Static Analysis for Probabilistic Programs: Inferring Whole Program Properties from Finitely Many Paths.

Sriram Sankaranarayanan
University of Colorado, Boulder.
srirams@colorado.edu

Aleksandar Chakarov
University of Colorado, Boulder.
aleksandar.chakarov@colorado.edu

Sumit Gulwani
Microsoft Research, Redmond.
sumitg@microsoft.com

Abstract

We propose an approach for the static analysis of probabilistic programs that sense, manipulate, and control based on uncertain data. Examples include programs used in risk analysis, medical decision making and cyber-physical systems. Correctness properties of such programs take the form of *queries* that seek the probabilities of assertions over program variables. We present a static analysis approach that provides guaranteed interval bounds on the values (assertion probabilities) of such queries. First, we observe that for probabilistic programs, it is possible to conclude facts about the behavior of the entire program by choosing a finite, *adequate set* of its paths. We provide strategies for choosing such a set of paths and verifying its adequacy. The queries are evaluated over each path by a combination of symbolic execution and probabilistic volume-bound computations. Each path yields interval bounds that can be summed up with a "coverage" bound to yield an interval that encloses the probability of assertion for the program as a whole. We demonstrate promising results on a suite of benchmarks from many different sources including robotic manipulators and medical decision making programs.

Categories and Subject Descriptors D.2.4 [**Software Engineering**] Software/Program Verification.

Keywords: Probabilistic Programming, Program Verification, Volume Bounding, Symbolic Execution, Monte-Carlo Sampling.

1. Introduction

The goal of this paper is to study static analysis techniques for proving properties of probabilistic programs that manipulate *uncertain* quantities defined by probability distributions. Uncertainty is a common aspect of many software systems, especially systems that manipulate error-prone data to make medical decisions (eg., should the doctor recommend drugs or dialysis to a patient based on the calculated eGFR score?); systems that predict long term risks of catastrophic events such as floods and earthquakes (eg., should a nuclear power plant be built at a certain location?); and control systems that operate in the presence of sensor errors and external environmental disturbances (eg., robotic manipulators and air traffic control systems). It is essential to learn how the presence of uncertainty in the inputs can affect the program's behavior.

In general, the problem of reasoning about uncertain systems is quite challenging due to (a) real-valued variables that are used to model physical quantities such as space, time and temperature; (b) imprecision in the input, modeled by different distributions such as uniform, Poisson and Gaussian; and (c) control flow that is mediated by conditions over the program variables and random variables. These aspects make the resulting programs infinite state, requiring techniques that go beyond the exhaustive exploration of concrete sets of states.

In this paper, we study infinite state programs that manipulate real-valued variables and uncertain quantities belonging to a wide variety of distributions. Our modeling approach is simple: we use standard imperative programs augmented with probabilistic constructs to model uncertainties. The overall goal of our approach is to answer questions about the probability of assertions at the exit of the program.

Monte-Carlo simulation [34] can be used to answer this question upto some desired level of precision. However, these techniques do not provide formal guarantees on the accuracy of the answers. In this work, we provide a static analysis approach that can place guaranteed interval bounds on the possible values of the assertion probabilities. As a side effect, our approach can perform a *stratified sampling* to provide more accurate answers than standard rejection sampling. Our static analysis approach has two main components: (A) choosing an adequate set of paths and (B) path probability bound computation.

Adequate Set of Paths: Unlike static analysis in the non-probabilistic case, it suffices to consider a carefully chosen finite set of *adequate program paths* for estimating probability bounds. Let us assume that a set of syntactic program paths S is provided with a lower bound c on their combined path probabilities. We can proceed to estimate the probability of any assertion φ over the finitely many paths in the set S and know that the contribution from the unexplored paths is at most $1 - c$.

Therefore, given some target "coverage" probability c (eg., $c = 0.95$), we wish to find a set S of syntactic program paths (if such a set exists) such that an execution of the program yields a path in the set S with probability at least c. We achieve this in two steps: (a) we use a statistical procedure to find a set S that achieves the coverage bound with high confidence, and (b) we then compute a formal lower bound on the actual coverage of S. Using a two-step approach guarantees that the symbolic execution is always performed over feasible program paths. Secondly, executing probabilistic programs is a natural way of sampling the paths of the program according to their path probabilities.

Path Probability Computation: We use symbolic execution along the chosen paths to compute bounds on the path probability: i.e, the probability that an execution of the program follows the chosen path. In turn, this reduces to finding the probability of satisfaction for a system of constraints, given probability distributions on the individual variables. This is a hard problem both in theory and in practice [2]. Therefore, rather than compute the precise answer, we seek to bound the probability of satisfaction. We present a scheme that can estimate probability bounds for linear assertions over integers and reals. Furthermore, given more time our technique can generate bounds that are tighter.

Our overall framework weaves the two ideas together to yield upper and lower bounds for the probability of the assertion for the program as a whole. The paper makes the following contributions:

1. We present an approach to infer bounds on assertion probability for the whole program by considering a suitably chosen finite set of program paths.
2. We present branch-and-bound techniques over polyhedra to derive tight bounds on path and assertion probabilities.
3. We present an experimental evaluation of our approach over a set of small but compelling benchmarks. We showcase some of the successes of our approach and identify limitations to motivate future work.

However, we have by no means solved every aspect of the complex problem of probabilistic program analysis.

1. Our approach is restricted to programs with linear assignments and conditionals (see Figure 3 in Section 3). As such, the high level ideas presented here directly extend to programs with non-linearities that arise from multiplications, exponentiations and bitwise operations. The current limitation lies in the volume computation technique (Section 4.2) that handles convex polyhedra over reals and integers.
2. We support the generation of random variables with known distributions and known correlations specified, for instance, by a given covariance matrix. However, our framework does not handle non-deterministic uncertainties or distributions with unknown (arbitrary) correlations.
3. Our focus is on estimating probability bounds for safety properties. We do not consider liveness properties (termination) or expectation queries in this paper.

2. Approach At A Glance

In this section, we first motivate the main contributions of this paper using an example from medical decision making. The probabilistic programming language and its semantics are described in the subsequent section.

Estimating Kidney Disease Risk in Adults Chronic kidney disease in adults is characterized by the gradual loss of kidney function. A common measure of chronic kidney disease is a quantity called the *Estimated Glomerular Filtration Rate* (eGFR), which is computed based on the age, gender, ethnicity and the measured serum Creatinine levels of the patient [1]. There are many formulae that are used for computing eGFR, which is typically a log-linear function of its inputs. Figure 1 shows the function that computes the value of eGFR according to the widely used CKD-EPI formula [2]. Kidney disease risk in adults is typically diagnosed if $\log(eGFR) < 4.5$, or equivalently $eGFR < 90$. However, the values of age and measured serum Creatinine levels can be erroneous. Age is typically rounded up or down causing a ± 1 variation, while the lab values of serum Creatinine levels are assumed to have a 10% variation. While gender and ethnicity are often mistake-free, it is sometimes possible to record a wrong gender or ethnicity due to

```
real estimateLogEGFR( real logScr, int age,
                      bool isFemale, bool isAA){
  var k,alpha: real;
  var f: real;
  f= 4.94;
  if (isFemale) {
     k = -0.357;
     alpha = -0.329;
  } else {
     k = -0.105;
     alpha = -0.411;
  }

  if ( logScr < k ) {
     f = alpha * (logScr - k);
  } else {
     f = -1.209 * (logScr - k);
  }
  f = f - 0.007 *  age;

  if (isFemale)  f = f + 0.017;
  if (isAA) f = f + 0.148;
  return f;
}
```

Figure 1. A program that computes the logarithm of the eGFR given the logarithm of the measured serum Creatinine value (logScr), the patient's age, whether the patient is female (isFemale) and whether the patient is African American (isAA). We assume that the inputs to the function are noisy and wish to know how the noise affects the output value f.

```
void compareWithNoise(real logScr, real age,
                      bool isFemale, bool isAA) {
  f1 = estimateLogEGFR(logScr, age, isFemale,isAA);
  logScr = logScr + uniformRandom(-0.1, 0.1);
  age = age + uniformRandomInt(-1,1);
  if (flip(0.01))
    isFemale = not( isFemale );
  if (flip(0.01))
    isAA = not( isAA );
  f2 = estimateLogEGFR(logScr, age, isFemale,isAA);
  estimateProbability (f1 - f2 >= 0.1);
  estimateProbability (f2 - f1 >= 0.1);
}
```

Figure 2. A probabilistic program that simulates noise in patient data, comparing the error between original and noisy outputs.

transcription mistakes in the (electronic) medical record. Figure 2 shows a model that adds the appropriate amount of noise to the inputs of the eGFR calculation. Given these input uncertainties, one naturally worries about the problem of patients being mis-classified purely due to the noise in the recorded patient data. Therefore, we ask the question: what is the probability that the presence of noise causes 10% or larger variation in the EGFR values (≥ 0.1 absolute variation in the logarithmic scale)? Such a query is expressed at the end of the function `compareWithNoise` in Figure 2.

Modeling Input Distributions: The distributions of serum Creatinine values are highly correlated with age, gender and race. A detailed population survey is provided by Jones et al. [19]. For illustration, we assume a 50% chance of the person being female and 10% chance of being African American. Based on these outcomes, different distributions are chosen for the patients based on their gender. We model input serum Creatinine levels as drawn from a normal distribution around the mean with a fixed standard devi-

[1] Cf. `http://www.kidney.org/kidneydisease/aboutckd.cfm`

[2] Cf. `http://nephron.com/epi_equation`.

ation of 0.1 around a mean that varies based on gender: 1.0 for women and 1.2 for men. A full model based on population data can be constructed and used in our static analysis. Table 3 summarizes the two basic primitives that we expect from any distribution supported by our analysis.

Estimating Probability of Assertions: For the example at hand, our approach proceeds in three steps:

1. We first generate a finite, adequate set of paths S such that the probability that a random execution yields a path in S is at least c for some fixed c. Fixing $c = 0.95$, we use the procedure in Algorithm 2 to yield 49 distinct paths.
2. Next, we analyze each of these paths and estimate intervals for the path probability and the probability for each of the assertions. For the example at hand, this requires less than 5 seconds of computational time for all paths and assertions.
3. We add up the path probability intervals for individual paths in the set S to infer a bound of $[0.98117, 0.98119]$ on the total probability of paths in S. This means that the paths in S are guaranteed to be encountered with probability at least 0.98117. Therefore, the probability of drawing a path not in S is at most 0.01883. We will make use of this fact in the next step.
4. We add up path probability intervals for the assertion `f1-f2 >= 0.1` to obtain a bound $[0.12630, 0.126306]$. However, since we did not cover all paths in the program, this bound is not valid for the whole program. To address this, we add 0.01883, an upper bound on the probability of encountering a path outside S. This yields a guaranteed bound of $[0.126306, 0.145136]$ for the probability that a given execution satisfies the assertion. Likewise, the probability of the assertion `f2-f1 >= 0.1` is estimated at $[0.08503, 0.10386]$.
5. We simulated the program for $1,000,000$ iterations in Matlab and estimated the probability of the assertions, taking nearly as much time as our static analysis. The bounds obtained by our analysis are confirmed.

Thus, our analysis provides bounds on the probability of obtaining a 10% or larger deviation in the computed risk score due to errors in the input data.

The rest of the paper is organized as follows: Section 3 presents the syntax and semantics of probabilistic programs, the process of bounding probabilities is explained in Section 4, Section 5 discusses our implementation and some experimental results on an interesting set of benchmarks. Related work is discussed in Section 6. We conclude by discussing future directions. Supplementary materials, including our prototype implementation and the benchmarks used in this paper are available on-line from our project page [3].

3. Probabilistic Programs

We now consider imperative programs with constructs that can generate random values according to a fixed set of distributions. We then describe queries over probabilistic programs that seek (conditional) probabilities of assertions.

Fig. 3 presents the key parts of the syntax for a probabilistic programming language for specifying programs. Let $X = \{x_1, \ldots, x_k\}$ be a set of program variables partitioned into subsets Q and I of real and integer typed variables, respectively. The language consists of assignments, conditional statements and loops. The expressions in the language are restricted in two ways: (1) The language is *strongly typed*: integer values cannot be converted to

program	$\rightarrow$	declarations init {initSpec*}stmt*queries
stmt	$\rightarrow$	assign | condStmt | while
initSpec	$\rightarrow$	intVariable := intConst | realVariable := realConst
	|	intVariable $\sim$ intRandom
	|	realVariable $\sim$ realRandom
assign	$\rightarrow$	intAssign | realAssign
condStmt	$\rightarrow$	if boolExpr stmt$^+$ else stmt$^+$
while	$\rightarrow$	while boolExpr stmt$^+$
intAssign	$\rightarrow$	intVariable := intExpr
realAssign	$\rightarrow$	realVariable := realExpr
intExpr	$\rightarrow$	intConst | intExpr $\pm$ intExpr
	|	intConst * intExpr | intRandom
realExpr	$\rightarrow$	realConst | realExpr $\pm$ realExpr
	|	realConst * realExpr | realRandom
intRandom	$\rightarrow$	uniformInt (intConst, intConst)
	|	Bernoulli (intConst, realConst)
	|	...
realRandom	$\rightarrow$	uniformReal (realConst, realConst)
	|	Gaussian (realConst, realConst)
	|	...
boolExpr	$\rightarrow$	boolExpr $\wedge$ boolExpr
	|	intExpr relop intExpr | realExpr relop realExpr
	|	*true* | *false*
relop	$\rightarrow$	$<$ | $>$ | $\geq$ | $\leq$ | $=$
intVariable	$\rightarrow$	I
realVariable	$\rightarrow$	Q
queries	$\rightarrow$	estimateProbability boolExpr(given boolExpr)?

Figure 3. Partial syntax specification for imperative language for modeling programs.

reals, or vice-versa. (2) The expressions involving integer and real variables are affine.

Inbuilt Random Value Generators: The programs in our language can call a set of inbuilt random number generators to produce values according to known distributions, as summarized in Table 2.

Some distributions can be automatically implemented based on other available ones. For instance, a coin flip with success probability $p \in [0, 1]$ can be simulated by a uniform random variable $r \sim \mathsf{uniformReal}(0, 1)$ and the conditional statement involving the condition $r \leq p$.

Initializer: The program has an initialization section that is assumed to initialize every variable $x_i \in X$. The initialization for a variable x_i can assign it to a concrete value c_i, or to a randomly distributed value by calling an in-built random number generator.

Queries: Queries in our language involve probabilities of Boolean expressions over program variables, possibly conditioned on other Boolean expressions. Table 1 summarizes the intended meaning of each of the two types of queries. For simplicity of presentation, we restrict queries to conjunctions of linear inequalities. However, queries involving disjunctions $\varphi_1 \vee \varphi_2$ can be expressed by using a special indicator variable x, which can be set to 1 iff either disjunct is true. The transformed query evaluates the probability of $x = 1$.

3.1 Semantics of Probabilistic Programs

The operational semantics of the execution of assignments, if-then-else, and while statements are quite standard. We focus on defining the semantics of the initialization, the random number generators, and the queries.

Let P be a probabilistic program written using the syntax in Fig. 3. We associate a set of *control locations*, *Loc* $= \{\ell_0, \ldots, \ell_k\}$,

[3] Cf. `http://systems.cs.colorado.edu/research/cyberphysical/probabilistic-program-analysis/`

Query	Remark
`estimateProbability(`φ`)`	The execution terminates AND variables at exit satisfy φ.
`estimateProbability(`φ `given` ψ`)`	The execution terminates AND variables satisfy φ under the condition that they satisfy ψ.

Table 1. Queries on the final state of the program and their informal meaning.

Name	Support	Density Function
`uniformReal(a,b)`	$[a,b] \subseteq \mathbb{R}$	$\frac{1}{b-a}\mathbb{1}_{[a,b]}$, $a < b$.
`uniformInt(a,b)`	$[a,b] \subseteq \mathbb{Z}$	$\begin{cases} \frac{1}{b-a}\mathbb{1}_{[a,b]} & b > a \\ 1 & a = b \end{cases}$.
`Binomial(n,p)`	$[0,n] \subseteq \mathbb{Z}$	$\binom{n}{x}p^x(1-p)^{n-x}$, $n \geq 1,\ p \in [0,1]$.
`Gaussian(m,s)`	$\mathbb{R}$	$\frac{1}{\sqrt{2\pi s}}e^{-\frac{1}{2}\frac{(x-m)}{s}^2}$, $s > 0$

Table 2. Specification of the inbuilt random value generators. Note that the function $\mathbb{1}_X(x)$ is the indicator function for the set X.

```
      state x: real;
      state c: int;
      init {  x := uniformReal(-1,3);
              c := 0;  }
--ℓ0: while (x <= 4) {
--ℓ1:    x := x + uniformReal(-1,3);
            c := c + uniformInt(0,2);
         }
--ℓF: estimateProbability(c <= 4);
```

Figure 4. A simple probabilistic program with a probability estimation query. We consider the assignments to variables x,c as a simultaneous assignment.

such that each statement in P is associated with a unique control location. The set of labels define a control flow graph with nodes corresponding to statement labels and edges corresponding to conditional tests and assignments between program variables. Let ℓ_0 be the location after the initializer and ℓ_F be the end of the program, where queries may be posed. A state of the program is a tuple $\langle \ell_j, \vec{x}_j \rangle$ given by its current control location $\ell_j \in Loc$ and valuations to the program variables $\vec{x}_j \in \mathbb{R}^{|Q|} \times \mathbb{Z}^{|I|}$.

The initializer assigns each variable to a fixed constant value or a random value sampled according to a known probability distribution. All random variables are assumed to be mutually independent.

Since our main goal is to define and evaluate queries, we define the semantics operationally. Let $\Pi = \{\pi_1, \ldots, \pi_N, \ldots\}$ be a set of syntactic program paths that start from location ℓ_0 and lead to the final location ℓ_F. Each path π_j is a finite sequence $\pi_j : \ell_0 \to \ell_1 \to \cdots \to \ell_l \to \ell_F$, where each (ℓ_i, ℓ_{i+1}) is an edge in the program's CFG. Note that locations may repeat in π_j.

Symbolic Execution: First, we define symbolic execution of paths in the presence of random variables.

Definition 3.1 (Symbolic State)**.** *The symbolic state of a program P is a tuple $s : \langle \ell, R, \psi, T \rangle$ wherein*

1. *ℓ denotes the current location.*
2. *$R = \{r_1, \ldots, r_k\}$ represents a set of random variables. This includes random variables for the initial condition, and the RHS of assignments involving calls to the random value generators (RVG). Fresh variables will be added to the set R, as RVGs are encountered along a path.*
3. *ψ is an assertion involving the variables in R representing the path condition. The syntax of the program ensures that ψ does not involve any program variables and is a conjunction of linear inequalities over R.*
4. *T is an affine transformation that maps each program variable $x_i \in X$ to an affine expression involving variables in R.*

Initial Symbolic State: The initial symbolic state s_0 is given by $\langle \ell_0, R_0, true, T_0 \rangle$ wherein R_0 is an initial set of random variables corresponding to each initialization of a program variable x_j by a call to an inbuilt random value generator. The transformation T_0 associates each program variable x_j with a constant c or a random variable $r_j \in R$ depending on how x_j is initialized.

Symbolic Step: We may now define the single-step execution of the program along a path $\pi : \ell_0 \rightsquigarrow \ell_F$ by means of symbolic states $s_0 \to s_1 \ldots \to s_F$. To do so, we specify the update of a symbolic state across an edge (ℓ, m) in the control-flow graph.

First, we handle any calls to random value generators on the edge (ℓ, m). Each such call results in the addition of a fresh random variable to the set R with appropriate type, set of support, and probability density function. Table 2 lists the formal specification of the random variable generated by various inbuilt random value generators. Next, depending on the type of the edge being traversed, we define the subsequent symbolic state:

1. The edge is labeled by a condition $\gamma[X, R]$, wherein calls to random value generators have been substituted by fresh random variables introduced in R. In this case, we update the path condition ψ to $\psi' : \psi \wedge \gamma[x_1 \mapsto T(x_1), \ldots, x_n \mapsto T(x_n)]$, wherein each occurrence of the program variable x_i in γ is substituted by $T(x_i)$.
2. The edge is labeled by an assignment $x_i := e[X, R]$ involving variables $x_j \in X$ and calls to random value generators replaced by fresh random variables. In this case, we update the transformation T to yield a new transformation T' defined as

$$T'(x_j) = \begin{cases} T(x_j) & x_j \neq x_i \\ e[x_1 \mapsto T(x_1), \ldots, x_n \mapsto T(x_n)] & \text{otherwise} \end{cases}$$

As a result, we have defined the transformation of the symbolic state $s : (\ell, R, \psi, T)$ to a new state $s' : (m, R', \psi', T')$ involving the addition of fresh random variables, updates to the path condition ψ or the transformation T.

Example 3.1. *Fig. 4 shows a simple program P that updates a real variable x and an integer c. Each step updates the value of x and c by drawing a uniform random variable of appropriate type from fixed ranges. Consider the path that iterates through the while loop twice: $\pi : \ell_0 \to \ell_1 \to \ell_0 \to \ell_1 \to \ell_0 \to \ell_F$. The initial state $s_0 : \langle \ell_0, R_0, \psi_0, T_0 \rangle$ is given by*

$$R_0 : \{r_1 : \mathsf{uniformReal}[-1,3]\}, \psi_0 : true,\ T_0 : (x \mapsto r_1, c \mapsto 0)$$

We may verify that the state at the end of the path is given by

$$\left\langle \ell_F, \left\{ \begin{array}{c} r_1, r_2, r_3, \\ z_1, z_2 \end{array} \right\}, \begin{array}{c} r_1 \leq 4 \wedge r_1 + r_2 \leq 4 \\ \wedge\, r_1 + r_2 + r_3 > 4 \end{array}, \left(\begin{array}{c} x : r_1 + r_2 + r_3, \\ c : z_1 + z_2 \end{array} \right) \right\rangle$$

The real-valued variables r_1, r_2, r_3 are distributed as $\mathsf{uniformReal}(-1,3)$, *while z_1, z_2 are distributed as* $\mathsf{uniformInt}(0,2)$.

Let $\pi : \ell_0 \rightsquigarrow \ell_F$ be a finite path from initial to exit node and $s(\pi) : \langle \ell_F, R_\pi, \psi_\pi, T_\pi \rangle$ be the final symbolic state obtained by propagating the initial symbolic state s_0 along the edges in π. A program path π is *feasible* iff the path condition ψ_π is satisfiable.

Lemma 3.1. *For any feasible state $s : (\ell, R, \psi, T)$ encountered during symbolic execution, the following properties hold:*

1. *ψ is a linear arithmetic assertion that can be decomposed into $\psi : \psi_Q \wedge \psi_I$ where ψ_Q involves real-valued random variables from R while ψ_I involves integer-valued random variables.*
2. *$T(x_j)$ is an affine expression for each x_j involving integer random variables/constants if x_j is integer typed and involving real-valued random variables/constants if x_j is real typed.*

3.2 Semantics of Queries

We may use the notion of a symbolic state propagated across program paths to define the semantics of queries. We first start with *unconditional queries* that estimate the probability of an assertion $\varphi[X]$ over the program variables. We then extend our definition to define the semantics of conditional probability queries.

To answer the queries, we start with the set of all feasible program paths $\Pi = \{\pi_j : \ell_0 \rightsquigarrow \ell_F \mid \pi_j \text{ is feasible}\}$, where $|\Pi|$ may be finite or countably infinite.

Probability of an Assertion: Consider a query that seeks the probability of an assertion φ over the program variables. We denote the outcome of the query as $\mathbb{P}(\varphi)$.

$$\mathbb{P}(\varphi) = \sum_{\pi_j \in \Pi} \mathbb{P}_{\pi_j}(\varphi)\,,$$

wherein $\mathbb{P}_{\pi_j}(\varphi)$ denotes the probability of the event that the execution proceeds along the path π_j, reaching the location ℓ_F *and* φ is true in the resulting state at location ℓ_F.

We now formally define $\mathbb{P}_{\pi_j}(\varphi)$ for $\pi_j \in \Pi$. First, let $s_j : \langle \ell_F, R_j, \psi_j, T_j \rangle$ denote the symbolic state obtained by executing along the path π_j. The transformation T_j maps the program variables at ℓ_F to affine expressions over R_j, the random variables encountered along π_j. Let $\varphi' : \varphi[X \mapsto T_j(X)]$ denote the transformation of the program variables in φ using T_j.

We partition R_j into integer valued variables Z_j and real valued variables Q_j. Next, we split ψ_j into an integer part ψ_j^I conjoined with a real part ψ_j^Q. Similarly, we split the condition φ' into φ^I and φ^Q. The value $\mathbb{P}_{\pi_j}(\varphi)$ can be defined in two parts:

1. For the real part, we define the integral over the sets of support for the variables in $Q_j = \{y_1, \ldots, y_k\}$
$$p_r : \int_{\mathcal{Q}} \mathbb{1}(\psi_j^Q \wedge \varphi^Q)\, p_1(y_1)p_2(y_2)\cdots p_k(y_k) dy_1 \cdots dy_k\,,$$
wherein each y_j has a density function $p_i(y_i)$ and $\mathcal{Q}$ is the joint region of support for the variables $y_1, \ldots, y_k$ taken as the Cartesian product of their individual intervals of support. The notation $\mathbb{1}(\varphi)$ for an assertion φ stands for the *indicator function* that maps to 1 wherever φ is satisfied and 0 elsewhere.
2. For the integer part, we define a summation over the sets of support for variables in $Z_j = \{z_1, \ldots, z_l\}$
$$p_z : \sum_{z_1 \in I_1} \cdots \sum_{z_l \in I_l} \mathbb{1}(\psi_j^I \wedge \varphi^I) i_1(z_1) i_2(z_2) \cdots i_l(z_l)\,,$$
wherein I_j is the set of support for z_j and $i_j(z_j)$ is the mass function for the distribution defining z_j.

The overall value $\mathbb{P}_{\pi_j}(\varphi)$ is the product $p_r \times p_z$. Note that the restrictions to the structure of the conditional statements and queries guarantee that ψ_j and φ' are defined by conjunctions of linear inequalities. Therefore, the sets defined by them are a union of polyhedra over the reals and Z-polyhedra over the integers. They are measurable under the Lebesgue and discrete measures, so that integrals and summations over them are well defined.

Example 3.2. *Consider again the symbolic execution from Ex. 3.1. The set of random variables is $R_F : \{r_1, r_2, r_3, z_1, z_2\}$ with the real-valued variables r_1, r_2, r_3 which are all of the type* uniformReal$(-1, 3)$. *The integer variables z_1, z_2 are of the type* uniformInt$(0, 2)$. *We wish to find the probability $\mathbb{P}_\pi(c \le 4)$ corresponding to the query* estimateProbability$(c \le 4)$. *First, we transform $c \le 4$ according to the transformation $c \mapsto z_1 + z_2$ to obtain $z_1 + z_2 \le 4$.*

The overall probability reduces to finding the probability that the following assertion holds

$$\psi^Q : r_1 + r_2 \le 4 \;\wedge\; r_1 + r_2 + r_3 > 4 \;\wedge\; \varphi^I : z_1 + z_2 \le 4$$

given the distributions of r_1, r_2, r_3, z_1, z_2. The constraint $r_1 \le 4$ is seen to be redundant given $r_1 \in [-1, 3]$ and therefore is dropped.

We split the computation into an integral over the real part

$$\int_{[-1,3]^3} \mathbb{1}(r_1 + r_2 \le 4 \wedge r_1 + r_2 + r_3 > 4) \left(\frac{1}{4}\right)^3 dr_1 dr_2 dr_3\,.$$

and an integer part $\sum_{z_1=0}^{2} \sum_{z_2=0}^{2} \mathbb{1}(z_1 + z_2 \le 4) \left(\frac{1}{3}\right)^2$.

The second half of this paper describes how to restrict the probability computation to a suitably chosen finite set of paths, while bounding the influence of the unexplored paths and for each path, place tight bounds on the summation and the integral above. We also use Monte-Carlo simulations to estimate the actual values to some degree of confidence.

Conditional Assertions: We now consider queries of the form estimateProbability$(\varphi_1$ given $\varphi_2)$ that seeks to estimate the probability of assertion φ_1 under the condition that the state at the exit satisfies φ_2. However, unlike the previous case, it is possible that the conditional probability may be ill defined. This is especially the case when no execution reaches the end of the program or all executions reaching the exit do not satisfy φ_2.

It is well-known that

$$\mathbb{P}(\varphi_1 | \varphi_2) \;=\; \frac{\mathbb{P}(\varphi_1 \wedge \varphi_2)}{\mathbb{P}\varphi_2} \;=\; \frac{\sum_{\pi \in \Pi} \mathbb{P}_\pi(\varphi_1 \wedge \varphi_2)}{\sum_{\pi \in \Pi} \mathbb{P}_\pi(\varphi_2)}\,.$$

As a result, we can use the definitions established for computing the probabilities of assertions unconditionally, and derive conditional probabilities. In general, however, it is incorrect to split the conditional probabilities as a whole as a summation of the conditional probabilities measured along paths.

4. Computing Probabilities

In this section, we present techniques to estimate the probabilities of assertions by sampling program paths. The probability of the assertion (overall probability) is simply the summation of the probabilities computed over individual paths. We consider three types of estimation: (a) computing guaranteed lower and upper bounds on the individual path probabilities and the overall probability of the assertion and (b) approximation (Monte-Carlo simulation) of path probabilities.

The overall path probability estimation proceeds in two steps: (a) choose a finite set of paths in the program using a heuristic strategy, so that with high confidence, the sum of the path probabilities of the chosen paths exceeds a *coverage bound c*; and (b) for each path chosen, estimate probability efficiently from the path condition. Finally, we show how both parts can be tied together to yield guaranteed whole program bounds and approximations.

Let us fix a program P and let φ be an assertion whose probability we wish to estimate. We assume a path coverage target

```
Data: Program $P$, assertion $\varphi$ and coverage $c \in (0,1)$.
Result: Probability estimate $\hat{p}$ and bounds $[\underline{p}, \overline{p}]$.
/* 1. Heuristically elect a set of paths.          */
$C := \mathbf{PathSelect}(P, c)$ ;
$\underline{q} := 0$ ;          /* lower bound on path probability */
$(\underline{p}, \overline{p}) := (0, 0)$ ;          /* path + assertion bounds */
$\hat{p} := 0$ ;                       /* Monte-Carlo Estimate */
/* 2. Iterate through paths in $C$                  */
for ($\pi \in C$) do
    /* 2.1 Compute path condition and transformation
       */
    $(R, \psi, T) := \mathbf{SymbolicExecute}(P, \pi)$;
    /* 2.2 Transform condition $\varphi$              */
    $\varphi' := \varphi[X \mapsto T(X)]$;
    /* 2.3 Lower bound on path probability          */
    $\underline{q} := \underline{q} + \mathbf{PolyhedronLowerBound}(R, \psi)$ ;
    /* 2.4 Bounds on path and assertion probability
       */
    $\underline{p} := \underline{p} + \mathbf{PolyhedronLowerBound}(R, \psi \wedge \varphi')$ ;
    $\overline{p} := \overline{p} + \mathbf{PolyhedronUpperBound}(R, \psi \wedge \varphi')$ ;
    /* 2.5 Monte-Carlo estimation                  */
    $\hat{p} := \hat{p} + \mathbf{MonteCarloSample}(R, \psi \wedge \varphi')$ ;
end
/* 3. Adjust upper bound estimate to account for
   uncovered paths.                                */
$\overline{p} := \overline{p} + (1 - \underline{q})$ ;
```
Algorithm 1: Algorithm for estimating the probability of an assertion.

$0 < c < 1$ wherein c is assumed to be very close to 1 (say $c = 0.99$). Let Π denote the set of all terminating, feasible program paths that lead from the initial location ℓ_0 to the final location ℓ_F. Algorithm 1 shows the overall algorithm for providing bounds and estimates of probabilities for an assertion. The idea is to compute a finite set of paths C according to the coverage criterion and compute the probabilities for the assertion φ to hold at the end of each path. Next, an actual lower bound on the total probability of all paths in C is also estimated and used to yield an upper bound on the overall assertion probability.

4.1 Heuristic Path Selection Strategy

The first step is to select finitely many feasible, terminating program paths $C = \{\pi_1, \ldots, \pi_k\}$. The path probability of a path π_j is defined as $\mathbb{P}_{\pi_j}(true)$. This denotes the probability that a randomly chosen initial condition results in taking the path π_j in the first place. We write $\mathbb{P}(\pi_j)$ to denote the path probability.

A subset $C \subseteq \Pi$ satisfies a coverage goal c iff $\sum_{\pi \in C} \mathbb{P}(\pi) \geq c$.

Satisfying any given coverage goal c is not guaranteed since it is not known a priori if a finite subset C satisfies the coverage goal. For instance, the program may fail to terminate with probability $p_{\text{non-terminating}} \geq 1 - c$. However, most programs of interest do terminate *almost surely* with probability 1.

Lemma 4.1. *Let P be an almost surely terminating program. For any coverage goal $c \in (0,1)$, there is a finite set of paths C such that $\sum_{\pi \in C} \mathbb{P}(\pi) \geq c$.*

Therefore, assuming that we are able to find enough paths C whose cumulative path probability is known to be at least c, we may proceed to estimate the probability of φ along paths $\pi \in C$. We use the notation $\mathbb{P}_C(\varphi)$ to denote the sum of path probabilities for φ along all paths belonging to the set C: $\mathbb{P}_C(\varphi) = \sum_{\pi \in C} \mathbb{P}_\pi(\varphi)$.

Theorem 4.1. *For any assertion φ and set of paths C that satisfy the coverage criterion c, $\mathbb{P}_C(\varphi) \leq \mathbb{P}(\varphi) \leq \mathbb{P}_C(\varphi) + (1 - c)$.*

```
Data: Program $P$, assertion $\varphi$, run length $K > 0$.
Result: Set of paths $C$ such that a run of $K$ continuously drawn
        program paths belong to $C$ .
count := 0 ;
$C = \{\}$ ;
while count $< K$ do
    $\pi$ := simulatePath(P) ;
    if $\pi \notin C$ then
        count := 0 ;
        $C := C \cup \{\pi\}$;
    else
        count := count + 1 ;
    end
end
```
Algorithm 2: Algorithm for collecting a set of paths C through random simulation until a run of $K > 0$ continuous paths already belonging to C is obtained.

Proof. Since we have $C \subseteq \Pi$, the lower bound is immediate. For the upper bound, we note that

$$\begin{array}{rcl} \mathbb{P}(\varphi) & = & \sum_{\pi \in \Pi} \mathbb{P}_\pi(\varphi) = \sum_{\pi \in C} \mathbb{P}_\pi(\varphi) + \sum_{\pi \notin C} \mathbb{P}_\pi(\varphi) \\ & \leq & \sum_{\pi \in C} \mathbb{P}_\pi(\varphi) + \sum_{\pi \notin C} \mathbb{P}_\pi(true) \\ & \leq & \sum_{\pi \in C} \mathbb{P}_\pi(\varphi) + (1 - c) \end{array}$$

□

Our approach uses a statistical approach based on simulation to select an initial set C of paths such that the sum of path probabilities of individual paths in C exceeds the coverage bound c with high confidence. Our overall strategy proceeds in two steps:

1. Fix integer parameter $K > 0$. The criterion for choosing K is described below.
2. The simple procedure shown in Algorithm 2 repeatedly runs the program and records the syntactic path taken by each run. If the current path was previously unseen, then it is added to the set C. If K consecutive paths belong to C, the algorithm terminates.

The strategy outlined above obtains a set of program paths C that is *likely*, but not guaranteed to satisfy the coverage criterion c desired by the user. As such, this strategy is unsuitable for static analysis. We employ it simply as a convenient heuristic to choose an initial set of paths. Our algorithm then proceeds to use probability bounding to find guaranteed bounds on the actual coverage. These bounds are then used to actually estimate coverage.

Selecting the Run Length: We now outline our heuristic strategy for selecting a suitable run length K. Suppose C be a current set of paths that is claimed to satisfy a coverage bound c. We wish to test this claim quickly. This can be viewed as a statistical hypothesis testing problem of choosing between two hypotheses:

$$\mathcal{H}_0 : \mathbb{P}(C) \geq c \text{ vs. } \mathcal{H}_1 : \mathbb{P}(C) < c .$$

In statistical testing terms, $\mathcal{H}_0$ is called the null hypothesis and $\mathcal{H}_1$, the alternative. Our goal is to witness $K > 0$ successive samples that belong to C, where K is set sufficiently large to convince us of $\mathcal{H}_0$ as opposed to $\mathcal{H}_1$. This can be attempted using the sequential probability ratio test (with a slight modification of $\mathcal{H}_0$ and $\mathcal{H}_1$ to introduce an indifference region, see Younes and Simmons [37] and references therein), or using a Bayesian factor test following Jeffreys (see Jha et al. [17], and references therein). Either approach gives rise to a formula that fixes K as a function of c. For illustration, Jeffreys test with uniform prior yields a simple formula

$$K \geq \left\lceil \frac{-\log B}{\log c} \right\rceil .$$

The factor B in the numerator is called the Bayes factor and can be set to 100 to yield a answer with high confidence. For instance, setting $c = 0.95$, we require $K \geq 90$ to satisfy the Bayes factor test at a confidence level given by $B = 100$.

However, this does not imply that we can be *absolutely certain* (rather than just *highly confident*!) that the set of paths satisfy the coverage guarantee. Therefore, we derive guaranteed lower bounds on the coverage of the set of paths C collected. This lower bound forms the basis of our bound estimations.

In our approach, the purpose of hypothesis testing is two-fold: (a) It guarantees that all chosen paths in the set C are feasible. As a result, our approach does not waste time exploring infeasible paths. (b) Sampling the program guarantees that paths with high probability are explored first before lower probability (tail) paths. This can potentially result in fewer paths explored by our analysis.

Path Slicing: The use of dynamic path slicing techniques can further enhance the efficiency of the procedure in Algorithm 2. The paths in the set C are sliced with respect to the variables involved in the query assertion φ being evaluated. The use of path slicing ensures that fewer path fragments in C are generated and each path fragment yields a simpler system of constraints over a smaller set of random variables. Our approach uses the path slicing approach originally proposed by Jhala et al. for predicate abstraction [18].

4.2 Computing Path Probabilities

We now describe techniques for computing the probability of an assertion φ at the exit of a given path π. As described previously, we perform a symbolic execution along π to obtain a symbolic state at the and of the program $\langle \ell_F, R, \psi, T \rangle$. We then consider the assertion $\psi \wedge \varphi[X \mapsto T(X)]$, involving random variables in R.

As a result, the computation of path probabilities reduces to the problem of estimating the probability that a sample point from R drawn according to the distribution of the individual variables also satisfies φ. We present algorithms for (a) computing lower and upper bounds; and (b) estimating the probability.

First, we observe that given the structure of the program, the path condition is a conjunction of linear inequalities and is therefore a convex polyhedron over R. How do we then compute the probability of drawing a sample point inside a given polyhedron? First, we note that computing the exact probability is closely related to the problem of computing the volume of a n-dimensional convex polyhedron. This is known to be $\sharp P$-hard and therefore computationally hard to solve precisely once the number of dimensions is large [2]. We proceed by using bounding boxes that bounds the region of interest from inside as well as outside.

Let $R = Q \cup Z$, where $Q = \{r_1, \ldots, r_k\}$ are the real-valued random variables and $Z = \{z_1, \ldots, z_l\}$ are the integer valued random variables. Let $\psi : \psi^Q \wedge \psi^Z$ be a conjunction of linear inequalities wherein ψ^Q involves the real-valued variables and ψ^Z involves the integer valued variables. Each random variable r_j (or z_k) is distributed according to a known probability distribution function $p_j(r_j)$ (or $p_k(z_k)$) over some set of support.

Our goal is to compute the probability that a randomly drawn sample $\hat{r} : (\hat{r_1}, \ldots, \hat{r_k}, \hat{z_1}, \ldots, \hat{z_l})$ satisfies the assertion ψ. We split this problem into two parts one for the integer part and one for the real part.

Random Variable Primitives: We ignore the details of the distributions, assuming instead that some primitive functions can be computed: (a) given a range $[a, b]$, we can compute its probability $\mathbb{P}_{r_i}([a, b]) = \int_a^b p_i(r_i) dr_i$; and (b) we can obtain samples $\widehat{r_{i,j}}$ for $j = 1, \ldots, N$ that are distributed according to p_i. The same assumptions are held for the integer-valued random variables as well. Table 3 summarizes the primitives.

sample ()	generate a (pseudo) random sample standard Monte-Carlo methods [34].
probability (a,b)	estimate probability mass of interval $[a, b]$ using cumulative distribution function (CDF).

Table 3. Primitives needed for each distribution type to support probability and expectation estimation.

```
Input: Polyhedron $\varphi[r_1, \ldots, r_n]$, each $r_i \sim \mathcal{D}_i$.
Output: Interval bounds $[\underline{p}, \overline{p}]$ and stratified MC sample
        estimate $\hat{p}$
$queue := \{\varphi\}$ ;
$(\underline{p}, \overline{p}, \hat{p}) := (0, 0, 0)$;
while $|\mathbf{queue}| > 0$ do
    $\xi := \mathbf{deque}(queue)$ ;
    if stopping criterion then
        $\mathcal{H} := \mathbf{boxOverApproximation}(\xi)$;
        $\mathcal{B} := \mathbf{boxUnderApproximation}(\xi)$;
        $\begin{pmatrix} \overline{p} \\ \underline{p} \end{pmatrix} := \begin{pmatrix} \overline{p} + \mathbf{boxProbability}(\mathcal{H}) \\ \underline{p} + \mathbf{boxProbability}(\mathcal{B}) \end{pmatrix}$;
        $\hat{p} := \hat{p} + \overline{p} * \mathbf{stratifiedSample}(\xi, \mathcal{H})$;
    else
        $(d, l) = \mathbf{selectDimension}(R)$ ;
        /* Perform Branch and Bound                */
        $(\xi_1, \xi_2) := (\xi \wedge r_d \geq l), (\xi \wedge r_d < l)$;
        $queue := \mathbf{enqueue}(queue, \{\xi_1, \xi_2\})$ ;
    end
end
```

Algorithm 3: Basic algorithm to compute bounds on the probability of a polyhedron represented by a linear assertion φ.

Computing Probability of Hypercubes: A hypercube over $\mathbb{R}^k$ can be written as the Cartesian product of intervals:

$$\mathcal{H} : [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_k, b_k] .$$

To compute the probability of $\mathcal{H}$, we use the available primitives directly. $\mathbb{P}_Q(\mathcal{H}) = \mathbb{P}_{r_1}([a_1, b_1])\mathbb{P}_{r_2}([a_2, b_2]) \cdots \mathbb{P}_{r_k}([a_k, b_k])$. The same calculation holds over the integers as well with minor modifications to ensure that a_j, b_j values are rounded to the nearest integer above/below, respectively. For a disjoint union of $N \geq 0$ hypercubes, we have $\mathbb{P}_Q(\mathcal{H}_1 \uplus \cdots \uplus \mathcal{H}_N) = \sum_{k=1}^{N} \mathbb{P}_Q(\mathcal{H}_k)$.

4.3 Computing Bounds on Polyhedral Probabilities

We now address the issue of computing bounds on the probability of a given polyhedral set specified by assertion φ over a set of real-valued variables $Q = \{r_1, \ldots, r_k\}$ (or integer valued variables $Z = \{z_1, \ldots, z_k\}$), where each r_i is drawn from a known distribution.

The basis of our approach is to over approximate $[\![\varphi]\!]$ by a union of hypercubes $\overline{H_1} \cup \overline{H_2} \cup \cdots \cup \overline{H_p}$ and under approximate by a union $\underline{H_1} \cup \cdots \cup \underline{H_s}$. The number of hypercubes p, s can be varied to control the quality of the approximation. The overall idea is to bound the probability of φ by computing the probability of the over approximation and the under approximation.

Lemma 4.2. *Let $\underline{H} \subseteq [\![\varphi]\!] \subseteq \overline{H}$. It follows that $\mathbb{P}_Q(\underline{H}) \leq \mathbb{P}_Q(\varphi) \leq \mathbb{P}_Q(\overline{H})$.*

We now consider how to obtain over approximations and under approximations of polyhedral sets.

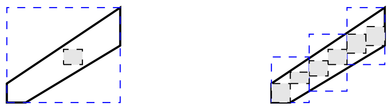

Figure 5. A worst-case scenario for over- and under approximation by a single hypercube (left) and a less pessimistic over- and under approximation using many hypercubes.

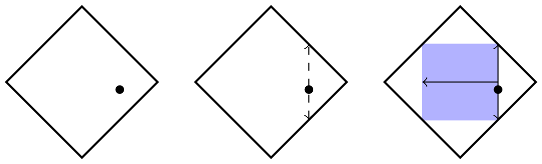

Figure 6. Underapproximation by interior ray shooting.

Over approximation: Linear programming (LP) solvers can be used to directly over approximate a given polyhedron by a single hypercube. To obtain upper (lower) bounds for r_j we solve the LP: $\max(\min)\ r_j$ s.t. φ. The resulting hypercube given by the product of intervals obtained for each r_j over approximates the region defined by φ. However, the hypercube may be a gross over approximation of the original set, yielding quite pessimistic upper bounds. Fig. 5 shows a polyhedron whose over approximation by a single hypercube is quite pessimistic. However, if multiple hypercubes were used, then the accuracy improves. As explained in Algorithm 3, this is achieved by repeated splitting along a chosen dimension and finding bounding boxes using LP solvers.

Under approximation: The goal is to find an under approximation of a polyhedron φ by a hypercube. As depicted in Fig. 5, there is no best under approximation for a given polyhedron. This is unlike the case for an over approximation, where LP solvers can yield a single best bounding box.

Our approach is to find a sample point $\widehat{r}$ in the interior of the polyhedron, treating it as a hypercube of volume zero where the upper and lower bounds along each dimension are specified by $[\hat{r_j}, \hat{r_j}]$. Next, we use a simple "*ray-shooting*" approach (Cf. Fig. 6) to expand the current box along each dimension while satisfying the constraints. This is carried out by iterating over the dimensions in a fixed order. Upon termination, this process is guaranteed to yield an under approximation. The size of the hypercube depends on the sample point chosen. In practice, we choose multiple sample points and choose the hypercube with the largest probability mass.

Z-polyhedra: As such, the approach presented so far extends to Z-polyhedra with minor modifications. One modification is to ensure that the bounds of each interval are rounded up/down to account for the integer variables. Additionally, the use of an ILP solver to derive the over approximation can yield tighter bounds. On the other hand, since ILP is a harder problem than LP, using LP solvers to over approximate the polyhedron yields safe upper bounds. The computation of an inner approximation can also be performed using ray shooting.

Monte-Carlo Estimation We have discussed techniques for bounding the probability using boxes. We can extend this to find unbiased estimators of the probability by sampling. This is performed by drawing numerous samples $\widehat{s_1}, \ldots, \widehat{s_N}$ according to the underlying distributions for each variable in R for a large number N and count the number N_φ that satisfies φ. We simply compute the ratio $\hat{p} = \frac{N_\varphi}{N}$.

In some cases, it may take a prohibitively large number N of samples to approximate $\hat{p}$ to some given confidence. To circumvent this, it is possible to fold the computation of $\hat{p}$ with the branch-and-bound decomposition in Algorithm 3. Here, we over approximate the polyhedron φ as the union of K disjoint hypercubes $[\![\varphi]\!] \subseteq \mathcal{H}_1 \uplus \cdots \uplus \mathcal{H}_K$. Next, we draw N_j samples from inside each $\mathcal{H}_j$ and estimate the number of samples $N_{\varphi,j}$ inside $\mathcal{H}_j$ that satisfy φ. This yields an estimate $\widehat{p_j}$ for $\mathcal{H}_j$. The overall probability is given by $\widehat{p} = \sum_{j=1}^{K} \widehat{p_j}\mathbb{P}(\mathcal{H}_j)$. Note that $\mathbb{P}(\mathcal{H}_j)$ is computed exactly and efficiently. This scheme integrates the ideas of *stratified sampling* and *importance sampling*, both of which are well-known variance reduction techniques for Monte-Carlo sampling [34].

ID	#var	Description
EGFR-EPI	11	log egfr calc. using the ckd-epi formula.
ARTRIAL	15	Framingham artrial fibrillation risk calculation.
CORONARY	16	Framingham coronary risk calculation.
INVPEND	7+	Inverted pendulum controller with noise.
PACK	10+	Packing objects with varying weights.
VOL	8+	Filling a tank with fluid at uncertain rates.
ALIGN-1,2	6+	Pointing to a target with error feedback.
CART	7+	Steering a cart against disturbances.

Table 4. Benchmarks used in our evaluation with descriptions. The benchmarks are available on-line at our project page, or upon request. #var: number of input variables to the program. + indicates random value generation inside loops.

5. Implementation and Experiments

We now describe the implementation and evaluation of the ideas presented thus far.

Implementation: Our prototype implementation accepts programs whose syntax is along the lines of Figure 3 in Section 3. Our front-end simulates the program and collects the unique paths observed until no new paths are seen for K consecutive iterations (Algorithm 2). The value of K was chosen for $c = 0.95$ using a Bayes factor test with $B = 100$. This yields $K = 90$ for achieving a coverage of at least 0.95 with a 99% confidence (under a uniform prior). Symbolic execution is performed along the collected paths, yielding the path conditions and transforming the probabilistic queries into assertions involving the random variables produced along the path. Finally, we implement the technique for bounds estimation described in Algorithms 1 and 3. Rather than use an expensive linear programming instance at each step of our branch and bound algorithm, we use a cheaper interval constraint propagation (ICP) to find bounding boxes efficiently [15]. Other optimizations used in our implementation include the use of on-the-fly Cartesian product decomposition of polyhedra and repeated simplification with redundancy elimination.

Benchmark Programs: We evaluate our program over a set of benchmarks described in Table 4. A detailed description of the benchmarks used along with our latest implementation are available on-line from our project page [4].

The first class of benchmarks consists of medical decision making formulae including the eGFR calculation described in Section 2 (EGFR-EPI). We also include a common heart risk calculator and a hypertension risk calculator that uses the results of the Framingham heart study (ARTRIAL, CORONARY) [5]. The results of this calculation are often taken into account if a patient is a candidate for drugs such as Statins to prevent long term heart attack risk.

[4] Cf. `http://systems.cs.colorado.edu/research/cyberphysical/probabilistic-program-analysis/`

[5] Cf. `http://www.framinghamheartstudy.org/`

ID	N_s	N_u	T_s, T_{vc}	p_{mc}	$[lb, ub]$
EGFR EPI	563	45	0.1, 1.1	0.97803	[0.97803,0.97803]
ARTIRAL	19547	2520	15.6, 1095	0.94094	[0.82809,1.0]
CORONARY	7181	1239	3.9, 998.5	0.91992	[0.87239,1.0]
INVPEND	90	1	18.5, 0.1	1	[1,1]
PACK	8043	1010	4.6, 24.1	0.95052	[0.95051,0.95052]
ALIGN-1	4464	529	3.9, 25.7	0.90834	[0.90769,0.90846]
ALIGN-2	17892	1606	8.9 , 75.1	0.93110	[0.9304,0.9384]
VOL	121	9	3.6 , 1766	0.835	**[0,1]**
CART	5799	774	4.2 , 1612	0.94898	**[0.0176,1]**

Table 5. Experimental evaluation of initial adequate path exploration strategy. Our coverage goal was $c = 0.95$ at a 95% confidence level. N_s, N_u: number of simulated paths and unique paths, respectively, T_s: simulation time in seconds, T_{vc}: total volume computation time (branch-and-bound depth is set to 15), p_{mc}: monte-carlo probability estimate, $[lb, ub]$: coverage bounds.

The second class of benchmarks consist of programs that carry out control tasks over physical artifacts. This includes models of linear feedback control systems with noise such as the inverted pendulum controller (INVPEND), a model of a robot that packs objects of uncertain weights in a carton with the goal that the overall weight of carton has to be within some tolerance (PACK), a model of an actuator that fills a tank with fluid where the rate at which the fluid flows can vary with the goal that the fluid waste has to be minimized (VOL) and models of a manipulator that attempts to point a laser at a desired stationary target, wherein alignments can fail due to actuator uncertainties. Each failed attempt results in a feedback that reveals the amount and direction of error that can be used to design the displacement for the next attempt. We propose two versions that differ in terms of the error distributions and the feedback correction mechanisms (ALIGN-1, ALIGN-2).

Experimental Evaluation: Table 5 shows the performance of the path selection and coverage estimation strategies. We note that for most of our benchmarks, the selection of adequate paths required less than 10^4 simulations and resulted in a smaller set of unique paths that are potentially adequate. The time for symbolic execution is a small fraction of the total time taken (T_s). The probability bounds computation (T_{vc}) time dominates the overall analysis time.

The probability bounds computation performs well on many of the examples, solving large problems with as many as 50 integer and real variables. In many cases, the probability bounds obtained are quite tight, indicating that the volume computation converged to fairly precise bounds rapidly. There are two notable exceptions that include the VOL and CART examples. The path conditions in these examples produce highly skewed polyhedra that are the worst-cases for a branch and bound approach. As a result, the bounds produced are quite pessimistic. We are investigating the use of zonotope and ellipsoidal based approximations that can handle such instances effectively. Next, we observe that while 0.95 is the target coverage, the coverage goal is not confirmed by the probability bounds in two instances (ALIGN-1,2). However, the actual reported coverages in these examples are fairly close to the target. Note that a failure to achieve coverage goal is not a failure of the technique. Our ability to deduce rigorous bounds on the path probability helps ensure that the overall result can be made sound regardless of what coverage we aim for. Furthermore, we may remedy the situation by iteratively searching for more paths until the goals are satisfied.

Table 6 shows the results on many probabilistic queries over the benchmarks. Once again, we report on the time taken for the volume computation and the bounds on the queries. We note that whenever path coverage returns a tight bound on the path probabilities, the estimation of assertion probabilities also yields similar tight bounds. The bounds provide non-trivial information about the

Query	T_{vc}	p_{mc}	$[lb, ub]$
ARTRIAL (2520 paths, $c \geq 0.82809$)			
score $>= 10$	252	0.13455	[0.12379,0.31685]
err $>= 5$	48	0.0003637	[0.000257,0.17754]
err $<= 5$	1076	0.94057	[0.82783,1]
CORONARY (1239 paths, $c \geq 0.87239$)			
err $>= 5.0$	22	0.000154	[0,0.12778]
err $>= 7.0$	17	0	[0,0.12778]
err $>= 10.0$	17	0	[0,0.12778]
err $<= -5.0$	16	0.0001	[0,0.12778]
err $<= -7.0$	17	0	[0,0.12778]
err $<= -10.0$	17	0	[0,0.12778]
ALIGN-1 (529 paths, $c \geq .90834$)			
attempts $>= 10$	17.0	0.00726	[.00722,0.09889]
attempts $>= 6$	24.7	0.11796	[0.11773,0.20997]
attempts $>= 2$	25.6	0.630489	[0.62991,0.72234]
err $>= 12$	11.1	0.001580	[0.001570,.093245]
err $<= -12$	10.5	0.001841	[0.001834,0.0935130]
err $>= 5$	21.7	0.084092	[0.084035,.175794]
err $<= -5$	21.8	0.084455	[0.084396,.176163]
PACK (1010 paths, $c \geq 0.95051$)			
count ≥ 5	24.3	0.95051	[0.95051 , 1.0]
count ≥ 6	24.3	0.40364	[0.40364 , 0.45312]
count ≥ 7	22.7	0.14192	[0.14192 , 0.19140]
count ≥ 10	21.0	0.000467	[0.000467 , 0.04995]
totalWeight ≥ 6	43.7	0.27269	[0.25223 , 0.3420]
totalWeight ≥ 5	34.9	0.67754	[0.61734 , 0.78481]
totalWeight ≥ 4	24.0	0.95051	[0.95051 , 1.0]
INVPEND (1 path, $c = 1$)			
pAng $<= 1$	4.1	0.05138	[0,0.13653]
pAng $>= -1$	0.1	1	[1,1]
pAng $<= 0.1$	0.1	0	[0,0]
pAng $>= -0.1$	0.1	1	[1,1]

Table 6. Query processing results on the benchmark examples. Table 5 explains the abbreviations used. Bounds in the table include the over-estimation from uncovered paths inferred from the lower bound on c reported in Table 5.

behavior of these programs and are quite hard to estimate by hand even if the programs in question are small.

6. Discussion and Related Work

Reasoning about infinite-state probabilistic programs is considered to be a hard problem. The execution of small and seemingly simple programs that use standard uniform random number generators can yield complex probability distributions over the program variables. Arithmetic operations often give rise to highly correlated program variables whose joint distributions cannot be factored as a product of marginals. Conditional branches and loops can lead to discontinuous distributions that cannot be described by simple probability density functions, in general.

Therefore, analysis algorithms for probabilistic programs must provide solutions to three basic questions: (a) *Representing* the possible intermediate distributions of variables, (b) *Propagating* the chosen representation according to the program semantics, and (c) *Reasoning* about the probabilities of assertions using the given intermediate distribution representation. Any viable solution for infinite state probabilistic programs must restrict the set of programs analyzed through some syntactic or semantic criteria, or deal with *information loss* due to the abstraction of intermediate probability distributions. We now compare our work against other approaches for probabilistic program analysis using these criteria.

Probabilistic Abstract Interpretation Monniaux's work constructs probabilistic analyses by annotating abstract domains such as intervals, octagons and polyhedra with *upper bounds* on the probability measure associated with the abstract objects [28]. However, the abstraction used by Monniaux associates a measure bound for the entire object, *without tracking how the overall measure is distributed amongst the individual states present in the concretization*. This restriction makes the resulting analysis quite conservative. The bounds associated with abstract objects become coarser during the course of analysis due to repeated meet and join operations. The domain of Mardziel et al [25] addresses this limitation for the case of integer-valued program variables. Their approach tracks upper and lower bounds for the object as a whole, as well as bounds on the measure associated with each lattice point. This allows the measure associated with an abstract object to be updated due to meet, join and projection operations.

Our "*domain*" can be seen as a disjunction of symbolic states (see Def. 3.1), and furthermore our analysis does not require meet or join operations over distributions. Similarly, no *upfront* explicit probabilistic bounds are attached to symbolic states during our analysis, unlike the work of Monniaux or Mardziel et al [25, 28]. On the other hand, the probabilistic bounds on symbolic states are computed "after the fact" (using probability bounds computation) when the entire path has been analyzed. The sources of loss in our approach include the probability bounds computation and the unexplored program paths.

The approaches mentioned thus far, including ours, do not handle non-linear operations. Furthermore, the high complexity of domain operations on convex polyhedra and approximate volume bound computations can limit the size of programs that can be analyzed. The work of Bouissou et al tackles these problems through the domain of *probabilistic affine forms* [3] by combining techniques from the AI community such as p-Boxes and Dempster-Shafer structures [9, 35] with affine forms [7]. Their approach represents variables as affine expressions involving two types of random variables (noise symbols): noise symbols that are correlated with the other symbols in an arbitrary, unspecified manner, and independent noise symbols. The concept of arbitrarily correlated noise symbols is a key contribution that supports transfer functions for nonlinear operations. However, at the time of writing, Bouissou et al do not provide meet, join and widening operations. Therefore, their technique is currently restricted to straight line probabilistic programs without conditional tests. Our approach uses symbolic states (see Def. 3.1) that are a special case of "constrained" probabilistic affine forms with linear inequality constraints over the random variables. However, our approach does not currently handle random variables with arbitrary, unspecified correlations. Extending our approach to treat correlated noise symbols will form an important part of our future work in this space.

Whereas the techniques described so far perform a forward propagation of the distribution information, it is possible to use backward abstract interpretation starting from an assertion whose probability is to be established, and exploring its preconditions backwards along the program paths. McIver and Morgan proposed a backward abstract interpretation for probabilistic programs with discrete Bernoulli random variables and demonic non-determinism. Their approach uses "expectations" that are real-valued functions of the program state [26] rather than assertions over the program variable. Expectations can be seen as abstractions of distributions. A notable aspect of their work lies in the use of *quantitative loop invariants* that are invariant expectations of loops in the program. The automatic inference of such invariants was addressed by the recent work of Katoen et al [20]. It is notable that very few approaches, including ours, have sought to provide a complete treatment of loops. It is common to assume that the loops terminate in all cases after a fixed number of iterations. The combination of quantitative loop invariants and the idea of using pre-/post-condition annotations are notable in the work of McIver and Morgan (ibid). A combined approach will form an interesting topic for future work.

DiPierro et al proposed an entirely different approach to probabilistic abstract interpretation that views concrete and abstract domains as Hilbert spaces, with the abstraction operator as a non-invertible linear operator [32]. Rather than defining the inverse in terms of a set of concrete states, as is traditionally done in abstract interpretation, the authors propose to use a Moore-Penrose pseudo-inverse, which corresponds to a least squares approximation in finite dimensional vector spaces. This approach exposes an interesting alternative to the traditional abstract interpretation approach to program analysis. However, a key problem lies in relating the results of the analysis proposed by DiPierro et al that provide a "close approximation" to the probability of a query assertion with the "classical" approach that provides a guaranteed interval bound enclosing the actual probability, such as the approach presented in this paper.

The recent work of Cousot and Monerau provides a general framework that encompasses a variety of probabilistic abstract interpretation schemes [6]. Various techniques mentioned thus far, including ours, can be seen as instances of this general framework [3, 6, 25, 28].

The program analysis techniques mentioned thus far focus on providing bounds on the probabilities of assertions for the program as a whole. In many cases, the same questions may be asked of a single path or a finite set of paths in the program. Recently, Geldenhuys et al proposed the integration of symbolic execution with exact model counting techniques to estimate the probabilities of violating assertions along individual paths of the program [13]. The tool LattE was used to count lattice points, and thus estimate polyhedral volumes for the constraints obtained along each path [8]. Like Geldenhuys et al, the approach here also performs a symbolic execution along a chosen set of program. However, our work goes further to infer *whole program bounds*. Our work computes probability bounds using a branch-and-bound technique, whereas Geldenhuys et al use the solver LattE off the shelf for computing precise probabilities. Finally, the approach of Geldenhuys et al is restricted to discrete, uniform random variables that take on a finite set of values.

An extension by Filieri et al., that is contemporaneous to our work, performs probabilistic symbolic execution with a user-specified cutoff on the path length [11]. Their approach uses the concept of "confidence" to estimate the probability of the paths that are cutoff prematurely. This is remarkably similar to the concept "coverage bounds" used in our work. On one hand, the work of Filieri et al. goes beyond ours to consider the effect of non-deterministic schedulers on multi-threaded programs. On the other, their work continues to use LattE as a lattice counting tool to estimate probabilities. Therefore, it is restricted to uniform distributions or discretized distributions over finite data domains. Integrating the contributions of Filieri et al. with our volume bounding approach can help us move towards a technique that can support integers along with reals and a wide variety of distributions without requiring discretization.

Visser et al presented a set of optimizations that can be used to reduce and simplify the constraints occurring in a program in a canonical form, so that the answer from one volume computation can be cached and reused [36]. Currently, we observe that different program paths explored in our framework necessarily yield different constraints. However, the idea of caching and reusing can be useful for intermediate polyhedra that are obtained during the branch and bound volume computation process. We plan to explore this idea as part of our future work.

A large volume of work has focused purely on verifying properties of finite state discrete and continuous time Markov chains and Markov decision processes [23]. The tool PRISM integrates many of these techniques, and has been used successfully in many scientific and engineering applications as showcased on-line [24]. A key aspect of PRISM is the integration of symbolic techniques for representing discrete distributions over a large but finite set of states using extensions to decision diagrams called MTBDDs [25]. Recently, infinite state systems such as probabilistic timed and hybrid automata have been considered (Cf. [16, 22], for instance) in the context of the model-checking approach embodied by PRISM. However, these extensions remain inside the decidable sub-classes. Our approach, on the other hand, considers Turing complete, infinite state programs, incorporating a rich set of probability distributions. Therefore, we do not compute probabilities precisely, settling for intervals that can be useful in many cases. Whereas work on PRISM uses a very rich temporal specification language, our work is currently restricted to (conditional) probabilities of assertions.

Sources of Probabilistic Programs: Probabilistic programs arise in many forms of computation involving erroneous or noisy inputs. These include risk analysis, medical decision making, data analysis [1], randomized algorithms [30], differential privacy mechanisms [10], simulations of fundamental physical processes (Monte-Carlo simulations) [12]. Certain loop optimizations such as *loop perforation* depend on the use of randomization to trade off performance against the precision of the computed results [27]. Program smoothing is an approach to program synthesis proposed by Chaudhuri et al [4]. It creates a probabilistic program that is a "smoothed version" of the original program obtained by adding noise to the inputs of the program.

Many programming language formalisms have been considered for expressing probabilistic programs including IBAL, a declarative language for specifying probabilistic models [31] probabilistic scheme [33] and Church, an extension of probabilistic scheme [14]. These languages support a variety of functions such as simulations, computing expectations, parameter estimations and computing marginal distributions based on Monte-Carlo simulations.

Probabilistic Program Semantics: The semantics of imperative probabilistic programs was first studied in detail by Kozen. [21]. Significantly, Kozen provides two types of semantics that are mathematically equivalent but represent different views of the probabilistic program. The first semantics is operational, wherein the program's execution is seen as essentially deterministic but governed by a countable set of random numbers generated up front. The second semantics considered by Kozen uses measure theoretic concepts to describe the program as a transformer of probability measure starting from the initial measure which is transformed by the execution of the program. This semantics has been the basis for further work, notably by Monniaux [29] and recently by Cousot and Monerau [6] on defining the semantics of programs that combine probabilistic and non-deterministic behaviors. Our work here uses a simple operational definition for the meaning of probabilistic programs. As a result, we lose the generality that can be achieved by a fundamental measure theoretic treatment espoused by the separate works of Kozen, Monniaux and Cousot. On the other hand, the simplicity of our semantics ensures that we can make inferences on individual program paths and combine them to reason about the program as a whole. Doing so also avoids a loss in precision for many of our benchmarks.

Abstract Monte-Carlo The *abstract Monte-Carlo* (AMC) approach of Monniaux reasons about programs that combine probabilistic and non-deterministic choices [28]. Therein, calls to random variables are dealt with using numerical sampling while the non-deterministic choices are explored using abstract interpretation. The AMC approach cannot, in general, be used to guarantee rigorous bounds on path probabilities, unlike our approach which can derive such bounds. However, our approach does not incorporate non-determinism.

Statistical Model Checking Statistical Model Checking (SMC) is yet another recent approach to verify probabilistic properties of systems with high confidence by using techniques from the field of statistical hypothesis testing [5, 37]. Using finitely many samples from the program, their approach can provide high confidence answers to questions about the probabilities of assertions. Whereas the SMC approach seeks to estimate whether a given lower or upper bound holds on the actual probability with high confidence, we attempt to estimate guaranteed bounds.

Volume Computation Our technique integrates the path constraints with the problem of finding the probability of satisfaction of constraints. This is known to be a $\sharp P$-complete problem and hence computationally hard (almost as hard as PSPACE-complete problems) [2]. The tool LattE Machiato integrates many state of the art techniques for counting lattice points in integer polyhedra and volume computation for convex polyhedra over the reals [8]. As mentioned earlier, the probabilistic symbolic execution approach of Geldenhuys et al [13] and the probabilistic abstract interpretation proposed by Mardziel et al [25] use the tool LattE to count the number of lattice points exactly. While LattE computes exact counts/volumes, our approach focuses on finding interval bounds for the probabilities. We observed two limitations of LattE that make it less ideal for probabilistic program analysis tasks: (a) the existing implementation does not handle non-uniform distributions (over the reals or integers), and (b) exact volume determination for real polyhedra is quite expensive, and often runs out of time/memory on the constraints that are obtained from our benchmarks. *We attempted to complete at least one of the benchmark examples in our approach using LattE, but were unsuccessful due to timeouts or more often out-of-memory errors.* Our approach, which focuses on finding interval bounds rather than exact computation, can handle programs over reals as well as integers, and a wide variety of distributions in addition to the uniform distribution.

7. Conclusion and Future Work

We have developed an analysis framework for probabilistic programs based on considering finitely many paths and estimating the probability of the remainder. Our initial experimental results are promising. However, we have also identified some challenges for motivating further work. Our future work will focus on more efficient polyhedral probability estimators using ellipsoidal and zonotope approximations, and the problem of synthesizing sketches, complementing recent approaches to this problem [4].

Acknowledgments

We thank Prof. Michael Hicks and the anonymous reviewers for their detailed comments. Sankaranarayanan and Chakarov acknowledge support from the US National Science Foundation (NSF) under award numbers CNS-0953941, CNS-1016994 and CPS-1035845. All opinions expressed are those of the authors and not necessarily of the NSF.

References

[1] C. C. Aggarwal and P. S. Yu. A survey of uncertain data algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 21(5), May 2009.

[2] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2009.

[3] O. Bouissou, E. Goubault, J. Goubault-Larrecq, and S. Putot. A generalization of p-boxes to affine arithmetic. *Computing*, 2012.

[4] S. Chaudhuri and A. Solar-Lezama. Smoothing a program soundly and robustly. In *CAV*, volume 6806 of *LNCS*, pages 277–292. Springer, 2011.

[5] E. Clarke, A. Donze, and A. Legay. Statistical model checking of analog mixed-signal circuits with an application to a third order $\delta - \sigma$ modulator. In *Hardware and Software: Verification and Testing*, volume 5394/2009 of *LNCS*, pages 149–163, 2009.

[6] P. Cousot and M. Monerau. Probabilistic abstract interpretation. In *ESOP*, volume 7211 of *LNCS*, pages 169–193. Springer, 2012.

[7] L. H. de Figueiredo and J. Stolfi. Self-validated numerical methods and applications. In *Brazilian Mathematics Colloquium monograph*. IMPA, Rio de Janeiro, Brazil, 1997. Cf. `http://www.ic.unicamp.br/~stolfi/EXPORT/papers/by-tag/fig-sto-97-iaaa.ps.gz`.

[8] J. De Loera, B. Dutra, M. Koeppe, S. Moreinis, G. Pinto, and J. Wu. Software for Exact Integration of Polynomials over Polyhedra. *ArXiv e-prints*, July 2011.

[9] A. Dempster. A generalization of bayesian inference. *Journal of the Royal Statistical Society*, 30:205–247, 1968.

[10] C. Dwork. Differential privacy: A survey of results. In *TAMC*, volume 4978 of *LNCS*, pages 1–19. Springer, 2008.

[11] A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *Intl. Conference on Software Engg. (ICSE)*, 2013. (To Appear, May 2013).

[12] D. Frenkel and B. Smit. *Understanding Molecular Simulation: From Algorithms to Applications*. Academic Press, 2002.

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

[14] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence*, pages 220–229, 2008.

[15] L. Granvilliers and F. Benhamou. Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques. *ACM Trans. On Mathematical Software*, 32(1):138–156, 2006.

[16] H. Hermanns, B. Wachter, and L. Zhang. Probabilistic CEGAR. In *CAV*, volume 5123 of *LNCS*, pages 162–175. Springer, 2008.

[17] S. K. Jha, E. M. Clarke, C. J. Langmead, A. Legay, A. Platzer, and P. Zuliani. A bayesian approach to model checking biological systems. In *CMSB*, volume 5688 of *Lecture Notes in Computer Science*, pages 218–234. Springer, 2009.

[18] R. Jhala and R. Majumdar. Path slicing. In *PLDI '05*, pages 38–47. ACM, 2005.

[19] C. Jones, G. McQuillan, and *et al.* Serum creatinine levels in the US population: Third national health and nutrition examination survey. *Am. J. Kidney Disease*, 32(6):992–999, 1998.

[20] J.-P. Katoen, A. McIver, L. Meinicke, and C. Morgan. Linear-invariant generation for probabilistic programs. In *Static Analysis Symposium (SAS)*, volume 6337 of *LNCS*, page 390406. Springer, 2010.

[21] D. Kozen. Semantics of probabilistic programs. *J. Computer and System Sciences*, 22:328–350, 1981.

[22] M. Kwiatkowska, G. Norman, and D. Parker. A framework for verification of software with time and probabilities. In *FORMATS*, volume 6246 of *LNCS*, pages 25–45. Springer, 2010.

[23] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[24] Kwiatkowska et al. The PRISM model checker. `http://www.prismmodelchecker.org`.

[25] P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa. Dynamic enforcement of knowledge-based security policies. In *Computer Security Foundations Symposium (CSF)*, pages 114–128, JUN 2011.

[26] A. McIver and C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer, 2004.

[27] S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically accurate program transformations. In *Static Analysis Symposium*, volume 6887 of *LNCS*, pages 316–333. Springer, 2011.

[28] D. Monniaux. An abstract monte-carlo method for the analysis of probabilistic programs. In *POPL*, pages 93–101. ACM, 2001.

[29] D. Monniaux. Abstract interpretation of programs as markov decision processes. *Sci. Comput. Program.*, 58(1-2):179–205, 2005.

[30] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[31] A. Pfeffer. IBAL: a probabilistic rational programming language. In *In Proc. 17th IJCAI*, pages 733–740. Morgan Kaufmann Publishers, 2001.

[32] A. D. Pierro, C. Hankin, and H. Wiklicky. Probabilistic λ-calculus and quantitative program analysis. *J. Logic and Computation*, 15(2): 159–179, 2005.

[33] A. Radul. Report on the probabilistic language scheme. In *DLS*, pages 2–10. ACM, 2007.

[34] R. Y. Rubinstein and D. P. Kroese. *Simulation and the Monte Carlo Method*. Wiley Series in Probability and Mathematical Statistics, 2008.

[35] G. Shafer. *A Mathematical Theory of Evidence*. Princeton University Press, 1976.

[36] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *SIGSOFT FSE*, page 58. ACM, 2012.

[37] H. L. S. Younes and R. G. Simmons. Statistical probabilitistic model checking with a focus on time-bounded properties. *Information & Computation*, 204(9):1368–1409, 2006.