8-2025

Equivalence and similarity refutation for probabilistic programs

Krishnendu CHATTERJEE

Ehsan Kafshdar GOHARSHADY

Petr NOVOTNÝ

Dorde ZIKELIC
Singapore Management University, dzikelic@smu.edu.sg

Equivalence and Similarity Refutation for Probabilistic Programs

KRISHNENDU CHATTERJEE, Institute of Science and Technology Austria (ISTA), Austria
EHSAN KAFSHDAR GOHARSHADY, Institute of Science and Technology Austria (ISTA), Austria
PETR NOVOTNÝ, Masaryk University, Czech Republic
ĐORĐE ŽIKELIĆ*, Singapore Management University, Singapore

We consider the problems of statically refuting equivalence and similarity of output distributions defined by a pair of probabilistic programs. Equivalence and similarity are two fundamental relational properties of probabilistic programs that are essential for their correctness both in implementation and in compilation. In this work, we present a new method for static equivalence and similarity refutation. Our method refutes equivalence and similarity by computing a function over program outputs whose expected value with respect to the output distributions of two programs is different. The function is computed simultaneously with an upper expectation supermartingale and a lower expectation submartingale for the two programs, which we show to together provide a formal certificate for refuting equivalence and similarity. To the best of our knowledge, our method is the first approach to relational program analysis to offer the combination of the following desirable features: (1) it is fully automated, (2) it is applicable to infinite-state probabilistic programs, and (3) it provides formal guarantees on the correctness of its results. We implement a prototype of our method and our experiments demonstrate the effectiveness of our method to refute equivalence and similarity for a number of examples collected from the literature.

CCS Concepts: • **Theory of computation → Program verification**; **Program analysis**; • **Software and its engineering → Formal software verification**; • **Mathematics of computing → Probability and statistics**.

Additional Key Words and Phrases: Probabilistic programming, Static program analysis, Probability distribution equivalence, Kantorovich distance, Martingales

ACM Reference Format:
Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, and Đorđe Žikelić. 2024. Equivalence and Similarity Refutation for Probabilistic Programs. *Proc. ACM Program. Lang.* 8, PLDI, Article 232 (June 2024), 25 pages. https://doi.org/10.1145/3656462

1 INTRODUCTION

Probabilistic programs. Probabilistic programs are imperative or functional programs extended with the ability to perform sampling from probability distributions and to condition data on observations [13, 47, 79]. They provide an expressive framework for specifying probabilistic models and have been adopted in a range of application domains including stochastic networks [41],

*Part of the work done while the author was at the Institute of Science and Technology Austria (ISTA).

Authors' addresses: Krishnendu Chatterjee, krishnendu.chatterjee@ist.ac.at, Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria; Ehsan Kafshdar Goharshady, ehsan.goharshady@ist.ac.at, Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria; Petr Novotný, petr.novotny@fi.muni.cz, Masaryk University, Brno, Czech Republic; Đorđe Žikelić, dzikelic@smu.edu.sg, Singapore Management University, Singapore.

ACM 2475-1421/2024/6-ART232
https://doi.org/10.1145/3656462

machine learning [44], security [9, 10] and robotics [76]. Instead of designing different inference and analysis techniques for probabilistic models that may arise in each of these domains, one can first specify the probabilistic model of interest as a probabilistic program and then utilize the existing techniques for probabilistic programs. This separation of model specification on one hand and inference and analysis on the other hand has sparked interest in the probabilistic programming paradigm, and recent years have seen the development of many probabilistic programming languages, e.g. Church [46], Pyro [17] or Edward [78]. Concurrently with studying the design and implementation of probabilistic programming languages, formal analysis of probabilistic programs has also become a very active research area.

Static analysis of probabilistic programs. Probabilistic programs are hard to reason about. While deterministic programs always produce the same output on a given input, probabilistic programs give rise to *output distributions*. This makes probabilistic programs extremely hard to analyze both in theory [55] and in practice [37, 66], as bugs in probabilistic program implementation may be very subtle and hard to detect.

Recent years have seen much work on static analysis of probabilistic programs, where the aim is to formally prove temporal or input/output properties by analyzing the source code directly and instead of repeatedly sampling randomized executions of probabilistic programs. There have been significant developments on static analysis with respect to termination [1, 19, 24–26, 56, 63], reachability [75], safety [7, 15, 16, 28, 73], cost [67, 81, 83], input/output [30], runtime [60], productivity for infinite streams [3], sensitivity [2, 8, 82] or differential privacy [4] properties.

Equivalence and similarity refutation. In this work, we focus on *relational analysis* of probabilistic programs. The goal of relational analysis is to prove properties of *pairs* of probabilistic programs. A prominent example of relational property is *equivalence*: two probabilistic programs are equivalent if they define the same output distributions. In this paper, we consider static analysis of equivalence and similarity of output distributions of probabilistic program pairs. Equivalence and similarity are two fundamental properties of probabilistic programming systems that are essential for their correctness *both in implementation and in compilation*. We study the following two problems:

(1) *Equivalence refutation problem.* Given a pair of probabilistic programs, prove that their output distributions are not equivalent (a notion formally defined in Section 4).
(2) *Similarity refutation problem.* Given a pair of probabilistic programs, prove a lower bound on Kantorovich distance [80] between their output distributions (we formally define Kantorovich distance and discuss its relation to other distances in Sections 3.3 and 4).

Relevance. Equivalence checking and refutation are crucial for ensuring probabilistic program correctness or for bug detection. For instance, if we have two different implementations of a probabilistic model or two randomized algorithms designed to solve a given problem, the equivalence refutation analysis allows us to detect whether the two probabilistic programs give rise to different output distributions [65]. Such an analysis allows, e.g., bug detection in samplers from probability distributions [20] or in implementations of cryptographic protocols [11]. Equivalence refutation analysis also allows bug detection in probabilistic program compilers. For instance, it was observed by [38] that a 10-line probabilistic program in Stan [43] executes over 6000 lines of code of Stan implementation. Hence, detecting compilation bugs by testing may be a challenging task even for small programs. Static equivalence refutation analysis allows bug detection in compilation by comparing the source code to its intermediate representation without program execution.

While equivalence refutation analysis only proves that output distributions of two programs are not equivalent, similarity analysis provides more fine-grained information and *quantifies the difference* between two output distributions (e.g., the difference between the output distribution induced by a sampler and the ground probability distribution whose samples we wish to generate [20]).

Prior work. Equivalence and similarity are *relational properties* that are defined with respect to a program *pair*. The prior work on relational reasoning about probabilistic programs focused on developing logical systems for such reasoning [33] rather than on automation; or on sensitivity analysis [2, 4, 8, 9, 54, 82], whose aims and assumptions differ from equivalence analysis (see Section 8 for detailed discussion). *Automated* methods for formal analysis of probabilistic program equivalence have been developed for *finite-state* probabilistic systems [12, 58, 65]. However, probabilistic programs defined over real or integer-valued variables or containing sampling from continuous probability distributions (such as normal or uniform) all give rise to infinite-state programs.

On the other hand, there is a huge body of work on sampling-based statistical testing of equivalence and similarity of two probability distributions [14, 21], see [18] for a survey. While these methods provide extremely useful information and do not impose syntactic restrictions on probability distributions that they can analyze, they suffer from two key limitations. First, guarantees on the correctness of their analyses are *statistical*, meaning that there is always a non-zero probability that the analysis results are incorrect. Second, sampling-based methods suffer from scalability issues if the probabilistic program needs to be executed for a long time. For instance, the two programs in Figure 1 both consist of 7 lines of code; however, each execution of either of the two programs requires millions of samples from uniform distribution. Static analysis methods would be much more appropriate for analyzing equivalence or similarity of such programs.

To the best of our knowledge, no prior work has proposed an *automated* method for equivalence and similarity refutation analyses in *infinite-state probabilistic programs* that provide *formal guarantees* on the correctness of their results.

Our approach – automated formal analysis via expectation martingales. We present a new method for static equivalence and similarity refutation analyses of probabilistic program pairs. To the best of our knowledge, we present the first method that provides the following desired features:

(1) *Automation.* Our method is fully automated.
(2) *Infinite-state programs.* Our method is applicable to infinite-state probabilistic programs.
(3) *Formal guarantees.* Our method provides formal guarantees on the correctness of its results.

Technical challenges. Given two programs, our method refutes their equivalence by computing a function f over their output variables such that the expected value of f at the output of the two programs differs. Our method searches for such a function by computing it simultaneously with an *upper expectation supermartingale (UESM)* for the first program and a *lower expectation submartingale (LESM)* for the second program. UESMs and LESMs, notions similar to cost supermartingales [83] or super- and sub-invariants [52] (see Remark 1 for a comparison), provide sound proof rules for deriving upper and lower bounds on the expected value of a function on program output in probabilistic programs. We show that UESMs and LESMs together with the function f over outputs provide sound proof rules for refuting equivalence and similarity of programs. To the best of our knowledge, no martingale-based approach has been used in prior work for static analysis of *relational* properties of probabilistic program pairs. The non-trivial challenge is to simultaneously compute the function f, along with two martingales (one submartingale and other supermatingale), which we achieve via a constraint solving-based approach.

Contributions. Our contributions can be summarized as follows:

```
    sent = 0, fail = 0
ℓ_init : while sent ≤ 8 000 000 and fail ≤ 0:
ℓ_1 :        if prob(0.999):
ℓ_2 :            sent = sent + 1
ℓ_3 :        else:
ℓ_4 :            fail = 1
ℓ_out : return sent
```

```
    sent = 0, fail = 0
ℓ_init : while sent ≤ 9 000 000 and fail ≤ 0:
ℓ_1 :        if prob(0.9995):
ℓ_2 :            sent = sent + 1
ℓ_3 :        else:
ℓ_4 :            fail = 1
ℓ_out : return sent
```

Fig. 1. Transmission protocol example.

(1) To our best knowledge, we present the first method for *static equivalence and similarity refutation* of probabilistic program pairs, which is *automated*, applicable to *infinite-state* probabilistic programs and provides *formal guarantees* on the correctness of its results.
(2) We formulate *sound proof rules for equivalence and similarity refutation* via UESMs and LESMs.
(3) We present *fully automated algorithms* for equivalence and similarity refutation analyses in probabilistic programs, based on the above proof rules. The algorithms simultaneously compute a UESM and an LESM for two probabilistic programs together with a function over their output variables. They are applicable to numerical probabilistic programs with polynomial arithmetic expressions that may contain sampling instructions from both discrete and continuous probability distributions. Moreover, our method and our algorithm for similarity refutation are also applicable to other distance metrics, such as Total Variation (TV), which can be reduced to the Kantorovich distance (see Section 3.3 for details).
(4) Our *experimental evaluation* demonstrates the ability of our method to refute equivalence and compute lower bounds on Kantorovich distance for a variety of program pairs.

2 OVERVIEW

We start by presenting an overview of our approach and illustrating it on the probabilistic program pair in Figure 1. We first overview our method for solving the equivalence refutation problem, and then show how our method can be extended to also solve the similarity refutation problem. We provide two more motivating examples for the equivalence and the similarity refutation problems in the extended version of the paper [27].

Example 2.1 (Simple programs with long execution times). Consider the probabilistic program pair in Figure 1. Each program models a simplified network protocol [15, 53] which aims to transmit n packets from the receiver to the sender. However, each packet may be lost with probability p, and the transmission stops whenever some packet is lost. For the program in Figure 1 left, we have $\mathsf{n} = 8\,000\,000$ and $\mathsf{p} = 0.001$ as in [15]. On the other hand, the protocol in Figure 1 right transmits $\mathsf{n} = 9\,000\,000$ packets with loss probability $\mathsf{p} = 0.0005$. Both programs output the number `sent` of successfully transmitted packets, hence the output distribution of each program is the probability distribution of the value of `sent` upon termination.

One easily sees that these two programs do not define equivalent output distributions. However, using sampling-based statistical testing to deduce this would be extremely inefficient. Indeed, sampling a single execution of either program requires $\mathsf{n} = 8\,000\,000$ or $\mathsf{n} = 9\,000\,000$ samples from Bernoulli distribution, respectively. A static analysis approach that does not need to sample program executions would be much more appropriate for refuting equivalence in this example.

Requirements. In the sequel, we consider a pair of probabilistic programs and assume that they satisfy the following requirements. First, we assume that the programs share a common set of *output variables* V_{out}. This is necessary for the output distributions to be defined over the same

space so that they can be compared. Second, we assume that both programs are *almost-surely terminating*, so that their output distributions are indeed probability distributions.

Equivalence refutation. Let $\mathbb{E}_{\mu_1}$ and $\mathbb{E}_{\mu_2}$ denote the expectation operators over output distributions defined by two probabilistic programs. Our method refutes equivalence by searching for a function $f : \mathbb{R}^{|V_{out}|} \to \mathbb{R}$ that maps program outputs to real numbers, whose expected values over two output distributions are not equal, i.e. $\mathbb{E}_{\mu_1}[f] \neq \mathbb{E}_{\mu_2}[f]$.

To find such a function f, our method simultaneously searches for an *upper expectation supermartingale (UESM)* for the first program and a *lower expectation submartingale (LESM)* for the second program, notions that we formally define in Section 5. For a probabilistic program and a function f over its outputs, a UESM for f (resp. LESM for f) provides a sound proof rule for deriving an upper bound (resp. lower bound) of the expected value of f on program output. Hence, in order to refute equivalence, our method searches for

(1) a function $f : \mathbb{R}^{|V_{out}|} \to \mathbb{R}$ over program outputs,
(2) an UESM for f in the first program, and
(3) an LESM for f in the second program,

such that the upper bound on $\mathbb{E}_{\mu_1}[f]$ implied by the UESM is strictly smaller than the lower bound on $\mathbb{E}_{\mu_2}[f]$ by by the LESM. In Section 5, we show that these three objects together formally certify that $\mathbb{E}_{\mu_1}[f] \neq \mathbb{E}_{\mu_2}[f]$ and thus that the output distributions of two programs are not equivalent.

Note that searching for a function f over outputs whose expectation differs in the two programs yields *both sound and complete* proof rule for refuting equivalence of output distributions. Indeed, we will prove soundness in Section 5 as stated above. On the other hand, for completeness, suppose that two output distributions are not equivalent. Then, there exists an event A over outputs such that $P_{\mu_1}[A] \neq P_{\mu_2}[A]$. Hence, with f being the indicator function $I(A)$, we have $E_{\mu_1}[I(A)] \neq E_{\mu_2}[I(A)]$.

Upper and lower expectation martingales. Consider a probabilistic program and a function f over its outputs. Intuitively, an *upper expectation supermartingale (UESM)* for f is a function U_f that assigns a real value to each program state (comprising of a location in the code and program variable values), which is required to satisfy the following two conditions:

(1) **Zero on output** The function U_f is equal to zero on termination, i.e. $U_f(s) = 0$ for every reachable terminal state s in the program.
(2) **Expected f-decrease** In every step of program computation, an increase in the value of f is matched in expectation by the decrease in the value of U_f. That is, for every reachable state s in the program, we have $U_f(s) - \mathbb{E}[U_f(s')] \geq \mathbb{E}[f((s')^{\text{out}})] - f(s^{\text{out}})$.

Here, we use the standard primed notation from program analysis: s' denotes the probabilistically chosen successor of the state s upon one computational step of the program. Also, s^{out} and $(s')^{\text{out}}$ denote the output variable valuations defined by states s and s'.

The expected f-decrease condition can be rewritten as $U_f(s) \geq \mathbb{E}[U_f(s')] + \mathbb{E}[f((s')^{\text{out}})] - f(s^{\text{out}})$. Intuitively, $U_f(s)$ is an upper bound on the expected difference between the value of f *in the current state* (which is $f(s^{\text{out}})$) and upon termination (which is a random variable over the output distribution of paths starting from s).

Lower expectation submartingales (LESMs) are defined similarly, with the only difference being that the expected f-decrease is replaced by the dual f-increase (by replacing $\geq$ with $\leq$).

We formally define UESMs and LESMs in Section 5. Furthermore, we prove that a UESM in the initial state of the program evaluates to an *upper bound* on the difference between the expected value of f on output and the value of f in the initial program state (subject to at least one of the so-called "Optional Stopping Theorem" conditions being satisfied, see Section 5 for details); and dually for LESMs. Hence, U/LESMs provide a sound proof rule for computing upper/lower bounds on the expected value of a function defined over program outputs. The names of UESMs and

LESMs emphasize their connection to supermartingale and submartingale processes in probability theory, respectively [84], which lie at the core of soundness proofs of our proof rules. Intuitively, supermartingales (resp. submartingales) are a class of stochastic processes that decrease (resp. increase) in expected value upon every one-step evolution of the process. In particular, in the case of UESMs, we see from the above definition that the sum of U_f and f intuitively behaves like a supermartingale, and similarly for LESMs and submartingales.

Example 2.2. Consider the programs shown in Figure 1 with output variables `sent` and `fail`. Define the function $f(\texttt{sent}, \texttt{fail}) = \texttt{sent} - \texttt{fail}$ over the outputs of programs. Furthermore, define the functions U_f mapping states in the left program to reals and L_f mapping states in the right program to reals via (as computed by our tool in Section 7, rounded to one decimal)

$$U_f\begin{pmatrix}\ell,\\ \texttt{sent},\\ \texttt{fail}\end{pmatrix} = \begin{cases} 998 - 998\cdot\texttt{fail}, & \text{if } \ell = \ell_{init}\\ 998 - 997\cdot\texttt{fail}, & \text{if } \ell = \ell_1\\ 999 - 998\cdot\texttt{fail}, & \text{if } \ell = \ell_2\\ -1 + \texttt{fail}, & \text{if } \ell = \ell_3\\ -1 + \texttt{fail}, & \text{if } \ell = \ell_4\\ 0, & \text{if } \ell = \ell_{out}\end{cases} \qquad L_f\begin{pmatrix}\ell,\\ \texttt{sent},\\ \texttt{fail}\end{pmatrix} = \begin{cases} 1997.5 - 1997.5\cdot\texttt{fail}, & \text{if } \ell = \ell_{init}\\ 1997.5 - 1996.5\cdot\texttt{fail}, & \text{if } \ell = \ell_1\\ 1998.5 - 1997.5\cdot\texttt{fail}, & \text{if } \ell = \ell_2\\ -1 + \texttt{fail}, & \text{if } \ell = \ell_3\\ -1 + \texttt{fail}, & \text{if } \ell = \ell_4\\ 0, & \text{if } \ell = \ell_{out}\end{cases}$$

Since both functions are equal to 0 at all reachable output states, it follows that they both satisfy the Zero on output condition. Furthermore, one can check by inspection that U_f satisfies the Expected f-decrease condition in the program on the left, and that L_f satisfies the Expected f-increase condition in the program on the right. Hence, U_f is an example of an UESM for f in the program in the left, and L_f is an example of an LESM for f in the program in the right.

UESMs and LESMs for equivalence refutation. To refute equivalence of two probabilistic programs, our method computes (1) a function f over probabilistic program outputs, (2) an UESM U_f^1 for f in the first program, and (3) an LESM L_f^2 for f in the second program, such that

$$U_f(s^1_{\text{init}}) + f((s^1_{\text{init}})^{\text{out}}) < L_f(s^2_{\text{init}}) + f((s^2_{\text{init}})^{\text{out}}),$$

where s^1_{init} and s^2_{init} are the initial states of the first and the second program, respectively. Note that the choice of computing UESMs for the first program and LESMs for the second program rather than the opposite is made without loss of generality. Indeed, by simply negating the function f, an UESM for f becomes an LESM for $-f$ and vice-versa. We formalize our proof rule for the equivalence refutation problem and prove its soundness in Section 5.3.

Example 2.3. Consider again the programs in Figure 1 and the function f, the UESM U_f, and the LESM L_f defined in Example 2.2. The initial state of the programs satisfies $\texttt{sent} = \texttt{fail} = 0$. Hence,

$$U_f(s^1_{\text{init}}) + f((s^1_{\text{init}})^{\text{out}}) = 998 < 1997.5 = L_f(s^2_{\text{init}}) + f((s^2_{\text{init}})^{\text{out}}).$$

Hence, our method refutes equivalence of output distributions of programs in Figure 1.

Automation: Simultaneous synthesis. The key challenge in automating the aforementioned idea is the effective computation of the function over outputs, the UESM and the LESM. Note that these objects cannot be computed separately – the computation must be guided by the objective of obtaining f, U_f^1 and L_f^2 such that $U_f(s^1_{\text{init}}) + f((s^1_{\text{init}})^{\text{out}}) < L_f(s^2_{\text{init}}) + f((s^2_{\text{init}})^{\text{out}})$.

We solve this challenge by employing a constraint-solving-based approach to compute these three objects *simultaneously*. While our theoretical results apply to the general arithmetic probabilistic programs, our automated method is applicable to probabilistic program pairs in which all arithmetic

expressions are polynomials over program variables. It first fixes a polynomial template for f by fixing a symbolic polynomial expression over output variables V_{out}. It also fixes polynomial templates for the UESM in the first program and the LESM in the second program by fixing one symbolic polynomial expression over program variables at each location of each program. The defining conditions of the UESM and the LESM are then encoded as constraints over the symbolic template variables. In addition, we add the *equivalence refutation constraint* $U_f(s^1_{\text{init}}) + f((s^1_{\text{init}})^{\text{out}}) < L_f(s^2_{\text{init}}) + f((s^2_{\text{init}})^{\text{out}}$. This results in a system of constraints whose every solution gives rise to a concrete instance of f, U^1_f and L^2_f that refute equivalence. Our synthesis then proceeds by solving the resulting system of constraints.

Note that considering f, U^1_f and L^2_f specified in terms of polynomials over program variables allows us to capture both expectations as well as higher moments of any random variable defined in terms of a polynomial expression over program variables in the output probability space of each program. We present our algorithm in Section 6.

Extension to similarity refutation. Our method for solving the equivalence refutation problem can be adapted to the similarity refutation problem. In particular, if we additionally require that the function f is 1-Lipschitz continuous, we show that

$$L_f(s^2_{\text{init}}) + f((s^2_{\text{init}})^{\text{out}}) - U_f(s^1_{\text{init}}) - f((s^1_{\text{init}})^{\text{out}})$$

evaluates to a lower bound on the Kantorovich distance between the output distributions of the two programs. We omit the details in order to keep this overview non-technical. We define Kantorovich distance and Lipschitz continuity in Section 3.3, prove the soundness of UESMs and LESMs for proving lower bounds on Kantorovich distance in Section 5.3 and show how to impose the additional 1-Lipschitz continuity condition in our automated synthesis procedure in Section 6.

Example 2.4. Going back to the probabilistic program pair in Figure 1 and Examples 2.2 and 2.3, since the function $f(\texttt{sent}, \texttt{fail}) = \texttt{sent} - \texttt{fail}$ is 1-Lipschitz with respect to the L^1-distance over $\mathbb{R}^2$, it immediately follows from our result in Example 2.3 that the Kantorovich distance between output distributions of these programs is bounded from below by

$$L_f(s^2_{\text{init}}) + f((s^2_{\text{init}})^{\text{out}}) - U_f(s^1_{\text{init}}) + f((s^1_{\text{init}})^{\text{out}}) = 1997.5 - 998 = 999.5.$$

Limitations. While our experimental results demonstrate the applicability of our method to a wide range of probabilistic program pairs, our approach has several limitations:

(1) Currently, our approach does not support programs with conditioning.
(2) In general, the lower bounds on the Kantorovich distance of output distributions computed by our approach might not be tight.
(3) From a practical perspective, the performance of our automated method is dependent on the quality of *supporting linear invariants* generated for both programs. In our approach, these are computed by off-the-shelf invariant generators. See Section 7 for details.

Remark 1 (Martingale-based approach to relational analysis). *Martingale-based approach has been widely studied for static analysis of probabilistic programs, and UESMs and LESMs used in our approach are based on cost supermartingales [83] or super- and sub-invariants for expectation bounds [52] in single programs. In contrast to these concepts, our key differences are: (a) we consider proof rules for* relational analysis *of equivalence and similarity properties of program pairs; (b) we consider proof rules based on* both *super-* and *submartingales; (c) we consider the two types of martingales (UESMs and LESMs)* simultaneously*; and (d) in addition to synthesizing a supermartingale and a submartingale, we also need to simultaneously synthesize a function f on outputs with respect to which the UESM and the LESM are defined.*

Furthermore, our results on UESMs and LESMs subsume and unify the results of [52, 83]. Moreover, while [52] make the assumption of non-negative program variables and leave the generalization to programs with both positive and negative variables as a direction of future work [52, page 26], our UESM/LESMs apply to both positive and negative variables under the same assumptions as in [52]. The non-negative variables assumption is also imposed by the methods [6, 81] for automated computation of bounds on expected values, whereas our UESM/LESMs are applicable to programs with both positive and negative variables. More detailed discussion of the differences is provided in Section 5.2.

3 PRELIMINARIES

We use boldface notation for vectors, e.g. $\mathbf{x}, \mathbf{y}$, etc. An i-component of vector $\mathbf{x}$ is denoted by $\mathbf{x}[i]$. For an n-dimensional vector $\mathbf{x}$, index $1 \leq i \leq n$, and number a we denote by $\mathbf{x}(i \leftarrow a)$ the vector $\mathbf{y}$ s.t. $\mathbf{y}[i] = a$ and $\mathbf{y}[j] = \mathbf{x}[j]$ for all $1 \leq j \leq n$ s.t. $j \neq i$. Throughout the paper, we work with vectors representing valuations of variables of some program. We assume some canonical ordering of the variables, denoting them $x_1, x_2, x_3, \ldots$, though in our examples we use aliases $x, y, z, \ldots$ for better readability. Hence, for a program with n variables $x_1, \ldots, x_n$, the number $\mathbf{x}[i]$ denotes the value of variable x_i in valuation $\mathbf{x} \in \mathbb{R}^n$.

We will operate with some basic notions of probability theory, such as *probability space*, *random variable*, *expected value,* etc. We review the formal definitions of these notions in the extended version of the paper [27]. We use the term *probability distribution* interchangeably with *probability measure*, particularly when the underlying sample space is (some subset of) an Euclidean space $\mathbb{R}^n$. For a finite or countable set A, we denote by $\mathcal{P}(A)$ the set of all probability distributions on A.

3.1 Program Syntax

Imperative-style syntax. We consider imperative arithmetic programs consisting of standard programming constructs: variable assignments, sequential composition, conditional branching, and loops. Right-hand sides of variable assignments are formed by expressions built from constants, program variables and Borel-measurable arithmetic operators (Borel measurability [84] is a standard assumption in probabilistic program analysis that is satisfied by all standard arithmetical operators). We denote by $E(\mathbf{x})$ the value of expression E in valuation $\mathbf{x}$ and assume $E(\mathbf{x})$ to be well-defined for all valuations $\mathbf{x}$. The guards of loops and conditional statements consist of *predicates.* A predicate Ψ is a logical formula obtained by a finite number of applications of conjunction, disjunction, and negation operations on *atomic predicates* of the form $E_1 \leq E_2$, where E_1, E_2 are expressions. We denote by $\mathbf{x} \models \Psi$ the fact that a predicate Ψ is satisfied by the valuation $\mathbf{x}$.

Probabilistic instructions. Our programs also admit two types of *probabilistic* statements. The first is *probabilistic branching,* in our examples represented by the command **if prob**(p) **then ... else ...**. Upon the execution of such a statement, the program enters the if-branch with probability p and the else-branch with probability $1 - p$. The second is *sampling* of a variable value from a given probability distribution, represented by the **sample(...)** statement in our examples. We allow sampling from both discrete and continuous probability distributions. In this work, we do not consider conditioning on observations.

Figure 1 shows the typical form of the programs we work with. However, our algorithm works with a more abstract and operational representation of programs called *probabilistic control-flow graphs (pCFGs).* The use of pCFGs is standard in probabilistic program analysis [1, 24], hence we use them as the primary syntactical representation of programs.

Probabilistic control-flow graphs. A *probabilistic control-flow graph (pCFG)* is an ordered tuple $C = (L, V, V_{out}, \ell_{init}, \mathbf{x}_{init}, \mapsto, G, Up)$, where:

- L is a finite set of *locations*;
- $V = \{x_1, \ldots, x_{|V|}\}$ is a finite set of *program variables*;

- $V_{out} = \{x_1, \ldots, x_{|V_{out}|}\} \subseteq V$ is a finite set of *output variables*;
- $\ell_{init} \in L$ is the *initial program location* and $\mathbf{x}_{init} \in \mathbb{R}^{|V|}$ is the initial variable valuation;
- $\mapsto \subseteq L \times \mathcal{P}(L)$ is a finite set of *transitions*. For each transition $\tau = (\ell, Pr)$, we say that ℓ is its *source location* and that $Pr : L \to [0, 1]$ is a probability distribution over *successor locations*.
- G is a map assigning to each transition $\tau = (\ell, Pr) \in \mapsto$ a *guard* $G(\tau)$, which is a predicate over V specifying whether τ can be executed.
- Up is a map assigning to each transition $\tau = (\ell, Pr) \in \mapsto$ an *update* $Up(\tau) = (j, u)$ where $j \in \{1, \ldots, |V|\}$ is a *target variable index* and u is an *update element* which can be:
  - the bottom element $u = \bot$, denoting no update;
  - a Borel-measurable arithmetic expression $u : \mathbb{R}^{|V|} \to \mathbb{R}$, denoting deterministic update;
  - a probability distribution $u = \delta$, denoting that variable value is sampled according to δ.

We assume the existence of a special *terminal location* ℓ_{out}. Terminal location ℓ_{out} only has one outgoing self-loop transition $\tau = (\ell_{out}, Pr)$ with $Pr(\ell_{out}) = 1$, $G(\tau) \equiv$ true and no variable update.

We require that each location ℓ has at least one outgoing transition and that the disjunction of guards of all transitions outgoing from ℓ is equivalent to *true*, i.e. $\bigvee_{\tau=(l,\_)} G(\tau) \equiv true$. These assumptions ensure that it is always possible to execute at least one transition and are imposed without loss of generality as we may always introduce an additional transition from ℓ to ℓ_{out}. We also require that guards of two distinct transitions τ_1 and τ_2 outgoing from ℓ are *mutually exclusive*, i.e. $G(\tau_1) \wedge G(\tau_2) \equiv false$, to ensure that there is no non-determinism in the programming language.

3.2 Program Semantics

We use operational semantics that views each pCFG as a (general state space) Markov process. This approach is standard in probabilistic program analysis [24, 56]. Towards the end of the subsection, we define the *output distribution* of a probabilistic program.

States, paths and runs. A *state* in a pCFG C is a tuple $(\ell, \mathbf{x})$, where ℓ is a location in C and $\mathbf{x} \in \mathbb{R}^{|V|}$ is a variable valuation. A transition $\tau = (\ell, Pr)$ is *enabled* at a state $(\ell', \mathbf{x})$ if $\ell = \ell'$ and $\mathbf{x} \models G(\tau)$. A state $(\ell', \mathbf{x}')$ is a *successor* of $(\ell, \mathbf{x})$, if there exists an enabled transition $\tau = (\ell, Pr)$ in C such that $Pr(\ell') > 0$ and we can obtain $\mathbf{x}'$ by applying the update of τ to $\mathbf{x}$. The state $(\ell_{init}, \mathbf{x}_{init})$ is the *initial state*. A state $(\ell, \mathbf{x})$ is said to be *terminal*, if $\ell = \ell_{out}$. We use $State^C$ to denote the set of all states in C.

A *finite path* in C is a sequence $(\ell_0, \mathbf{x}_0), (\ell_1, \mathbf{x}_1), \ldots, (\ell_k, \mathbf{x}_k)$ of states with $(\ell_0, \mathbf{x}_0) = (\ell_{init}, \mathbf{x}_{init})$ and with $(\ell_{i+1}, \mathbf{x}_{i+1})$ being a successor of $(\ell_i, \mathbf{x}_i)$ for each $0 \leq i \leq k-1$. A state $(\ell, \mathbf{x})$ is *reachable* in C if there exists a finite path in C whose last state is $(\ell, \mathbf{x})$. A *run* (or an *execution*) in C is an infinite sequence of states whose each finite prefix is a finite path. We use $Fpath^C$ and Run^C to denote the set of all finite paths and all runs in C, respectively. We also use $Reach^C$ to denote the set of all reachable states in C.

Next valuation function. Let $\tau \in \mapsto$ be a transition and $\mathbf{x}$ a valuation. By $Next(\tau, \mathbf{x})$ we denote a random vector representing the successor valuation after τ is taken in a state whose current valuation is $\mathbf{x}$. Formally, let $(i, u) = Up(\tau)$. Then $Next(\tau, \mathbf{x})[j] = \mathbf{x}[j]$ for all variable indices $1 \leq j \leq |V|$ with $j \neq i$ that are not updated by the transition, and

$$Next(\tau, \mathbf{x})[i] = \begin{cases} \mathbf{x}[i] & \text{if } u = \bot, \\ u(\mathbf{x}) & \text{if } u \text{ is a Borel-measurable arithmetic expression,} \\ X_\delta & \text{if } u = \delta \text{ is a probability distribution} \\ & \text{(here } X_\delta \text{ is a random variable following the distribution } \delta). \end{cases}$$

Semantics of pCFGs. A pCFG C defines a discrete-time Markov process taking values in the set of states of C, whose trajectories correspond to runs in C. Intuitively, the process starts in the initial

state $(\ell_{init}, \mathbf{x}_{init})$ and in each time step it samples the next state along the run from the probability distribution defined by the current state. Suppose that, at time step i, the process is in the state $(\ell_i, \mathbf{x}_i)$. The next state $(\ell_{i+1}, \mathbf{x}_{i+1})$ is chosen as follows:

- Let $\tau = (\ell_i, Pr_i)$ be the unique transition enabled at $(\ell_i, \mathbf{x}_i)$. Recall, our assumptions on pCFGs ensure that at each state in C there is a unique enabled transition.
- Sample the successor location ℓ_{i+1} from the probability distribution Pr_i.
- Sample a value of the random vector $Next(\tau, \mathbf{x}_i)$ to get $\mathbf{x}_{i+1}$.

The above intuition can be formalized by a construction of a probability space whose sample space is Run^C. The construction is standard (see, e.g., [64]) and we omit it. We denote by $\mathbb{P}^C$ the probability measure over the runs of C which results from this construction and which thus formally captures the dynamics intuitively explained above.

Termination. Our equivalence analysis is restricted to probabilistic programs that terminate almost-surely. This is both a conceptual assumption since we want our probabilistic programs to define valid probability distributions over their outputs, and also a technical assumption required by our approach. Given a pCFG C, a run $\rho = (\ell_0, \mathbf{x}_0), (\ell_1, \mathbf{x}_1), \cdots \in Run^C$ is *terminating* if it reaches some terminal state. We use $Term \subseteq Run^C$ to denote the set of all terminating runs in Run^C. A pCFG C terminates *almost-surely (a.s.)* if $\mathbb{P}^C[Term] = 1$. Automated almost-sure termination proving for linear and polynomial arithmetic probabilistic programs can be achieved by synthesizing a ranking supermartingale (RSM) [19, 22, 24]. We define the *termination time* of ρ via $TimeTerm(\rho) = \inf_{i \geq 0}\{i \mid \ell_i = \ell_{out}\}$, with $TimeTerm(\rho) = \infty$ if ρ is not terminating.

Output distribution. Every a.s. terminating pCFG defines a probability distribution over its outputs. For every variable valuation $\mathbf{x} \in \mathbb{R}^{|V|}$, let $\mathbf{x}^{out}$ be the projection of $\mathbf{x}$ to the components corresponding to variables in V_{out}. Then, for a terminating run ρ that reaches a terminal state $(\ell_{out}, \mathbf{x})$, we say that $\mathbf{x}^{out}$ is its *output variable valuation* (or, simply, its *output*). An a.s. terminating pCFG C defines a probability distribution over the space of all output variable valuations $\mathbb{R}^{|V_{out}|}$ as follows. For each Borel-measurable subset $B \subseteq \mathbb{R}^{|V_{out}|}$, we define

$$Output(B) = \left\{\rho \in Run^C \mid \rho \text{ reaches a terminal state } (\ell_{out}, \mathbf{x}) \text{ with } \mathbf{x}^{out} \in B\right\}.$$

A *output distribution* μ^C of C is defined by putting

$$\mu^C[B] = \mathbb{P}^C\Big[Output(B)\Big]$$

for each Borel-measurable subset B of $\mathbb{R}^{|V_{out}|}$. Since C is a.s. terminating, we have $\mu^C[\mathbb{R}^{|V_{out}|}] = 1$.

3.3 Kantorovich Distance of Probability Distributions

To measure the similarity of output distributions, we use the established *Kantorovich distance* (also known as 1-Wasserstein distance). The definition of this distance is parameterized by a choice of a *metric* in $\mathbb{R}^{|V_{out}|}$; this is an Euclidean space and thus can be equipped with a number of standard metrics such as as discrete, L^1 (i.e. Manhattan), L^2 (i.e. Euclidean) or L^∞ (i.e. uniform).

The standard, "primal", definition of Kantorovich distance [80] between two distributions μ_1, μ_2, which involves *couplings* between the two distributions, is somewhat technical and we omit it. However, since we consider distances of $\mathbb{R}^{|V_{out}|}$-valued distributions, we can use an equivalent *dual* definition, which we present below.

Kantorovich distance: definition. The Kantorovich distance is only well-defined for pairs of distributions that have *finite first moments* w.r.t. the underlying metric. Given a metric $d \colon \mathbb{R}^{|V_{out}|^2} \to \mathbb{R}_{\geq 0}$, we say that a probability measure μ has a *finite first moment* w.r.t. d if there exists $\mathbf{x}_0 \in \mathbb{R}^{|V_{out}|}$

s.t. the function $g_{\mathbf{x}_0} : \mathbb{R}^{|V_{out}|} \to \mathbb{R}_{\geq 0}$ defined by $g_{\mathbf{x}_0}(\mathbf{x}) = d(\mathbf{x}_0, \mathbf{x})$ satisfies $\mathbb{E}_\mu[g_{\mathbf{x}_0}] < \infty$. Note that due to the triangle inequality property of metrics, $\mathbb{E}_\mu[g_{\mathbf{x}_0}] < \infty$ iff $\mathbb{E}_\mu[g_{\mathbf{y}}] < \infty$ for all $\mathbf{y} \in \mathbb{R}^{|V_{out}|}$.

Definition 3.1 (Kantorovich distance of output distributions). Let C_1, C_2 be two pCFGs with the same set of output variables V_{out}. Further, let d be a metric in $\mathbb{R}^{|V_{out}|}$ such that μ^{C_1} and μ^{C_2} have finite first moments w.r.t. d. The *Kantorovich (or 1-Wasserstein) distance* between μ^{C_1} and μ^{C_2} is defined via

$$\mathcal{K}_d(\mu^{C_1}, \mu^{C_2}) = \sup_{f \in L^1_d(\mathbb{R}^{|V_{out}|})} \left| \mathbb{E}_{\mu_1}[f] - \mathbb{E}_{\mu_2}[f] \right|,$$

where

$$L^1_d(\mathbb{R}^{|V_{out}|}) = \left\{ f : \mathbb{R}^{|V_{out}|} \to \mathbb{R} \;\middle|\; |f(x) - f(y)| \leq d(x, y) \text{ for all } x, y \in \Omega \right\}$$

is the set of all 1-Lipschitz continuous functions defined over the metric space $(\mathbb{R}^{|V_{out}|}, d)$, and $\mathbb{E}_{\mu_1}$ and $\mathbb{E}_{\mu_2}$ denote expectation operators with respect to μ_1 and μ_2.

When defined with respect to the discrete metric (which assigns 0 distance to pairs of identical elements and unit distance to all distinct pairs), Kantorovich distance is equal to another well known distance of probability distributions: the Total Variation distance [80], which has been previously used in verification of finite-state probabilistic systems [31, 57]. Moreover, for finite-state probabilistic models, the notion of simulation distance is based on the notion of *optimal transport* [61, 77], and Kantorovich distance generalizes this notion to infinite-state models. The survey paper [35] gives an overview of various uses of Kantorovich distance in probabilistic verification.

4 PROBLEM STATEMENT

In what follows, let C_1 and C_2 be two pCFGs. Since we can only compare two probability distributions if they are defined over the same space, we require that the two pCFGs share a common output variable set V_{out}. Furthermore, we assume that both C_1 and C_2 are a.s. terminating.

4.1 Equivalence Refutation Problem

We say that C_1 and C_2 are *output equivalent*, if for every Borel-measurable set $B \subseteq \mathbb{R}^{|V_{out}|}$ we have

$$\mu^{C_1}[B] = \mu^{C_2}[B]. \tag{1}$$

(Recall that μ^{C_1} and μ^{C_2} denote output distributions of C_1 and C_2, respectively.)

Problem 1 (Equivalence refutation problem). Given two a.s. terminating pCFGs C_1 and C_2 with the same output variable set V_{out}, prove that C_1 and C_2 are <u>not</u> output equivalent.

4.2 Similarity Refutation Problem

The similarity refutation problem is parameterized by a metric over the output space which gives rise to a Kantorovich distance of distributions over this space. Our theoretical results in this work are applicable to any metric. Our algorithmic approach will consider standard metrics such as discrete, L^1 (i.e. Manhattan), L^2 (i.e. Euclidean) or L^∞ (i.e. uniform). We provide the definition of each of these metrics in the extended version [27].

Let d be a metric over $\mathbb{R}^{|V_{out}|}$ such that the output distributions μ^{C_1}, μ^{C_2} have finite first moments w.r.t. d. We say that C_1 and C_2 are *ϵ-output close*, if

$$\mathcal{K}_d\left(\mu^{C_1}, \mu^{C_2}\right) < \epsilon. \tag{2}$$

The definition of the similarity refutation problem follows straightforwardly.

Problem 2 (Similarity refutation problem). Given two a.s. terminating pCFGs C_1 and C_2 with the same output variable set V_{out}, a metric d over $\mathbb{R}^{|V_{out}|}$ such that μ^{C_1} and μ^{C_2} have finite first moments w.r.t d, and $\epsilon > 0$, prove that C_1 and C_2 are <u>not</u> ϵ-output close.

Finite first moment assumption. The Similarity refutation problem assumes that the output distributions of the two programs have finite first moments w.r.t. the output state metric. Our algorithm (see Section 6) checks this assumption (or more precisely, its sufficient conditions) automatically. The extended version of the paper [27] contains a discussion of what to do when we aim to analyze a pair of programs that violate the assumption.

5 MARTINGALE-BASED REFUTATION RULES

Our approach to equivalence and similarity refutation is based on the notions of upper and lower expectation supermartingales, which generalize and unify cost supermartingales [83] and super/sub-invariants [52]. Given an a.s. terminating probabilistic program and a function f over its output variables, upper expectation supermartingales provide a sound proof rule for deriving upper bounds on the expected value of f at output in probabilistic programs, and similarly for lower expectation submartingales and lower bounds.

In this section, we start by fixing the necessary terminology. Then, we state proof rules which subsume and unify the proof rules of [52, 83]. Finally, we state our proof rules for equivalence and similarity refutation in probabilistic program pairs.

5.1 Expectation Supermartingales and Submartingales: Definition

State and predicate functions, invariants. Let $C = (L, V, V_{out}, \ell_{init}, \mathbf{x}_{init}, \mapsto, G, Up)$ be a pCFG:

- A *state function* η in C is a function which to each location $\ell \in L$ assigns a Borel-measurable function $\eta(\ell) : \mathbb{R}^{|V|} \to \mathbb{R}$ over program variables. We interchangeably use $\eta(\ell)(\mathbf{x})$ and $\eta(\ell, \mathbf{x})$.
- A *predicate function* Π in C is a function which to each location $\ell \in L$ assigns a predicate $\Pi(\ell)$ over program variables. It naturally induces a set of states $\{(\ell, \mathbf{x}) \mid \mathbf{x} \models \Pi(\ell)\}$. With a slight abuse of notation, we also use Π to refer to this set of states.
- A predicate function Π is an *invariant* if Π contains all reachable states in C, i.e. if for each reachable state $(\ell, \mathbf{x}) \in Reach^C$ we have $\mathbf{x} \models I(\ell)$.

Upper expectation supermartingales. We now define upper expectation supermartingales (UESMs). Consider a pCFG C and let $f : \mathbb{R}^{|V_{out}|} \to \mathbb{R}$ be a Borel-measurable function over its outputs. A UESM for f is a state function U_f that satisfies certain conditions in every reachable state.

Since it is generally not feasible to compute the set of all reachable states in a program, we define UESMs with respect to a supporting invariant that over-approximates the set of all reachable states. This is done with later automation in mind, and our algorithmic approach in Section 6 will first automatically synthesize this supporting invariant (or, alternatively, invariants can be provided by the user) before proceeding to the synthesis of an UESM. Example 2.2 shows an example UESM.

Definition 5.1 (Upper expectation supermartingale (UESM)). Let $C = (L, V, V_{out}, \ell_{init}, \mathbf{x}_{init}, \mapsto, G, Up)$ be an a.s. terminating pCFG, I be an invariant in C and $f : \mathbb{R}^{|V_{out}|} \to \mathbb{R}$ be a Borel-measurable function over the output variables of C. An *upper expectation supermartingale (UESM) for* f with respect to the invariant I is a state function U_f satisfying the following two conditions:

(1) *Zero on output.* For every $\mathbf{x} \models I(\ell_{out})$, we have $U_f(\ell_{out}, \mathbf{x}) = 0$.
(2) *Expected f-decrease.* For every location $\ell \in L$, transition $\tau = (\ell, Pr) \in \mapsto$, and valuation $\mathbf{x}$ s.t. $\mathbf{x} \models I(\ell) \wedge G(\tau)$, we require the following: for $\mathbf{N} = Next(\tau, \mathbf{x})$ it holds

$$U_f(\ell, \mathbf{x}) \geq \sum_{\ell' \in L} Pr(\ell') \cdot \mathbb{E}[U_f(\ell', \mathbf{N}) + f(\mathbf{N}^{out})] - f(\mathbf{x}^{out}) \tag{3}$$

(where $\mathbf{N}^{out}$ is the projection of the random vector $\mathbf{N}$ onto the V_{out}-indexed components). Intuitively, this condition requires that, in any step of computation, any increase in the f-value of the current valuation (projected onto the output variables) is matched, in expectation, by the decrease of the U_f-value.

Lower expectation submartingales. A lower expectation submartingale is defined analogously as an UESM, with the expected f-decrease condition replaced by the dual expected *f-increase* condition. Example 2.2 shows an example LESM.

Definition 5.2 (Lower expectation submartingale (LESM)). Let $C = (L, V, V_{out}, \ell_{init}, \mathbf{x}_{init}, \mapsto, G, Up)$ be an a.s. terminating pCFG, I be an invariant in C and $f : \mathbb{R}^{|V_{out}|} \to \mathbb{R}$ be a Borel-measurable function over the output variables of C. A *lower expectation submartingale (LESM) for f* with respect to the invariant I is a state function L_f satisfying the following two conditions:

(1) *Zero on output.* For every $\mathbf{x} \models I(\ell_{out})$, we have $L_f(\ell_{out}, \mathbf{x}) = 0$.
(2) *Expected f-increase.* For every location $\ell \in L$, transition $\tau = (\ell, Pr) \in \mapsto$, and valuation $\mathbf{x}$ s.t. $\mathbf{x} \models I(\ell) \wedge G(\tau)$, we require the following: for $\mathbf{N} = Next(\tau, \mathbf{x})$ it holds

$$L_f(\ell, \mathbf{x}) \leq \sum_{\ell' \in L} Pr(\ell') \cdot \mathbb{E}[L_f(\ell', \mathbf{N}) + f(\mathbf{N}^{out})] - f(\mathbf{x}^{out}). \tag{4}$$

5.2 Expectation Bounds via U/LESMs

In this subsection, we state Theorem 5.4, which shows that under certain conditions, a U/LESM for a function f provides an upper/lower bound on the expected value of f at output. The theorem is used to prove soundness of our proof rules for equivalence and similarity refutation in Section 5.3.

The result of Theorem 5.4 subsumes and unifies the proof rules of [52, 83]. Both these papers use Optional Stopping Theorem (OST) [84] to formulate conditions under which U/LESMs provide sound proof rules for computing expectation bounds. The proof rule of [52] uses the classical OST [84] (though only for lower bounds, see the discussion below), whereas the work of [83] derives what they call Extended OST to relax and replace some of the classical OST conditions.

Our approach to the formulation and proof of Theorem 5.4 differs from the proof rules in [52, 83] in the following aspects: First, in [83], the upper/lower cost supermartingales provided bounds on the expected value of a single *cost* variable whereas we consider arbitrary functions of the output variables. Second, the approach in [52] considers functions f taking values in the *non-negative* and *extended* real interval $[0, \infty]$. In such a case, the space of all such functions forms a complete lattice, which allows the use of Park induction [69] to obtain upper bounds. In contrast, we work with functions taking values in $(-\infty, \infty)$, which precludes such approach. We need to work with the co-domain $(-\infty, \infty)$ to achieve automation: our algorithm, presented in Section 6, works with functions represented via polynomials, which in general have this co-domain. Hence, we use OST for both upper and lower bounds.

We start by stating the conditions under which U/LESMs provide the required bounds, which we call the *OST-soundness conditions*. The first three conditions are conditions imposed by the classical OST [84], whereas the fourth condition is imposed by the Extended OST [83]. In the following, we use the notion of *conditional expectation*. For the sake of brevity, we omit its formal definition. Intuitively, when dealing with a pCFG C and a random variable X over the runs of C, we denote by $E[X \mid \mathcal{F}_t]$ the *conditional expected value* of X given the knowledge of the first t steps of C's run.

Definition 5.3 (OST-soundness). Let C be a pCFG, η be a state function in C, and $f : \mathbb{R}^{|V_{out}|} \to \mathbb{R}$ be a Borel measurable function. Denote by $Z_i(\rho)$ the i-the state along a run ρ, and let Y_i be defined by $Y_i := \eta(Z_i) + f(\mathbf{X}_i^{out})$ for any $i \geq 0$. We say that the tuple (C, η, f) is *OST-sound* if $\mathbb{E}[|Y_i|] < \infty$ for every $i \geq 0$ and moreover, at least one of the following conditions (C1)–(C4) holds:

(C1) There exists a constant c such that $\mathit{TimeTerm} \leq c$ with probability 1 (i.e., the termination time of the program is uniformly bounded).
(C2) There exists a constant c such that for each $t \in \mathbb{N}$ and each run ρ it holds that

$$|Y_{\min\{t,\mathit{TimeTerm}(\rho)\}}(\rho)| \leq c$$

(i.e., $Y_i(\rho)$ is uniformly bounded from below and above up until the point of termination).
(C3) $\mathbb{E}[\mathit{TimeTerm}] < \infty$, $\mathbb{E}[|Y_0|] < \infty$, and there exists a constant c such that for every $t \in \mathbb{N}$ it holds $\mathbb{E}[|Y_{t+1} - Y_t| \mid \mathcal{F}_t] \leq c$ (i.e., the expected one-step change of Y_i is uniformly bounded over the program runtime, even if conditioned by the whole past history of the program).
(C4) There exist real numbers M, c_1, c_2, d such that (i) for all sufficiently large $n \in \mathbb{N}$ it holds $\mathbb{P}(\mathit{TimeTerm} > n) \leq c_1 \cdot e^{-c_2 \cdot n}$; and (ii) for all $t \in \mathbb{N}$ it holds $|Y_{n+1} - Y_n| \leq M \cdot n^d$.

Our algorithm presented in Section 6 will automatically enforce OST-soundness. We are now ready to state the U/LESM soundness theorem.

Theorem 5.4 (Soundness of U/LESMs). *Let $C = (L, V, V_{out}, \ell_{init}, \mathbf{x}_{init}, \mapsto, G, Up)$ be an a.s. terminating pCFG with output distribution μ^C and $f : \mathbb{R}^{|V_{out}|} \to \mathbb{R}$ a Borel measurable function over the outputs of C. Let U_f and L_f be an upper (respectively lower) expectation supermartingale for f w.r.t. some invariant. Assume that (C, U_f, f) and (C, L_f, f) are OST-sound. Then $\mathbb{E}_{\mu^C}[f(\mathbf{x}^{out})]$ is well-defined and*

$$U_f(\ell_{init}, \mathbf{x}_{init}) + f(\mathbf{x}^{out}_{init}) \geq \mathbb{E}_{\mu^C}[f(\mathbf{x}^{out})],$$
$$L_f(\ell_{init}, \mathbf{x}_{init}) + f(\mathbf{x}^{out}_{init}) \leq \mathbb{E}_{\mu^C}[f(\mathbf{x}^{out})].$$

Proof (sketch). Let Z_n denote the n-th state along a run of C and $\mathbf{X}_n$ denotes the n-th valuation encountered along a run. For L_f, we define a stochastic process $Y = (Y_n)_{n=0}^{\infty}$ by putting $Y_n := L_f(Z_n) + f(\mathbf{X}_n^{out})$. The inequality for L_f follows from application of the (extended) optional stopping theorem [83, 84] to Y, which is permissible due to the OST-soundness assumption. The argument for U_f is analogous. Full proof can be found in the extended version of the paper [27]. □

5.3 Proof Rules for Equivalence and Similarity Refutation

We now show how to use the results in the previous section to derive refutation rules for the equivalence and similarity problems. Example 2.3 shows an application of this proof rule for equivalence refutation, and Example 2.4 for similarity refutation.

Theorem 5.5 (Soundness of equivalence and similarity refutation). *Consider two a.s. terminating pCFGs $C_1 = (L^1, V^1, V_{out}, \ell^1_{init}, \mathbf{x}^1_{init}, \mapsto^1, G^1, Up^1)$ and $C_2 = (L^2, V^2, V_{out}, \ell^2_{init}, \mathbf{x}^2_{init}, \mapsto^2, G^2, Up^2)$. Assume that there exists a Borel-measurable function $f : \mathbb{R}^{|V_{out}|} \to \mathbb{R}$ and two state functions, U_f for C_1 and L_f for C_2, such that the following holds:*

- *U_f is a UESM for f in C_1 such that (C_1, U_f, f) is OST-sound;*
- *L_f is an LESM for f in C_2 such that (C_2, L_f, f) is OST-sound;*
- *$U_f(\ell^1_{init}, \mathbf{x}^1_{init}) + f((\mathbf{x}^1_{init})^{out}) < L_f(\ell^2_{init}, \mathbf{x}^2_{init}) + f((\mathbf{x}^2_{init})^{out})$.*

Then C_1 and C_2 do not define equivalent output distributions.

Moreover, if f is 1-Lipschitz continuous under a metric d of the output space $\mathbb{R}^{|V_{out}|}$, then

$$\mathcal{K}_d\left(\mu^{C_1}, \mu^{C_2}\right) \geq L_f(\ell^2_{init}, \mathbf{x}^2_{init}) + f((\mathbf{x}^2_{init})^{out}) - U_f(\ell^1_{init}, \mathbf{x}^1_{init}) - f((\mathbf{x}^1_{init})^{out}).$$

Proof. From Theorem 5.4 we have

$$\begin{aligned}\mathbb{E}_{\mu^{C_1}}[f(\mathbf{x}^{out})] &\leq U_f(\ell^1_{init}, \mathbf{x}^1_{init}) + f((\mathbf{x}^1_{init})^{out}) \\ &< L_f(\ell^2_{init}, \mathbf{x}^2_{init}) + f((\mathbf{x}^2_{init})^{out}) \leq \mathbb{E}_{\mu^{C_2}}[f(\mathbf{x}^{out})] \end{aligned} \tag{5}$$

Hence, $\mathbb{E}_{\mu^{C_1}}[f(\mathbf{x}^{out})] < \mathbb{E}_{\mu^{C_2}}[f(\mathbf{x}^{out})]$, and so the output distributions μ^{C_1} and μ^{C_2} are not equivalent (otherwise, any measureable f would have the same expectation under both measures).

The second part follows directly from (5) and from the definition of the Kantorovich distance. □

To conclude this section, we highlight that searching for a Borel-measurable function $f : \mathbb{R}^{|V_{out}|} \to \mathbb{R}$ such that $\mathbb{E}_{\mu^{C_1}}[f(\mathbf{x}^{out})] \neq \mathbb{E}_{\mu^{C_2}}[f(\mathbf{x}^{out})]$ yields *both sound and complete* proof rule for refuting equivalence of output distributions. While soundess follows from Theorem 5.5, to prove completeness suppose that two output distributions are not equivalent. Then, there exists an event A over outputs such that $\mu^{C_1}[A] \neq \mu^{C_1}[A]$. Hence, with f being the indicator function $I(A)$ of the event A, which is indeed a Borel-measurable function, we have $E_{\mu^{C_1}}[I(A)] \neq E_{\mu^{C_1}}[I(A)]$.

6 AUTOMATED CONSTRAINT SOLVING-BASED ALGORITHM

In this section, we present our algorithms for automated equivalence and similarity refutation. In the sequel, let $C_1 = (L^1, V^1, V_{out}, \ell^1_{init}, \mathbf{x}^1_{init}, \mapsto^1, G^1, Up^1)$ and $C_2 = (L^2, V^2, V_{out}, \ell^2_{init}, \mathbf{x}^2_{init}, \mapsto^2, G^2, Up^2)$ be two a.s. terminating pCFGs with a common output variable set V_{out}.

Assumptions. Our algorithms impose the following assumptions:

- *Polynomial programs.* We consider probabilistic programs in which all arithmetic expressions are *polynomials* over program variables. Furthermore, by introducing dummy variables for expressions appearing in transition guards, we without loss of generality assume that arithmetic expressions appearing in transition guards are linear.
- *Finite moments of probability distributions.* We assume that each probability distribution δ appearing in sampling instructions has *finite moments* which are accessible to the algorithm, i.e. for each $p \in \mathbb{N}$, the p-th moment $m_\delta(p) = \mathbb{E}_{X\sim\delta}[|X|^p]$ is finite and can be computed by the algorithm. This is a standard assumption in static probabilistic program analysis and allows sampling instructions from most standard probability distributions.
- *Linear invariants.* Recall, in Definition 5.1 and Definition 5.2 we defined U/LESMs with respect to supporting invariants. We assume that we are provided with *linear invariants* I_1 and I_2 for C_1 and C_2, respectively. Linear invariant generation is a well-studied problem in program analysis; in our implementation, we use the methods of [39, 74] to synthesize supporting linear invariants I_1 and I_2.
- *OST-soundness.* Recall from Section 5.2 that we need to impose one of the OST-soundness conditions in Definition 5.3 on each pCFG for the proof rules based on U/LESMs to be sound. These conditions impose restrictions on the pCFG as well as on the function on outputs and the U/LESMs that we need to synthesize. In what follows, we state the restrictions imposed on pCFGs by each of the OST-conditions. In principle, a different restriction can be imposed on each of the two pCFGs. To streamline the presentation, we consider imposing the same condition on both pCFGs, in an order of preference specified in the next subsection. Then, depending on the OST-condition that the algorithm uses for the pCFGs, we will also constrain our output functions and U/LESMs to satisfy the corresponding restrictions. We use the same enumeration of OST-conditions as in Definition 5.3. We use $TimeTerm_1$ and $TimeTerm_2$ to denote random variables defined by termination time in C_1 and C_2:

(C1) The pCFGs have bounded termination time, i.e. there exists $c > 0$ s.t. $TimeTerm_1(\rho) \leq c$ for all runs ρ in C_1 and $TimeTerm_2(\rho) \leq c$ for all runs ρ in C_2. To enforce this condition, it suffices to restrict our attention to programs in which all loops are statically bounded.

(C2) This condition imposes no restrictions on pCFGs.

(C3) The pCFGs have bounded *expected* termination time, i.e. $\mathbb{E}^{C_i}[TimeTerm_i] < \infty$ for $i \in \{1, 2\}$. To verify this, it suffices to synthesize a ranking supermartingale (RSM) [19]. Automated synthesis of RSMs in polynomial programs was considered in [22, 24].

(C4) There exist real numbers c_1, c_2 such that, for all sufficiently large $n \in \mathbb{N}$, it holds $\mathbb{P}(\mathit{TimeTerm}_i > n) \leq c_1 \cdot e^{-c_2 \cdot n}$ for $i \in \{1, 2\}$. It was shown in [83] that, in polynomial programs, to verify the first condition it suffices to synthesize an RSM as in (C3).

6.1 Algorithm for the Equivalence Refutation Problem

Algorithm outline. Our algorithm uses constraint solving-based synthesis to simultaneously compute a function f over output variables V_{out}, an UESM U_f^1 for f in C_1 and an LESM L_f^2 for f in C_2. The algorithm proceeds in four steps. First, it fixes symbolic polynomial templates for f, U_f^1 and L_f^2. Second, it collects the defining constraints of UESMs and LESMs, the equivalence refutation constraint, and the constraints that encode OST-condition restrictions. Third, it translates these constraints into a linear programming (LP) instance. Fourth, it uses an LP solver to solve the resulting LP instance, with each solution giving rise to a valid triple of f, U_f^1 and L_f^2.

Our algorithm builds on classical constraint solving-based methods for static analysis of polynomial (probabilistic) programs for termination [22], reachability [5], safety [23] or cost [83, 86] properties. Hence, we keep our exposition brief and focus on the Step 2 of our algorithm which contains the main algorithmic novelty, since it collects the defining constraints of U/LESMs and the relational constraint for equivalence refutation.

Algorithm parameters. Our algorithm takes as an input a natural number parameter $d \in \mathbb{N}$ which denotes the maximal degree of polynomials that the algorithm uses for synthesis. Also, it determines which of the OST-soundness conditions to impose on the pCFGs as follows:

(1) If the pCFGs have bounded termination time (e.g. they only contain statically bounded loops), then our algorithm imposes (C1) on them since this condition does not introduce any constraints on f, U_f^1 and L_f^2.
(2) Else, if the pCFGs have bounded updates, i.e. there exists $M > 0$ such that every update element in each pCFG changes variable value by at most M, then our algorithm imposes (C4) on them. This is because it was shown in [83] that, for a polynomial program with bounded updates, (C, η, f) is OST-sound with (C4) satisfied for state function η and output function f. Hence, the algorithm need not introduce any constraints on f, U_f^1 and L_f^2.
(3) Else, if the pCFGs have bounded expected termination time (e.g. the method of [22] successfully synthesizes RSMs), then our algorithm imposes (C3) on them as this condition introduces milder constraints on f, U_f^1 and L_f^2 than (C2).
(4) Else, our algorithm imposes (C2) on the pCFGs.

Step 1: Symbolic templates. The algorithm fixes a symbolic polynomial template for f in the form of a symbolic polynomial of degree at most d over output variables V_{out}. It also fixes a symbolic polynomial template for the UESM U_f^1 for f in C_1, in the form of a symbolic polynomial $U_f^1(\ell)$ of degree at most d over program variables V^1 for each location $\ell \in L^1$. A template for the LESM L_f^2 for f in C_2 is fixed analogously.

This is formally done as follows. Let $\mathit{Mono}_d(V)$ and $\mathit{Mono}_d(V_{out})$ denote the sets of all monomials of degree at most d over the variables V and V_{out}, respectively. The templates for f, for $U_f^1(\ell)$ at each location $\ell \in L^1$ and for $L_f^2(\ell)$ at each location $\ell \in L^2$ are respectively defined by fixing the following symbolic polynomial expressions

$$\sum_{m \in \mathit{Mono}_d(V_{out})} f_m \cdot m, \qquad \sum_{m \in \mathit{Mono}_d(V)} u_m^\ell \cdot m, \qquad \sum_{m \in \mathit{Mono}_d(V)} l_m^\ell \cdot m,$$

where f_m, u_m^ℓ and l_m^ℓ are a real-valued *symbolic template variables* for each m and ℓ.

Step 2: Constraint collection. The algorithm collects all defining constraints for U_f^1 and L_f^2 to be a UESM and an LESM, the equivalence refutation constraint and the OST-soundness constraints. For every collected constraint that contains an expectation operator, the algorithm symbolically evaluates the expected values to obtain expectation-free polynomial expressions over symbolic template and program variables. This is possible since all involved expressions are polynomials over program variables, all moments of probability distributions are finite and can be computed, and moreover, the expectations integrate over a single variable in the polynomial expression (since in pCFGs, at most one variable is updated in each step).

(1) *UESM constraints.* By Definition 5.1, an UESM must satisfy the Zero on output condition and the Expected f-decrease condition. Both constraints are defined with respect to the supporting invariant I_1. As explained above, such invariants can be synthesized automatically, e.g. by methods of [39, 74]. The algorithm collects the following constraints:
- *Zero on output.* $\forall \mathbf{x} \in \mathbb{R}^{|V^1|}. \mathbf{x} \models I_1(\ell^1_{out}) \Rightarrow U_f^1(\ell^1_{out}, \mathbf{x}) = 0$.
- *Expected f-decrease.* For every transition $\tau = (\ell, Pr) \in \mapsto^1$ and for $\mathbf{N} = Next(\tau, \mathbf{x})$:

$$\forall \mathbf{x} \in \mathbb{R}^{|V^1|}. \mathbf{x} \models I_1(\ell) \wedge G^1(\tau) \Rightarrow U_f^1(\ell, \mathbf{x}) \geq \sum_{\ell' \in L^1} Pr(\ell') \cdot \mathbb{E}[U_f^1(\ell', \mathbf{N}) + f(\mathbf{N}^{out})] - f(\mathbf{x}^{out})$$

(2) *LESM constraints.* Analogously, the algorithm collects constraints for L_f^2 to be an LESM for f in C_2. As in Definition 5.2, constraints are defined w.r.t. the supporting invariant I_2.
- *Zero on output.* $\forall \mathbf{x} \in \mathbb{R}^{|V^2|}. \mathbf{x} \models I_2(\ell^2_{out}) \Rightarrow L_f^2(\ell^2_{out}, \mathbf{x}) = 0$.
- *Expected f-increase.* For every transition $\tau = (\ell, Pr) \in \mapsto^2$ and for $\mathbf{N} = Next(\tau, \mathbf{x})$:

$$\forall \mathbf{x} \in \mathbb{R}^{|V^2|}. \mathbf{x} \models I_2(\ell) \wedge G^2(\tau) \Rightarrow L_f^2(\ell, \mathbf{x}) \leq \sum_{\ell' \in L^2} Pr(\ell') \cdot \mathbb{E}[L_f^2(\ell', \mathbf{N}) + f(\mathbf{N}^{out})] - f(\mathbf{x}^{out})$$

(3) *Equivalence refutation constraint.* The algorithm collects the equivalence refutation constraint, according to Theorem 5.5:

$$U_f(\ell^1_{init}, \mathbf{x}^1_{init}) + f((\mathbf{x}^1_{init})^{out}) < L_f(\ell^2_{init}, \mathbf{x}^2_{init}) + f((\mathbf{x}^2_{init})^{out})$$

(4) *OST-soundness constraints.* Finally, the algorithm collects the constraints for OST-soundness conditions in Definition 5.3. Depending on which of the conditions $(C1)-(C3)$ in Definition 5.3 we impose (see Algorithm parameters above), we collect the following constraints:

(C1) No additional constraints are necessary.

(C4) No additional constraints are necessary.

(C2) We require that there exists a constant $C > 0$ such that the absolute value of the sum of the U/LESM and f is bounded from above by C at every reachable state. Thus, we introduce an additional symbolic variable for C, collect the constraint $C > 0$ and collect the following constraints for each $\ell \in L^1$ and $\ell \in L^2$, respectively:

$$\forall \mathbf{x} \in \mathbb{R}^{|V^1|}. \mathbf{x} \models I_1(\ell) \Rightarrow \left| U_f^1(\ell, \mathbf{x}) + f(\mathbf{x}^{out}) \right| \leq C$$

$$\forall \mathbf{x} \in \mathbb{R}^{|V^2|}. \mathbf{x} \models I_2(\ell) \Rightarrow \left| L_f^2(\ell, \mathbf{x}) + f(\mathbf{x}^{out}) \right| \leq C$$

(C3) We require that there exists a constant $C > 0$ such that the sum of the U/LESM and f has bounded expected one-step change at every reachable state. However, this condition yields a constraint which is not of the form as in eq. (6) that is needed in Step 3 for reduction to an LP instance. In order to allow for an automated synthesis by reduction to LP, we instead collect a stricter condition of bounded *maximal* one-step change at every reachable state. In particular, we introduce a symbolic variable for C, collect $C > 0$ constraint, and

collect the following constraint for each $\tau = (\ell, Pr) \in \mapsto^1$ and $\ell' \in L^1$ with $Pr(\ell') > 0$, and for each $\tau = (\ell, Pr) \in \mapsto^2$ and $\ell' \in L^2$ with $Pr(\ell') > 0$, respectively:

$$\forall \mathbf{x} \in \mathbb{R}^{|V^1|}, \mathbf{N} \in \mathit{supp}(\mathbf{N}).\, \mathbf{x} \models I_1(\ell) \wedge G^1(\tau) \Rightarrow \left| U^1_f(\ell, \mathbf{x}) + f(\mathbf{x}^{out}) - U^1_f(\ell', \mathbf{N}) - f(\mathbf{N}^{out}) \right| \leq C$$

$$\forall \mathbf{x} \in \mathbb{R}^{|V^2|}, \mathbf{N} \in \mathit{supp}(\mathbf{N}).\, \mathbf{x} \models I_2(\ell) \wedge G^2(\tau) \Rightarrow \left| L^2_f(\ell, \mathbf{x}) + f(\mathbf{x}^{out}) - L^2_f(\ell', \mathbf{N}) - f(\mathbf{N}^{out}) \right| \leq C$$

Step 3: Conversion to an LP instance. This step of our algorithm is analogous to [5, 22, 83, 86]. Observe that the equivalence refutation constraint is a linear and purely existentially quantified constraint over the symbolic template variables of f, U^1_f and L^2_f, since the initial variable valuations $\mathbf{x}^1_{init}$ and $\mathbf{x}^2_{init}$ are fixed. On the other hand, upon symbolically evaluating the expected values appearing in constraints, all other constraints collected in Step 2 above are of the form

$$\forall \mathbf{x} \in \mathbb{R}^{|V|}.\, \mathit{lin\text{-}exp}_1(\mathbf{x}) \geq 0 \wedge \cdots \wedge \mathit{lin\text{-}exp}_k(\mathbf{x}) \geq 0 \Rightarrow \mathit{poly\text{-}exp}(\mathbf{x}) \geq 0, \tag{6}$$

where $V \in \{V^1, V^2\}$, $\mathit{lin\text{-}exp}_i$ is a linear expression over variables in V for each $1 \leq i \leq k$, and *poly-exp* is a polynomial expression over variables in V (equalities and absolute values are encoded by two inequality constraints). This is due to our algorithm assumptions that the supporting invariants and transition guards are all defined in terms of linear expressions. Furthermore, the linear coefficients in each $\mathit{lin\text{-}exp}_i$ are constant values determined by transition guards or by supporting invariants, hence they do not contain any symbolic template variables.

It was shown in [5, 22, 83, 86] that entailments as in eq. (6) can be translated into purely existentially quantified *linear constraints* over the symbolic template variables (and auxiliary variables introduced by the translation), by using Handelman's theorem [51]. Using this translation, we convert the system of constraints collected in Step 2 into a system of purely existentially quantified linear constraints over the symbolic template variables and auxiliary variables introduced in translation. Thus, we obtain a linear programming (LP) instance without an optimization objective.

Step 4: LP solving. We feed the resulting LP instance to an off-the-shelf LP solver. The algorithm returns "Not output-equivalent" and outputs the computed f, U^1_f and L^2_f if the LP is successfully solved, or returns "Unknown" otherwise.

The following theorem establishes soundness of our algorithm for the equivalence refutation analysis. The proof can be found in the extended version [27].

Theorem 6.1 (Correctness of Equivalence Refutation). *Suppose that the algorithm outputs "Not output-equivalent". Then C_1 and C_2 are indeed not output-equivalent, and U^1_f and L^2_f are valid UESM and LESM for f, respectively.*

6.2 Algorithm for the Similarity Refutation Problem

We now outline the key additional steps needed to extend our algorithm in Section 6.1 to an algorithm for the Similarity refutation problem. For the interest of space, we omit the details and defer them to the extended version of the paper [27].

In addition to the parameters listed in Section 6.1, our algorithm for the Similarity refutation problem is also parameterized by the choice of a metric d over outputs and a lower bound $\epsilon > 0$ on the Kantorovich distance that we wish to prove. We allow any of the following standard metrics: L^p-metric, discrete metric, and uniform metric (all defined in Section **??**). The rest of the algorithm proceeds analogously as in Section 6.1, with the only difference being that in Step 2 of the algorithm we need to collect two additional constraints: (1) relational constraint on the lower bound on Kantorovich distance, and (2) 1-Lipschitz continuity of the function f on outputs.

Optimization of the Kantorovich distance. We note that our algorithm for the Similarity refutation problem reduces the synthesis of f, U_f^1 and L_f^2 to an LP instance without the optimization objective. Thus, our method can also *optimize* the lower bound on the Kantorovich distance by treating ϵ as a variable and adding the optimization objective to maximize ϵ.

7 EXPERIMENTAL RESULTS

We implemented a prototype[1] of our methods for equivalence and similarity refutation (the latter in terms of Kantorovich distance with respect to the L^1 metric). Our prototype takes as input two probabilistic programs having the same set of output variables V_{out}. The tool then checks (i) whether the output distributions of two programs are equivalent, and if not, (ii) whether it can compute a lower bound on their Kantorovich distance.

Implementation. We implemented our prototype in Java. We used Gurobi [50] to solve LP instances and ASPIC [39] and STING [74] to generate supporting linear invariants. Our implementation uses rational numbers for storing coefficients and variable values to avoid rounding and floating-point errors. For each input program pair, our prototype attempts to synthesize UESM/LESMs of a varying polynomial degree ranging from 1 to 5, progressively increased in case of failure. All experiments were run on an Ubuntu 22.04 machine with an 11th Gen Intel Core i5 CPU and 16 GB RAM with a timeout of 10 minutes.

Baseline. To refute the equivalence of two probabilistic programs, an alternative approach would be to use a state of the art symbolic integration tool such as PSI [42] to first compute probability density functions of output distributions of two probabilistic programs, and then check whether the two density functions are identical. Hence, we compare our method against a baseline which first uses PSI [42] to compute the probability density functions and then uses Mathematica [85] to compare the density functions. We note that PSI (and so our baseline) is only applicable to programs with statically bounded loops.

Benchmarks. As our benchmark set, we consider loopy probabilistic programs collected from [60, 83]. These benchmarks model various different applications, ranging from classical examples of random walks and their variants, coupon collector, to examples of academic interest such as queueing network and species fight, to realistic examples including Bit-coin mining. These programs have been verified to have finite expected termination time and bounded variable updates, which are sufficient conditions for the satisfaction of the (C4) OST-soundness condition as discussed in Section 6. Programs with statically bounded loops also satisfy the (C1) OST-soundness condition. As some of these programs contain unbounded loops which are not supported by PSI and so by our baseline, for each collected program we also consider three modifications where we force the loops to terminate after at most 10, 100 and 1000 iterations, respectively.

Each collected program was used to construct a pair of programs for equivalence and similarity analysis as follows: if the program contained non-deterministic branching, we constructed two programs of which one always chooses the if-branch and the other chooses the else-branch of the non-deterministic choice. For programs without non-determinism, we obtain the second program by injecting a small perturbation into exactly one sampling instruction, without further changes.

Discussion of Results. Table 1 shows our experimental results on the benchmark set described above together with the illustrating example from Section 2. Our method shows much better scalability compared to the baseline as the loop bound parameter is increased, demonstrating the advantage of a static analysis method that does not rely on (symbolic) execution of probabilistic programs with long executions. Moreover, our method is also able to compute Kantorovich distance

[1]We will make our prototype tool publicly available. Link to the implementation hidden for double blind reviewing.

lower bounds for most benchmarks. To the best of our knowledge, our method is the first automated method to compute lower bounds on Kantorovich distance.

Table 1. Experimental results showing: (1) comparison of our equivalence refutation method and the baseline, (2) lower boundson Kantorovich distance computed by our similarity refutation method, and (3) time taken to solve each instance. A ✓ in the "Eq. Ref." column represents that the tool successfully refuted equivalence of the two input programs, "TO" and "NA" stand for "timeout" and "Not Applicable", respectively.

	Name	Loop	Our Method			PSI + Mathematica	
			Eq. Ref.	Time(s)	Distance	Time(s)	Eq. Ref.	Time(s)
Benchmarks from [83]	Simple Example	10	✓	0.74	6.667	0.76	✓	6.30
		100	✓	0.41	66.667	0.40	TO	–
		1000	✓	0.34	666.667	0.34	TO	–
		original	✓	0.30	266.667	0.25	NA	–
	Nested Loop	10	✓	5.31	1.667	5.13	TO	–
		100	✓	3.30	16.667	3.13	TO	–
		1000	✓	2.84	166.667	2.82	TO	–
		original	✓	0.31	50	0.33	NA	–
	Random Walk	10	✓	0.76	2	0.69	✓	3.37
		100	✓	0.33	20	0.28	TO	–
		1000	✓	0.27	200	0.29	TO	–
		original	✓	0.24	9	0.22	NA	–
	Goods Discount	10	✓	0.88	0.125	0.46	TO	–
		100	✓	0.36	0.125	0.56	TO	–
		1000	✓	0.30	0.125	0.53	TO	–
		original	✓	0.35	0.008	0.56	NA	–
	Pollutant Disposal	10	✓	2.68	3.272	2.76	TO	–
		100	✓	3.05	32.727	2.83	TO	–
		1000	✓	3.34	327.272	3.18	TO	–
		original	✓	0.44	0.026	0.46	NA	–
	2D Robot	10	✓	0.59	TO	–	TO	–
		100	✓	0.57	TO	–	TO	–
		1000	✓	0.62	TO	–	TO	–
		original	✓	13.90	TO	–	NA	–
	Bitcoin Mining	10	✓	0.40	0.005	0.46	✓	1.22
		100	✓	0.23	0.05	0.21	✓	34.79
		1000	✓	0.34	0.5	0.46	TO	–
		original	✓	0.25	0.05	0.22	TO	–
	Bitcoin Mining Pool	10	✓	205.51	225.25	140.7	✓	1.90
		100	✓	121.43	22525	124.17	TO	–
		1000	✓	235.57	2252500	258.49	TO	–
		original	✓	129.98	122761.25	131.06	NA	–
	Species Fight	10	TO	–	TO	–	TO	–
		100	TO	–	TO	–	TO	–
		1000	TO	–	TO	–	TO	–
		original	✓	0.90	TO	–	NA	–
	Queuing Network	10	✓	0.99	TO	–	TO	94.42
		100	✓	0.77	TO	–	TO	–
		1000	✓	0.81	TO	–	TO	–
		original	✓	0.79	TO	–	TO	–
Benchmarks from [60]	coupon_collector	10	✓	0.67	0.167	0.89	✓	1.21
		100	✓	0.64	0.458	0.85	TO	6.45
		1000	✓	1.45	0.496	2.41	TO	–
		original	✓	1.10	0.5	1.41	NA	–
	coupon_collector4	10	✓	17.28	0.216	19.83	TO	3.04
		100	✓	16.92	1.215	18.49	TO	–
		1000	✓	31.18	1.471	27.93	TO	–
		original	✓	70.96	TO	TO	NA	–
	random_walk_1d_intvalued	10	✓	0.25	2	0.34	✓	1.49
		100	✓	0.20	20	0.19	✓	6.24
		1000	✓	0.41	200	0.41	TO	–
		original	✓	0.32	1.2	0.38	NA	–
	random_walk_1d realvalued	10	✓	0.29	2	0.33	TO	–
		100	✓	0.22	20	0.28	TO	–
		1000	✓	0.48	200	0.54	TO	–
		original	✓	0.27	3.841	0.50	NA	–
	random_walk_1d_adversary	10	✓	0.28	12.425	0.22	✓	1.49
		100	✓	0.26	138.425	0.22	TO	91.59
		1000	✓	0.59	1398.425	0.56	TO	–
		original	✓	0.38	0.768	0.40	NA	–
	random_walk_2d_demonic	10	✓	0.25	2	0.34	TO	–
		100	✓	0.27	20	0.27	TO	–
		1000	✓	0.65	200	0.67	TO	–
		original	✓	0.58	0.668	0.58	NA	–
	random_walk_2d_variant	10	✓	0.40	2	0.37	TO	–
		100	✓	0.31	20	0.37	TO	–
		1000	✓	0.83	200	0.94	TO	–
		original	✓	0.75	0.501	0.76	NA	–
Sec. 2	Transmission protocol (Figure 1)	10	✓	0.27	0.005	0.24	✓	1.19
		100	✓	0.19	0.05	0.21	✓	12.23
		1000	✓	0.47	0.5	0.57	TO	–
		original	✓	0.21	1.001	0.19	TO	–
Count			69	–	59	–	15	–
Average			–	12.88	–	12.94	–	17.77

As mentioned above, for the purpose of our experiments in Table 1, we used the (C4) OST-soundness condition which has been verified to be satisfied by all benchmarks. However, conditions (C1)-(C3) are also applicable to many of our benchmarks. An experimental comparison of the performance of our tool with different OST-soundness conditions is provided in the extended version of the paper [27].

We also observe one practical limitation of our approach: the performance of our automated method is dependent on the quality of *supporting linear invariants* generated for both programs. For some benchmarks in Table 1 (e.g. `coupon_collector`) the computed lower bound on distance does not scale with the number of loop iterations. We believe this is due to linear invariants generated for these programs being imprecise. When more precise invariants are available (e.g. `Nested Loop`), our method derives tighter bounds on distance. Moreover, while linear invariant generation is very efficient in most of our benchmarks, in some cases (e.g. `Bitcoin Mining Pool`) this was a highly computationally expensive task, leading to larger runtimes of our tool. There are also cases like the `2D robot` benchmark, where STING times out without returning any invariants, which is why our tool fails to compute a distance lower-bound. However, the invariants returned by ASPIC are strong enough for our tool to disprove equivalence. In other benchmarks where equivalence is refuted but no lower bound on distance is computed, we observe that supporting invariants are unbounded and so our tool could not normalize the function f on outputs to make it 1-Lipschitz continuous. Lastly, in cases like the finite loop instances of the `Species Fight` benchmark, non-polynomial functions are required for disproving equivalence, thus our tool fails.

Summary of Results. Our experiments demonstrate that our automated method can refute equivalence and compute lower bounds on the Kantorovich distance for a wide variety of probabilistic programs (Table 1). These results highlight scalability and efficiency of the method, especially when compared to the baseline based on symbolic integration. Hence, while suffering from certain limitations discussed above, we conclude that our method is applicable to a wide range of programs.

8 RELATED WORK

We discuss many of the existing static analyses for single probabilistic programs and statistical testing techniques for probability distributions in Section 1. Moreover, we provide a discussion on the comparison of our UESMs and LESMs to cost supermartingales of [83] and super- and sub-invariants of [52] in Remark 1 and in Section 5. Hence, we omit repetition and in the rest of this section we overview some other prior works on relational analysis of (probabilistic) programs.

Sensitivity analysis. Sensitivity analysis is a relational property that has received a lot of attention in static analysis of probabilistic programs [2, 8, 54, 82]. Given a probabilistic program and two inputs, the goal of sensitivity analysis is to derive bounds on the distance between output distributions on those inputs, towards verifying e.g. differential privacy [4, 9]. A prominent method for sensitivity analysis in probabilistic programs is based on coupling proofs [4, 8, 9]. While sensitivity analysis considers two inputs given the *same* probabilistic program, in this work we study the equivalence and similarity refutation problems for *probabilistic program pairs*. Our method does not assume any level of syntactic similarity of control flows of the two programs. In contrast, sensitivity analysis and methods based on coupling proofs often exploit the "aligned" control flow of two executions on sufficiently close inputs.

Equivalence analysis for finite-state probabilistic programs. There is a significant body of work on comparing *finite-state* Markov chains and Markov decision processes w.r.t. various notions of equivalence and similarity. This includes computing the total variation distance [31, 57], trace equivalence [59], contextual equivalence [62], and probabilistic bisimilarity [61, 77]. These works focus on finite-state models, whereas we consider Turing-complete probabilistic programs.

Accuracy of probabilistic inference. Several works have studied accuracy of probabilistic inference algorithms, e.g. by considering auxiliary inference divergence [34], bidirectional Monte Carlo [48, 49] and symmetric divergence over simulations [36]. The key distinction between these works and ours is that the former provide statistical guarantees, such as bounds for Kullback-Leibler (KL) divergence in expectation, while our work provides provably valid guarantees via static analysis.

Relational analyses in non-probabilistic programs. Prior work has studied static analysis with respect to a number of relational properties in non-probabilistic programs, including equivalence proving [40, 45], semantic differencing [70, 71], continuity [29] or differential cost analysis [32, 72, 86]. In particular, the method of [86] computes a bound on the difference in cost usage of a program pair by simultaneously computing an upper bound on cost for one program and a lower bound on cost for the other program, similarly to what we do with the synthesis of UESMs and LESMs. However, the key algorithmic difference is that we also *simultaneously synthesize the function on outputs* with respect to which the UESM and the LESM are defined. In contrast, in differential cost analysis a cost function is known a priori.

9 CONCLUSION

We presented a new martingale-based method for refuting the equivalence and similarity of output distributions of probabilistic programs. Our approach is fully automated, applicable to infinite-state programs, and provides formal guarantees on the correctness of its result. Our experimental results demonstrate the effectiveness of our approach on a range of probabilistic program pairs.

An interesting direction for future work is the extension of our methods to probabilistic programs with *observe* statements [68], that can express *epistemic* uncertainty about the system modeled by the program. The observe statements condition the output distribution by the event that all observations along the run are satisfied, and the task would be to refute the equivalence and similarity of such conditional distributions. Another direction is to consider improving our method for similarity refutation for programs with unbounded outputs, as discussed in Section 7. Yet another direction is to study the equivalence and similarity refutation problems for probabilistic programs that do not terminate almost-surely. Such programs define sub-distributions over their outputs, hence methods for these problems would need to reason about pairs of sub-distributions.

ACKNOWLEDGEMENTS

This research was partially supported by the ERC CoG 863818 (ForM-SMArt) grant. Petr Novotný is supported by the Czech Science Foundation grant no. GA23-06963S.

REFERENCES

[1] Sheshansh Agrawal, Krishnendu Chatterjee, and Petr Novotný. 2018. Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. *Proc. ACM Program. Lang.* 2, POPL (2018).
[2] Alejandro Aguirre, Gilles Barthe, Justin Hsu, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2021. A pre-expectation calculus for probabilistic sensitivity. *Proc. ACM Program. Lang.* 5, POPL (2021).
[3] Alejandro Aguirre, Gilles Barthe, Justin Hsu, and Alexandra Silva. 2018. Almost Sure Productivity. In *ICALP*.
[4] Aws Albarghouthi and Justin Hsu. 2018. Synthesizing coupling proofs of differential privacy. *Proc. ACM Program. Lang.* 2, POPL (2018).
[5] Ali Asadi, Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Mohammad Mahdavi. 2021. Polynomial reachability witnesses via Stellensätze. In *PLDI*.
[6] Martin Avanzini, Georg Moser, and Michael Schaper. 2020. A modular cost analysis for probabilistic programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020).
[7] Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. 2022. Data-Driven Invariant Learning for Probabilistic Programs. In *CAV*.
[8] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2018. Proving expected sensitivity of probabilistic programs. *Proc. ACM Program. Lang.* 2, POPL (2018).

[9] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. Proving Differential Privacy via Probabilistic Couplings. In *LICS*.

[10] Gilles Barthe, Marco Gaboardi, Justin Hsu, and Benjamin C. Pierce. 2016. Programming language techniques for differential privacy. *ACM SIGLOG News* 3, 1 (2016).

[11] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *POPL*.

[12] Gilles Barthe, Charlie Jacomme, and Steve Kremer. 2022. Universal Equivalence and Majority of Probabilistic Programs over Finite Fields. *ACM Trans. Comput. Log.* 23, 1 (2022).

[13] Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva. 2020. *Foundations of probabilistic programming*. Cambridge University Press.

[14] Tugkan Batu, Lance Fortnow, Ronitt Rubinfeld, Warren D. Smith, and Patrick White. 2013. Testing Closeness of Discrete Distributions. *J. ACM* 60, 1 (2013).

[15] 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 *TACAS*.

[16] Raven Beutner, C.-H. Luke Ong, and Fabian Zaiser. 2022. Guaranteed bounds for posterior inference in universal probabilistic programming. In *PLDI*.

[17] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.* 20 (2019).

[18] Clément L Canonne. 2020. A survey on distribution testing: Your data is big. But is it blue? *Theory of Computing* (2020).

[19] Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *CAV*.

[20] Sourav Chakraborty and Kuldeep S. Meel. 2019. On Testing of Uniform Samplers. In *AAAI*.

[21] Siu-on Chan, Ilias Diakonikolas, Paul Valiant, and Gregory Valiant. 2014. Optimal Algorithms for Testing Closeness of Discrete Distributions. In *SODA*.

[22] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *CAV*.

[23] Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. 2020. Polynomial invariant generation for non-deterministic recursive programs. In *PLDI*.

[24] Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. 2018. Algorithmic Analysis of Qualitative and Quantitative Termination Problems for Affine Probabilistic Programs. *ACM Trans. Program. Lang. Syst.* 40, 2 (2018).

[25] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Tobias Meggendorfer, and Dorde Zikelic. 2022. Sound and Complete Certificates for Quantitative Termination Analysis of Probabilistic Programs. In *CAV*.

[26] Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, Jiri Zárevúcky, and Dorde Zikelic. 2021. On Lexicographic Proof Rules for Probabilistic Termination. In *FM*.

[27] Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, and Đorđe Žikelić. 2024. Equivalence and Similarity Refutation for Probabilistic Programs. arXiv:2404.03430 [cs.PL]

[28] Krishnendu Chatterjee, Petr Novotný, and Dorde Zikelic. 2017. Stochastic invariants for probabilistic termination. In *POPL*.

[29] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2010. Continuity analysis of programs. In *POPL*.

[30] Mingshuai Chen, Joost-Pieter Katoen, Lutz Klinkenberg, and Tobias Winkler. 2022. Does a Program Yield the Right Distribution? - Verifying Probabilistic Programs via Generating Functions. In *CAV*.

[31] Taolue Chen and Stefan Kiefer. 2014. On the Total Variation Distance of Labelled Markov Chains. In *CSL-LICS*.

[32] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational cost analysis. In *POPL*.

[33] Ryan Culpepper and Andrew Cobb. 2017. Contextual equivalence for probabilistic programs with continuous random variables and scoring. In *ESOP*.

[34] Marco F. Cusumano-Towner and Vikash K. Mansinghka. 2017. AIDE: An algorithm for measuring the accuracy of probabilistic inference algorithms. In *NIPS*.

[35] Yuxin Deng and Wenjie Du. 2009. The Kantorovich metric in computer science: A brief survey. *Electronic Notes in Theoretical Computer Science* 253, 3 (2009).

[36] Justin Domke. 2021. An Easy to Interpret Diagnostic for Approximate Inference: Symmetric Divergence Over Simulations. *CoRR* abs/2103.01030 (2021).

[37] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing probabilistic programming systems. In *FSE*.

[38] Saikat Dutta, Wenxian Zhang, Zixin Huang, and Sasa Misailovic. 2019. Storm: program reduction for testing and debugging probabilistic programming systems. In *FSE*.

[39] Paul Feautrier and Laure Gonnord. 2010. Accelerated Invariant Generation for C Programs with Aspic and C2fsm. In *TAPAS@SAS*.
[40] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating regression verification. In *ASE*.
[41] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *ESOP*.
[42] Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *CAV*.
[43] Andrew Gelman, Daniel Lee, and Jiqiang Guo. 2015. Stan: A probabilistic programming language for Bayesian inference and optimization. *Journal of Educational and Behavioral Statistics* 40, 5 (2015).
[44] Zoubin Ghahramani. 2015. Probabilistic machine learning and artificial intelligence. *Nat.* 521, 7553 (2015), 452–459.
[45] Benny Godlin and Ofer Strichman. 2013. Regression verification: proving the equivalence of similar programs. *Softw. Test. Verification Reliab.* 23, 3 (2013).
[46] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Kallista A. Bonawitz, and Joshua B. Tenenbaum. 2008. Church: a language for generative models. In *UAI*.
[47] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic programming. In *FOSE*.
[48] Roger B. Grosse, Siddharth Ancha, and Daniel M. Roy. 2016. Measuring the reliability of MCMC inference with bidirectional Monte Carlo. In *NIPS*.
[49] Roger B. Grosse, Zoubin Ghahramani, and Ryan P. Adams. 2015. Sandwiching the marginal likelihood using bidirectional Monte Carlo. *CoRR* abs/1511.02543 (2015).
[50] Gurobi Optimization, LLC. 2023. Gurobi Optimizer Reference Manual. https://www.gurobi.com
[51] David Handelman. 1988. Representing polynomials by positive linear functions on compact convex polyhedra. *Pacific J. Math.* 132, 1 (1988).
[52] Marcel Hark, Benjamin Lucien Kaminski, Jürgen Giesl, and Joost-Pieter Katoen. 2020. Aiming low is harder: induction for lower bounds in probabilistic program verification. *Proc. ACM Program. Lang.* 4, POPL (2020).
[53] Leen Helmink, M. P. A. Sellink, and Frits W. Vaandrager. 1993. Proof-Checking a Data Link Protocol. In *TYPES*.
[54] Zixin Huang, Zhenbang Wang, and Sasa Misailovic. 2018. PSense: Automatic Sensitivity Analysis for Probabilistic Programs. In *ATVA*.
[55] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2019. On the hardness of analyzing probabilistic programs. *Acta Informatica* 56, 3 (2019).
[56] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2018. Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms. *J. ACM* 65, 5 (2018).
[57] Stefan Kiefer. 2018. On Computing the Total Variation Distance of Hidden Markov Models. In *ICALP*.
[58] Stefan Kiefer, Andrzej S. Murawski, Joël Ouaknine, Björn Wachter, and James Worrell. 2011. Language Equivalence for Probabilistic Automata. In *CAV*.
[59] Stefan Kiefer and Qiyi Tang. 2020. Comparing Labelled Markov Decision Processes. In *FSTTCS*.
[60] Satoshi Kura, Natsuki Urabe, and Ichiro Hasuo. 2019. Tail Probabilities for Randomized Program Runtimes via Martingales for Higher Moments. In *TACAS*.
[61] Kim G. Larsen and Arne Skou. 1991. Bisimulation through probabilistic testing. *Information and Computation* 94, 1 (1991).
[62] Axel Legay, Andrzej S. Murawski, Joël Ouaknine, and James Worrell. 2008. On Automated Verification of Probabilistic Programs. In *TACAS*.
[63] Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. 2018. A new proof rule for almost-sure termination. *Proc. ACM Program. Lang.* 2, POPL (2018).
[64] Sean P Meyn and Richard L Tweedie. 2012. *Markov chains and stochastic stability*. Springer Science & Business Media.
[65] Andrzej S. Murawski and Joël Ouaknine. 2005. On Probabilistic Program Equivalence and Refinement. In *CONCUR*.
[66] Chandrakana Nandi, Dan Grossman, Adrian Sampson, Todd Mytkowicz, and Kathryn S. McKinley. 2017. Debugging probabilistic programs. In *MAPL@PLDI*.
[67] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. In *PLDI*.
[68] Federico Olmedo, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Annabelle McIver. 2018. Conditioning in Probabilistic Programming. *ACM Trans. Program. Lang. Syst.* 40, 1 (2018).
[69] David Park. 1969. Fixpoint induction and proofs of program properties. *Machine intelligence* 5 (1969).
[70] Nimrod Partush and Eran Yahav. 2013. Abstract Semantic Differencing for Numerical Programs. In *SAS*.
[71] Nimrod Partush and Eran Yahav. 2014. Abstract semantic differencing via speculative correlation. In *OOPSLA*.

[72] Weihao Qu, Marco Gaboardi, and Deepak Garg. 2021. Relational cost analysis in a functional-imperative setting. (2021).
[73] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *PLDI*.
[74] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. 2004. Constraint-Based Linear-Relations Analysis. In *SAS*.
[75] Toru Takisaka, Yuichiro Oyabu, Natsuki Urabe, and Ichiro Hasuo. 2021. Ranking and Repulsing Supermartingales for Reachability in Randomized Programs. *ACM Trans. Program. Lang. Syst.* 43, 2 (2021).
[76] Sebastian Thrun. 2000. Probabilistic Algorithms in Robotics. *AI Mag.* 21, 4 (2000).
[77] Mathieu Tracol, Josée Desharnais, and Abir Zhioua. 2011. Computing Distances between Probabilistic Automata. In *QAPL*.
[78] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep probabilistic programming. In *ICLR*.
[79] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. *CoRR* abs/1809.10756 (2018). http://arxiv.org/abs/1809.10756
[80] Cédric Villani. 2021. *Topics in optimal transportation*. Vol. 58. American Mathematical Soc.
[81] Di Wang, Jan Hoffmann, and Thomas W. Reps. 2021. Central moment analysis for cost accumulators in probabilistic programs. In *PLDI*.
[82] Peixin Wang, Hongfei Fu, Krishnendu Chatterjee, Yuxin Deng, and Ming Xu. 2020. Proving expected sensitivity of probabilistic programs with randomized variable-dependent termination time. *Proc. ACM Program. Lang.* 4, POPL (2020).
[83] Peixin Wang, Hongfei Fu, Amir Kafshdar Goharshady, Krishnendu Chatterjee, Xudong Qin, and Wenjun Shi. 2019. Cost analysis of nondeterministic probabilistic programs. In *PLDI*.
[84] David Williams. 1991. *Probability with Martingales*. Cambridge University Press.
[85] Wolfram Research, Inc. 2022. *Mathematica 13.2*. https://www.wolfram.com
[86] Dorde Zikelic, Bor-Yuh Evan Chang, Pauline Bolignano, and Franco Raimondi. 2022. Differential cost analysis with simultaneous potentials and anti-potentials. In *PLDI*.

Received 2023-11-16; accepted 2024-03-31