varphi_f]}$ the *semantic check* of trace τ. Comparing this definition with the standard definition in Equation (2), although both policies and schedulers resolve non-determinism, schedulers account for the concrete valuation, whereas policies can be arbitrary and do not consider semantics. This definition hence intuitively "delays" the step-wise valuation transition within a computation to the final semantic check, allowing the selection of the next transition to be *arbitrary*. More specifically, in the standard definition, for any computation $(\ell_0, v_0) \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} (\ell_n, v_n)$, one must ensure the transition between each v_i and v_{i+1} is valid. The new definition, however, delays all step-wise obligations to an overall semantic check, i.e., $\mathbf{1}_{[[\sigma_1\ldots\sigma_n]](v_0)\models\neg\varphi_f]}$. This new definition (Equation (3)) thus distinguishes itself by separating semantic checks from probability computation, enabling a potential division between semantic verification and policy selection.

This approach, however, inevitably introduces nonsensical traces, which lack corresponding computations and are evaluated to $\bot$. Nevertheless, we argue that by considering *all* possible policies, the overall probabilities are the same as the standard definition. Formally, we have the following theorem, with a detailed proof provided in the extended version of this paper.

Theorem 4.2. *The two definitions in Equations (2) and (3) are equivalent for any A, φ_e and φ_f:*

$$\mathbb{P}^{\mathcal{D}}\left[\vDash \{\varphi_e\}\ A\ \{\varphi_f\}\right] = \mathbb{P}\left[\vDash \{\varphi_e\}\ A\ \{\varphi_f\}\right] \tag{4}$$

The theorem then concludes the validity of viewing a PCFA as an MDP, laying down a solid foundation for the structural abstraction below.

As a brief recapitulation, to introduce our new definition, we defined several key concepts, e.g., the policies, especially the simple ones, the evaluation of traces, feasibility and the path conditions. They all play significant roles in the below introduction as we will focus on the new definition.

4.2 Structural Abstraction

Based on the aforementioned definition, the concept of *structural abstraction* emerges naturally; this approach disregards computational semantics and utilises the underlying MDP of the PCFA to abstract the probabilistic program. Formally, in Eq. (3), the semantic check $\mathbf{1}_{[[\tau]](v)\models\neg\varphi_f]}$ is ignored, and is consistently treated as 1. This therefore yields an upper bound of the violation probability:

Definition 4.3 (Structural Upper Bound). The *structural upper bound* of a PCFA A with initial and ending locations ℓ_0 and ℓ_e is given by:

$$\mathbb{P}_{\mathcal{U}}[A] := \mathsf{MaxReach}_{\mathcal{D}(A)}(\ell_0, \ell_e) = \sup_{\psi\in\mathcal{S}(A)} \sum_{\tau\in\mathcal{L}(A^\psi)} wt(\tau) \tag{5}$$

By definition and Equation (4), we instantly have:

$$\mathbb{P}\left[\vDash \{\varphi_e\}\ A\ \{\varphi_f\}\right] = \mathbb{P}^{\mathcal{D}}\left[\vDash \{\varphi_e\}\ A\ \{\varphi_f\}\right] \leq \mathbb{P}_{\mathcal{U}}[A] \tag{6}$$

Thus, while calculating the exact value in (3) is challenging, the structural upper bound can be efficiently computed with standard algorithms [1] commonly used to determine maximum reachability probabilities in MDP.

Besides the principles of abstraction above, we also developed the following result for the new definition as the principle behind *refinement*, which essentially exclude non-violating traces while retaining the violating ones. A proof is provided using the new definition in the extended paper.

Lemma 4.4. *For two PCFA A_1 and A_2, when they have the same set of violating traces against the given pre- and post-conditions φ_e and φ_f, then, we have:*

$$\mathbb{P}\left[\vDash \{\varphi_e\}\ A_1\ \{\varphi_f\}\right] = \mathbb{P}^{\mathcal{D}}\left[\vDash \{\varphi_e\}\ A_1\ \{\varphi_f\}\right] = \mathbb{P}^{\mathcal{D}}\left[\vDash \{\varphi_e\}\ A_2\ \{\varphi_f\}\right] = \mathbb{P}\left[\vDash \{\varphi_e\}\ A_2\ \{\varphi_f\}\right]$$

Notably, the lemma only requires the two PCFA to have the same set of violating traces, and has no requirement on the non-violating traces: A_1 and A_2 are even allowed to have different sets of statements Σ.

As the refinement procedure involves removal of the non-violating traces from A, which essentially yields another PCFA A', the result essentially enables the *cross-structural* comparison of the violating probability between different PCFA.

4.3 The Refinement of Structural Abstraction

General PCFA. Prior to exploring the details of refinement, let us first introduce the concept of *general PCFA*. As discussed in Section 3, the notion of PCFA is essentially restricted. We then introduce the concept of a general PCFA by easing these restrictions, so that in such a notion: (1) the transition function Δ is now a transition relation $\Delta : L \times \Sigma \times L$, hence $\ell \xrightarrow{\sigma} \ell'$ then denotes $(\ell, \sigma, \ell') \in \Delta$; (2) the ending location is not unique, and now there is a set $L_e \subseteq L$ called the ending location set; (3) transitions are allowed to start from an ending location. The terminologies for general PCFA are inherited directly from PCFA. This definition essentially renders general PCFA simply any *Non-deterministic Finite Automata* (NFA) without restriction.

Example 4.5 (General PCFA). The generalised automata in Figs. 6 and 7 for traces 1 and 2 are general PCFA but NOT PCFA, as they are non-deterministic, also, there are transitions starting from the ending location. □

Building on Lemma 4.4, consider a PCFA A and a general PCFA V that over-approximates the violating traces of A against the given pre- and post-conditions φ_e and φ_f. One might naturally attempt to derive a refined bound as $\mathbb{P}_{\mathcal{U}}[A \cap V]$ by directly applying Lemma 4.4. However, a technical obstacle arises: to apply Lemma 4.4, both automata A_1 and A_2 must be PCFA conforming to Definition 3.1. Recall that a PCFA imposes three structural constraints: (a) determinism, (b) a unique ending location, and (c) no out-transitions from the ending location. Since V is a general PCFA, the intersection $A \cap V$ may violate these constraints and thus fail to be a PCFA.

To resolve this issue, we apply the *deterministic minimisation* operation $\mathsf{min}(-)$ to $A \cap V$, which restores the PCFA structure. The following lemma formalises this transformation:

Lemma 4.6. *For any PCFA A and general PCFA V, $\mathsf{min}(A \cap V)$ is a PCFA.*

Proof. To show this, observe the following straightforward properties [8]: (1) all PCFA (hence A) satisfy the property that: (**) *every accepting trace of the automaton is not a prefix of another accepting trace*; (2) any deterministically minimal automaton satisfies Property (b) and (c) (i.e., a single ending location without out-edge) iff it satisfies (**); (3) as $\mathcal{L}(A \cap V)$ is a subset of $\mathcal{L}(A)$, $A \cap V$ also satisfies (**). So that, by (2), we can conclude that $\mathsf{min}(A \cap V)$ is a PCFA, as deterministic

[8] Detailed proof is provided in the extended version of this paper.

minimisation already guarantees determinism (i.e., Property (a)). Notably, pure determinisation is not sufficient here, as it cannot guarantee (b) uniqueness of the ending location. □

By the rationale above, we can now apply Lemma 4.4 to conclude the following theorem:

Theorem 4.7. *Consider a PCFA A and a (general) PCFA V that over-approximates the violation traces of A against the given pre- and post-conditions φ_e and φ_f. Let the deterministic automata minimisation operation be denoted by* $\mathsf{min}(-)$. *The following equality holds:*

$$\mathbb{P}\left[\vdash \{\varphi_e\}\ A\ \{\varphi_f\}\right] = \mathbb{P}\left[\vdash \{\varphi_e\}\ \mathsf{min}(A \cap V)\ \{\varphi_f\}\right]$$

Proof. Let $A_1 = A$ and $A_2 = \mathsf{min}(A \cap V)$. By Lemma 4.6, A_2 is a PCFA. Since V over-approximates the violating traces of A, A_1 and A_2 have the same set of violating traces. The equality follows from Lemma 4.4. □

With the theorem established, refining structural abstraction fundamentally thus entails identifying an appropriate *general PCFA*, termed the *refinement automaton* V. The new structural upper bound $\mathbb{P}_{\mathcal{U}}[\mathsf{min}(A \cap V)]$ serves as a *refined structural upper bound* for the violating probability $\mathbb{P}\left[\vdash \{\varphi_e\}\ A\ \{\varphi_f\}\right]$.

Separating Probability from Semantics. Notably, Theorem 4.7 facilitates a clear separation between probability and semantics. Since the construction of V must exclusively include violating traces, and given the definition of the evaluation of probabilistic statements as $[\![\mathsf{Pb}_{(i,d)}]\!]_v = v$ — identical to *skip* and $*_i$ statements — probabilistic statements do not influence whether a trace is violating. This implies that non-random techniques are *directly* applicable in constructing V. This principle will be fully leveraged and demonstrated in the subsequent section.

5 Automating Refinement of Structural Abstraction

This section introduces algorithms for automatically verifying the thresholds problem, building on the introduced structural abstraction (Equation (6)), and the principles of the refinement (Theorem 4.7). In the following, we first propose a general CEGAR framework capable of integrating non-random techniques, as detailed in Section 5.1, exploiting the separation of probability and semantics afforded by these foundations. Within the general framework, we present concrete instantiations through trace abstraction in Sections 5.2 and 5.3. The former provides a direct instantiation, while the latter introduces a further optimisation to achieve refutational completeness. We argue that this demonstration offers a modular and well-structured description, where our framework handles the probabilistic aspects of verification, while leaving the semantic aspects to be addressed by established non-random techniques.

5.1 A General CEGAR Framework

This part presents a general CEGAR-based automatic verification framework for probabilistic programs, compatible with non-probabilistic verification and analysis techniques. We begin by introducing the concept of counterexamples, followed by a formal presentation of the key framework. This then leads to two subtle technical challenges: first, verifying whether a counterexample has indeed been identified, and second, resolving the incompatibility of path conditions between violating traces. The two points arise to complement the overall framework and present unique challenges as compared to the non-random verification.

Counterexamples. Following common CEGAR principles [14, 28, 33], our procedure either verifies the threshold or identifies a *counterexample* (CE) disproving it. Drawing inspiration from [25, 33], a *counterexample* in this framework is a finite set of *compatible* violating traces, certifying the existence of a run exceeding the violation probability threshold. Formally, a set of violating traces

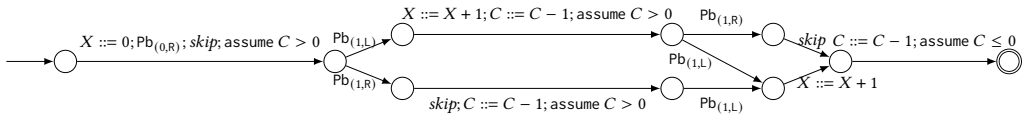


Fig. 9. Example of A Counterexample of Example 2.1 when $\beta = 0.3$

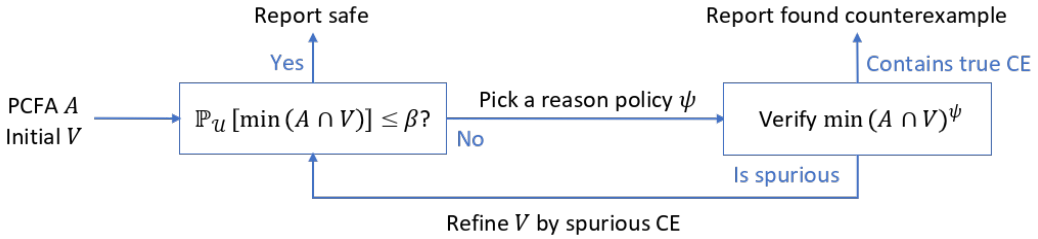


Fig. 10. General CEGAR Framework

Θ is compatible if: (1) the path condition of Θ is satisfiable; and (2) any two distinct traces τ_1 and τ_2 share a prefix (possibly ε) after which they diverge with probabilistic statements $\mathsf{Pb}_{(i,\mathsf{L})}$ and $\mathsf{Pb}_{(i,\mathsf{R})}$ sharing the same distribution tag i. The latter ensures that the traces form a PCFA that is effectively an MC, and hence the overall set Θ forms a subset of violating computations of a potential violating run. Finally, a lemma affirms the validity of these counterexamples [9].

Lemma 5.1. *Consider a PCFA A with pre- and post-conditions φ_e and φ_f, and a threshold $\beta \in [0, 1]$. The violation probability $\mathbb{P}\left[\nvdash \{\varphi_e\}\ A\ \{\varphi_f\}\right]$ exceeds β, iff there is a violating run of A, iff there is a finite set of compatible and violating traces of A.*

Example 5.2 (Counterexample). Referring to Example 2.1, consider the threshold $\beta = 0.3$ (instead of 0.5 in Section 2). A counterexample with three traces is shown in Fig. 9. Some trivial locations are abbreviated using semicolons ";" for brevity. All three traces are violating with a total probability of 0.375 when the initial valuation is $C = 2$, indicating that the violating probability of Example 2.1 exceeds the threshold $\beta = 0.3$. Notably, this set is compatible as they have a path condition $C = 2$.

Framework Overview. With the concept of counterexample established, we are ready to present our framework, as shown in Fig. 10. Initially, the process is provided with a PCFA A and an initial refinement automaton V, which over-approximates the violating traces of A. Then, the process computes the refined structural bound based on the current V. Should the bound be $\leq \beta$, the process deems it safe, for which the soundness is indicated by Eq. (6). Otherwise, in accordance with well-established conclusions for MDP [1], there must be a *simple* policy ψ such that $\sum_{\tau \in \mathcal{L}(\min(A \cap V)^{\psi})} wt(\tau) = \mathbb{P}_{\mathcal{U}}[\min(A \cap V)] > \beta$. We call such a policy ψ a *reason policy* (that induces the bound). Referring to the previous discussion on counterexamples, the traces of $\min(A \cap V)^{\psi}$ that are accepted already meet the second condition. Hence, we call the automaton and the accepting traces of $\min(A \cap V)^{\psi}$ a *candidate counterexample*. With such a reason policy, the process proceeds to ascertain whether there indeed exists a counterexample within $\min(A \cap V)^{\psi}$ – that a finite set of traces satisfying the first condition of counterexamples. Should such a set exist, by Lemma 5.1, it is justifiable to report a violation with the discovered counterexample. Otherwise, this intuitively suggests that the current V remains too coarse and necessitates refinement of V with the knowledge that the current candidate counterexample, i.e., $\min(A \cap V)^{\psi}$, is *spurious*.

[9] More details are in the extended version of this paper.

Example 5.3 (Reason Policy and Candidate Counterexample). Casting our minds back to Example 2.1, suppose the initial refinement automaton V is the trivial automaton that permits all traces. As a consequence, the initial PCFA $\mathsf{min}(P \cap V)$ retains its original structure P (Fig. 2).

Within this figure, the majority of the locations are elementary in that they permit at most one action, with the exception of the loop's starting location, which possesses two actions. In this scenario, the sole reason policy ψ is determined by opting for the action $\langle \texttt{assume}\ \neg(C > 0)\rangle$ at this particular location and defaulting to the only action available for the remainder of the locations.

By applying this reason policy ψ to P, the resulting PCFA P^{ψ} (the candidate counterexample) is exactly the structure portrayed in Fig. 8. □

Verify Candidate Counterexample by Enumeration. To verify the candidate counterexample, inspired by [25, 33], we enumerate traces from $\mathcal{L}(\mathsf{min}(A \cap V)^{\psi})$ to find a finite subset of violating traces with satisfiable path conditions. During the process, we maintain and enlarge a set Θ of violating traces in an on-the-fly manner.

The enumeration proceeds in two phases:

- *Explicit enumeration:* We iteratively enumerate traces from the (potentially infinite) candidate counterexample $\mathsf{min}(A \cap V)^{\psi}$ in descending order wrt. the weights of the traces. For each trace, we verify its violation status using SMT solvers [13, 15]; only violating traces are retained in the accumulating set Θ.
- *Symbolic subset search:* For the current finite set Θ of enumerated violating traces, we search for the maximum-weight *compatible* subset. This constitutes a combinatorial optimisation problem: given Θ's weighted path conditions, find the subset with satisfiable conjunction and maximum total weight. Formally, we are to solve the following formula (let $\mathbb{I}[P]$ denotes the Iverson bracket that if P is true, it returns 1, otherwise 0):

$$\mathsf{MaxArg}_{\Theta' \subseteq \Theta} \left(\mathbb{I}[\mathsf{PathCond}(\Theta') \text{ is satisfiable}] \cdot wt(\Theta')\right)$$

  This problem can be solved using the MAX-SMT approach [7].

The enumeration terminates when either: (1) The maximum compatible subset weight exceeds threshold β, yielding a valid counterexample; or (2) The sum of the current maximum and remaining unenumerated weights cannot exceed β, confirming spuriousness.

Example 5.4 (Trace Enumeration). Continuing the story in Example 5.3, let us assume again $\beta = 0.3$. By employing the enumeration algorithm as delineated in [25] to the candidate counterexample depicted in Fig. 8, both traces have an equal weight of 0.5, and thus, either could be enumerated first. When both traces are enumerated, since both of them are safe, the process terminates upon enumerating the two traces, as the residual probability becomes $0 \leq \beta = 0.3$, confirming the spuriousness of the counterexample. □

Before introducing the customisable refinement procedure, there is one more subtlety to address: the instant divergence that may occur from incompatible path conditions of violating traces. We address this by *pre-condition splitting*.

Splitting the Pre-conditions. Notice that the existence of violating yet incompatible traces potentially leads to instant divergence of the CEGAR process. Consider the following example:

Example 5.5 (Divergence from Incompatibility). Consider the following Hoare style program whose PCFA is depicted in Fig. 11, let the threshold β be 0.75.

$$\{\texttt{True}\}\ (\textbf{if}\ X > 0\ \textbf{then}\ Y ::= 0\ \textbf{else}\ Y ::= 1) \oplus (\textbf{if}\ X > 0\ \textbf{then}\ Y ::= 1\ \textbf{else}\ Y ::= 0)\ \{Y = 0\}$$

In this example, consider the candidate counterexample comprising the two blue traces with total weights 1. After enumeration, both traces are found violating but incompatible. However, standard

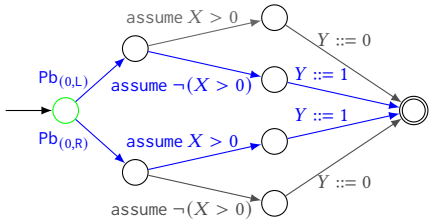


Fig. 11. PCFA of Example 5.5

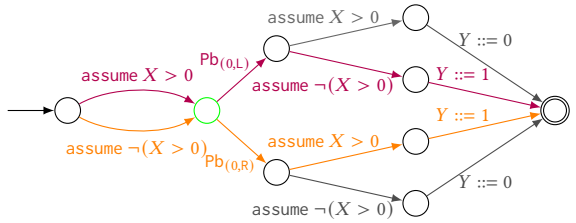


Fig. 12. PCFA of Example 5.5 with Split Conditions

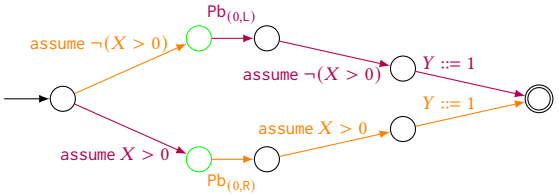


Fig. 13. Refined PCFA of Example 5.5 with Split Conditions

refinement cannot distinguish them: since V must contain *all* violating traces, no refinement automaton can separate these two traces. Hence, the CEGAR process diverges by repeatedly picking the blue candidate counterexample as it has structural upper bound $1 > 0.75 = \beta$. This issue also occurs when distinguishing traces with different values of C in Example 2.1. □

To resolve the kind of divergence as illustrated in the previous example, we propose "pre-condition splitting" to ensure traces with incompatible path conditions cannot appear in the same candidate counterexample again. Specifically, consider the previous example, we append the conflicting path conditions $\mathtt{assume}\ X > 0$ and $\mathtt{assume}\ \neg(X > 0)$ before the traces in Figure 11, yielding Figure 12. Consequently, both the red and orange traces now contain contradictory assumption conditions $\mathtt{assume}\ X > 0$ and $\mathtt{assume}\ \neg(X > 0)$, rendering them infeasible rather than violating. We call the above prepended conditions *split conditions*. These infeasible traces are subsequently eliminated during refinement, eventually producing the refined PCFA as shown in Figure 13. The resulting PCFA contains only two traces under different reason policies, yielding an exact structural upper bound of $0.5 \leq 0.75 = \beta$ and confirming safety. We generalise the above idea as follows.

PCFA with Split Conditions. Formally, a set of *split conditions* $\{\varphi_e^1, \ldots, \varphi_e^n\}$ partitions the pre-condition φ_e such that $\bigvee_{i=1}^n \varphi_e^i = \varphi_e$ and $\varphi_e^i \wedge \varphi_e^j$ is unsatisfiable for $i \neq j$. A PCFA with split conditions enforces that edges from the initial location ℓ_0 are labeled by $\mathtt{assume}\ \varphi_e^i$ for all split conditions φ_e^i and do not return to ℓ_0. This structure is used for the refinement automaton V in our framework, with the set of split conditions maintained on-the-fly during CEGAR. Initially, the set is the singleton $\{\varphi_e\}$. Specifically, when instantiating the framework with an initial refinement automaton V, we construct the actual automaton V' by introducing a fresh initial location ℓ_0' with edge $\mathtt{assume}\ \varphi_e$ leading to the original initial location.

This resembles introducing a fresh initial location with split condition edges as shown in Figure 12 versus Figure 11. By further refinement, the original location may be partitioned as in Figure 13, where green locations correspond to the original initial location in Figures 11 and 12.

Modified Intersection Operation. Incorporating split conditions into the refinement automaton V necessitates modifications to the intersection operation. Since V now contains split conditions while the original PCFA A does not, we adapt the intersection $A \cap V$ as follows: First, we synchronise

A with the split conditions by constructing $A' := (L \uplus \{\ell'_0\}, \Sigma, \Delta', \ell'_0, \ell_e)$ where ℓ'_0 is a fresh initial location and Δ' is additionally enlarged with the transitions $\ell'_0 \xrightarrow{\texttt{assume } \varphi^i_e} \ell_0$, where ℓ_0 is the original initial location of A, for each split condition φ^i_e currently in V. The intersection is then computed as $A' \cap V$. This renewed method is employed in computing the refined upper bound as in the left blue box of Figure 10.

Deriving Split Conditions. Split conditions are derived on-the-fly during the CEGAR process, tightly integrated with the MAX-SMT and enumeration procedures. Notably, as shown above, now with the split conditions, every candidate counterexample must have a unique split condition — by the definition of policies (in Section 4.1), and every `assume` statement must be an independent action. When a candidate counterexample is deemed spurious, let its current split condition be φ^i_e and the enumerated set of violating traces be Θ, we extract the path condition E of the maximum-weight compatible trace set of Θ from the MAX-SMT analysis. We then replace the original split condition φ^i_e with two refined conditions: $\varphi^i_e \wedge E$ and $\varphi^i_e \wedge \neg E$.

The above ideas are integrated into the refinement procedure below.

Refinement Procedure. With split condition derivation established, the refinement procedure in Figure 10 proceeds through three phases. Below let E denote the identified path condition from the MAX-SMT analysis as mentioned previously.

(1) *Split condition update:* Replace the transition $\ell_0 \xrightarrow{\texttt{assume } \varphi^i_e} \ell$ in both the current V and the spurious candidate counterexample with two transitions: $\ell_0 \xrightarrow{\texttt{assume } (\varphi^i_e \wedge E)} \ell$ and $\ell_0 \xrightarrow{\texttt{assume } (\varphi^i_e \wedge \neg E)} \ell$, where ℓ_0 is the initial location and ℓ is the destination location (intuitively representing the potentially partitioned original initial location).

(2) *Trace relabeling:* For refinement techniques requiring enumerated traces Θ, we update trace labels accordingly: traces in the maximum-weight set have their first statement `assume` φ^i_e replaced by `assume` $(\varphi^i_e \wedge E)$, while the remaining traces are updated with `assume` $(\varphi^i_e \wedge \neg E)$.

(3) *Refinement:* The updated objects — including V, the candidate counterexample, and Θ — are processed by the customisable refinement method to be introduced below to compute a new general PCFA V with split conditions.
Crucially, split conditions do not alter behaviors of refinement techniques adopted. Specifically, the refinement procedure can be intuitively described as: given a control-flow automaton (V in our case) and its subset of identified non-violating traces (from the candidate counterexample), refine the automaton to eliminate as much as possible more non-violating traces from V. This intuition is generally shared among standard non-probabilistic refinement techniques, e.g., predicate abstraction [14] and trace abstraction [28] to be exemplified below in Section 5.2. By substituting new split conditions (`assume` $(\varphi^i_e \wedge E)$ and `assume` $(\varphi^i_e \wedge \neg E)$) for `assume` φ^i_e, both the modified candidate counterexample and Θ remain subsets of $\mathcal{L}(V)$, allowing refinement methods to proceed unaware of split conditions.

Split conditions prevent re-enumeration of identical candidate counterexamples, as traces with path conditions contradicting the split condition become infeasible rather than violating, hence effectively reducing divergence. For finite-state programs like Example 5.5, since the number of potential split conditions is finite, this approach guarantees convergence of the CEGAR process.

Remark 1 (Methodology of Adapting the Framework). *Within the framework, most components are predetermined, leaving only* two *customisable elements: the construction of the initial V and the refinement method. As demonstrated in the principle of refinement in Theorem 4.7, the construction and refinement of V does not require handling probabilities as the probabilistic statements will not affect whether a trace is violating or not. This renders typically straightforward instantiations.*

To show our claim of the straightforwardness of the adaptation, in the following, we exemplify the instantiation of our framework with trace abstraction. We begin by a direct instantiation with the non-random technique briefly introduced and straightforwardly applied to adapt to the above methodology. Further, we explore a potential optimisation based on the direct instantiation, which further retains the *refutational completeness* property of trace abstraction, which is a guarantee hardly seen in probabilistic verification beyond finite-state (incl. bounded) techniques. In fact, we also investigated more instances of the instantiations, e.g., predicate abstraction and value analysis. As they are all direct applications of the well-established techniques, in the interests of space, the demonstration is within the extended version of this paper.

5.2 Direct Instantiation with Trace Abstraction

Next, we shall adapt the framework by *trace abstraction* [28], a successful non-probabilistic verification technique. In this part, again, we first present a brief account of the approach of trace abstraction and then shift to a discussion on the direct instantiation of the general CEGAR framework with the renowned technique.

Trace Abstraction. In the non-probabilistic context, trace abstraction over-approximates the non-violating traces of the original control-flow automaton (CFA) [10] A by a series of certified CFA $Q_1, \ldots, Q_n$ with $\mathcal{L}(A) \subseteq \bigcup_{i=1}^{n} \mathcal{L}(Q_i)$, where a certified CFA contains only the non-violating traces.

The CEGAR-driven refinement algorithm for trace abstraction entails constructing a series of *Floyd-Hoare automata*, which serve as the certified CFA that contains no violating trace. A Floyd-Hoare automaton is a (general) CFA $(L, \Sigma, \Delta, \ell_0, L_e)$, augmented with a predicate assignment function $\lambda : L \to \Phi$. For each transition $\ell \xrightarrow{\sigma} \ell'$, the condition represented by a Hoare triple [47] $\{\lambda(\ell)\}\ \sigma\ \{\lambda(\ell')\}$ must hold. Although previously mentioned, we here formally present an account for this concept. The validity of a Hoare triple $\{\varphi_1\}\ \sigma\ \{\varphi_2\}$ is given by that for all valuations $v \models \varphi_1$, the valuation $[\![\sigma]\!]_v$ either is $\bot$ or satisfies φ_2.

In the $(n+1)$-th iteration of trace abstraction refinement, a trace (candidate counterexample in the non-probabilistic scenario) is selected from the rest of the automaton $A \setminus \bigcup_{i=1}^{n} \mathcal{L}(Q_i)$, which is followed by an attempt to verify it. Should the trace indeed be violating, the counterexample is reported; otherwise, the trace is generalised into a Floyd-Hoare automaton Q_{n+1}. The iteration goes until the m-th round when $A \setminus \bigcup_{i=1}^{m} \mathcal{L}(Q_i) = \emptyset$.

In the above process, the generalisation process encompasses two steps: *interpolation* [13, 41] and the construction of a Floyd-Hoare automaton. More specifically, given a non-violating trace $\sigma_1 \ldots \sigma_n$, the interpolation of the trace produces a *tagged trace*, of form: $\{\varphi_0\}\ \sigma_1\ \{\varphi_1\}\ \ldots\ \{\varphi_{n-1}\}\ \sigma_n\ \{\varphi_n\}$, where the propositions φ_i are called *interpolants* and the Hoare triple $\{\varphi_{i-1}\}\ \sigma_i\ \{\varphi_i\}$ holds for every segment. At the same time, φ_e implies φ_0 and φ_n implies φ_f. Utilising the tagged trace, a Floyd-Hoare automaton is constructed by establishing a location set corresponding to each generated predicates $\{\varphi_0, \ldots, \varphi_n\}$ [11]. In what follows, the transitions $\ell \xrightarrow{\sigma} \ell'$ for σ in the statement set of the program CFA A are added, whenever the Hoare triple $\{\lambda(\ell)\}\ \sigma\ \{\lambda(\ell')\}$ holds.

Example 5.6 (Generalisation). Notably, this generalisation is directly applicable to our context. Referring back to traces 1 and 2, the interpolation process by the established tool SMTInterpol [13] returns the following tagged traces for Traces 1 and 2:

$$\{\texttt{True}\}\ X ::= 0\ \{X = 0\}\ \texttt{Pb}_{(0,\texttt{L})}\ \{X = 0\}\ C ::= 0\ \{X = 0\}\ \texttt{assume}\ \neg(C > 0)\ \{X = 0\} \tag{7}$$

[10] Simply given by (general) PCFA without probabilistic statements.
[11] Duplicate predicates are naturally removed.

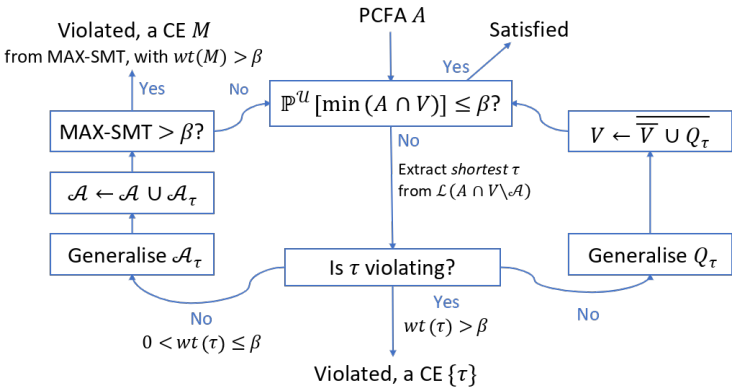


Fig. 14. Optimised CEGAR with Trace Abstraction

$$\begin{array}{l}\{\texttt{True}\}\ X ::= 0\ \{\texttt{True}\}\ \texttt{Pb}_{(0,\texttt{L})}\ \{\texttt{True}\}\ C ::= 0\ \{C \leq 0\}\ \texttt{assume}\ C > 0\ \{\texttt{False}\} \\ \texttt{Pb}_{(1,\texttt{R})}\ \{\texttt{False}\}\ \mathit{skip}\ \{\texttt{False}\}\ C ::= C - 1\ \{\texttt{False}\}\ \texttt{assume}\ \neg(C > 0)\ \{\texttt{False}\}\end{array} \tag{8}$$

Then, by introducing the location set for the interpolants for each tagged trace and subsequently adding the transitions, the generalised certified general PCFA Figs. 6 and 7 are obtained. □

Probabilistic Refinement with Trace Abstraction. The theoretical aim of the instantiation involves to construct a $V = \overline{\bigcup_{i=1}^{n} Q_i}$ where each Q_i is a certified (general) PCFA. Specifically:

- the initial V is the trivial PCFA that accepts all traces Σ^*; and,
- when a spurious counterexample is found, generalise the enumerated non-violating traces and union all the Floyd-Hoare automata obtained by generalisation from each trace into a PCFA Q. Next, update V in an incremental manner (so there is no need to keep every generated Q) by: $V \leftarrow \overline{\overline{V} \cup Q}$.

5.3 Optimised Instantiation with Trace Abstraction for Refutational Completeness

When we probe closer to the flexibility on examining each traces provided by trace abstraction, we discern that there is room for further optimisation. In this section, we shall dig deeper into the integration with trace abstraction to explore potential optimisations and their impact.

In general, the verification problem is undecidable, which, in light of our CEGAR-driven algorithm, implies that the refinement loop might never terminate. Nevertheless, we can still modify the algorithm to ensure termination and a valid counterexample when the problem is genuinely unsatisfiable; this is referred to as *refutational completeness*. It is worth noting that, this property suggests a theoretical guarantee on automatically proving an *arbitrarily close* lower bound.

To achieve this effect, it is worth noting the following trivial facts:

(1) The main objects we are dealing with are now single traces (rather than states), allowing us to consider each trace separately;
(2) The counterexample consists of only finite traces as declared in Lemma 5.1;
(3) There are only a finite number of traces of a certain length in any PCFA A; and,
(4) It is possible to decide the existence of a counterexample in a finite trace set.

Based on the facts, to achieve refutational completeness, two principles can be derived: (1) storing the enumerated violating traces in a specific storage, from and only from which counterexamples are sought; and, (2) preventing the re-enumeration of traces within the storage. As there are only

finite traces, the structure of the storage can be either a set of separated traces or a general PCFA, which does not hinder the operations below.

Optimised CEGAR Framework. Incarnating the principles, the direct instantiation can be optimised to form the algorithmic procedure depicted in Fig. 14. Compared to the direct instantiation, this new algorithm possesses some subtle differences.

The algorithm consists of two loops – the left loop modifies and examines the mentioned storage $\mathcal{A}$, while the right loop eliminates discovered non-violating traces. Initially, rather than extracting a reason policy directly, the process picks a *shortest* trace τ from $A \cap V \setminus \mathcal{A}$, thereby avoiding re-enumerating traces in the storage. Next, verify this single trace τ, from which three kinds of results may be yielded. If the trace is found to be non-violating, the standard generalisation process is employed to construct a certified general PCFA Q_τ. After this, the aforementioned updating procedure is applied to V. When τ is violating, a simple scenario occurs if its weight surpasses the threshold independently, for which we report a counterexample with this single trace.

A more intriguing scenario arises otherwise. As the examination of $\mathcal{A}$ is rather computationally intensive, a specialised generalisation procedure is also executed to add traces with "similar reasons for violation" to the discovered τ. This process, unlike the aforementioned generalisation for non-violating traces, faces two main hurdles. First, the interpolation process cannot be applied, which we'll explain in more detail below. Second, the process must avoid introducing loops to make use of Fact 4 mentioned earlier. In response to these challenges, a novel finite generalisation process for violating traces will be introduced below. For now, one just needs to know that the finite generalisation results in a (general) PCFA, denoted by $\mathcal{A}_\tau$, such that $\tau \in \mathcal{L}(\mathcal{A}_\tau)$ and there are only finite traces in $\mathcal{L}(\mathcal{A}_\tau)$.

With the finitely generalised automaton $\mathcal{A}_\tau$ for the picked trace τ, the traces are then integrated into the storage $\mathcal{A}$. Subsequently, the storage $\mathcal{A}$ retains only finite traces. Recall (in Section 5.1) that the existence of a potential counterexample for finite traces can be decided by encoding the problem in MAX-SMT [7, 33]. Also, one may notice that the direct instantiation of trace abstraction Section 5.2 is also complete for finite traces. Hence, adopting the analysis process may also be a possible choice for examining $\mathcal{A}$. As a result from the examinations, if a counterexample is identified, i.e., MAX-SMT produces a result $> \beta$, the subset identified by MAX-SMT is reported; otherwise, the process recommences from the beginning for the next iteration.

Finite Generalisation. We now turn our attention to the finite generalisation process for violating traces. The fundamental steps for this process resemble those for non-violating traces: one first computes a set of *tagged traces*, from which an automaton is subsequently built. However, the construction methods for these formalisms significantly *differ* from those used for generalising non-violating traces. Let us delve deeper into the details.

Tagging Traces by Weakest Pre-condition. The interpolation process used for non-violating traces cannot be applied when tagging violating traces. To understand why, consider the details of interpolation [31, 41]: for an *unsatisfiable* proposition $\varphi \wedge \varphi'$, comprising sub-propositions φ and φ', a proposition ψ exists such that: (1) φ implies ψ; (2) $\psi \wedge \varphi'$ is *unsatisfiable*; and (3) ψ contains only variables present in both φ and φ'. Given the *unsatisfiability* requirement, it is evident why this cannot be applied to violating traces, where reaching φ_f is, by definition, *satisfiable*.

To address this challenge, we use the *weakest pre-condition* technique, introduced by Dijkstra [16]. The tagging procedure involves computing backwardly from φ_f. The weakest pre-condition function $\mathsf{wp}(\sigma, \varphi)$ yields the weakest proposition that ensures φ after executing statement σ, defined as:

$$\mathsf{wp}(X ::= E, \varphi) := \varphi[X/E] \quad \mathsf{wp}(\texttt{assume}\ \varphi', \varphi) := \varphi' \to \varphi \quad \mathsf{wp}(\sigma, \varphi) := \varphi$$

Here, $\varphi[X/E]$ substitutes every occurrence of X in φ with E, and $\to$ denotes logical implication.

Given a violating trace $\sigma_1 \dots \sigma_n$ with pre- and post-conditions φ_e and φ_f, one can compute a tagged trace that: $\{\varphi_0\}\ \sigma_1\ \{\varphi_1\}\ \dots\ \{\varphi_{n-1}\}\ \sigma_n\ \{\varphi_n\}$, where: $\varphi_i := \mathsf{wp}(\sigma_{i+1}, \varphi_{i+1})$ for $i \in \{1, \dots, n-1\}$, with $\varphi_n := \varphi_f$ and $\varphi_0 := \varphi_e \wedge \mathsf{wp}(\sigma_1, \varphi_1)$.

Constructing Finitely Generalised Automaton. With tagged traces introduced, we can now construct the finitely generalised automaton. However, applying the original method directly remains infeasible due to the risk of introducing loops. To address this, we extend the usual Floyd-Hoare automaton by introducing the concept of *order*, resulting in the *ordered Floyd-Hoare automaton*. Consider a standard Floyd-Hoare automaton, (A, λ), consists of a PCFA A with location set L. An *ordered Floyd-Hoare automaton* extends this to a triple (A, λ, ρ), where $\rho : L \to \mathbb{N}$ assigns a unique priority to each location. For a transition $\ell \xrightarrow{\sigma} \ell'$ to belong to the automaton, it must satisfy both the usual condition $\{\lambda(\ell)\}\ \sigma\ \{\lambda(\ell')\}$ and the additional requirement $\rho(\ell) < \rho(\ell')$.

Given a tagged trace $\{\varphi_0\}\ \sigma_1\ \{\varphi_1\}\ \dots\ \{\varphi_{n-1}\}\ \sigma_n\ \{\varphi_n\}$, we construct an ordered Floyd-Hoare automaton as follows. First, define a location set $\{\ell_0, \dots, \ell_n\}$, whose size matches the length of the trace, unlike the non-violating case where locations correspond to distinct propositions. Next, assign $\lambda(\ell_i) := \varphi_i$ and $\rho(\ell_i) := i$ for each location ℓ_i. Set ℓ_0 as the initial location and ℓ_n as the final location. Finally, include all transitions satisfying the conditions for an ordered Floyd-Hoare automaton.

Refutational Completeness. We here first present a formal account for the property.

Theorem 5.7 (Refutational Completeness). *Given a PCFA A, a pre- and a post-condition φ_e and φ_f with a threshold β, if $\mathbb{P}\left[\nu\ \{\varphi_e\}\ A\ \{\varphi_f\}\right] > \beta$, then the process introduced above will terminate and report a valid counterexample.*

Given the above process, refutational completeness follows as below. When the violating probability surpasses the threshold, a counterexample with finite traces exists, per Fact 2. For any potential counterexample, let N be the length of its longest trace. By Fact 3, and since traces are not re-enumerated, there will be an iteration where the shortest trace in $\mathcal{L}(A \cap V \setminus \mathcal{A})$ exceeds N. Thus, all violating traces constituting the valid example will be stored in $\mathcal{A}$. Finally, by Fact 4, this counterexample must be detected through examining storage $\mathcal{A}$.

6 Empirical Evaluation

To evaluate the effectiveness and efficiency of our proposed approach, we implemented our framework instantiated with trace abstraction. We tested it on a diverse set of examples and compared its performance with state-of-the-art tools.

Implementation. Our prototype, developed in Java and Scala, integrates high-performance libraries to tackle key challenges. For satisfiability modulo theories (SMT), we employed SMTInterpol [13], selected for its robust interpolation capabilities vital to trace abstraction. Automata manipulation is handled by the flexible Brics [45] library, while Breeze [24] facilitates computationally intensive numerical tasks, such as MDP and MC reachability analysis. An input file is a program like that in Example 2.1, which is internally parsed and converted to PCFA.

Baseline Tools. We compared our prototype against two state-of-the-art tools for violation bound analysis: cegispro2 [5] and PSI [20]. Cegispro2, a recent advancement based on fixed-point and pre-expectation techniques, has shown superior performance over tools such as Exist [2] and storm [30] in prior studies. PSI, a well-established tool, is known for its precise bounds, fast computation, and robust handling of both discrete and continuous random variables. We assessed the time performance of these tools using identical validation objectives and benchmark examples. Notably, tools and benchmarks from [33, 54], though technically relevant, were not included in the evaluation due to fundamental compatibility issues. Specifically, [33] takes a PRISM-variant input

format, hard to convert from and to probabilistic programs. While [54] requires manually provided invariants, hence unsuitable to compare with fully automated tools like ours, CegisPro2, and PSI.

Benchmarks. We identify four challenging features when selecting benchmarks:

(P1) Unbounded loops (i.e., loops with an unbounded number of iterations),
(P2) Bounded loops with over 100 times of iterations,
(P3) Loops with over 5 conditional and probabilistic branches in the body, and
(P4) Sequential or nested loops.

Following these criteria, we curated a diverse and representative set of examples from the literature, including:

- We tested *all loopy* examples from [50] [12], a comprehensive benchmark suite with both simple and complex cases featuring intricate branching structures. This suite remains widely adopted in recent tools [55, 57].
- We tested *all* assertion benchmarks from CegisPro2 [5], which collected examples from diverse literature sources and compared against multiple state-of-the-art tools. However, we found that their suite has limited diversity, consisting mostly of single loops with simple bodies and restricted number of iterations.
- To enrich our test set, we incorporated additional loopy examples from broader literature, including PSI [20], Exist [2], Amber [43], [8, 52], [54], and Diabolo [57]. Some derive from real-world applications: "Reliability" from [8, 52] models a pixel-block search algorithm from x264 video encoders with hardware failures; "Herman3" from [57] implements the probabilistic self-stabilisation algorithm [32] with three threads; "Coupon5" from [57] represents the classical coupon collector problem [19], applied in commercial analysis [56] and biology [11].
- Finally, to address the lack of nested loop benchmarks, we developed "LimitV" and "LimitVP" by modifying the "Limit" example from Example 2.1, featuring nested loops and unbounded state spaces.

Overall, the experimental dataset consists of 37 examples from diverse sources. In these benchmarks, continuous distributions were adapted to the tools by discretization. All tests were conducted on a MacBook Pro (M4 Pro, 48GB RAM).

Experimental Evaluation. The experimental results for examples from [50] are presented in Table 1, while results for the other examples are summarized in Table 2. For each example, we tested multiple parameter settings with different proof bounds (β). Following established practices [5, 20], we preserved original thresholds from CegisPro2's benchmark suite and tested additional tighter thresholds. For examples where PSI computed exact solutions, we selected β values near the exact bounds. For remaining examples, we used the standard guess-and-refine approach, commonly employed in the field. Importantly, all threshold selections are consistent across tools for fair comparison.

The results first of all confirm our tool's correctness through consistency with CegisPro2 and PSI for both upper and lower bounds. In contrast, CegisPro2 only reports whether bounds are proved and will crash or fail to terminate for improper bounds, unlike our tool which extracts counterexamples.

Statistically, we tested 68 verification tasks across both tables. Our method handles 55 tasks, timing out in 11, with 2 tasks ("InvPend" and "Vol") beyond our current integer-only implementation. These two cases also exceed CegisPro2's capacity due to negative numbers, a theoretical limitation as per [5], and cause PSI timeouts due to excessive iterations. In comparison, CegisPro2 handles 22 tasks, while PSI handles only 4. Our method demonstrates superior performance (in terms of time)

[12] Available at https://github.com/eth-sri/psi/tree/master/test/colorado.

Table 1. Examples from [50]: Each example is tested with multiple parameters, where "UB" denotes unbounded; for each parameter, different "Bound" values (i.e., threshold β) are tested, forming a *verification task*. "Type" indicates which criteria the example satisfies. In "Time" columns, numbers represent seconds, "TO" denotes time-out (over 500s), "N/A" means not applicable; bold font shows the fastest time among methods. In our "Result" column, "SAT" indicates safety of the bound, "UnSAT(n)" indicates violation with n counterexample traces. CegisPro2's "Result" is simpler: it proves or reports errors. When the bound is a proper upper bound, we denote this by ✓; for lower bounds, we use the tool's "lower bound" functionality, denoting success by "✓ (Lower)". PSI's "Prob" column shows exact violation probability values.

Example Name	Parameter	Type	Bound	Our Method		CegisPro2		PSI	
				Time (s)	Result	Time (s)	Result	Time (s)	Prob
BeauquierEtal3	count<1	P1, P3	0.6	**0.281**	SAT	N/A	-	N/A	-
Cav1	x>250	P1	0.6	**84.971**	SAT	TO	-	N/A	-
	x>187.5	P1	0.1	**51.799**	SAT	TO	-	N/A	-
Cav2	h-t>4	P1	0.05	**1.329**	UnSAT (4)	16.453	✓ (Lower)	N/A	-
			0.06	**1.891**	UnSAT (7)	41.043	✓ (Lower)	N/A	-
Cav3	x-estX>10	P3	0.5	TO	-	TO	-	TO	-
Cav4	x<=3	P1	0.1	0.702	SAT	**0.65**	✓	N/A	-
	x<=8	P1	0.01	**1.285**	SAT	1.612	✓	N/A	-
Cav5	i>10	P1, P3	0.5	**3.978**	SAT	TO	-	N/A	-
			0.1	**0.898**	UnSAT (8)	TO	-	N/A	-
Cav6	N=3	P3	0.58	**0.836**	SAT	N/A	-	TO	-
	N=4	P3	0.56	**5.619**	SAT	N/A	-	TO	-
	N=5	P3	0.55	**103.212**	SAT	N/A	-	TO	-
Cav7	count>10	P1	0.3	**223.926**	UnSAT (229944)	TO	-	N/A	-
			0.5	TO	-	TO	-	N/A	-
BookMod	count<=20	P1, P3	0.001	**4.402**	SAT	N/A	-	N/A	-
	count<=40	P1, P3	0.00025	**19.562**	SAT	N/A	-	N/A	-
Cart	count<5	P1, P3	0.5	TO	-	TO	-	N/A	-
Carton5	count<8	P1, P3	0.2	**3.754**	SAT	TO	-	N/A	-
	count<7	P1, P3	0.2	**33.327**	SAT	TO	-	N/A	-
	count<6	P1, P3	0.2	**199.67**	UnSAT (820)	TO	-	N/A	-
	count<5	P1, P3	0.2	**22.979**	UnSAT (205)	TO	-	N/A	-
Fig6	c>5	P1	0.5	**21.422**	UnSAT (572)	TO	-	N/A	-
			0.9	**34.11**	SAT	TO	-	N/A	-
Fig7	x<=1000	P1	0.0015	3.422	UnSAT (3)	**0.515**	✓ (Lower)	N/A	-
			0.00198	**1.607**	SAT	53.183	✓	N/A	-
InvPend	pAng>100	P3	-	N/A	-	N/A	-	TO	-
LoopPerf	guardOK>2	P3, P4	0.5	TO	-	N/A	-	TO	-
Vol	count>3	P2, P3	-	N/A	-	N/A	-	TO	-
Herman	count <5	P2, P3	0.7	**0.549**	SAT	N/A	-	TO	-
Israeli-jalfon-3	count>=1	P1, P3	0.3	**0.741**	UnSAT (3)	64.455	✓ (Lower)	N/A	-
			0.6	**0.688**	SAT	82.902	✓	N/A	-
Israeli-jalfon-5	count>=1	P1, P3	0.3	**0.741**	UnSAT (3)	64.455	✓ (Lower)	N/A	-

in 50 tasks, while CegisPro2 excels in 7 and PSI in 2. When our tool reports results, it is typically much faster — over 100× faster than CegisPro2 in "Israeli-jalfon-3" and over 800× faster than PSI in "Coupon5" — not to mention they are often TO while we report results in a few seconds.

Delving deeper into the types of the tasks. Observably, CegisPro2 performs well mostly in verification tasks falling into (P2) with relatively simple loop bodies. However, their method is

Table 2. Other Examples: Columns have the same meaning as Table 1

Example Name	Parameter	Type	Bound	Our Method		CegisPro2		PSI	
				Time (s)	Result	Time (s)	Result	Time (s)	Prob
RwMultiStep	UB	P1	0.45	**3.782**	SAT	TO	-	N/A	-
			0.39	**2.123**	UnSAT (4)	TO	-	N/A	-
	200	P2	0.3	154.691	SAT	**10.543**	✓	TO	-
brp	8000000000	P2	0.001	TO	-	**16.218**	✓	TO	-
			0.00001	TO	-	**10.095**	✓	TO	-
	200	P2	0.0001	**2.609**	SAT	3.314	✓	TO	-
Chain	10000	P2	0.4	**83.136**	SAT	TO	-	TO	-
ChainStepSize	10000000	P2, P3	0.7	TO	-	**11.279**	✓	TO	-
			0.6	TO	-	TO	-	TO	-
			0.55	TO	-	TO	-	TO	-
	200	P2, P3	0.8	**0.652**	SAT	19.623	✓	TO	-
EqualProbGrid	UB	P1	0.6	**0.272**	SAT	TO	-	N/A	-
			0.4	**1.374**	UnSAT (3)	TO	-	N/A	-
Geo	x>3	P1	0.3	**0.584**	UnSAT (1)	1.028	✓ (Lower)	N/A	-
	x>5	P1	0.9	**0.359**	SAT	1.797	✓	N/A	-
Grid	100	P2	0.93	**5.061**	SAT	TO	-	12.865	0.5
ZeroConf	100000000	P2	0.53	TO	-	**17.806**	✓	TO	-
			0.526	TO	-	TO	-	TO	-
	200	P2	0.6	**0.138**	SAT	4.844	✓	TO	-
Coupon5	count<100	P2	1.5E-9	**0.538**	SAT	1.451	✓	491.158	1.273E-9
Herman3	count<100	P2, P3	0.01	**0.304**	SAT	28.064	✓	TO	-
Reliability	unrel>0	P3, P4	0.01	**5.434**	SAT	N/A	-	TO	-
SeqLoop	y >3	P4	0.3	**0.304**	SAT	N/A	-	TO	-
NestedLoop	y >1030	P4	0.4	**194.848**	SAT	N/A	-	TO	-
Limit	X == 0	P1	0.5	**0.15**	SAT	N/A	-	N/A	-
LimitV	X == 0	P3, P4	0.4	**5.052**	UnSAT (7)	N/A	-	N/A	-
			0.5	**0.182**	SAT	N/A	-	N/A	-
LimitVP	X == 0	P3, P4	0.3	**2.376**	UnSAT (3)	N/A	-	N/A	-
			0.45	**0.275**	SAT	N/A	-	N/A	-
Birthday	UB	P1, P3	0.5	**26.574**	UnSAT (361)	TO	-	N/A	-
			0.7	**45.301**	UnSAT (1561)	TO	-	N/A	-
			0.99	**409.902**	UnSAT (43888)	TO	-	N/A	-
	3	P3	0.4	17.002	SAT	TO	-	**3.703**	0.388
	5	P3	0.7	46.692	UnSAT (1261)	TO	-	**43.231**	0.85
Cards	sum != 1000	P2	0.3	**0.563**	SAT	TO	-	TO	-

highly sensitive to the structure of the loop body, loop conditions and the distance to the real value, and hence, for examples falling into other categories, it is easily overwhelmed. The data also shows that these challenging cases are difficult for PSI, which is a renowned tool for *exact* solution computation. It is hence solely capable of tackling bounded loops stemming from its theoretical foundation, while from the experimental results, it is also not working swiftly for bounded loops with relatively large iterations. For examples falling into (P1), the unrolling-based method struggles to handle the situation.

In contrast, the experimental results highlight the advantages of our method in addressing more general and versatile cases. Specifically, the data demonstrates that our method exhibits evident superior performance in handling examples of type (P1). It also typically resolves instances of (P2)

efficiently with reasonable loop counts, while showing greater robustness to variations in loop bodies and conditions. This is evidenced by the fact that our method outperforms CegisPro2 in 9 tasks for (P2), compared to only 5 tasks where the opposite is true. Moreover, our method robustly handles the more complex and general cases (P3) and (P4), which are particularly challenging for CegisPro2 and PSI, especially when combined with (P1) and (P2). Essentially, for (P3) and (P4), thanks to the separation of concerns offered by the refinement principle, the complex structures, including nested and sequential loops, are seamlessly handled by the established method of trace abstraction, which has shown its excellence in non-random verification.

Beyond efficiency, our method is often capable of computing tight bounds. For example, in the "Grid" and "Coupon5" cases in Table 2, the bounds computed by our method are very close to the exact values computed by PSI, and these bounds are swiftly provable by our method. In fact, our method can even prove the upper bound of "Reliability" in Table 2 by 0.0016 as compared to the original 0.0072 produced in [52] and 0.0051 from Rely [8]. Also, the theoretical refutational completeness also proves itself in the experimental data; e.g., when unbounded, the "Birthday" example should have probability 1 of violating, then even when we set the threshold to be 0.99, our tool still successfully reports the counterexample.

In summary, our method demonstrates superior performance, effectively addressing a broader range of cases and excelling in handling complex program structures. Even in scenarios where other tools are applicable, our method offers greater stability and efficiency, successfully covering a substantial portion of the combined capabilities of cegispro2 and PSI.

7 Related Work

Automaton and MDP. When integrating graphical program representation via Control Flow Automata (CFA) or Control Flow Graph (CFG) with Markov Decision Processes (MDP), the most straightforward approach is to use MDP as the program's semantics. This is achieved by unrolling the CFA / CFG while exploring the program's concrete state space. This principle supports probabilistic model checking [30, 39], a successful lightweight probabilistic program analysis technique. Practical tools like PRISM [37] have been developed with industrial strength, successfully deployed across diverse domains, some beyond computer science [17, 26, 38, 46]. Additionally, bounded model checking for probabilistic programs [35] has been developed, analysing programs with potentially infinite valuations. In this work, we propose a different approach to integrate PCFA with MDP. This method, entirely structural / syntactic, converts a PCFA into an MDP by erasing computational semantics, rather than unrolling the potential state space.

Probabilistic CEGAR. The study conducted by Hermanns et al. [33] bears a significant connection to our work. They explored the extension of *predicate abstraction* within a probabilistic setting using a CEGAR-guided algorithm [14]. We drew inspiration from their work for our broad CEGAR framework and the idea of counterexamples, as well as their method of verification through enumeration. Despite these similarities, the theoretical foundation is the key divergent point between our projects with theirs. Given the immense flexibility provided by the new refinement principle we developed in Theorem 4.7, we are able to take one more step towards generalising the framework to a versatile one that is capable of being straightforwardly instantiated by various non-probabilistic techniques, including the predicate abstraction considered in their work. Especially, we discuss the extension with trace abstraction, which could hardly cooperate with their method.

Trace Abstraction and Probabilities. Trace abstraction [28, 49] is a key technique in non-probabilistic program verification. It views a program as a set of traces rather than a state space, leading to award-winning tools like [27]. Beyond imperative program verification, it has applications in domains like termination analysis [29]. For combining trace abstraction with probabilities,

pioneering work by Smith et al. [52] addressed this via program synthesis. Unlike our method, which exposes probabilities at the control-flow level, Smith et al. encapsulated them at the statement level. For probabilistic distributions on statements, symbolic representations of violation probabilities are synthesised for candidate counterexample traces. Due to their handling of probabilities, traces remain probabilistically independent, akin to the non-probabilistic case. This approach, however, places the burden of handling distributions on each trace, limiting the use of existing non-probabilistic verification techniques. Moreover, their use of the union bound [3] also impedes completeness, even for finite trace examples like $\mathbb{P}\left[\lightning \{\texttt{True}\}\ X \sim \mathit{flip}; Y \sim \mathit{flip}\ \{X \wedge Y\}\right]$. Besides, trace abstractions are also applied in termination analysis, e.g., Chen et al. [12] aim to automatically prove almost sure termination (AST) by decomposing programs into certified ω-regular automata.

Probabilistic Threshold Problem. The problem considered in this work, determining whether $\mathbb{P}\left[\lightning \{\varphi_e\}\ P\ \{\varphi_f\}\right] \leq \beta$, has been addressed by other works. Smith et al. [52] also consider this problem. Wang et al. [54] analyse violation probabilities by synthesising bounds with exponential templates. This problem can also be tackled using pre-expectation calculus and expectation-transformer semantics [23, 40], an elegant probabilistic counterpart of Dijkstra's predicate-transformer semanticsf in his seminal work [16]. This approach has been exploited recently in CegisPro2 [5]. The semantics computes expectations, which can elegantly compute violation probabilities [40]. The key obstacle to automation, however, is the same as predicate-transformer semantics – synthesising (probabilistic) invariants, which works like [4] aim to overcome. We believe incorporating such information in the interpolation process could enhance interpolant quality, potentially resolving previously unsolvable problems or improving solving time.

Static Analysis of Bayesian Programs. Besides the topics, there is a rising trend in the analysis of the posterior distribution of Bayesian programs, which additionally equip usual probabilistic programs with the `observe` structure to express conditioning. Various techniques have been proposed to address this novel problem, including typing [6], fixed-point & Optional Stopping Theorem [55], and novel semantics [57].

8 Conclusion

This work introduces structural abstraction for verifying the threshold query for probabilistic programs. In this approach, transitions in a Probabilistic Control-Flow Automaton (PCFA) are treated as pure labels, abstracting the semantics and enabling a view of the PCFA as a Markov Decision Process (MDP). From the theoretical establishment of this abstraction and the principles of refinement, our method shows the ability to separate probability from semantics. We hence developed a general CEGAR framework capable of leveraging non-random techniques, for which we demonstrated the instantiation via trace abstraction, with a further optimised refutationally complete method. Our implementation, benchmarked against advanced tools on diverse examples, highlights its adaptability and superiority in various scenarios.

Acknowledgments

We would like to thank Prof. C.-H. Luke Ong and the anonymous reviewers for their helpful comments and discussions. This research was supported in part by Shanghai Trusted Industry Internet Software Collaborative Innovation Center, the National Natural Science Foundation of China (No. 62072267, No. 62021002 and No. 62172271) and the National Research Foundation, Singapore, under its RSS Scheme (NRF-RSS2022-009).

Data Availability Statement

All data, source code, and compiled binaries reported in the experimental evaluation are publicly available at https://zenodo.org/records/15760713 (with DOI 10.5281/zenodo.15760713).

References

[1] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT press.

[2] Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. 2022. Data-driven invariant learning for probabilistic programs. In *International Conference on Computer Aided Verification*. Springer, 33–54.

[3] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. A program logic for union bounds. *arXiv preprint arXiv:1602.05681* (2016).

[4] Ezio Bartocci, Laura Kovács, and Miroslav Stankovič. 2019. Automatic generation of moment-based invariants for prob-solvable loops. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 255–276.

[5] Kevin Batz, Mingshuai Chen, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2023. Probabilistic program verification via inductive synthesis of inductive invariants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 410–429.

[6] Raven Beutner, C-H Luke Ong, and Fabian Zaiser. 2022. Guaranteed bounds for posterior inference in universal probabilistic programming. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 536–551.

[7] Nikolaj S Bjørner and Anh-Dung Phan. 2014. *νZ*-Maximal Satisfaction with Z3. *Scss* 30 (2014), 1–9.

[8] Michael Carbin, Sasa Misailovic, and Martin C Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. *ACM SIGPLAN Notices* 48, 10 (2013), 33–52.

[9] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of statistical software* 76, 1 (2017), 1–32.

[10] Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic program analysis with martingales. In *International Conference on Computer Aided Verification*. Springer, 511–526.

[11] Anne Chao. 1984. Nonparametric estimation of the number of classes in a population. *Scandinavian Journal of statistics* (1984), 265–270.

[12] Jianhui Chen and Fei He. 2020. Proving almost-sure termination by omega-regular decomposition. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 869–882.

[13] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. 2012. SMTInterpol: An interpolating SMT solver. In *International SPIN Workshop on Model Checking of Software*. Springer, 248–254.

[14] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*. Springer, 154–169.

[15] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[16] Edsger W Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457.

[17] M. Duflot, M. Kwiatkowska, G. Norman, and D. Parker. 2004. A Formal Analysis of Bluetooth Device Discovery. In *Proc. 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*.

[18] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography: Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006. Proceedings 3*. Springer, 265–284.

[19] Pál Erdős and Alfréd Rényi. 1961. On a classical problem of probability theory. *A Magyar Tudományos Akadémia Matematikai Kutató Intézetének Közleményei* 6, 1-2 (1961), 215–220.

[20] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact symbolic inference for probabilistic programs. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I 28*. Springer, 62–83.

[21] Shafi Goldwasser and Silvio Micali. 1984. Probabilistic encryption. *Journal of computer and system sciences* 28, 2 (1984), 270–299.

[22] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. 2012. Church: a language for generative models. *arXiv preprint arXiv:1206.3255* (2012).

[23] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. 2014. Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. *Performance Evaluation* 73 (2014), 110–132.

[24] David Hall, Daniel Ramage, Jason Zaugg, Alexander Lehmann, Jonathan Merritt, Keith Stevens, Jason Baldridge, Timothy Hunter, Dave DeCaprio, Daniel Duckworth, et al. 2009. ScalaNLP: Breeze. https://github.com/scalanlp/breeze

[25] Tingting Han, Joost-Pieter Katoen, and Damman Berteun. 2009. Counterexample generation in probabilistic model checking. *IEEE transactions on software engineering* 35, 2 (2009), 241–257.

[26] J. Heath, M. Kwiatkowska, G. Norman, D. Parker, and O. Tymchyshyn. 2008. Probabilistic model checking of complex biological pathways. *Theoretical Computer Science* 319, 3 (2008), 239–257.

[27] Matthias Heizmann. [n.d.]. Uni-Freiburg : SWT - Ultimate. https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=tool&tool=automizer November 02, 2021.
[28] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2009. Refinement of trace abstraction. In *International Static Analysis Symposium*. Springer, 69–85.
[29] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2014. Termination Analysis by Learning Terminating Programs. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 797–813.
[30] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. 2022. The probabilistic model checker Storm. *International Journal on Software Tools for Technology Transfer* (2022), 1–22.
[31] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L McMillan. 2004. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 232–244.
[32] Ted Herman. 1990. Probabilistic self-stabilization. *Inform. Process. Lett.* 35, 2 (1990), 63–67.
[33] Holger Hermanns, Björn Wachter, and Lijun Zhang. 2008. Probabilistic cegar. In *International Conference on Computer Aided Verification*. Springer, 162–175.
[34] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.
[35] Nils Jansen, Christian Dehnert, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Lukas Westhofen. 2016. Bounded Model Checking for Probabilistic Programs. In *Automated Technology for Verification and Analysis*, Cyrille Artho, Axel Legay, and Doron Peled (Eds.). Springer International Publishing, Cham, 68–85.
[36] Laura Kallmeyer and Wolfgang Maier. 2013. Data-driven parsing using probabilistic linear context-free rewriting systems. *Computational Linguistics* 39, 1 (2013), 87–119.
[37] Marta Kwiatkowska, Gethin Norman, and David Parker. 2002. PRISM: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 200–204.
[38] M. Kwiatkowska, G. Norman, and D. Parker. 2012. Probabilistic Verification of Herman's Self-Stabilisation Algorithm. *Formal Aspects of Computing* 24, 4 (2012), 661–670.
[39] Marta Kwiatkowska, Gethin Norman, and David Parker. 2018. Probabilistic model checking: Advances and applications. *Formal System Verification: State-of the-Art and Future Trends* (2018), 73–121.
[40] Annabelle McIver, Carroll Morgan, and Charles Carroll Morgan. 2005. *Abstraction, refinement and proof for probabilistic systems*. Springer Science & Business Media.
[41] Kenneth L McMillan. 2006. Lazy abstraction with interpolants. In *International Conference on Computer Aided Verification*. Springer, 123–136.
[42] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press.
[43] Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. 2021. The probabilistic termination tool amber. In *International Symposium on Formal Methods*. Springer, 667–675.
[44] Rajeev Motwani and Prabhakar Raghavan. 1995. *Randomized algorithms*. Cambridge university press.
[45] Anders Møller. 2012. dk.brics.automaton. https://www.brics.dk/automaton/
[46] G. Norman, D. Parker, M. Kwiatkowska, S. Shukla, and R. Gupta. 2003. Using Probabilistic Model Checking for Dynamic Power Management. In *Proc. 3rd Workshop on Automated Verification of Critical Systems (AVoCS'03) (Technical Report DSSE-TR-2003-2, University of Southampton)*, M. Leuschel, S. Gruner, and S. Lo Presti (Eds.). 202–215.
[47] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2022. *Logical Foundations*. Software Foundations, Vol. 1. Electronic textbook.
[48] Michael O Rabin. 1976. *Probabilistic Algorithms Algorithms and Complexity: New Directions and Recent Results*. Academic Press New York.
[49] Bat-Chen Rothenberg, Daniel Dietsch, and Matthias Heizmann. 2018. Incremental Verification Using Trace Abstraction. In *Static Analysis*, Andreas Podelski (Ed.). Springer International Publishing, Cham, 364–382.
[50] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 447–458. https://doi.org/10.1145/2491956.2462179
[51] Eugene S Santos. 1969. Probabilistic Turing machines and computability. *Proceedings of the American mathematical Society* 22, 3 (1969), 704–710.
[52] Calvin Smith, Justin Hsu, and Aws Albarghouthi. 2019. Trace abstraction modulo probability. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–31.
[53] David Tolpin, Jan Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. *arXiv preprint arXiv:1608.05263* (2016).

[54] Jinyi Wang, Yican Sun, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. 2021. Quantitative analysis of assertion violations in probabilistic programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1171–1186.

[55] Peixin Wang, Tengshun Yang, Hongfei Fu, Guanyan Li, and C-H Luke Ong. 2024. Static posterior inference of Bayesian probabilistic programming via polynomial solving. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1361–1386.

[56] Jaegeol Yim. 2016. Design of a smart coupon system. *International Journal of Multimedia and Ubiquitous Engineering* 11, 3 (2016), 187–198.

[57] Fabian Zaiser, Andrzej S Murawski, and C-H Luke Ong. 2025. Guaranteed Bounds on Posterior Distributions of Discrete Probabilistic Programs with Loops. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 1104–1135.

Received 2025-03-26; accepted 2025-08-12