

PROPER:a Tool for Analyzing Termination and Assertions for Affine Probabilistic Programs

Xuhui Zhao¹[0000–1111–2222–3333], Yuxin Deng²[1111–2222–3333–4444], and
Hongfei Fu³[2222–3333–4444–5555]

¹ East China Normal University, Shanghai, China
51184501088@stu.ecnu.edu.cn

² East China Normal University, Shanghai, China
yxdeng@sei.ecnu.edu.cn

³ Shanghai Jiao Tong University, Shanghai, China
fuhf@cs.sjtu.edu.cn

Abstract. Probabilistic programs combine probabilistic reasoning models with Turing complete programming languages, unify formal descriptions of calculation and uncertain knowledge, and can effectively deal with complex relational models and uncertain problems. In this paper, we provide a tool for analyzing termination and assertions for affine probabilistic programs, PROPER. On one hand, it can help to analyze the termination property of affine probabilistic programs both qualitatively and quantitatively. It can check whether a probabilistic program terminates with probability 1, estimate the upper bound of expected termination time, and calculate the number of steps after which the termination probability of the given probabilistic program decreases exponentially. On the other hand, it can estimate the correct probability interval for a given assertion to hold, which helps to analyze the influence of uncertainty of variables on the results of probabilistic programs. Our work is implemented and the effectiveness of PROPER is demonstrated through various affine probabilistic programs.

Keywords: Probabilistic Programming · Program Verification · Termination · Ranking Supermartingale.

1 Introduction

In traditional precise logic, the representation and reasoning of knowledge is mechanical and deterministic, just like Demon de Laplace, capable of inferences about the future based on the known facts. But in real world, there are a large number of uncertainty problems, which require us to use prerequisite knowledge and deductive reasoning to predict the results, that is to say, to make decisions on nondeterministic problems through probabilistic reasoning. Probability is an important tool for the representation of imprecise and incomplete knowledge. Therefore, probabilistic programs are put forward for that purpose.

Probabilistic programs are a kind of logic programs with probabilistic facts. Probabilistic programs are often used to build generation models and reason

about its hidden process. They make probabilistic reasoning models easier to build and can estimate the possibilities of certain events to occur. They can also be used to quantify the uncertainty in the prediction instead of predicting a single value. Several practical probabilistic programming languages (such as Edward [9], Anglican [7], WebPPL [17] and Church [16]) now exist, they can build powerful models and are used in a wide range of applications, such as business, military, scientific research and daily life. Probability is becoming more and more important in actual calculations, such as risk analysis, medical decision-making, differential privacy mechanisms, etc. With the increasing demand for high confidence software, the analysis and verification of probabilistic programs has also received widespread attention in academia and industry. Therefore, our work is a valuable and challenging, which the reliability and stability of the program can be better guaranteed. Program verification is an incalculable problem in computer theory. In view of this difficulty, a realistic consideration is that we only verify those properties that are more important and can be verified automatically. In the current work, we present PROPER, a tool for probabilistic program verification, namely, analyzing termination and assertions for affine probabilistic programs (probabilistic programs whose all arithmetic expressions are linear). The tool builds on [5, 6, 18].

Currently, due to the demand for high-confidence software, verification of program termination is an important part of software design and development. Termination analysis is an important part of software complete correctness. Failure to terminate the program may cause the system to become unresponsive, resulting in system failure. For termination analysis, our goal is to analyze the termination property of a given affine probabilistic program qualitatively and quantitatively. The qualitative analysis aims to answer whether an affine program terminates with probability 1 (almost-sure termination). The quantitative analysis aims to approximate the expected termination time (expectation problem) and compute a bound N such that the probability to terminate after N steps decreases exponentially (concentration problem). The method we adopt is to synthesis of polynomial ranking supermartingales, which is an extension of linear ranking-supermartingale.

For assertion analysis, our goal is to estimate the correct probability interval of the assertion that meets the given specifications in an affine probabilistic program. It has great potential value in artificial intelligence fields such as incomplete knowledge processing systems, meta-reasoning, cognitive query, and conformant probability planning. The main application scenarios include fuzzy information description, incomplete information for reasoning or noise in information. We observe that for probabilistic programs, by selecting a finite set of appropriate paths, facts about the behavior of the entire program can be derived. Therefore, we used the static analysis method for probabilistic programs that can perceive, manipulate and control based on uncertain data. Firstly, We select a finite set of adequate program paths through certain strategies for estimating probability bounds. Then evaluate each path through symbolic execution and

probabilistic volumebound calculations. The sum of interval boundaries generated by each path is the probability of assertion for the program as a whole.

To summarize, this paper makes the following contributions:

1. Defining and parsing probabilistic programs. More than ten kinds of probability distributions are built in, which is convenient to be called to build probabilistic models.
2. Reducing the termination analysis of a given program to a linear programming problem. We also consider the concentration results under the premise that the program is terminating.
3. Reducing the estimation of a given assertion to a polyhedron solving problem. In particular, it can handle probabilistic programs with infinitely many states.

The source code and the experimental data given in this paper are all available in the repository <https://github.com/Healing1219/PROPER>.

2 Preliminaries

Probabilistic programs is extending classical imperative programs, which can be constructed by generating integer and real random values according to a fixed set of distributions. In this subsection, we illustrate the syntax and semantics of affine probabilistic programs that we study.

```

program := typeSpecifier main{stmt*}
stmt := assign | condStmt | while
assign := intAssign | realAssign
condStmt := ifStmt | ifElseStmt
while := while (test) stmt*
intAssign := intVar = intConst | intVar ~ intRandom
realAssign := realVar = realConst | realVar ~ realRandom
intRandom := uniformInt(intConst, intConst)
           — Bernoulli(intConst, intConst)
           ...
realRandom := uniformReal(realConst, realConst)
           — Gaussian(realConst, realConst)
           ...
intExpr := intConst | intRandom | intExpr ± intExpr
          intConst* intExpr | intExpr / intConst
realExpr := realConst | realRandom | realExpr ± realExpr
          realConst* realExpr | realExpr / realConst
boolExpr := true | false | boolExpr ∧ boolExpr
           intExpr relop intExpr | realExpr relop realExpr
relop := | | < | ≥ | ≤ | ==

```

Fig. 1: Syntax specification of a probabilistic language

Syntax. In order to analyze and verify probabilistic programs, we first need a probabilistic language sufficiently expressive and easy to understand. We define a probabilistic language, whose syntax specification is shown in Figure 1. The statements in the probabilistic language are similar to those in classic imperative languages, mainly composed of three types of statements: assignment, condition-branch (if-else) and loop statements (while). The main difference is that we now have a collection of random value generators (discrete distribution, Binomial distribution, Poisson distribution, Integer Uniform distribution, Real Uniform distribution, Exponential distribution, Gamma distribution, Beta distribution, Geometric distribution), which can be used to simulate different probability distributions. Variables are classified into two types: program variables X and sampling variables R . Program variables are ordinary variables that can be assigned to integer, real, boolean variables or arithmetic expressions. Boolean variables are mainly used for condition-branch and loop statements. Sampling variables are assigned with dynamically generated values when the program is running, which is subject to a continuous or discrete probability distribution. The values may be different each time. In addition, assertion expressions also have certain specifications. The format of assertions is a Boolean expression that contains variables. In the case of compound conditions, we also support logical operators “and” (&&) in assertion expressions.

Semantics. We use control flow graphs (CFGs) to express the semantics of probabilistic programs, which we define below:

Formally, a CFG is a tuple in the form $(L, X, R, \mapsto, \perp)$, where:

- L is a finite set of labels $L = \{\ell_0, \ell_1, \dots, \ell_n\}$ used to represent control locations. Each statement in a program has a unique label. For example, ℓ_0 usually indicates the starting location.
- $X = \{x_0, \dots, x_n\}$ is a set of program variables and $R = \{r_1, \dots, r_m\}$ is a set of sampling variables.
- \mapsto is a transition relation. Its element is in the form (ℓ, α, ℓ') , representing one step of transition from control location ℓ to ℓ' , by an update function $\alpha: \mathbb{R}^{|X \cup R|} \rightarrow \mathbb{R}^{|X|}$.
- \perp is a sign of the exit of the program.

Obviously, any probabilistic program can be transformed to a CFG naturally. Each label represents the control location in the execution of the probabilistic program, where the statement of the control location is the next program position to be executed, as shown in Figure 2. The following, we assume that the probabilistic program P with two linear sequences $X = \{x_0, \dots, x_n\}$ and $R = \{r_0, \dots, r_{|m|}\}$, interpreting each vector $x \in \mathbb{R}^{|X|}$ as a program variable and $r \in \mathbb{R}^{|R|}$ as a sampling variable. And let $(L, X, R, \mapsto, \perp)$ is the CFG associated with the probabilistic program P . For notational convenience we assume the configuration is a tuple (ℓ, x) , where ℓ is a location of P and x is a valuation of program variables. We fix (ℓ_0, x_0) to be the initial configuration of the program P . The program updates its state according to the transition relation. The transition relation (ℓ, α, ℓ') specifies the transitions between labels together

with the additional information specific to different types of labels. They also can be viewed as a set $f = \{f_0, \dots, f_{|n|}\}$, where each f_i denotes $\mathbb{R}^{X \cup R} \rightarrow \mathbb{R}$. The execution of P is an infinite sequence of configurations whose each finite prefix is a finite path $(\ell_0, x_0) \dots (\ell_k, x_k)$. For each $0 \leq i \leq k$, enable the transition relation (ℓ_i, f, ℓ_{i+1}) in (ℓ_i, x_i) , and evaluate the random variable r so that $x_{i+1} = f(x_i, r)$. If there is a finite path starting from (ℓ_0, x_0) and ending with (ℓ_k, x_k) , you can access the configuration (ℓ_k, x_k) from the starting configuration (ℓ_0, x_0) . According to the different types of statements, the transition relation of each label can be divided into the following situations. Suppose the current configuration is (ℓ_i, x_i) .

- Assignment Statements: for $x := exp$, where x is a variable and exp is an arithmetic expression or constant. Through the transition $(\ell, f(x, exp), \ell')$, the configuration of P transit to (ℓ_{i+1}, x_{i+1}) , where $x_{i+1} = f(x_i, r)$
- Conditional-branch Statements: For **if** ϕ ℓ_a **else** ℓ_b , $x_i = x_{i+1}$, if the value of ϕ is true($x_i \models f$), the configuration of P transit to (ℓ_a, x_a) , otherwise (ℓ_b, x_b) .
- Loop Statements: For **while** ϕ **do** ℓ_{i+1} , if the value of ϕ is true($x_i \models f$), the configuration of P transit to (ℓ_{i+1}, x_{i+1}) , otherwise (ℓ_{out}, x_{out}) , where ℓ_{out} means out of the loop.
- Location of termination: $\ell_{i+1} = \ell_i = \perp$ marks the end of P , and $x_i = x_{i+1}$

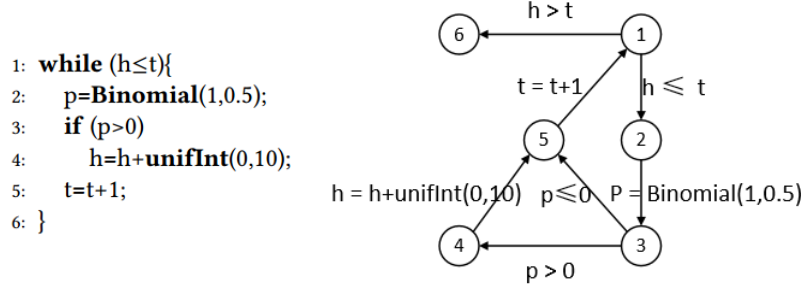


Fig. 2: A probabilistic program with its CFG

Example 1 In Figure 2, we show a probabilistic program on the left and its control flow graph on the right. In the program, t is a program variable, p and h are sampling variables. Here, two probability distributions are used: **Binomial**(1,0.5) represents a binomial distribution, which yields values 0 and 1 with equal probability; **unifInt**(0,10) represents an integer uniform distribution, which yields the integers between 0 and 10 with equal probability. The numbers 1-6 on the left are the program control locations, where 1 is the starting location and 6 is the terminal location. The probabilistic program simulates a scene of the tortoise hare race. The tortoise's initial position is t and the hare's initial position is h . The tortoise always moves at the speed of 1. When $p > 0$, that is

to say, there is a probability of 0.5, the hare increases its speed by a random number between 0 and 10. The program terminates when the hare passes the tortoise.

3 Architecture Design

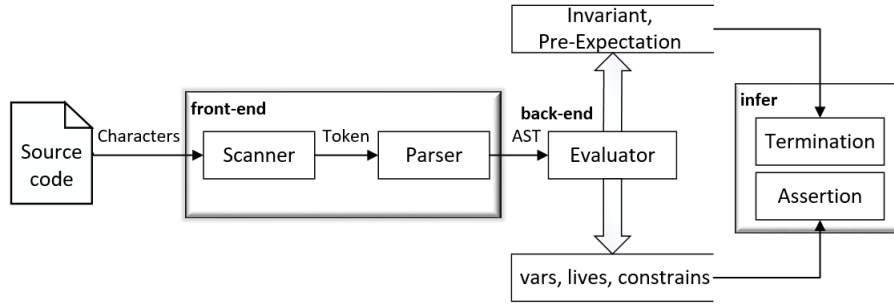


Fig. 3: The architecture of PROPER

PROPER’s architecture is mainly composed of three parts: front-end, back-end and infer, as shown in Figure ??.

Front-end and back-end can be regarded as a whole, based on the interpreter, which converts the source code into some effective intermediate representations and executes them line by line. The front-end work content is similar to the compiler front-end, mainly including scanner and parsers. Generally, scanner input the source code and converts it into token stream. This process is called lexical analysis. Then the output of the scanner is used as the input of parser, which is used to construct the abstract syntax tree. In this process, the symbol table is created. It is essentially a database, which is used to store related information such as variables and function calls. It needs four features: high speed, easy maintenance, flexibility and repeatability. In order to meet the above requirements, we use a chain hash table structure to achieve, which helps to quickly and effectively insert new data nodes, access and update existing data nodes. The average time complexity of insertion and deletion is $O(1)$. The specific structure is shown on the right side of Figure 4.

All variables are stored in a hash table, and variables hashed to the same location are connected by a double linked list, thus solving the problem of supporting repeatability. An example is shown on the left side of Figure 4. The linked list at the top is used to store the starting pointers of variables at different levels. Here, the global variables “apple” and the function name “Fruit” belong to the first-level variables, “banana” and “peer” belong to the second-level variables, and “peer” and “mango” belong to the third-level variables. Because the variable name is used for hashing, the same variable name “peer” is hashed to the same

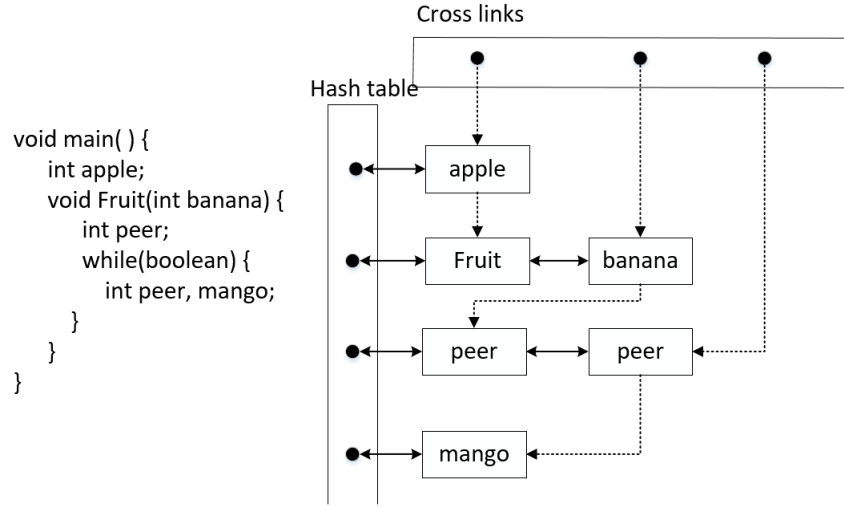


Fig. 4: Data Structure of the Symbol Table

location. Similarly, different variable names may also be hashed to the same location. We assume that “banana” is hashed to the same location as “Fruit”, and the same level is connected by a linked list.

The back-end executes the code according to the abstract syntax tree. It will recursively traverse each child node. Each non-terminal symbol corresponds to such a node, the structure of the node is shown in Figure 5(b). “parent” and “children” correspond to the parent node and child node list respectively; “value” and “text” are used to store the parsed value and text information of the symbol; “type” is the type of non-terminal, such as “STMT”, “EXPR”, “UNARY”, etc.; “production” corresponds to the sequence number of the parsed syntax expression. And “symbol” corresponds to the record in the symbol table, the structure is shown in Figure 5(a). “name” is the name of the symbol; “level” represents the level of the variable; “duplicate” indicates whether it is a variable with the same name. If the symbol corresponding to the function name, “args” points to the function parameters list, and “next” points to the next variable symbol at the same level.

While executing the syntax tree, we also collect the relevant information for inferring. For termination analysis, we need invariant and Pre-Expectation, which we will introduce these two concepts in detail in section 4. For assertion analysis, we need vars, lives, and constraints of each path. Vars is the set of variables included in the program; lives is the set of variables involving the assertion, used for path slice; constraints are related constraints.

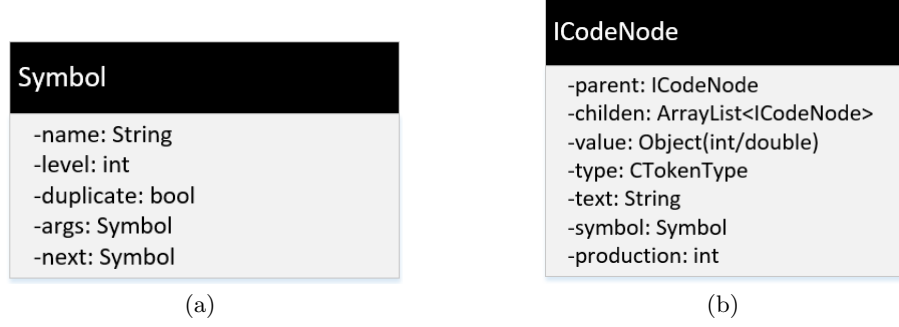


Fig. 5: The internal structure of “Symbol” and “ICodeNode” class

4 Termination Analysis

Termination analysis is an important part of program verification. If a loop program terminates for all initial values that can enter the loop, the program is said to be terminating. Ensuring termination is a necessary condition for many properties of programs such as total correctness.

Generally speaking, the termination of a program is undecidable [21], but for some subclasses of programs, the termination can be verified. In this paper, we prove the termination of the affine probabilistic programs by synthesizing ranking supermartingales [4]. We first extend the traditional linear ranking supermartingale concept to polynomial ranking supermartingales, and verify the termination of the affine probability program by synthesizing polynomial rank supermartingales. PROPER implements Algorithm 1 [5, 6] to analyze probabilistic termination qualitatively and quantitatively.

Qualitative analysis It mainly analyzes whether a probabilistic program will terminate with probability 1 (almost sure termination). To better understand the following content, we need to comprehend the concept of polynomial ranking supermartingale, invariants and pre-expectation.

- **Invariant** is a function I that assigns a set $I(\ell)$ on variables X to each label ℓ . Thus for all configurations (ℓ, x) that can be reachable from the initial configurations (ℓ_0, x_0) by running the program, it will hold that $x \models I(\ell)$. In PROPER, we only support linear invariants, which can be represented by a finite number of linear inequalities.
- **Pre-Expectation** is about the expression over the variables across a transition \mapsto , the specific calculation formula can refer to step 2.2 of Algorithm 1. Intuitively, pre-expectation is the expected value of polynomial ranking supermartingale $g(\ell, \cdot)$ in the next step of program execution.
- **Polynomial Ranking Supermartingale** [4] (pRSM) is a collection of functions at each label, represented by the template $g(\ell, \cdot)$ with the maximal degree d . It satisfies $pre(\ell, x) \leq g(\ell, x) - \varepsilon$ and $g(\ell, x) \geq 0$ at non-terminal

location. $g(\ell, x) = K$ at terminal location, where $K \leq -\varepsilon$ and $\varepsilon \geq 0$. The dividing line between termination and non-termination is 0, that shows each function is non-negative at non-terminal location. So this method can not handle the distribution without lower bound, such as Gaussian distribution.

Algorithm 1 Termination Analysis.

Require: Program P;

Ensure: Judge whether the program is terminating, isT;

The probability to terminate after N steps decreases exponentially, N.

- 1: Set template $g(\ell, \mathbf{x})$ with a natural number as the maximal degree. Each location has the common template with unique coefficient. If ℓ_{\perp} , the template $g(\ell, \mathbf{x})=K$.
 - 2: Traverse the programs parse tree
 - 2.1: Collect the invariant $I[\]$ for each location.
 - 2.2: Calculate the pre-expectation for each location.
 - case “assignment statement” :
 $pre(\ell, \mathbf{x}) = \mathbb{E}(g(\ell', \alpha, \mathbf{x}))$.
 - case “condition(if-else) or loop(while) statement” :
 $pre(\ell, \mathbf{x}) = g(\ell', \mathbf{x})$.
 - case “terminal statement” :
 $pre(\ell, \mathbf{x}) = g(\ell, \mathbf{x}) = K$.
 - 3: By the concept of half-space, $H = g(\ell, \mathbf{x}) - pre(\ell, \mathbf{x}) - \epsilon$, where $\epsilon > 0$ and $K \leq -\epsilon$
 - 4: Pattern extraction. By Farkas’ Lemma, $H' = \sum_{i=1}^d a_i \mu_i$, where $\mu_i \in \mu$, $\mu = \{ \prod_{i=1}^n I_i | n \in \mathbb{N}_0 \text{ and } I_1, \dots, I_n \in I \}$ and a_i is a non-negative real number.
 - 5: Solve linear programming. H and H’ are corresponding coefficient relations. If solvable, the program P can be terminating, otherwise return.
 - 6: Calculate the upper bound of expected termination time according to $EP(P) \leq UB(P) := \frac{g(\ell_0, x_0) - K}{\epsilon}$
 - 7: Calculate the difference interval $[a, b]$ satisfying the condition $a \leq g(\ell', \mathbf{x}) - g(\ell, f(\mathbf{x}, \mathbf{r})) \leq b$
 - 8: Obtain N, according to $\mathbb{P}(\mathbf{T}_P > N) \leq e^{-\frac{2(\epsilon(N-1) - g(\ell_0, x_0))^2}{(N-1)(b-a)^2}}$
-

Refer to Algorithm 1 steps 1 to 5. The specific idea is to calculate the martingale of each location and the value of location ℓ' minus the value of the previous location ℓ is ϵ , where ϵ is greater than 0. The invariants and pre-expectation are analyzed and calculated by traversing the abstract syntax tree. For the sake of simplicity, in PROPER, we take the maximal degree of the template as 2, that is, the template $g(\ell, \cdot)$ is a quadratic equation. For Example 1, the quadratic template g is shown in the following formula, where $n = 1, \dots, 5$. For terminal location $g(\ell_6, h, t) = k$.

$$g(\ell_n, h, t, p) = a_{n0} + a_{n1} \cdot h + a_{n2} \cdot h^2 + a_{n3} \cdot ht + a_{n4} \cdot hp + a_{n5} \cdot t + a_{n6} \cdot t^2 + a_{n7} \cdot tp + a_{n8} \cdot p + a_{n9} \cdot p^2$$

The next step is to calculate pre-expectation. We still refer to Example 1, for the assignment statement at location ℓ_2 :

$$\begin{aligned} pre(\ell_2, h, t, p) &= \mathbb{E}(g(\ell_3, h, t, p)) \\ &= (a_{30} + 0.25a_{39} + 0.5a_{38}) + (a_{31} + 0.5a_{34}) \cdot h + a_{32} \cdot h^2 + \\ &\quad a_{33} \cdot ht + (a_{35} + 0.5a_{37}) \cdot t + a_{36} \cdot t^2 \end{aligned}$$

For condition statement at location ℓ_3 , If the condition $p > 0$ holds:

$$\begin{aligned} pre(\ell_3, h, t, p) &= g(\ell_4, h + \text{unifInt}(0, 10), t, p) \\ &= (a_{40} + 5.5a_{41} + 0.25a_{42}) + (a_{41} + 11a_{42}) \cdot h + a_{42} \cdot h^2 + a_{43} \cdot ht + a_{44} \cdot hp \\ &\quad + (5.5a_{43} + a_{45}) \cdot t + a_{46} \cdot t^2 + a_{47} \cdot tp + (a_{48} + 5.5a_{44}) \cdot p + a_{49} \cdot p^2 \end{aligned}$$

Otherwise, its next location is ℓ_5 :

$$\begin{aligned} pre(\ell_3, h, t, p) &= g(\ell_5, h, t + 1, p) \\ &= (a_{50} + a_{55} + a_{56}) + (a_{51} + a_{53}) \cdot h + a_{52} \cdot h^2 + a_{53} \cdot ht + a_{54} \cdot hp \\ &\quad + (a_{55} + 2a_{56}) \cdot t + a_{56} \cdot t^2 + a_{57} \cdot tp + (a_{57} + a_{58}) \cdot p + a_{59} \cdot p^2 \end{aligned}$$

Then, through steps 3 and 4, H and H' can be computed. H is transformed by the inequality $pre(\ell, x) \leq g(\ell, x) - \epsilon$. According to Farkas' Lemma [10], we know that the coefficients of H and H' match exactly. So we obtain a linear programming problem. The coefficients of template g can be solved. Below we show the result of Example 1 in Table 1.

Table 1: The RSM for Example 1

Label	Invariant	The RSM-map
1	$h \leq t + 9$	$3 \cdot t - 3 \cdot h + 27$
2	$h \leq t$	$3 \cdot t - 3 \cdot h + 26$
3	$h \leq t$	$3 \cdot t - 3 \cdot h - 14 \cdot p + 32$
4	$h \leq t \wedge p = 1$	$3 \cdot t - 3 \cdot h + 17$
5	$h \leq t + 10$	$3 \cdot t - 3 \cdot h + 31$
6	$t < h$	-1

The "Label" column lists control locations of the program. The "Invariant" column gives the logical formulas that the reachable labelled locations satisfy when the program is executed from the starting location. The "The RSM-map" column denotes the ranking supermartingale on the labels. The expected value of the RSM decreases by at least a positive amount ϵ after each execution of a statement. In addition, ϵ is taken as 1 and K as -1 for our tool PROPER. We can see that, when going from from label 1 to label 2, the expected value of the supermartingale decreases by 1. Similarly in the step from label 2 to label 3, the condition is met because $p = \mathbb{E}(\text{Binomial}(1, 0.5)) = 0.5$. And when the "while" condition is not met, the expected value is k.

Quantitative analysis We aim to estimate the upper bound of expected termination time and calculate the bound N , so that the probabilistic program concentrates on termination before N steps.

We focus on the approximation of the expected termination time, as given in steps 6 to 8 in Algorithm 1. According to the previous steps, we can find the coefficients of the polynomial template pRSM. The existence of an pRSM ensures a finite upper bound on the expected termination time. If we know the initial values of variables, we can see that the value at the first location is $g(\ell, x_0)$, the value is K at the terminated location ℓ_\perp and the difference between two consecutive locations is ϵ . Therefore, when program P is almost surely terminating, we can get the upper bound on termination time for the given initial condition: $ET(P) \leq UB(P) = \frac{g(\ell, x_0) - K}{\epsilon}$. Then we focus on the concentration problem, that is, the probability of termination after N steps shows an exponential decrement. Exponential sum is one of the most commonly used specific function families in nonlinear approximation theory. Our main idea is based on martingale inequality of Azuma's Inequality [2], Hoeffding's Inequality [12, 15] and Bernstein's Inequality [3, 15]. In probability theory, the Azuma's inequality gives a concentration result for the values of martingales that have bounded differences. Hoeffding's Inequality is a special case of Azuma's Inequality. It proposes an upper bound on the probability that the sum of random variables deviates from its expected value. Bernstein's Inequality is a generalization of Hoeffding. It can handle not only independent variables but also weak independent variables. By Chatterjee *et al.* [6], we know that if $\epsilon(N-1) > g(\ell_0, \mathbf{x}_0)$, the inequality $\mathbb{P}(\mathbf{T}_P > N) \leq e^{-\frac{2((N-1) - g(\ell_0, \mathbf{x}_0))^2}{(N-1)(b-a)^2}}$ holds.

Consider again Example 1 with the initial values $t=30$ and $h=5$. We have $ET(P) \leq \frac{3 \cdot 30 - 3 \cdot 5 + 27 - K}{\epsilon} = 103$, where K and ϵ are set to be -1 and 1, respectively. The difference interval exists, which is $[-28, 14]$. Suppose we make the query $\mathbb{P}(\mathbf{T}_P > N) \leq 10^{-3}$, the number N can be computed to be 6296 approximately.

5 Estimating the probabilities of assertions

The goal of this section is to estimate the probability that a given assertion is correct at the exit of a program. For the specific algorithm, we refer to Sankaranarayanan *et al.* [18]. There are two main steps.

The first step is to generate a sufficient and appropriate path set S with high confidence coverage. The finite set $S = \{s_1, \dots, s_i, \dots, s_n\}$ contains distinct paths that are terminating, where s_i is the transition sequences $\ell_0 \rightarrow \dots \rightarrow \ell_k \rightarrow \dots \rightarrow \ell_\perp$. An infinite number of paths may be generated by the control flow of loop statements over program variables and sampling variables. Therefore, the symbolic execution [?] method is adopted to collect path sets. It can improve the coverage of path set s to all paths in a limited time. Without requiring specific input, symbolic execution can systematically explore as many unique execution paths as possible, ensuring high coverage. All execution paths of a program can be expressed as a tree, called an execution tree, as show in Figure 6. Each execution state, represented by a box, shows the statement to be executed. In the

upper row of the box, X represents symbol storage that associates variables with a specific value or an expression on the symbolic value x_i . And π represents path constraints, that is, a formula representing a set of hypotheses on the symbol x_i due to the branch taken during execution to reach stmt. At the beginning, $\pi = \text{true}$. Both X and π are updated during symbol execution. In the bottom row of the box, 'stmt' is the next statement to be evaluated, it can be assignment, condition-branch or loop statement.

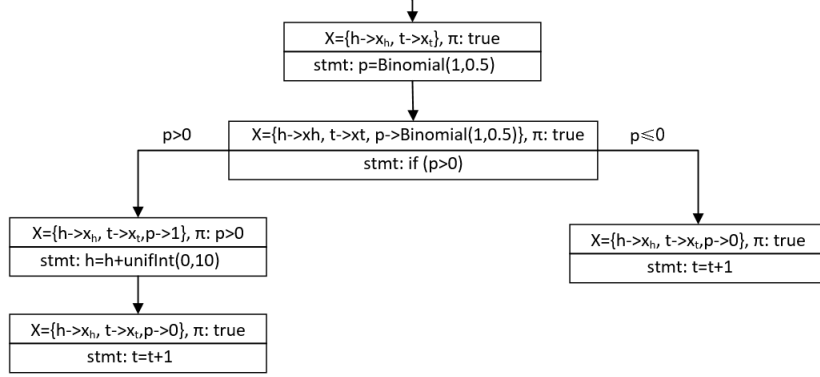


Fig. 6: Symbolic execution tree for Example 1

Through a symbolic execution, we traverse the program execution tree and collect the symbolic states and path constraints, so that each path of the program execution is based on the original probability distribution. At the same time, we also do some simplification work to avoid interference from irrelevant steps (if there is an condition-branch, but neither the if-branch nor the else-branch will affect the variables related to the assertion, then the condition-branch belongs to the irrelevant setp). Suppose the set of paths S has a lower bound c on their combined path probabilities. We can still consider the estimated probability of a given assertion φ on path set S as an approximation over the whole program, and know that the contribution from the unexplored path is at most $1 - c$. In this way, our method is able to handle the problem that program P contains program variables and sampling variables with a wide range of distributions or even infinitely many states.

The second step is path probability bound computation and estimating the probability of a given assertion. For the path set S collected above, we can analyze each path in turn, then estimate the path probability c . Each path has a series of variables and constraints. We need to estimate the probability of satisfying all constraints given the probability distributions on variables. In an overall view, it comes down to finding the satisfaction probability of the constrained system when the probability distribution of a single variable is given. Same as calculating path probability, we can seek the probabilities of asser-

tions over variables, which can provide guaranteed interval bounds on assertion probabilities. The path condition is a conjunction of linear inequalities, so it is a convex polyhedron on \mathbb{R} . Each variable can be regarded as a dimension of convex polyhedron. Since the calculation of the accurate probabilities is closely related to the volume of n -dimensional convex polyhedron, when the dimension increases, the calculation becomes very difficult and the time complexity is high [1]. Usually, the computation may fail due to the calculation or insufficient memory. On the premise of weighing the efficiency and accuracy of calculation, we focus on calculating lower and upper bounds of a given assertion rather than the estimated value.

Algorithm 2 mcBoundProbability

Require: Polytop $p=(\text{vars}, \text{constraints})$, maxDepth

Ensure: Probability interval $[p_1, p_2]$

```

1: if isABox() || maxDepth <= 0 then
2:    $[p_1, p_2] += \text{computeBoundingProbability}()$ ;
3: end if
   // If at least two (or more) non trivial clusters remain, then perform the decom-
   position.
4: if isDecomposable(p) then
5:    $\text{dim} = \text{selectBranchDimension}()$ ;
6:    $\text{branches} = \text{branchAndBound}(\text{dim}, n\text{Branch})$ ;
7:   for  $b$  in  $\text{branches}$  do
8:      $\text{mcBoundProbability}(b, \text{maxDepth} - 1)$ ;
9:   end for
10: end if

```

The calculation of path probability can be reduced to a polyhedron solving problem [?]. However, the solution of a convex polyhedron by linear programming is too rough. Therefore, we first decompose the convex polyhedron into several non-intersecting convex bodies [?] and then add up the calculation results, as described in Algorithm 2. The input to the algorithm is *maxDepth* and a polygon structure composed of variables and constraints. The *maxDepth* is the maximum recurrence dept, in order to avoid infinite decomposition, with the default value 12 in PROPER. The output is the upper bound and the lower bound of the probability. The function *isABox()* return true if each dimension is bounded. The function *isDecomposable()* is used to determine whether the current dimension can be decomposed. The function *selectBranchDimension()* is based on the idea of importance sampling, selecting an unbounded or semi-bounded dimension or else it will select the maximum width dimension. The function *branchAndBound()* is to perform the decomposition according to the selected dimension. Here, *nBranch* is the number of segmentation, with the default value 2. The function *computeBoundingProbability()* is used to calculate the upper and lower bounds.

The general idea of algorithm 2 is to use multiple hypercubes to approximate a given polyhedron. The boundary of hypercube is found by continuously selecting the most influential dimension and decomposing along the selected dimension. When the boundary box is found or the maximum recursion depth exceeds $maxDepth$, the upper and lower bounds will be calculated. The paths in the set S are calculated in turn. Since there is no intersection of each path, the path probability is equal to the sum of each paths' probability. The method of calculating the assertion probability is the same as above. The assertion can be regarded as a set of constraints. Therefore, the constraint is equivalent to the disjunctive normal form of path constraint and assertion. Let us assume that the calculation result of path probability is $[p_1, p_2]$ and assertion probability is $[p_1', p_2']$. It indicates that the maximum $1 - p_1$ path has not been explored. The upper bound of the assertion probability needs to be corrected by adding $1 - p_1$. Therefore, the probability of assertion ultimately is $[p_1', p_2' - p_1 + 1]$

6 Implementation and Experimental Evaluation

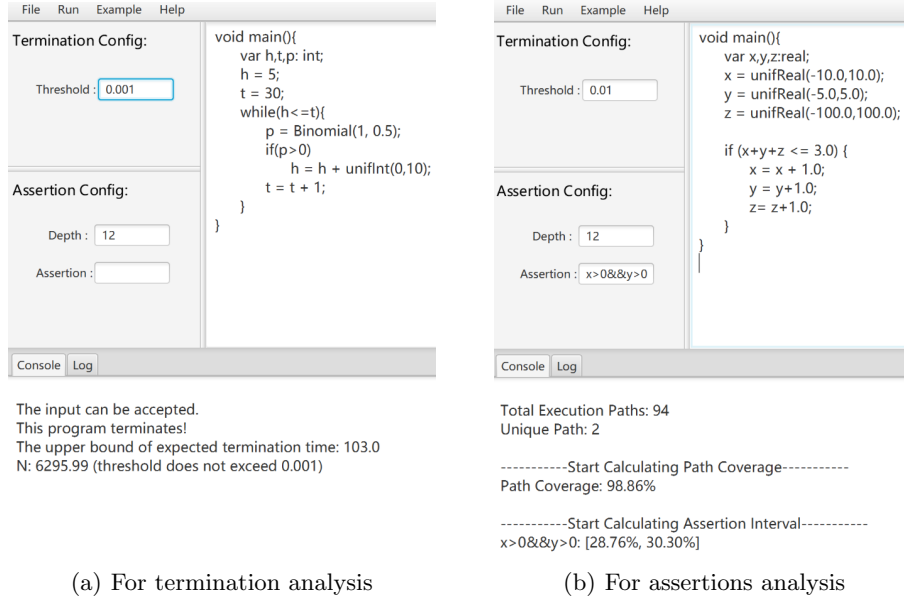


Fig. 7: PROPER User Interface

The user interface is shown in Figure 7. Subgraph (a) is a running example of terminating analysis, and subgraph (b) is a running example of assertion analysis. At the top of the interface is a row of menu bars, which are displayed

in the form of pull list, mainly including importing files, saving files, running and other functions. The concentration termination time can be set at the upper left of the interface; the recursion depth and assertion can be set at the lower left. Probabilistic programs can be written on the right, alternatively, imported from an external file. We have also built in several simple examples, you can try them. First, check the syntax specification. If it is accept, then by clicking “Termination Analysis” or “Assertion Analysis” in the “Run” drop-down menu, we start the analysis process. The analysis results are displayed in the console, with two decimal places reserved.

PROPER can deal with basic probability programs with various structures, for which we have written a few simple but typical probabilistic programs, such as, single while loop(eg.Simple), multiple nesting of while loops(eg.NestedLoop), conditions statement(if) in a while loop(eg.Tortoise-Hare), conditions-branch(if-else) in a while loop(eg.RandomWalk), nesting conditions statement in a while loop(eg.Bitcoin mining) and nesting conditions-breanch statement in a while loop(eg.Gambler2). We shall now present some experimental results of analyzing probabilistic programs using PROPER.

Table 2: Termination analysis

Example	x_0	$g(\ell_0, x_0)$	UB(P)	N (time \leq 0.01 sec)
Simple	$x = 100$	$6x$	601	1474.82
NestedLoop	$x = 1, m = 2$	$1040 \cdot m - 1040 \cdot x + 208$	1249	158141.27
Award	$bonus = 0$	$-4.0 \cdot bonus + 440$	441	4522.28
RandomWalk	$position = 0$	$-20 \cdot position + 120$	121	4695.66
Gambler	$money = 3$	$400 \cdot money + 400$	1601	1506477.30
Gambler2	$money = 10$	$45.37 \cdot money + 226.86$	908.44	3358.51
Bitcoin mining	$coin = 10$	$5.32 \cdot coin$	54.18	6.39E7
Tortoise-Hare	$h = 5, t = 30$	$3 \cdot t - 3 \cdot h + 27$	103	4264.27

In Table 2, we display the experimental results about termination analysis, where x_0 means the initial value of each variable, $g(\ell_0, x_0)$ is the polynomial ranking supermartingale about the starting location, $UB(P)$ is the upper bound of expected termination time and N is the bound that the probability of termination after N steps shows an exponential decrement (we set the threshold to be 0.01).

The results of estimating the probability intervals of assertions as shown in Table 3 are consistent with those in [18]. A little difference lies in the syntax of assertions. We can support the conjunction operator “&&” in PROPER that are not available before. It can analyze situations where two or more conditions are met at the same time. In the table, an assertion is a boolean expression; c is the lower bound of path coverage; the column headed by Bounds gives the upper and lower bounds of the given assertion.

Table 3: Estimating the probability intervals of assertions

Example	Assertion	c	Bounds
framingham	$points \geq 10$	96.36%	[13.73%, 17.36%]
	$pointsErr - points \geq 5$	96.36%	[77.81%, 84.93%]
	$points - pointsErr \geq 5$	96.36%	[17.76%, 24.68%]
sum-three	$x > 5 \ \&\& \ y > 0$	98.86%	[15.05%, 16.36%]
	$x > 0 \ \&\& \ y > 0 \ \&\& \ z > 0$	98.86%	[12.56%, 13.73%]
	$x + y > z + 10$	98.86%	[44.77%, 47.07%]
	$x + y + z > 100$	98.86%	[1.24%, 2.59%]

7 Related Tools

Software tools for the analysis of probabilistic programs has not yet received much attention. As far as we know, the only existing tools are ProbFuzz [8], PSense [?] and PSI [20]. PSI is a symbolic inference tool that approximates the probability density function represented by a probabilistic program. PSense is an automated verification tool that generates tight upper bounds for the sensitivity of probabilistic programs over initial inputs. ProbFuzz is a tool for testing probabilistic programs. Both the aforementioned tools do not consider termination analysis of probabilistic programs. For example, ProbFuzz focuses on testing rather than verification of probabilistic programs, PSI considers only inference and PSense solves sensitivity instead. Moreover, PSI/PSense requires that the input probabilistic while loop has a bounded number of loop iterations, while we can handle probabilistic loops with an unbounded number of loop iterations.

8 Conclusion and Future Work

Uncertainty exists in many software systems, including data analysis, stochastic algorithms and Monte Carlo simulation [11]. We have designed PROPER to provide convenience and support for analyzing probabilistic programs. PROPER is helpful to perform qualitative and quantitative analysis on the termination of probabilistic programs. It can also collect path sets with high confidence coverage and compute probability intervals for assertions to hold in probabilistic programs.

However, we have only solved some aspects of the complex problem of probabilistic program analysis and verification. There are still many ways to improve the tool.

1. Currently, PROPER only deals with linear programs. That is, it cannot handle variable multiplication, division and exponents, etc., regardless of the termination analysis or the estimation of assertion probability intervals.
2. Non-deterministic probabilistic programs are also not supported. PROPER requires the the behavior of the input program to be fully probabilistic, and there is no non-deterministic transitions.

3. In the future, we hope to find a better way to compute accurate termination time and probability intervals under the premise of ensuring high efficiency.

References

1. Arora, Sanjeev and Lund, Carsten and Motwani, Rajeev and Sudan, Madhu and Szegedy, Mario: Proof verification and the hardness of approximation problems. *Journal of the Acm* **45**(3), 501–555 (1998)
2. Azuma, K.: Weighted sums of certain dependent random variables. *Tohoku Mathematical Journal* **19**(3), 357–367 (2010)
3. Bennett G.: Probability Inequalities for the Sum of Independent Random Variables. *Journal of the American Statistical Association* **57**(297), 33–45 (1962)
4. Chakarov A., Sankaranarayanan S.: Probabilistic Program Analysis with Martin-gales. *International Conference on Computer Aided Verification Springer, Berlin, Heidelberg*, (2013)
5. Chatterjee, K., Fu, H., Goharshady, A.K.: Termination Analysis of Probabilistic Programs through Positivstellensatz’s. In: CAV, pp. 3–22 (2016)
6. Chatterjee, K., Fu, H., Novotný, P., Hasheminezhad, R.: Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In: POPL, pp. 327–342 (2016)
7. David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and FrankWood: Design and Implementation of Probabilistic Programming Language Anglican. In: IFL. *ACM*, 6:1–6:12. (2016)
8. Dutta S., Legunsen O., Huang Z., et al.: Testing probabilistic programming systems. the 26th ACM Joint Meeting. (2018)
9. Dustin Tran and Alp Kucukelbir and Adji B. Dieng and Maja Rudolph and Dawen Liang and David M. Blei: Edward: A library for probabilistic modeling, inference, and criticism. (2016) <https://doi.org/1610.09787>
10. Farkas J.: A fourier-féle mechanikai elv alkalmazásai (Hungarian). *Mathematikai és Természettudományi Értesítő* **12**, 457–472 (1894)
11. HASTINGS and W., K.: Monte Carlo sampling methods using Markov chains and their applications. *Journal of the Acm*, 97–109 (1970)
12. Hoeffding W.: Probability Inequalities for Sums of Bounded Random Variables. *Encyclopedia of Statistical Sciences. American Cancer Society* (2004)
13. Huang Z., Wang Z., Misailovic S.: PSense: Automatic Sensitivity Analysis for Probabilistic Programs. *Automated Technology for Verification and Analysis - 16th International Symposium*. (2018)
14. L.M.F. Fioriti, H. Hermanns: Probabilistic termination : Soundness, completeness, and compositionality. In: POPL, pp. 489–501 *ACM*(2015)
15. McDiarmid C.: Concentration. In *Probabilistic Methods for Algorithmic Discrete Mathematics. Algorithms in combinatorics* **16**, 195–248 (1998)
16. Noah D Goodman, Vikash K Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B Tenenbaum: Church: a language for generative models. In: *UAI*, pp. 220–229. *AUAI Press* (2008)
17. Noah D Goodman and Andreas Stuhlmüller: The Design and Implementation of Probabilistic Programming Languages. (2014)
18. Sankaranarayanan, Sriram and Chakarov, Aleksandar and Gulwani, Sumit.: Static Analysis for Probabilistic Programs: Inferring Whole Program Properties from Finitely Many Paths. *acm sigplan notices* **48**(6), 447–458 (2013)

19. Swarat Chaudhuri and Azadeh Farzan. Vechev: PSI: Exact Symbolic Inference for Probabilistic Programs. *Computer Aided Verification - 28th International Conference*, 62–83 (2016)
20. Timon Gehr and Sasa Misailovic and Martin T. Vechev: PSI: Exact Symbolic Inference for Probabilistic Programs. *Computer Aided Verification - 28th International Conference*, 62–83 (2016)
21. Turing A.: On Computable Numbers, with an Application to the Entscheidungsproblem. *Alan Turing His Work & Impact*, 13–115 (2013)