CrossMark

Approximate counting in SMT and value estimation for probabilistic programs

Dmitry Chistikov[1] · **Rayna Dimitrova[2]** ·
Rupak Majumdar[2]

Abstract #SMT, or model counting for logical theories, is a well-known hard problem that generalizes such tasks as counting the number of satisfying assignments to a Boolean formula and computing the volume of a polytope. In the realm of satisfiability modulo theories (SMT) there is a growing need for model counting solvers, coming from several application domains (quantitative information flow, static analysis of probabilistic programs). In this paper, we show a reduction from an approximate version of #SMT to SMT. We focus on the theories of integer arithmetic and linear real arithmetic. We propose model counting algorithms that provide approximate solutions with formal bounds on the approximation error. They run in polynomial time and make a polynomial number of queries to the SMT solver for the underlying theory, exploiting "for free" the sophisticated heuristics implemented within modern SMT solvers. We have implemented the algorithms and used them to solve the value problem for a model of loop-free probabilistic programs with nondeterminism.

1 Introduction

Satisfiability modulo theories (SMT) is a foundational problem in formal methods, and the research landscape is not only enjoying the success of existing SMT solvers, but also generating demand for new features. In particular, there is a growing need for *model counting*

✉ Rayna Dimitrova
  rayna@mpi-sws.org

  Dmitry Chistikov
  d.chistikov@warwick.ac.uk

  Rupak Majumdar
  rupak@mpi-sws.org

[1] Department of Computer Science, University of Warwick, Coventry, United Kingdom

[2] Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern and Saarbrücken, Germany

🙋 Springer

solvers; for example, questions in quantitative information flow and in static analysis of probabilistic programs are naturally cast as instances of model counting problems for appropriate logical theories [24,43,52].

We define the #SMT problem that generalizes several model counting questions relative to logical theories, such as computing the number of satisfying assignments to a Boolean formula (#SAT) and computing the volume of a bounded polyhedron in a finite-dimensional real vector space. Specifically, to define model counting modulo a *measured theory*, first suppose every variable in a logical formula comes with a domain which is also a measure space. Assume that, for every logical formula φ in the theory, the set of its models $[\![\varphi]\!]$ is measurable with respect to the product measure; the *model counting (or #SMT)* problem then asks, given φ, to compute the measure of $[\![\varphi]\!]$, called the *model count* of φ.

In our work we focus on the model counting problems for the theories of bounded integer arithmetic and linear real arithmetic. These problems are complete for the complexity class **#P**, so fast exact algorithms are unlikely to exist.

We extend to the realm of SMT the well-known hashing approach from the world of #SAT, which reduces *approximate* versions of counting to decision problems. From a theoretical perspective, we solve a model counting problem with a resource-bounded algorithm that has access to an oracle for the decision problem. From a practical perspective, we show how to use unmodified existing SMT solvers to obtain approximate solutions to model-counting problems. This reduces an approximate version of #SMT to SMT.

Specifically, for integer arithmetic (not necessarily linear), we give a randomized algorithm that approximates the model count of a given formula φ to within a multiplicative factor $(1+\varepsilon)$ for any given $\varepsilon > 0$. The algorithm makes $O(\frac{1}{\varepsilon} |\varphi|)$ SMT queries of size at most $O(\frac{1}{\varepsilon^2} |\varphi|^2)$ where $|\varphi|$ is the size of φ.

For linear real arithmetic, we give a randomized algorithm that approximates the model count with an additive error γN, where N is the volume of a box containing all models of the formula, and the coefficient γ is part of the input. The number of steps of the algorithm and the number of SMT queries (modulo the combined theory of integer and linear real arithmetic) are again polynomial.

As an application, we show how to solve the value problem (cf. [52]) for a model of loop-free probabilistic programs with nondeterminism.

Techniques

Approximation of **#P** functions by randomized algorithms has a rich history in complexity theory [33,34,58,62]. Jerrum, Valiant, and Vazirani [34] described a hashing-based **BPP^NP** procedure to approximately compute any **#P** function, and noted that this procedure already appeared implicitly in previous papers by Sipser [54] and Stockmeyer [58]. The procedure works with encoded computations of a Turing machine and is thus unlikely to perform well in practice. Instead, we show a direct reduction from approximate model counting to SMT solving, which allows us to retain the structure of the original formula. An alternate approach could eagerly encode #SMT problems into #SAT, but experience with SMT solvers suggests that a "lazy" approach may be preferable for some problems.

For the theory of linear real arithmetic, we also need an ingredient to handle continuous domains. Dyer and Frieze [19] suggested a discretization that introduces bounded additive error; this placed approximate volume computation for polytopes—or, in logical terms, approximate model counting for quantifier-free linear real arithmetic—in **#P**. Motivated by the application in the analysis of probabilistic programs, we extend this technique to handle formulas with existentially quantified variables, while Dyer and Frieze only work with quantifier-free formulas. To this end, we prove a geometric result that bounds the effect of

projections: this gives us an approximate model counting procedure for existentially quantified linear arithmetic formulas. Note that applying quantifier elimination as a preprocessing step can make the resulting formula exponentially big; instead, our approach works directly on the original formula that contains existentially quantified variables.

We have implemented our algorithm on top of the Z3 SMT solver [17] and applied it to formulas that encode the value problem for probabilistic programs. Our initial experience suggests that simple randomized algorithms using off-the-shelf SMT solvers can be effective on small examples.

Counting in SMT

#SMT is a well-known hard problem whose instances have been studied before, e. g., in volume computation [19], in enumeration of lattice points in integer polyhedra [2], and as #SAT [28]. Indeed, very simple sub-problems, such as counting the number of satisfying assignments of a Boolean formula or computing the volume of a union of axis-parallel rectangles in $\mathbb{R}^n$ (called Klee's measure problem [37]) are already #**P**-hard (see Sect. 2 below).

Existing techniques for #SMT either incorporate model counting primitives into propositional reasoning [5,44,63] or are based on enumerative combinatorics [24,40,43]. Typically, exact algorithms [24,40,44] are exponential in the worst case, whereas approximate algorithms [43,63] lack provable performance guarantees. In contrast to exact counting techniques, our procedure is easily implementable and uses "for free" the sophisticated heuristics built in off-the-shelf SMT solvers. Although the solutions it produces are not exact, they provably meet user-provided requirements on approximation quality. This is achieved by extending the hashing approach from SAT [10,21,27,28] to the SMT context.

A famous result of Dyer, Frieze, and Kannan [20] states that the volume of a convex polyhedron can be approximated with a multiplicative error in probabilistic polynomial time (without the need for an SMT solver). In our application, analysis of probabilistic programs, we wish to compute the volume of a projection of a Boolean combination of polyhedra; in general, it is, of course, non-convex. Thus, we cannot apply the volume estimation algorithm of [20], so we turn to the "generic" approximation of #**P** using an **NP** oracle instead. Our #SMT procedure for linear real arithmetic allows an additive error in the approximation; it is known that the volume of a polytope does not always have a small exact representation as a rational number [41].

An alternative approach to approximate #SMT is to apply Monte Carlo methods for volume estimation. They can easily handle complicated measures for which there is limited symbolic reasoning available. Like the hashing technique, this approach is also exponential in the worst case [33]: suppose the volume in question, p, is very small and the required precision is a constant multiple of p. In this case, Chernoff bound arguments would suggest the need for $\Omega(\frac{1}{p})$ samples; the hashing approach, in contrast, will perform well. So, while in "regular" settings (when p is non-vanishing) the Monte Carlo approach performs better, "singular" settings (when p is close to zero) are better handled by the hashing approach. The two techniques, therefore, are complementary to each other (see the remark at the end of Sect. 5.5).

Related work

Probably closest to our work is a series of papers by Chakraborty, Meel, Vardi et al. [8–10], who apply the hashing technique to uniformly sample satisfying assignments of SAT formulas [9]. They use CryptoMiniSat [55] as a practical implementation of an **NP**(SAT) oracle, as it has built-in support for XOR (addition modulo 2) constraints that are used for hashing. Their recent work [8] supports weighted sampling and weighted model counting, where different satisfying assignments are associated with possibly different probabilities

(this can be expressed as a discrete case of #SMT). Concurrently, Ermon et al. [21] apply the hashing technique in the context of counting problems, relying on CryptoMiniSat as well. Ermon et al. also consider a weighted setting where the weights of satisfying assignments are given in a factorized form; for this setting, as a basic building block, they invoke an optimization solver ToulBar2 [1] to answer MAP (maximum a posteriori assignment) queries. More recently and concurrently with (the conference version of) our work, Belle, Van den Broeck, and Passerini [4] apply the techniques of Chakraborty et al. in the context of so-called weighted model integration. This is an instance of #SMT where the weights of the satisfying assignments (models) are computed in a more complicated fashion. Belle et al. adapt the procedure of Chakraborty et al., also using CryptoMiniSat, but additionally rely on the Z3 SMT solver to check candidate models against the theory constraints (real arithmetic in this case) encoded by the propositional variables, and use the LattE tool [40] for computing the volume of polyhedra.

We briefly review the problem settings of Ermon et al. [21] and Belle et al. [4,5] in Sect. 2. In our work, the problem setting is more reminiscent of those in Chakraborty et al. [10] and Ermon et al. [21], and the hashing approach itself is the same as the one described, e.g., in [10] for the #SAT case. We lift this idea to the SMT world, in particular for the cases of bounded integer arithmetic and linear real arithmetic with existential quantification. Our implementation is a proof of concept for the extension, to SMT, of the hashing approach to approximate model counting. While we discuss some preliminary experiments in Sect. 6, a scalable implementation and extensive empirical evaluation are beyond the scope of this paper. We now outline some challenges towards a scalable tool for #SMT.

From an implementation perspective, bounded integer arithmetic can be reduced to the Boolean case, which is readily handled by approximate #SAT tools such as ApproxMC [10]. Modern SMT solvers such as Z3 [17] contain conversion and preprocessing heuristics to bit-blast arithmetic formulas. Our approach, on the other hand, handles bounded integer arithmetic formulas directly, relying on the SMT solver for performing word-level reasoning. As in SMT solving, the relative performance of the two techniques (direct theory reasoning vs. bitblasting) is likely to depend on the considered benchmarks, and choosing between them in a practical tool remains an open problem.

Our use of hashing introduces many Boolean XOR constraints. Modern SAT solvers perform poorly on XOR constraints, unless they implement specialized heuristics (see, e.g., the CryptoMiniSat solver [55]). Our implementation currently uses an unmodified theory solver with an additional pre-processor that solves the system of XOR equations (see Sect. 5.6). A better implementation would replace the "usual" SAT solver within the SMT solver to one that has special heuristics for XOR constraints, e.g., those implemented in CryptoMiniSat. An open question is whether there is a different family of hash functions that combines well with theory reasoning. A step in this direction was taken by Chakraborty et al. in their recent work [11], where they use word-level hashing functions to enable better usage of the power of modern SMT solvers. Chakraborty et al. show, empirically, that on a large number of benchmarks word-level reasoning leads to improved performance compared to the bit-level XOR reasoning. However, they also establish that these word-level hash functions do not help for formulas involving word-level multiplication—and, in fact, the XOR-based approach performs better on several such benchmarks [11].

Contributions

We extend, from SAT to SMT, the hashing approach to approximate model counting:

1. We formulate the notion of a measured theory (Sect. 2) that gives a unified framework for model-counting problems.

2. For the theory of bounded integer arithmetic, we provide a direct reduction (Theorem 1 in Sect. 2) from approximate counting to SMT.
3. For the theory of bounded linear real arithmetic, we give a technical construction (Lemma 2 in Sect. 3.3) that lets us extend the results of Dyer and Frieze to the case where the polyhedral set is given as a projection of a Boolean combination of polytopes; this leads to an approximate model counting procedure for this theory (Theorem 2 in Sect. 2).
4. As an application, we show that the value problem for small loop-free probabilistic programs with nondeterminism reduces to #SMT (Sect. 5).

The conference version of this paper appeared as [13].

2 The #SMT problem

We present a framework for a uniform treatment of model counting both in discrete theories like SAT (where it is literally counting models) and in linear real arithmetic (where it is really volume computation for polyhedra). We then introduce the notion of approximation and give an algorithm for approximate model counting by reduction to SMT.

Preliminaries: Counting Problems and #P

A relation $R \subseteq \Sigma^* \times \Sigma^*$ is a *p-relation* if (1) there exists a polynomial $p(n)$ such that if $(x, y) \in R$ then $|y| = p(|x|)$ and (2) the predicate $(x, y) \in R$ can be checked in deterministic polynomial time in the size of x. Intuitively, a p-relation relates inputs x to solutions y. It is easy to see that a decision problem L belongs to **NP** if there is a p-relation R such that $L = \{x \mid \exists y. R(x, y)\}$.

A *counting problem* is a function that maps Σ^* to $\mathbb{N}$. A counting problem $f \colon \Sigma^* \to \mathbb{N}$ belongs to the class **#P** if there exists a p-relation R such that $f(x) = |\{y \mid R(x, y)\}|$, i.e., the class **#P** consists of functions that count the number of solutions to a p-relation [61]. *Completeness* in **#P** is with respect to Turing reductions; the same term is also (ab)used to encompass problems that reduce to a fixed number of queries to a **#P** function (see, e.g., [19]).

#SAT is an example of a **#P**-complete problem: it asks for the number of satisfying assignments to a Boolean formula in conjunctive normal form (CNF) [61]. Remarkably, **#P** characterizes the computational complexity not only of "discrete" problems, but also of problems involving real-valued variables: approximate volume computation (with additive error) for bounded rational polyhedra in $\mathbb{R}^k$ is **#P**-complete [19].

Measured Theories and #SMT

We will now define the notion of model counting that generalizes #SAT and volume computation for polyhedra. Suppose $\mathcal{T}$ is a logical theory. Let $\varphi(x)$ be a formula in this theory with free first-order variables $x = (x_1, \ldots, x_k)$. Assume that $\mathcal{T}$ comes with a fixed interpretation which specifies domains of the variables, denoted $D_1, \ldots, D_k$, and assigns a meaning to predicates and function symbols in the signature of $\mathcal{T}$. Then a tuple $a = (a_1, \ldots, a_k) \in D_1 \times \cdots \times D_k$ is called a *model* of φ if the sentence $\varphi(a_1, \ldots, a_k)$ holds, i.e., if $a \models_{\mathcal{T}} \varphi(x)$. We denote the set of all models of a formula $\varphi(x)$ by $[\![\varphi]\!]$; the *satisfiability problem* for $\mathcal{T}$ asks, for a formula φ given as input, whether $[\![\varphi]\!] \neq \emptyset$.

Consider the special cases of #SAT and volume computation for polyhedra; the corresponding satisfiability problems are SAT and linear programming. For #SAT, atomic predicates are of the form $x_i = b$, for $b \in \{0, 1\}$, the domain D_i of each x_i is $\{0, 1\}$,

and formulas are propositional formulas in conjunctive normal form. For volume computation, atomic predicates are of the form $c_1 x_1 + \cdots + c_k x_k \leq d$, for $c_1, \ldots, c_k, d \in \mathbb{R}$, the domain D_i of each x_i is $\mathbb{R}$, and formulas are conjunctions of atomic predicates. Sets $[\![\varphi]\!]$ in these cases are the set of satisfying assignments and the polyhedron itself, respectively.

Suppose the domains $D_1, \ldots, D_k$ given by the fixed interpretation are measure spaces: each D_i is associated with a σ-algebra $\mathscr{F}_i \subseteq 2^{D_i}$ and a measure $\mu_i : \mathscr{F}_i \to \mathbb{R}$. This means, by definition, that $\mathscr{F}_i$ and μ_i satisfy the following properties: $\mathscr{F}_i$ contains $\emptyset$ and is closed under complement and countable unions, and μ_i is non-negative, assigns 0 to $\emptyset$, and is σ-additive.[1]

In our special cases, these spaces are as follows. For #SAT, each $\mathscr{F}_i$ is the set of all subsets of $D_i = \{0, 1\}$, and $\mu_i(A)$ is simply the number of elements in A. For volume computation, each $\mathscr{F}_i$ is the set of all Borel subsets of $D_i = \mathbb{R}$, and μ_i is the Lebesgue measure.

Assume that each measure μ_i is σ-finite, that is, the domain D_i is a countable union of measurable sets (i.e., of elements of $\mathscr{F}_i$, and so with finite measure associated with them). This condition, which holds for both special cases, implies that the Cartesian product $D_1 \times \cdots \times D_k$ is measurable with respect to a unique *product measure* μ, defined as follows. A set $A \subseteq D_1 \times \cdots \times D_k$ is *measurable* (that is, μ assigns a value to A) if and only if A is an element of the smallest σ-algebra that contains all sets of the form $A_1 \times \cdots \times A_k$, with $A_i \in \mathscr{F}_i$ for all i. For all such sets, it holds that $\mu(A_1 \times \cdots \times A_k) = \mu_1(A_1) \ldots \mu_k(A_k)$.

In our special cases, the product measure $\mu(A)$ of a set A is the number of elements in $A \subseteq \{0, 1\}^k$ and the volume of $A \subseteq \mathbb{R}^k$, respectively.

We say that the theory $\mathscr{T}$ is *measured* if for every formula $\varphi(x)$ in $\mathscr{T}$ with free (first-order) variables $x = (x_1, \ldots, x_k)$ the set $[\![\varphi]\!]$ is measurable. We define the *model count* of a formula φ as $\mathsf{mc}(\varphi) = \mu([\![\varphi]\!])$. Naturally, if the measures in a measured theory can assume non-integer values, the model count of a formula is not necessarily an integer. With every measured theory we associate a *model counting problem*, denoted #SMT[$\mathscr{T}$]: the input is a logical formula $\varphi(x)$ in $\mathscr{T}$, and the goal is to compute the value $\mathsf{mc}(\varphi)$.

The #SAT and volume computation problems are just special cases as intended, since $\mathsf{mc}(\varphi)$ is equal to the number of satisfying assignments of a Boolean formula and to the volume of a polyhedron, respectively.

Note that one can alternatively restrict the theory to a fixed number of variables k, i.e., to $x = (x_1, \ldots, x_k)$, where $x \in D_1 \times \cdots \times D_k$, and introduce a measure μ directly on $D_1 \times \cdots \times D_k$; that is, μ will not be a product measure. Such measures arise, for instance, when μ comes in a factorized form where factors span non-singleton subsets of $\{x_1, \ldots, x_k\}$. A toy example, with $k = 3$, might have μ induced by the probability density function $Z \cdot f_1(x_1, x_2) \cdot f_2(x_2, x_3)$, where f_1 and f_2 are non-negative and absolutely continuous, and the normalization constant Z (sometimes called *the partition function*) is chosen in such a way that $\mu(D_1 \times D_2 \times D_3) = 1$. Note that computing Z, given f_1 and f_2, is itself a #SMT- (i.e., model counting) question: the associated theory has measure $\bar{\mu}$ induced by $f_1 \cdot f_2$, and the goal is to compute $\mathsf{mc}(\mathsf{true})$, where we assume that true is a formula in the theory with $[\![\mathsf{true}]\!] = D_1 \times D_2 \times D_3$. (Much more sophisticated) problems of this form arise in machine learning and have been studied, e.g., by Ermon et al. [21].

Remark A different stance on model counting questions, under the name of weighted model integration (for real arithmetic), was recently suggested by Belle, Passerini, and Van den Broeck [5]. Their problem setting starts with a tuple of real-valued (theory) variables $x = (x_1, \ldots, x_k)$ and a logical formula φ over x and over standalone propositional variables, $p = (p_1, \ldots, p_s)$. All theory atoms in the formula are also abstracted as (different) propositional

---

[1] The reader is referred to standard textbooks on probability and/or measure theory for further background; see, e.g., [18, Chapter 1].

variables, $q = (q_1, \ldots, q_t)$. All literals l of propositional variables p, q are annotated with weight functions $f_l(x)$, which (can) depend on x. Take any total assignment to p, q that satisfies the propositional abstraction of φ and let L be the set of all satisfied literals. The weight of this assignment to p, q is the integral $\int \prod_{l \in L} f_l(x) \, dx$ taken over the area restricted in $\mathbb{R}^k$ by the conjunction of atoms that are associated with literals $l \in L$. The weighted model integral of φ is then the sum of weights of all assignments (to p, q) that satisfy the propositional abstraction of φ.

We discuss several other model counting problems in the following subsection.

Approximate Model Counting

We now introduce *approximate* #SMT and show how approximate #SMT reduces to SMT. We need some standard definitions. For our purposes, a *randomized algorithm* is an algorithm that uses internal coin-tossing. We always assume, whenever we use the term, that, for each possible input x to $\mathscr{A}$, the overall probability, over the internal coin tosses r, that $\mathscr{A}$ outputs a wrong answer is at most $1/4$. (This error probability $1/4$ can be reduced to any smaller $\alpha > 0$, by taking the median across $O(\log \alpha^{-1})$ independent runs of $\mathscr{A}$.)

We say that a randomized algorithm $\mathscr{A}$ *approximates* a real-valued functional problem $\mathscr{C} \colon \Sigma^* \to \mathbb{R}$ *with an additive error* if $\mathscr{A}$ takes as input an $x \in \Sigma^*$ and a rational number $\gamma > 0$ and produces an output $\mathscr{A}(x, \gamma)$ such that

$$\Pr\big[|\mathscr{A}(x, \gamma) - \mathscr{C}(x)| \leq \gamma \, \mathscr{U}(x)\big] \geq 3/4,$$

where $\mathscr{U} \colon \Sigma^* \to \mathbb{R}$ is some specific and efficiently computable upper bound on the absolute value of $\mathscr{C}(x)$, i.e., $|\mathscr{C}(x)| \leq \mathscr{U}(x)$, that comes with the problem $\mathscr{C}$. Similarly, $\mathscr{A}$ *approximates* a (possibly real-valued) functional problem $\mathscr{C} \colon \Sigma^* \to \mathbb{R}$ *with a multiplicative error* if $\mathscr{A}$ takes as input an $x \in \Sigma^*$ and a rational number $\varepsilon > 0$ and produces an output $\mathscr{A}(x, \varepsilon)$ such that

$$\Pr\big[(1 + \varepsilon)^{-1} \mathscr{C}(x) \leq \mathscr{A}(x, \varepsilon) \leq (1 + \varepsilon) \mathscr{C}(x)\big] \geq 3/4.$$

The computation time is usually considered relative to $|x| + \gamma^{-1}$ or $|x| + \varepsilon^{-1}$, respectively (note the inverse of the admissible error). Polynomial-time algorithms that achieve approximations with a multiplicative error are also known as fully polynomial-time randomized approximation schemes (FPRAS) [34].

Algorithms can be equipped with *oracles* solving auxiliary problems, with the intuition that an external solver (say, for SAT) is invoked. In theoretical considerations, the definition of the running time of such an algorithm takes into account the preparation of *queries* to the oracle (just as any other computation), but not the answer to a query—it is returned within a single time step. Oracles may be defined as solving some specific problems (say, SAT) as well as any problems from a class (say, from **NP**). The following result is well-known.

Proposition 1 (generic approximate counting [34,58]) *Let* $\mathscr{C} \colon \Sigma^* \to \mathbb{N}$ *be any member of* **#P**. *There exists a polynomial-time randomized algorithm* $\mathscr{A}$ *which, using an* **NP**-*oracle, approximates* $\mathscr{C}$ *with a multiplicative error.*

In the rest of this section, we present our results on the complexity of model counting problems, #SMT[$\mathscr{T}$], for measured theories. For these problems, we develop randomized polynomial-time approximation algorithms equipped with oracles, in the flavour of Proposition 1. We describe the proof ideas in Sect. 3, and details are provided in Appendix. We formally relate model counting and the value problem for probabilistic programs in Sect. 5; in the implementation, we substitute an appropriate solver for the theory oracle. We illustrate our approach on an example in Sect. 4.

Integer arithmetic. By IA we denote the *bounded* version of integer arithmetic: each free variable x_i of a formula $\varphi(x_1, \ldots, x_k)$ comes with a bounded domain $D_i = [a_i, b_i] \subseteq \mathbb{Z}$, where $a_i, b_i \in \mathbb{Z}$. We use the counting measure $| \cdot | \colon A \subseteq \mathbb{Z} \mapsto |A|$, so the model count $\mathsf{mc}(\varphi)$ of a formula φ is the number of its models. In the formulas, we allow existential (but not universal) quantifiers at the top level. The model counting problem for IA is #**P**-complete.

Example 1 Consider the formula

$$\varphi(x) = \exists y \in [1, 10]. \, (x \geq 1) \wedge (x \leq 10) \wedge (2x + y \leq 6)$$
$$= \exists y. \, (y \geq 1) \wedge (y \leq 10) \wedge (x \geq 1) \wedge (x \leq 10) \wedge (2x + y \leq 6)$$

in the measured theory IA. This formula has one free variable x and one existentially quantified variable y, let's say both with domain $[0, 10]$. It is easy to see that there exist only two values of x, $x \geq 1$, for which there exists a $y \geq 1$ with $2x + y \leq 6$: these are the integers 1 and 2. Hence, $\mathsf{mc}(\varphi) = 2$.   □

Theorem 1 *The model counting problem for* IA *can be approximated with a multiplicative error by a polynomial-time randomized algorithm that has oracle access to satisfiability of formulas in* IA.

Linear real arithmetic. By RA we denote the *bounded* version of linear real arithmetic, with possible existential (but not universal) quantifiers at the top level. Each free variable x_i of a formula $\varphi(x_1, \ldots, x_k)$ comes with a bounded domain $D_i = [a_i, b_i] \subseteq \mathbb{R}$, where $a_i, b_i \in \mathbb{R}$. The associated measure is the standard Lebesgue measure, and the model count $\mathsf{mc}(\varphi)$ of a formula φ is the volume of its set of models. (Since we consider linear constraints, any quantifier-free formula defines a finite union of polytopes. It is an easy geometric fact that its projection on a set of variables will again be a finite union of bounded polytopes. Thus, existential quantification involves only finite unions.)

Example 2 Consider the same formula

$$\varphi(x) = \exists y \in [1, 10]. \, (x \geq 1) \wedge (x \leq 10) \wedge (2x + y \leq 6)$$
$$= \exists y. \, (y \geq 1) \wedge (y \leq 10) \wedge (x \geq 1) \wedge (x \leq 10) \wedge (2x + y \leq 6),$$

this time in the measured theory RA, where $x \in \mathbb{R}$ and $y \in \mathbb{R}$. Note that now $\varphi(x)$ is equivalent to $(x \geq 1) \wedge (x \leq 2.5)$, and thus $\mathsf{mc}(\varphi) = 1.5$: this is the length of the line segment defined by this constraint.   □

We denote the combined theory of (bounded) integer arithmetic and linear real arithmetic by IA + RA. In the model counting problem for RA, the a priori upper bound $\mathscr{U}$ on the solution is $\prod_{i=1}^{k} (b_i - a_i)$; additive approximation of the problem with $\mathscr{U} = 1$ is #**P**-complete.

Theorem 2 *The model counting problem for* RA *can be approximated with an additive error by a polynomial-time randomized algorithm that has oracle access to satisfiability of formulas in* IA + RA.

3 Proof techniques

In this section we explain the techniques behind Theorems 1 and 2. The detailed analysis can be found in Appendix.

3.1 Intuition: hashing-based approximate counting

Let us first explain how the hashing-based approach to approximate counting works. In this subsection we will describe the intuition behind the approach on an abstract level using very simple examples and without referring to any implementation issues. We will later (Sect. 3.2 and 3.3) present the approach in more generality and explain how it can be implemented in practice.

The core of the hashing approach is the following high-level observation (see, e.g., Jerrum et al. [34], and historical notes in the introduction above). Let $\mathcal{H}_m$ be a family of hash functions of the form $h \colon D \to \{0, 1\}^m$ with properties to be fixed below. Intuitively, one expects that, for each element $a \in D$, if a function h is picked at random from $\mathcal{H}_m$, then the image $h(a)$ attains all values from $\{0, 1\}^m$ with equal probabilities. For example, the probability that $h(a) = 0^m$ should equal $1/2^m$. Moreover, this behaviour should, in a way, extend from single elements $a \in D$ to sets: with high probability, the number of elements of a set $S \subseteq D$ that satisfy $h(a) = 0^m$ should be close to $|S|/2^m$. Since this number is, in fact, always integral, one can expect it to be positive if $|S| \gg 2^m$ and equal to zero if $|S| \ll 2^m$. Obviously, for each set S there will be individual functions $h \in \mathcal{H}_m$ violating these inequalities, but for the majority of functions $h \in \mathcal{H}_m$ these inequalities will hold.

Now the idea is to use this observation for estimating the cardinality of a set that is not given to us explicitly. In the scenario we are interested in, the set S will be the set of all models of a given formula. More formally, consider a formula $\varphi(x)$ in some measured theory with one free variable. For simplicity, suppose the theory is IA, integer arithmetic with a bounded domain $D = [0, M]$, where the measure of a set $A \subseteq D$ is simply the cardinality of A. Denote by S the set of all models of the formula $\varphi(x)$, i.e., $S = [\![\varphi]\!]$. If, as above, the hash function $h \colon D \to \{0, 1\}^m$ is chosen at random from an appropriate family $\mathcal{H}_m$, then with high probability the formula $\varphi(x) \wedge (h(x) = 0^m)$ is satisfiable if $\mathsf{mc}(\varphi) \gg 2^m$ and unsatisfiable if $\mathsf{mc}(\varphi) \ll 2^m$.

Notice that we do not a priori know $|S|$, but we do know that it is between 0 (the formula is unsatisfiable) and the entire volume D. So, we can iteratively search over this range to approximate $|S|$. Let us therefore arrange the following process to estimate $\mathsf{mc}(\varphi)$. We shall first check if the formula $\varphi(x)$ is satisfiable; if it is not, $\mathsf{mc}(\varphi) = 0$ and the process terminates immediately. Suppose $\varphi(x)$ is satisfiable; we will go over the values of m from 1 to about $\log M$ in increasing order and for each of them decide, admitting a certain element of uncertainty, whether $\mathsf{mc}(\varphi) \gg 2^m$ or $\mathsf{mc}(\varphi) \ll 2^m$. Specifically, for each m we will draw a hash function h at random from the family $\mathcal{H}_m$ and check satisfiability of the formula $\varphi(x) \wedge (h(x) = 0^m)$. If the formula is unsatisfiable, this will suggest that $\mathsf{mc}(\varphi) \ll 2^m$ or $\mathsf{mc}(\varphi) \approx 2^m$, and we will therefore terminate the process. If the formula is satisfiable, this will suggest that $\mathsf{mc}(\varphi) \gg 2^m$ or $\mathsf{mc}(\varphi) \approx 2^m$, and we will therefore continue the process, going on to the increased value of m. (Note that if $\mathsf{mc}(\varphi) \approx 2^m$ for some m, the formula $\varphi(x) \wedge (h(x) = 0^m)$ is about equally likely to be satisfiable and unsatisfiable.) Rare events aside, we should expect the process to terminate when the value of m is such that $2^m \approx \mathsf{mc}(\varphi)$.

Example 3 Suppose $\varphi(x)$ is the formula $x = 42$, and the domain of the variable x is $D = [0, 255]$. The set $S = [\![\varphi]\!]$ is a singleton: $S = \{42\}$. Since $S \neq \emptyset$, that is, the formula $\varphi(x)$ is satisfiable, we start the process described above.

We set $m = 1$ at first and draw a hash function $h_1 \colon D \to \{0, 1\}$ at random from the set $\mathcal{H}_1$. Let us omit the description of the set $\mathcal{H}_1$; suppose the hash function that we draw happens to be $h_1(x) = x \bmod 2$. We now check satisfiability of the formula $\varphi(x) \wedge (x \bmod 2 = 0)$,

which is equivalent to $(x = 42) \land (x \bmod 2 = 0)$. As $x = 42$ is a model of this formula, we proceed to $m = 2$. Now we need to draw a hash function from $\mathscr{H}_2$. Suppose it has the form $h_2(x) = \lfloor x/64 \rfloor$ where the result is interpreted as an element of $\{0, 1\}^2$ in a natural way. Since $\lfloor 42/64 \rfloor = 0$, the formula $(x = 42) \land (h_2(x) = (0, 0))$ is satisfiable. Once we have determined this, we proceed to $m = 3$. Here we need to draw a hash function at random from the set $\mathscr{H}_3$; suppose we draw $h_3(x) = (h_{31}, h_{32}, h_{33})$ where $h_{31} = (x + 1) \bmod 2$; then, regardless of how h_{32} and h_{33} are defined, the formula $(x = 42) \land (h_3(x) = (0, 0, 0))$ will be unsatisfiable. Therefore, our process will terminate at $m = 3$.

What will be the outcome of the process? The exact answer is tightly related to the properties of the families of the hash functions $\mathscr{H}_m$. More precisely, we asserted previously that with high probability the formula $\varphi(x) \land (h(x) = 0^m)$ where $h \in \mathscr{H}_m$ is satisfiable if $\mathsf{mc}(\varphi) \gg 2^m$ and unsatisfiable if $\mathsf{mc}(\varphi) \ll 2^m$. The precise meaning of $\gg$ and $\ll$ will, in fact, influence the final estimate of $\mathsf{mc}(\varphi)$. From the fact that in our run the process terminates at $m = 3$ we can draw the conclusion that (with high probability) $\mathsf{mc}(\varphi)$ belongs to the interval $[u_* 2^m, u^* 2^m] = [8u_*, 8u^*]$ where u_* and u^* are positive constants that do not depend on the formula φ and form a part of the description of our algorithm. One can imagine, for instance, that $u_* = 1/2$ and $u^* = 1$; in our case this will give us the interval $[4, 8]$. (The actual formulas defining u_* and u^* can be found in Appendix A.2.) Of course, in our case this answer will not be very satisfactory, because the correct value of $\mathsf{mc}(\varphi)$ is 1. If, however, we compute the probability of such an outcome, i.e., the probability that the process will only terminate at $m \geq 3$ on input φ, we will see that this is a moderately rare event. If each bit of all hash functions from $\mathscr{H}_m$ is chosen independently (imagine, for example, that picking h from $\mathscr{H}_m$ corresponds to picking the values of each $h(x)$ independently and uniformly—this corresponds to the "ideal" hashing), then this probability will be $1/8$. In comparison, with probability $1/2$ the process will stop at $m = 1$, which corresponds to the interval $[1/2, 1]$—and this interval contains the correct value. Standard error reduction techniques will help us amplify the probability of such successful outcomes, thus making it very likely (according to our choice of α) that the guessed interval will contain the correct value of $\mathsf{mc}(\varphi)$. In general, with high probability, the higher the values of m that the process attains, the larger the estimate of $\mathsf{mc}(\varphi)$. $\square$

3.2 Approximate discrete model counting

We now explain the idea behind Theorem 1 in more detail, zooming in on some aspects that we only sketched previously. Let $\varphi(x)$ be an input formula in IA and let $x = (x_1, \ldots, x_k)$ be the free variables of φ. Suppose M is a big enough integer such that all models of φ have components not exceeding M, i.e., $[\![\varphi]\!] \subseteq [0, M]^k$.

Our approach to approximating $\mathsf{mc}(\varphi) = |[\![\varphi]\!]|$ works as follows. Suppose our goal is to find a value v such that $v \leq \mathsf{mc}(\varphi) \leq 2v$, and we have an oracle $\mathscr{E}$, for "Estimate", answering questions of the form $\mathsf{mc}(\varphi) \geq^? N$. Then it is sufficient to make such queries to $\mathscr{E}$ for $N = N_m = 2^m$, $m = 0, \ldots, k \log(M + 1)$, and the overall algorithm design is reduced to implementing such an oracle efficiently.

As we already know, such an implementation can be done with the help of *hashing*. Suppose that a hash function h, taken at random from some family $\mathscr{H}$, maps elements of $[0, M]^k$ to $\{0, 1\}^m$. If the family $\mathscr{H}$ is chosen appropriately, then each potential model w is mapped by h to, say, 0^m with probability 2^{-m}; moreover, one should expect that any set $S \subseteq [0, M]^k$ of size d has roughly $2^{-m} \cdot d$ elements in $h^{-1}(0^m) = \{w \in [0, M]^k \mid h(w) = 0^m\}$. In other words, if $|S| \geq 2^m$, then $S \cap h^{-1}(0^m)$ is non-empty with high probability, and if $|S| \ll 2^m$, then $S \cap h^{-1}(0^m)$ is empty with high probability. So—rephrasing slightly

---

Algorithm 1: Approximate model counting for IA

Input: formula $\varphi(x)$ in IA
Output: value $v \in \mathbb{R}$
Parameters: $\varepsilon \in (0, 1)$, /* approximation factor */
                 $\alpha \in (0, 1)$, /* error probability */
                 $a \in \mathbb{N}$ /* enumeration limit for SMT solver */
Compute values m^*, q, p, r based on parameters (see text);

1   **if** $(e := \mathrm{SMT}(\varphi, p + 1)) \le p$ **then return** e;
2   $\psi(x, x') = \varphi(x) \wedge t(x, x')$;
3   $\psi_q(\mathbf{x}, \mathbf{x}') = \psi(x^1, x'^1) \wedge \psi(x^2, x'^2) \wedge \cdots \wedge \psi(x^q, x'^q)$;
4   $k' :=$ number of bits in $\mathbf{x}'$;
5   **for** $m = 1, \ldots, m^*$ **do**
6      $c := 0$; /* majority vote counter */
7   **for** $j = 1, \ldots, r$ **do**
8   **if** $\mathscr{E}(\psi_q, k', m, a)$ **then** $c := c + 1$
9   **if** $c \le r/2$ **then break**;
10 **return** $\sqrt[q]{a \cdot 2^{m-0.5}}$

---

the observations outlined above—our task is reduced to distinguishing between empty and non-empty sets. This, in turn, is a satisfiability question and, as such, can be entrusted to the IA solver. As a result, we reduced the approximation of the model count of φ to a series of satisfiability questions in IA.

Our algorithm posts these questions as SMT queries of the form

$$\varphi(x) \wedge t(x, x') \wedge (h'(x') = 0^m), \tag{1}$$

where x and x' are tuples of integer variables, each component of x' is either 0 or 1, the formula $t(x, x')$ says that x' is binary encoding of x, and the IA formula $h'(x') = 0^m$ encodes the computation of the hash function h on input x.

Algorithm 2: Satisfiability "oracle" $\mathscr{E}$

Input: formula $\psi_q(\mathbf{x}, \mathbf{x}')$ in IA; $k', m, a \in \mathbb{N}$
Output: *true* or *false*
1   $h' := \text{PICK- HASH}(k', m)$;
2   $\psi_{h'}(\mathbf{x}, \mathbf{x}') = \psi_q(\mathbf{x}, \mathbf{x}') \wedge (h'(\mathbf{x}') = 0^m)$;
3   **return** $(\mathrm{SMT}(\psi_{h'}, a) \ge a)$ /* check if $\psi_{h'}$ has at least a models */

Algorithm 1 is the basis of our implementation. It returns a value v that satisfies the inequalities $(1 + \varepsilon)^{-1} \mathrm{mc}(\varphi) \le v \le (1 + \varepsilon) \, \mathrm{mc}(\varphi)$ with probability at least $1 - \alpha$. Algorithm 1 uses a set of parameters to discharge small values by enumeration in the SMT solver (parameters a, p) and to query the solver for larger instances (parameters m^*, q, r). The procedure $\mathscr{E}$ given as Algorithm 2 asks the SMT solver for IA to produce a models (for a positive integer parameter a) to formulas of the form (1) by calling the procedure SMT.

To achieve the required precision with the desired probability, the algorithm constructs a conjunction of q copies of the formula (over disjoint sets of variables), where the number of copies q is defined[2] as

---

[2] We refer the reader to Appendix A.2 for a detailed description.

$$q = \left\lceil \frac{1 + 4\log(\sqrt{a+1}+1) - 2\log a}{2\log(1+\varepsilon)} \right\rceil;$$

we refer the reader to Appendix A.2 for a detailed description. This results in a formula with $k' = qk\lceil\log(M+1)\rceil = O(|\varphi|/\varepsilon)$ binary variables, where $|\varphi|$ denotes the size of the original formula φ. Then, in lines 5–9, Algorithm 1 performs for each dimension of the hash function in the range $\{1, \ldots, m^*\}$ a majority vote over r calls to the procedure $\mathscr{E}$, where the values of m^* and r are computed as follows:

$$m^* = \lfloor k' - 2\log(\sqrt{a+1}+1) \rfloor, \quad r = \left\lceil 8 \cdot \ln\left(\frac{1}{\alpha} \cdot \lfloor k' - 2\log(\sqrt{a+1}+1)\rfloor\right) \right\rceil.$$

For a formal derivation of these values, see Appendix A.3.

In a practical implementation, early termination of the majority-vote loop is possible as soon as the number of positive answers given by $\mathscr{E}$ exceeds $r/2$.

For formulas φ with up to $p = \lceil(\sqrt{a+1}-1)^{2/q}\rceil$ models, Algorithm 1 returns the exact model count $\mathsf{mc}(\varphi)$ (line 1 in Algorithm 1) computed by the procedure SMT, which repeatedly calls the solver, counting the number of models up to $p + 1$.

The values of m^*, q, p, and r used in Algorithm 1, as well as the choice of the return value $v = \sqrt[q]{a \cdot 2^{m-0.5}}$, guarantee its correctness and are formally derived in Appendix.

For a fixed approximation factor ε the number q of copies depends only on the parameter a. More precisely, the larger the parameter a is, the fewer copies q are necessary. While, in general, smaller values for q result in fewer variables in the queries to the SMT solver, the number of queries at each step of the loop in Algorithm 1 increases with a, albeit not drastically. One possible heuristic for balancing this trade-off is choosing as a the smallest value after which the value for q stabilizes. We have observed empirically that applying this heuristic leads to good performance, and have used it to select the values for a for the experiments on which we report in Sect. 5.6.

The family of hash functions $\mathscr{H}$ used by PICK- HASH in Algorithm 2 needs to satisfy the condition of *pairwise independence*: for any two distinct vectors $x_1, x_2 \in [0, M]^k$ and any two strings $w_1, w_2 \in \{0, 1\}^m$, the probability that a random function $h \in \mathscr{H}$ satisfies $h(x_1) = w_1$ and $h(x_2) = w_2$ is equal to $1/2^{2m}$. The condition of pairwise independence is used by Algorithm 1 via the following proposition, known as (a simple form of) the *Leftover Hash Lemma*. It was originally proved by Impagliazzo, Levin, and Luby [32], and here we use a formulation due to Trevisan [59].

Lemma 1 *Let $\mathscr{H}$ be a family of pairwise independent hash functions $h \colon \{0, 1\}^n \to \{0, 1\}^m$. Let $S \subseteq \{0, 1\}^n$ be such that $|S| \geq 4/\rho^2 \cdot 2^m$. For $h \in \mathscr{H}$, let ξ be the cardinality of the set $\{w \in S \colon h(w) = 0^m\}$. Then*

$$\Pr\left[\left|\xi - \frac{|S|}{2^m}\right| \geq \rho \cdot \frac{|S|}{2^m}\right] \leq \frac{1}{4}.$$

There are several constructions for pairwise independent hash functions; we employ a commonly used family, that of random XOR constraints [3,9,28,62]. Given k' and m, the family contains (in binary encoding) all functions $h' = (h'_1, \ldots, h'_m)\colon \{0, 1\}^{k'} \to \{0, 1\}^m$ with $h'_i(x_1 \ldots, x_{k'}) = a_{i,0} + \sum_{j=1}^{k'} a_{i,j}x_j$, where $a_{i,j} \in \{0, 1\}$ for all i and $+$ is the XOR operator (addition in GF(2)). By randomly choosing the coefficients $a_{i,j}$ we get a random hash function from this family. The size of each query is thus bounded by $O(k'^2) = O(\frac{1}{\varepsilon^2}|\varphi|^2)$, where $|\varphi|$ is again the size of the original formula φ, and there will be at most $m^* + 1 \leq k' + O(1) = O(\frac{1}{\varepsilon}|\varphi|)$ queries in total.

Table 1 Input and runtime parameters

i	ε	α	a	Bits	m	Time (s)	Result
8	0.2	0.1	100	21	10	8.16	41.6801
16	0.2	0.1	100	45	10	18.87	41.6801
32	0.2	0.1	100	93	10	44.81	41.6801

ε, parameter in the multiplicative approximation factor $(1 + \varepsilon)$; α, maximum error probability; a, the SMT enumeration threshold (number of models the SMT solver checks for); Bits, number of binary variables in the formula given to the solver; m, maximal hash size; Result, approximate model count

Example 4 Consider the formula $\varphi(x) = (x \leq 42)$, where the integer variable x ranges over the sets $M_i = [1, 2^{i-1} - 1]$, for $i \in \{8, 16, 32\}$. The model count $\mathsf{mc}(\varphi) = 42$ is small, while the size of the variable domain changes with i and for $i = 32$ is quite significant. Table 1 illustrates the performance of our approximate counting algorithm on input φ for this set of values of i. The parameter ε in the multiplicative approximation factor $(1+\varepsilon)$ is set to 0.2, and the maximum error probability α is set to 0.1. We report the number of Boolean variables in the formula given to the solver (after making the respective number of copies), and the running time in seconds. The table shows that the running time, as well as the number of calls to the SMT solver, are small, which reflects the small model count (the main loop of Algorithm 1 terminates early). As the size of the domain increases, the size of the SMT queries also increases, which, however, leads to only a moderate increase in the overall running time. □

Note that the entire argument remains valid even if φ has existentially quantified variables: queries (1) retain them as is. The prefix of existential quantifiers could simply be dropped from (1), as searching for models of quantifier-free formulas already captures existential quantification. It is important, though, that the model enumeration done by the procedure SMT in Algorithms 1 and 2 only count distinct assignments to the *free* variables of φ and $\psi_{h'}$ respectively.

3.3 Approximate continuous model counting

In this subsection we explain the idea behind Theorem 2. Let φ be a formula in RA; using appropriate scaling, we can assume without loss of generality that all its variables share the same domain. Suppose $[\![\varphi]\!] \subseteq [0, M]^k$ and fix some γ, with the prospect of finding a value v that is at most $\varepsilon = \gamma M^k$ away from $\mathsf{mc}(\varphi)$ (we take M^k as the value of the upper bound $\mathscr{U}$ in the definition of additive approximation). We show below how to reduce this task of approximate continuous model counting to additive approximation of a model counting problem for a formula with a discrete set of possible models, which, in turn, will be reduced to that of multiplicative approximation.

We first show how to reduce our continuous problem to a discrete one. Divide the cube $[0, M]^k$ into s^k small cubes with side δ each, $\delta = M/s$. For every $y = (y_1, \ldots, y_k) \in \{0, 1, \ldots, s - 1\}^k$, set $\psi'(y) = 1$ if at least one point of the cube $C(y) = \{y_j \delta \leq x_j \leq (y_j + 1) \delta, 1 \leq j \leq k\}$ satisfies φ; that is, if $C(y) \cap [\![\varphi]\!] \neq \emptyset$.

Imagine that we have a formula ψ such that $\psi(y) = \psi'(y)$ for all $y \in \{0, 1, \ldots, s - 1\}^k$, and let ψ be written in a theory with a uniform measure that assigns "weight" M/s to each point $y_j \in \{0, 1, \ldots, s - 1\}$; one can think of these weights as coefficients in numerical integration. From the technique of Dyer and Frieze [19, Theorem 2] it follows that for a quantifier-free φ and an appropriate value of s the inequality $|\mathsf{mc}(\psi) - \mathsf{mc}(\varphi)| \leq \varepsilon/2$ holds.

Indeed, Dyer and Frieze prove a statement of this form in the context of volume computation of a polyhedron, defined by a system of inequalities $Ax \leq b$. However, they actually show a stronger statement: given a collection of m hyperplanes in $\mathbb{R}^k$ and a set $[0, M]^k$, an appropriate setting of s will ensure that out of s^k cubes with side $\delta = M/s$ only a small number J will be *cut*, i.e., intersected by some hyperplane. More precisely, if $s = \lceil mk^2 M^k/(\varepsilon/2) \rceil$, then this number J will satisfy the inequality $\delta^k \cdot J \leq \varepsilon/2$. Thus, the total volume of cut cubes is at most $\varepsilon/2$, and so, in our terms, we have $|\mathsf{mc}(\psi) - \mathsf{mc}(\varphi)| \leq \varepsilon/2$ as desired.

However, in our case the formula φ need not be quantifier-free and may contain existential quantifiers at the top level. If $\varphi(x) = \exists u.\Phi(x, u)$ where Φ is quantifier-free, then the constraints that can "cut" the x-cubes are not necessarily inequalities from Φ. These constraints can rather arise from projections of constraints on variables x and, what makes the problem more difficult, their combinations. However, we are able to prove the following statement:

Lemma 2 *The number $\bar{J}$ of points $y \in \{0, 1, \ldots, s-1\}^k$ for which cubes $C(y)$ are cut satisfies $\bar{\delta}^k \cdot \bar{J} \leq \varepsilon/2$ if $\bar{\delta} = M/\bar{s}$, where $\bar{s} = \lceil 2^{\overline{m}+2k} k^2 M^k/(\varepsilon/2) \rceil = \lceil 2^{\overline{m}+2k} k^2/(\gamma/2) \rceil$ and $\overline{m}$ is the number of atomic predicates in Φ.*

Proof Observe that a cube $C(y)$ is cut if and only if it is intersected by a hyperplane defined by some predicate in variables x. Such a predicate does not necessarily come from the formula Φ itself, but can arise when a polytope in variables (x, u) is projected to the space associated with variables x. Put differently, each cut cube $C(y)$ has some d-dimensional face with $0 \leq d \leq k-1$ that "cuts" it; this face is an intersection of $C(y)$ with some affine subspace π in variables x.

Consider this subspace π. It can be, first, the projection of a hyperplane defined in variables (x, u) by an atomic predicate in Φ or, second, the projection of an intersection of several such hyperplanes. Now note that each predicate in (x, u) defines exactly one hyperplane; an intersection of hyperplanes in (x, u) projects to some specific affine subspace in variables x. Therefore, each "cutting" affine subspace π is associated with a distinct subset of atomic predicates in Φ, where, since the domain is bounded, we count in constraints $0 \leq x_j \leq M$ as well. This gives us at most $2^{\overline{m}+2k}$ cutting subspaces, so it remains to apply the result of Dyer and Frieze with $m = 2^{\overline{m}+2k}$.   □

A consequence of the lemma is that the choice of the number $\bar{s}$ ensures that the formula $\psi(y) = \exists x.(\varphi(x) \wedge x \in C(y))$ written in the combined theory $\mathsf{IA} + \mathsf{RA}$ satisfies the inequality $|\mathsf{mc}(\psi) - \mathsf{mc}(\varphi)| \leq \varepsilon/2$. Here we associate the domain of each free variable $y_j \in \{0, 1, \ldots, \bar{s}-1\}$ with the uniform measure $\mu_j(v) = M/\bar{s}$. Note that the value of $\bar{s}$ chosen in Lemma 2 will still keep the number of steps of our algorithm polynomial in the size of the input, because the number of bits needed to store the integer index along each axis is $\lceil \log(\bar{s}+1) \rceil$ and not $\bar{s}$ itself.

As a result, it remains to approximate $\mathsf{mc}(\psi)$ with additive error of at most $\varepsilon' = \varepsilon/2 = \gamma M^k/2$, which can be done by invoking the procedure from Theorem 1 that delivers approximation with multiplicative error $\beta = \varepsilon'/M^k = \gamma/2$.

4 A fully worked-out example

We now show how our approach to #SMT, developed in Sects. 2 and 3 above, works on a specific example, coming from the value problem for *probabilistic programs*. Probabilistic programs are a means of describing probability distributions; the model we use combines

probabilistic assignments and nondeterministic choice, making programs more expressive, but analysis problems more difficult.

For this section we choose a relatively high level of presentation in order to convey the main ideas in a more understandable way; a formal treatment follows in Sect. 5, where we discuss (our model of) probabilistic programs and their analysis in detail.

The Monty Hall problem [50,53]

We describe our approach using as an example the following classic problem from probability theory. Imagine a television game show with two characters: the player and the host. The player is facing three doors, numbered 1, 2, and 3; behind one of these there is a car, and behind the other two there are goats. The player initially picks one of the doors, say door i, but does not open it. The host, who knows the position of the car, then opens another door, say door j with $j \neq i$, and shows a goat behind it. The player then gets to open one of the remaining doors. There are two available strategies: *stay* with the original choice, door i, or *switch* to the remaining alternative, door $k \notin \{i, j\}$. The Monty Hall problem asks, which strategy is better? It is widely known that, in the standard probabilistic setting of the problem, the switching strategy is the better one: it has payoff 2/3, i.e., it chooses the door with the car with probability 2/3; the staying strategy has payoff of only 1/3.

Modeling with a probabilistic program

We model the setting of the Monty Hall problem with the probabilistic program in Procedure 3: "Switch" strategy in Monty Hall problem, which implements the "switch" strategy.

Procedure 3: "Switch" strategy in Monty Hall problem

```
c ~ Uniform({1, 2, 3})                              /* position of the car */
i := 1                                   /* initial choice of the player */
choice:
    case: j := 2; assume(j ≠ c)
    case: j := 3; assume(j ≠ c)
/* the host opens door j with a goat */
if i ≠ c then accept else reject          /* the player switches from door i */
```

In this problem, there are several kinds of uncertainty and choice, so we briefly explain how they are expressed with the features of our programming model.

First, there is uncertainty in what door hides the car and what door the player initially picks. It is standard to model the initial position of the car, c, by a random variable distributed uniformly on $\{1, 2, 3\}$; we simply follow the information-theoretic guidelines here. At the same time, due to the symmetry of the setting we can safely assume that the player always picks door $i = 1$ at first, so here choice is modeled by a deterministic assignment.

Second, there is uncertainty in what door the host opens. We model this with nondeterministic choice. Since the host knows that the car is behind door c and does not open door c accordingly, we restrict this choice by stipulating that $j \neq c$. For the semantics of the program, this means that for different outcomes of the probabilistic assignment $c \sim \mathsf{Uniform}(\{1, 2, 3\})$ different sets of paths through the program are available (some paths are excluded, because they are incompatible with the results of observations stipulated by **assume** statements[3]).

---

[3] Our **assume** statement has the same semantics as the **observe** statement in [29].

Table 2 Semantics of the probabilistic program in procedure 3: "Switch" strategy in Monty Hall problem

Nondeterministic branches:	Probabilistic outcomes:		
	$c = 1$ $\Pr = 1/3$	$c = 2$ $\Pr = 1/3$	$c = 3$ $\Pr = 1/3$
—with $j = 2$	reject	×	accept
—with $j = 3$	reject	accept	×
Verdict	rejected	accepted	accepted
Belongs to Accept	no	yes	yes
Belongs to Term	yes	yes	yes

Note that we don't know the nature of the host's choice in the case that more than one option is available (when $c = 1$, either element of $\{2, 3\}$ can be chosen as j). In principle, this choice may be cooperative (the host helps the player to win the car), adversarial (the host wants to prevent the player from winning), probabilistic (the host tosses a coin), or any other. In our example, the cooperative and the adversarial behavior of the host are identical, so our model is compatible with either of them. For now, let us defer the in-depth discussion of the treatment of nondeterminism to Sect. 5.3.

Finally, uncertainty in the final choice of the player is modeled by fixing a specific behaviour of the player and declaring acceptance if the result is successful. Our procedure implements the "switching" strategy; that is, the player always switches from door i. The analysis of the program will show how good the strategy is.

Semantics and value of the program

Informally, consider all possible outcomes of the probabilistic assignments. Restrict attention to those that may result in the program reaching (nondeterministically) at least one of accept or reject statements—such elementary outcomes form the set Term (for "termination"); only these scenarios are compatible with the observations. Similarly, some of these outcomes may result in the program reaching (again, nondeterministically) an accept statement—they form the set Accept; the interpretation is that for these scenarios the strategy is successful.

These sets Term and Accept are events in a probability space. The *value* of the program (in this case interpreted as the payoff of the player's strategy) is the probability of acceptance conditioned on termination:[4]

$$\mathsf{val}(\mathsf{Switch}) = \Pr[\mathsf{Accept} \mid \mathsf{Term}] = \frac{\Pr[\mathsf{Accept}]}{\Pr[\mathsf{Term}]},$$

where, in general, we assume $\Pr[\mathsf{Term}] > 0$ and the last equality follows because Accept ∩ Term = Accept. In general, this semantics corresponds to the cooperative behavior of the host, but in our case the adversarial behavior would be identical: there is no value of c such that one nondeterministic choice leads to accept and another leads to reject. (We can also deal with adversarial nondeterminism, see Sect. 5.3.)

Indeed, consider Table 2, which illustrates the semantics of the probabilistic program in Procedure 3: "Switch" strategy in Monty Hall problem. There are three probabilistic assignments $c = 1, 2, 3$, each associated with probability $1/3$. For $c = 1$ there are two paths to reject, and for each of $c = 2, 3$ there is a single path to accept and a path that hits a violated assume, indicated by the symbol ×. Therefore, the nondeterministic execution for $c = 1$ is rejecting, and the nondeterministic executions for $c = 2$ and $c = 3$ are accepting. The set

---

[4] As we consider loop-free probabilistic programs, all executions are finite. Thus, here a "terminating" execution is one that satisfies all assume statements which it encounters, and reaches accept or reject.

Accept thus includes the assignments $c = 2$ and $c = 3$, and the set **Term** all three assignments $c = 1, 2, 3$; as a result, $\mathsf{val}(\mathsf{Switch}) = \Pr[\mathsf{Accept}]/\Pr[\mathsf{Term}] = (2/3)/(3/3) = 2/3$, as intended.

Remark Probably the most common mistake that occurs in the analysis of the Monty Hall example (as a puzzle in probability theory) is an inadequate choice of the probability space. Note that our model only associates probabilities with the choice of position of the car ($c \in \{1, 2, 3\}$). The **assume** statements in the program do not act on these probabilistic assignments directly: rather, they eliminate certain paths through the program (more precisely, the paths that hit a violated **assume**). If for a particular probabilistic outcome all paths are eliminated, then this outcome is removed from the set **Term**, thus rescaling the probability weight for all other outcomes (this does not happen in the Monty Hall example). In all other aspects, however, the space of all probabilistic outcomes ($c \in \{1, 2, 3\}$) remains the same, and each individual outcome is classified as accepted or rejected according to the standard (cooperative) semantics of the induced nondeterministic execution.

Reduction of value estimation to model counting

To estimate the value of the program, we first reduce its computation to a model counting problem (as defined in Sect. 2) for an appropriate logical theory. We write down the verification condition $\mathsf{vc}(N, P)$ that defines a valid computation of the program, by asserting a relation between (values of) nondeterministic and probabilistic variables N and P. Then we construct existential formulas of the form

$$\varphi_{\mathsf{acc}}(P) = \exists\, N \,.\, \mathsf{vc}(N, P) \wedge \mathsf{accept} \quad \text{and}$$
$$\varphi_{\mathsf{term}}(P) = \exists\, N \,.\, \mathsf{vc}(N, P) \wedge (\mathsf{accept} \vee \mathsf{reject}),$$

which assert that the program terminates with "accept" (resp. "accept" or "reject"), and whose sets of models (i.e., satisfying assignments) are exactly the sets **Accept** and **Term** defined above. For the Monty Hall program, these formulas $\varphi_{\mathsf{acc}}(c)$ and $\varphi_{\mathsf{term}}(c)$, with $c \in \{1, 2, 3\}$, will be equivalent to $c \neq 1$ and **true**, respectively. The value of the program is the ratio $\mathsf{mc}(\varphi_{\mathsf{acc}})/\mathsf{mc}(\varphi_{\mathsf{term}})$, where $\mathsf{mc}(\cdot)$ denotes the model count of a formula, as in Sect. 2. Technically, we can use IA, the theory of integer arithmetic, with the domain $\{1, 2, 3\}$ for the free variable c and with the counting measure $|\cdot|: A \mapsto |A|$, also following Sect. 2. So in our example, $\mathsf{mc}(\varphi_{\mathsf{acc}}) = 2$ and $\mathsf{mc}(\varphi_{\mathsf{term}}) = 3$.

Computing the value of the program

We show how our method (see Sect. 3.2) estimates $\mathsf{mc}(\varphi_{\mathsf{acc}})$. We make several copies of the variable c, denoted $c^1, \ldots, c^q$. The formula

$$\varphi(\mathbf{c}) = \varphi_{\mathsf{acc}}(c^1) \wedge \varphi_{\mathsf{acc}}(c^2) \wedge \cdots \wedge \varphi_{\mathsf{acc}}(c^q)$$

has 2^q models, and we can estimate $\mathsf{mc}(\varphi_{\mathsf{acc}})$ by estimating $\mathsf{mc}(\varphi)$ and taking the qth root of the estimate. Enlarging φ_{acc} to φ and then taking the qth root increases precision: for example, if the approximation procedure gives a result up to a factor of 2, the qth root of the estimate for $\mathsf{mc}(\varphi)$ gives an approximation for $\mathsf{mc}(\varphi_{\mathsf{acc}})$ up to a factor of $2^{1/q}$.

Now observe that for a hash function h with values in $\{0, 1\}^m$, taken at random from an appropriate family, the expected model count of the formula

$$\varphi(\mathbf{c}) \wedge (h(\mathbf{c}) = 0^m) \tag{2}$$

is $\mathsf{mc}(\varphi) \cdot 2^{-m}$. By a Chernoff bound argument, the model count is concentrated around the expectation. Our algorithm will, for increasing values of m, sample random hash functions

Table 3 Typical run for the Monty Hall example

m	Satisfiable	Unsatisfiable	Majority vote
0…6	62	0	Sat
7…9	61	1	Sat
10	55	7	Sat
11	50	12	Sat
12	48	14	Sat
13	21	41	Unsat

from an appropriate class, construct the formula (2), and give the formula to an SMT solver to check satisfiability. (Note that such formulas are purely existential—in variables **c** as well as in q copies of N.) With high probability, the first m for which the sampled formula is unsatisfiable will give a good enough estimate of $\mathsf{mc}(\varphi)$ and, by the reduction above, of $\mathsf{mc}(\varphi_{\mathsf{acc}})$.

Let us give some concrete values to support the intuition. We encode the number $c \in \{1, 2, 3\}$ in binary, as $c \equiv c_0 c_1$. We make $q = 12$ copies, and this will ensure that we will obtain the *exact* value of $\mathsf{mc}(\varphi_{\mathsf{acc}})$ by taking qth root of $\mathsf{mc}(\varphi)$, where φ is as above (for exact rather than approximate solution, a multiplicative gap of less than $3/2$ suffices in our setting). In reality, $\mathsf{mc}(\varphi_{\mathsf{acc}}) = 2$ and so $\mathsf{mc}(\varphi) = 2^{12}$, but we only know a priori that $\mathsf{mc}(\varphi_{\mathsf{acc}}) \in [0, 3]$ and $\mathsf{mc}(\varphi) \leq 3^{12}$. We iterate over the dimension m of the hash function and perform the SMT query (2) for each m. Using standard statistical techniques, we can reduce the error probability α by repeating each random experiment a sufficiently large number of times, r; in our case $r = 62$ leads to $\alpha = 0.01$. A typical run of our implementation is demonstrated in Table 3; for each m we show how many of the sampled formulas are satisfiable, and how many are not. The "Majority vote" column is used by our procedure to decide if the number of models is more than 2^m times a constant factor. From the table, our procedure will conclude that $\mathsf{mc}(\varphi)$ is between $0.17 \cdot 2^{12}$ and $11.66 \cdot 2^{12}$ with probability at least 0.99 (see Appendix for derivation of the constants 0.17 and 11.66). This gives us the interval [1.73, 2.45] for $\mathsf{mc}(\varphi_{\mathsf{acc}})$; since $\mathsf{mc}(\varphi_{\mathsf{acc}})$ is integer, we conclude that $\mathsf{mc}(\varphi_{\mathsf{acc}}) = 2$ with probability at least 0.99.

As mentioned above, the same technique will deliver us $\mathsf{mc}(\varphi_{\mathsf{term}}) = 3$ and hence, $\mathsf{val}(\mathrm{Switch}) = 2/3$.

5 Value estimation for probabilistic programs

In this section we show how our approach to #SMT applies to the *value problem* for probabilistic programs.

What are probabilistic programs?

Probabilistic models such as Bayesian networks, Markov chains, probabilistic guarded-command languages, and Markov decision processes have a rich history and form the modeling basis in many different domains (see, e.g., [16,22,38,45]). More recently, there has been a move toward integrating probabilistic modeling with "usual" programming languages [25,46]. Semantics and abstract interpretation for probabilistic programs with angelic and demonic non-determinism has been studied before [15,39,45,47], and we base our semantics on these works.

Probabilistic programming models extend "usual" nondeterministic programs with the ability to sample values from a distribution and condition the behavior of the programs based on observations [29]. Intuitively, probabilistic programs extend an imperative programming language like C with two constructs: a nondeterministic assignment to a variable from a range of values, and a probabilistic assignment that sets a variable to a random value sampled from a distribution. Designed as a modeling framework, probabilistic programs are typically treated as descriptions of probability distributions and not meant to be implemented and executed as usual programs.

Section summary

We consider a core *loop-free* imperative language extended with *probabilistic statements*, similarly to [52], and with *nondeterministic choice*. Under each given assignment to the probabilistic variables, a program accepts (rejects) if there is an execution path that is compatible with the observations and goes from the initial vertex to the accepting (resp., rejecting) vertex of its control flow automaton. Consider all possible outcomes of the probabilistic assignments in a program $\mathscr{P}$. Restrict attention to those that result in $\mathscr{P}$ reaching (nondeterministically) at least one of the accepting or rejecting vertices—such elementary outcomes form the set Term (for "termination"); only these scenarios are compatible with the observations. Similarly, some of these outcomes may result in the program reaching (again, nondeterministically) the accepting vertex—they form the set Accept. Note that the sets Term and Accept are events in a probability space; define $\mathsf{val}(\mathscr{P})$, the *value* of $\mathscr{P}$, as the conditional probability $\Pr[\mathsf{Accept} \mid \mathsf{Term}]$, which is equal to the ratio $\frac{\Pr[\mathsf{Accept}]}{\Pr[\mathsf{Term}]}$ as $\mathsf{Accept} \subseteq \mathsf{Term}$. We assume that programs are well-formed in that $\Pr[\mathsf{Term}]$ is bounded away from 0.

Now consider a probabilistic program $\mathscr{P}$ *over a measured theory* $\mathscr{T}$, i.e., where the expressions and predicates come from $\mathscr{T}$. Associate a separate variable r with each probabilistic assignment in $\mathscr{P}$ and denote the corresponding distribution by $\mathsf{dist}(r)$. Let R be the set of all such variables r.

Proposition 2 *There exists a polynomial-time algorithm that, given a program $\mathscr{P}$ over $\mathscr{T}$, constructs logical formulas $\varphi_{\mathsf{acc}}(R)$ and $\varphi_{\mathsf{term}}(R)$ over $\mathscr{T}$ such that* $\mathsf{Accept} = [\![\varphi_{\mathsf{acc}}]\!]$ *and* $\mathsf{Term} = [\![\varphi_{\mathsf{term}}]\!]$, *where each free variable $r \in R$ is interpreted over its domain with measure* $\mathsf{dist}(r)$. *Thus,* $\mathsf{val}(\mathscr{P}) = \mathsf{mc}(\varphi_{\mathsf{acc}})/\mathsf{mc}(\varphi_{\mathsf{term}})$.

Proposition 2 reduces the *value problem*—i.e., the problem of computing $\mathsf{val}(\mathscr{P})$—to model counting. This enables us to characterize the complexity of the value problem and solve this problem approximately using the hashing approach from Sect. 3. These results appear as Theorem 4 in Sect. 5.5 below.

In the remainder of this section we define the syntax (Sect. 5.1) and semantics (Sect. 5.2) of our programs and the value problem. By reducing this problem to #SMT (Sect. 5.5) we show an application of our approach to approximate model counting (an experimental evaluation is provided in Sect. 5.6). We also discuss modeling different kinds of nondeterminism: cooperative and adversarial (Sect. 5.3), and give an short overview of known probabilistic models subsumed by ours (Sect. 5.4).
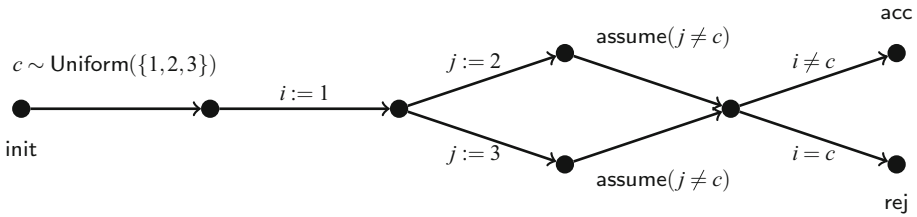
Fig. 1 CFA for the probabilistic program given as procedure 3: "Switch" strategy in Monty Hall problem

5.1 Syntax

A program has a set of variables $\mathscr{X}$, partitioned into Boolean, integer, and real-valued variables. We assume expressions are type correct, i.e., there are no conversions between variables of different types. The **basic statements** of a program are:

- skip (do nothing),
- deterministic assignments $x := e$,
- probabilistic assignments $x \sim \mathsf{Uniform}(a, b)$,
- assume statements $\mathsf{assume}(\varphi)$,

where e and φ come from an (unspecified) language of expressions and predicates, respectively.

The (deterministic) assignment and assume statements have the usual meaning: the deterministic assignment $x := e$ sets the value of the variable x to the value of the expression on the right-hand side, and $\mathsf{assume}(\varphi)$ continues execution only if the predicate is satisfied in the current state (i.e., it models observations used to condition a distribution). The probabilistic assignment operation $x \sim \mathsf{Uniform}(a, b)$ samples the uniform distribution over the range $[a, b]$ with constant parameters a, b and assigns the resulting value to the variable x. For example, for a real variable x, the statement $x \sim \mathsf{Uniform}(0, 1)$ draws a value uniformly at random from the segment $[0, 1]$, and for an integer variable y, the statement $y \sim \mathsf{Uniform}(0, 1)$ sets y to 0 or 1 with equal probability.

The **control flow** of a program is represented using directed acyclic graphs, called control flow automata (CFA), whose nodes represent program locations and whose edges are labeled with program statements. Let $\mathscr{S}$ denote the set of basic statements; then a *control flow automaton* (CFA) $\mathscr{P} = (\mathscr{X}, V, E, \mathsf{init}, \mathsf{acc}, \mathsf{rej})$ consists of a set of variables $\mathscr{X}$, a labeled, directed, acyclic graph (V, E), with $E \subseteq V \times \mathscr{S} \times V$, and three designated vertices init, acc, and rej in V called the *initial*, *accepting*, and *rejecting* vertices.

Figure 1 depicts the CFA for the probabilistic program shown in Procedure 3: "Switch" strategy in Monty Hall problem. The accept and reject statements from the procedure correspond to the acc and rej vertices of the CFA respectively.

We assume init has no incoming edges and acc and rej have no outgoing edges. We write $v \xrightarrow{s} v'$ if $(v, s, v') \in E$. We also assume programs are in static single assignment (SSA) form, that is, each variable is assigned at most once along any execution path. A program can be converted to SSA form using standard techniques [31,48].

Since control flow automata are acyclic, our programs do not have looping constructs. Loops can be accommodated in two different ways: by assuming that the user provides loop invariants [35], or by assuming an outer (statistical) procedure that selects a finite set of executions that is sufficient for the analysis up to a given confidence level [51,52]. In either

case, the core analysis problem reduces to analyzing finite-path unwindings of programs with loops, which is exactly what our model captures.

Although our syntax only allows uniform distributions, we can model some other distributions. For example, to simulate a Bernoulli random variable x that takes value 0 with probability p and 1 with probability $1 - p$, we write the following code:

$$X \sim \mathsf{Uniform}(0, 1);$$
$$\text{if } (X \leq p)\{x := 0; \} \quad \text{else } \{x := 1; \}$$

We can similarly encode uniform distributions with non-constant boundaries as well as (approximately encode) normal distributions (using repeated samples from uniform distributions and the central limit theorem). To encode uniform distributions with non-constant boundaries, we use assume conditioning: e.g., to simulate a random variable x that has distribution $\mathsf{Uniform}(-y^2, 1 + 2y)$ where $y \in [0, 10]$ is a previously assigned variable, we write the following code:

$$x \sim \mathsf{Uniform}(-100, 21);$$
$$\mathsf{assume}(-y * y \leq x \leq 1 + 2 * y);$$

The semantics of this conditioning is explained in the following subsection.

5.2 Semantics

The semantics of a probabilistic program is given as a superposition of nondeterministic programs, following [15,39]. Intuitively, when a probabilistic program runs, an oracle makes all random choices faced by the program along its execution up front. With these choices, the program reduces to a usual nondeterministic program.

We first provide some intuition behind our semantics. Let us partition the variables $\mathscr{X}$ of a program into random variables R (those assigned in a probabilistic assignment) and nondeterministic variables $N = \mathscr{X} \setminus R$ (the rest). (The partition is possible because programs are in static single assignment form.) We consider two events. The (normal) *termination* event (resp. the *acceptance* event) states that under a scenario ω for the random variables in R, there is an assignment to the variables in N such that the program execution under this choice of values reaches acc or rej (resp. reaches acc). The termination is "normal" in that all assumes are satisfied. Our semantics computes the conditional probability, under all scenarios, of the acceptance event given that the termination event occurred.

We now formalize the semantics. A *state* of a program is a pair $(v, \mathbf{x})$ of a control node $v \in V$ and a type-preserving assignment of values to all program variables in $\mathscr{X}$. Let Σ denote the set of all states and Σ^* the set of finite sequences over Σ.

Let $(\Omega, \mathscr{F}, \mathsf{Pr})$ be the probability space associated with probabilistic assignments in a program $\mathscr{P}$; elements of Ω will be called *scenarios*. The probabilistic semantics of $\mathscr{P}$, denoted $\langle\!\langle \mathscr{P} \rangle\!\rangle$, is a function from Ω to 2^{Σ^*}, mapping each scenario $\omega \in \Omega$ to a collection of maximal executions of the nondeterministic program obtained by fixing ω. It is defined with the help of an extension of $\langle\!\langle \cdot \rangle\!\rangle$ from programs to states, which, in turn, is defined inductively as follows:

- $(\mathsf{acc}, \mathbf{x}) \in \langle\!\langle \mathsf{acc} \rangle\!\rangle \omega$ and $(\mathsf{rej}, \mathbf{x}) \in \langle\!\langle \mathsf{rej} \rangle\!\rangle \omega$ for all $\mathbf{x}$;
- $(v, \mathbf{x})(v', \mathbf{x})\sigma \in \langle\!\langle v \rangle\!\rangle \omega$ if $v \xrightarrow{\mathsf{skip}} v'$ and $(v', \mathbf{x})\sigma \in \langle\!\langle v' \rangle\!\rangle \omega$;

- $(v, \mathbf{x})(v', \mathbf{x}')\sigma \in \langle\!\langle v \rangle\!\rangle \omega$ if $v \xrightarrow{x := e} v', \mathbf{x}' = \mathbf{x}[x := \mathsf{eval}(e)(\mathbf{x}, \omega)]$, and $(v', \mathbf{x})\sigma \in \langle\!\langle v' \rangle\!\rangle \omega$; similarly, if $v \xrightarrow{x \sim \mathsf{Uniform}(a,b)} v'$, we have $\mathbf{x}' = \mathbf{x}[x := c]$ where c is the value chosen for x in the scenario ω;
- $(v, \mathbf{x})(v', \mathbf{x})\sigma \in \langle\!\langle v \rangle\!\rangle \omega$ if $v \xrightarrow{\mathsf{assume}(\varphi)} v', \mathsf{eval}(\varphi)(\mathbf{x}, \omega) = \mathit{true}$, and $(v', \mathbf{x})\sigma \in \langle\!\langle v' \rangle\!\rangle \omega$.

Finally, define $\langle\!\langle \mathscr{P} \rangle\!\rangle \omega = \langle\!\langle \mathsf{init} \rangle\!\rangle \omega$. Here $\mathsf{eval}(e)(\mathbf{x}, \omega)$ (resp. $\mathsf{eval}(\varphi)(\mathbf{x}, \omega)$) denotes the value of the expression e (resp. predicate φ) taken in the scenario ω under the current assignment $\mathbf{x}$ of values to program variables, and $\mathbf{x}[x := c]$ is the assignment that maps variable x to the value c and agrees with $\mathbf{x}$ on all other variables.

Let $\Phi \subseteq \Sigma^*$ be a set of paths of a program $\mathscr{P}$. The probability that the run of $\mathscr{P}$ has a property Φ is defined as

$$\mathsf{Pr}\,[\text{run of } \mathscr{P} \text{ satisfies } \Phi] = \int_\Omega \mathbb{1}\big[\langle\!\langle \mathscr{P} \rangle\!\rangle \cap \Phi \neq \emptyset\big]\, d\mathsf{Pr}(\omega)$$

where $\mathbb{1}\big[\langle\!\langle \mathscr{P} \rangle\!\rangle \cap \Phi \neq \emptyset\big]$ denotes the indicator event that at least one execution path from $\langle\!\langle \mathscr{P} \rangle\!\rangle$ belongs to Φ. Specifically, let $\Phi_{\mathsf{acc}} \subseteq \Sigma^*$ be the set of all sequences that end in a state $(\mathsf{acc}, \mathbf{x})$ for some $\mathbf{x}$, and $\Phi_{\mathsf{term}} \subseteq \Sigma^*$ be the set of all sequences that end in either $(\mathsf{acc}, \mathbf{x})$ or $(\mathsf{rej}, \mathbf{x})$. We define the *termination* and *acceptance* events as

$$\mathsf{Term} = [\text{run of } \mathscr{P} \text{ satisfies } \Phi_{\mathsf{term}}],$$
$$\mathsf{Accept} = [\text{run of } \mathscr{P} \text{ satisfies } \Phi_{\mathsf{acc}}].$$

The *value* $\mathsf{val}(\mathscr{P})$ of a program $\mathscr{P}$ is defined as the conditional probability $\mathsf{Pr}[\mathsf{Accept} \mid \mathsf{Term}]$, which is equal to the ratio $\frac{\mathsf{Pr}[\mathsf{Accept}]}{\mathsf{Pr}[\mathsf{Term}]}$ as $\mathsf{Accept} \subseteq \mathsf{Term}$. Thus, the value of a program is the conditional probability

$$\mathsf{Pr}_\omega[\,\exists \mathbf{z} \,.\, \mathscr{P}(\omega, \mathbf{z}) \text{ reaches } \mathsf{acc} \mid \exists \mathbf{z} \,.\, \mathscr{P}(\omega, \mathbf{z}) \text{ reaches } \mathsf{acc} \text{ or } \mathsf{rej}\,].$$

For simplicity of exposition, we restrict attention to well-formed programs, for which $\mathsf{Pr}[\mathsf{Term}]$ is bounded away from 0. The *value problem* takes as input a program $\mathscr{P}$ and computes $\mathsf{val}(\mathscr{P})$.

Before we show in Sect. 5.5 how the value problem reduces to model counting, we first discuss the features and expressivity of our model of probabilistic programs. In Sect. 5.3 we discuss the semantics of nondeterminism and in Sect. 5.4 we relate our programming model to well-known probabilistic models.

5.3 Cooperative versus adversarial nondeterminism

Our semantics corresponds to a *cooperative* understanding of nondeterminism, in the following sense. For each individual scenario ω, the set $\langle\!\langle \mathscr{P} \rangle\!\rangle \omega$ can have one of the following four forms:

1. there are no paths to acc nor rej (for any assignment $\mathbf{z}$ for the nondeterministic variables in N),
2. there is a path to rej, but no paths to acc,
3. there is a path to acc, but no paths to rej,
4. there are paths to both acc and rej (under different assignments $\mathbf{z}, \mathbf{z}'$ for the nondeterministic variables).

The conditional probability measure

$$\mathsf{Pr}_\omega[\,\cdot \mid \mathsf{Term}\,] = \mathsf{Pr}_\omega[\,\cdot \mid \exists \mathbf{z} \,.\, \mathscr{P}(\mathbf{z}, \omega) \text{ reaches } \mathsf{acc} \text{ or } \mathsf{rej}\,]$$

restricts the attention to ω of the forms 2, 3, 4. Now our definition of Accept says that all ω of the form 4 are counted towards acceptance. The value of the program is accordingly defined as the (conditional) probability of options 3, 4.

In the Monty Hall problem in Sect. 4, this semantics worked as intended only because there are no scenarios ω of the form 4. However, a cooperative interpretation may not always be desirable. Imagine, for instance, that in a game, for some fixed strategy of the player all scenarios ω have the form 4, which means that the outcome of the game depends on the host's choice. Our semantics evaluates the strategy as perfect, with the value 1, although using the strategy may even lead to *losing* with probability 1 once nondeterminism is interpreted adversarially.

We can distinguish between semantics with cooperative and adversarial (also known as angelic and demonic) nondeterminism by defining the *upper* and *lower* values of a program by

$$\overline{\mathsf{val}}(\mathscr{P}) = \mathsf{Pr}_\omega[\,\exists \mathbf{z} . \, \mathscr{P}(\mathbf{z}, \omega) \text{ reaches acc} \mid \mathsf{Term}\,] \quad \text{and}$$
$$\underline{\mathsf{val}}(\mathscr{P}) = \mathsf{Pr}_\omega[\,\nexists \mathbf{z} . \, \mathscr{P}(\mathbf{z}, \omega) \text{ reaches rej} \mid \mathsf{Term}\,].$$

The upper value $\overline{\mathsf{val}}(\mathscr{P})$ coincides with $\mathsf{val}(\mathscr{P})$ as defined in Sect. 5.2, and the lower value $\underline{\mathsf{val}}(\mathscr{P})$ indeed corresponds to the adversarial interpretation of nondeterministic choice: only scenarios of the form 3 are counted towards acceptance, and scenarios of the form 2 and, most importantly, 4 towards rejection. Obviously, $\underline{\mathsf{val}}(\mathscr{P}) \leq \overline{\mathsf{val}}(\mathscr{P})$, with equality if and only if the set of scenarios of the form 4 has (conditional) measure zero, as in Sect. 4.

Observe now that the problem of computing $\underline{\mathsf{val}}(\mathscr{P})$ reduces to the problem of computing $\overline{\mathsf{val}}(\mathscr{P})$: the reason for that is the equality

$$\underline{\mathsf{val}}(\mathscr{P}) = 1 - \overline{\mathsf{val}}(\mathscr{P}^*),$$

where for a program $\mathscr{P} = (\mathscr{X}, V, E, \mathsf{init}, \mathsf{acc}, \mathsf{rej})$ we define the corresponding *dual* program $\mathscr{P}^* = (\mathscr{X}, V, E, \mathsf{init}, \mathsf{rej}, \mathsf{acc})$. The details are easily checked.

Note that the type of nondeterminism is interpreted at the level of programs and not on the level of individual statements. Mixing statements with different type of nondeterminism is equivalent to considering probabilistic programs with alternation, which raises the complexity of the value problem: even non-probabilistic loop-free programs with two kinds of nondeterminism on the per-statement basis are **PSPACE**-hard to analyze.

Also note that our semantics resolves the nondeterminism after the probabilistic choice. This indicates that the nondeterministic choice can "look in the future." For example, consider a program that first chooses a bit x nondeterministically, then chooses a bit r uniformly at random, and then accepts if $x = r$ and rejects if $x \neq r$. Under our semantics, the program always accepts: there is a way for the nondeterministic choice to guess correctly. This feature of our model can be undesirable in certain cases: in formal approaches to security, for example, a scheduler that uses the power to look into the future when resolving nondeterminism is unrealistic; its existence, however, can lead to classifying secure protocols as insecure [12].

We now briefly discuss the *synthesis* question in which the nondeterminism is resolved before the probabilistic choice. A more general setting, where nondeterministic and probabilistic choice alternate, is **PSPACE**-complete [49].

Verification vs. synthesis. In this paper, we consider the *verification* question: given a probability space over random inputs, the value of the program is the conditional probability of acceptance, given the program terminates. As stated above, nondeterminism is resolved after probabilistic choice. In decision making under uncertainty, one is also interested in the *synthesis* question: is there a strategy (a way to resolve nondeterministic choices) such that the

resulting probabilistic program achieves a certain value. That is, the value synthesis problem asks to compute, for a given $p \geq 0$, if

$$\exists \mathbf{z} . \mathsf{val}(\mathscr{P}(\cdot, \mathbf{z})) \geq p.$$

The complexity of the synthesis problem is, in general, harder than that of the verification problem. The precise complexity characterization is $\mathbf{NP^{\#P}}$, the class of problems solvable by a nondeterministic polynomial-time Turing machine with access to a $\mathbf{\#P}$ oracle. Intuitively, the $\mathbf{NP}$-computation guesses the values of variables in $\mathbf{z}$, and asks a $\mathbf{\#P}$ oracle to resolve the resulting verification problem. Moreover, the problem is $\mathbf{NP^{\#P}}$-hard already for Boolean programs, by using a reduction from E-MAJSAT, a canonical $\mathbf{NP^{\#P}}$-complete problem.

Proposition 3 *Synthesis for probabilistic programs over* IA *and* RA *is* $\mathbf{NP^{\#P}}$*-complete.*

In general, one can study models with arbitrary interleavings of probabilistic and nondeterministic choice. For such models, the static analysis problem reduces to *stochastic SMT*, which is known to be **PSPACE**-complete [49].

We leave the study of "approximate synthesis" techniques for the future.

5.4 Related models

Our programming model captures (finite-path) behaviors of several different probabilistic models that have been considered before, including the programming models studied recently [31,51,52]. In contrast to models that only capture probabilistic behavior, such as (dynamic) Bayesian networks, we additionally allow nondeterministic choices. We show a few additional probabilistic models that can be expressed as programs.

(Dynamic) Bayesian networks [16,38]. A Bayesian network over V is a directed acyclic graph $G = (V, E)$, where each vertex $v \in V$ represents a random variable and each edge $(u, v) \in E$ represents a direct dependence of the random variable v on the random variable u. Each node v is labeled with a conditional probability distribution: that of v conditioned on the values of the random variables $\{u \mid (u, v) \in E\}$. A Bayesian network can be represented as a probabilistic program that encodes the conditional probability distribution for each node using a sequence of conditionals and the Bernoulli distribution.

A temporal graphical model is a probabilistic model for states that evolve over time. In such a model, there is a set of random variables $X^{(t)}$ indexed by a time t, and the distribution of a variable $v^{(t+1)} \in X^{(t+1)}$ is given by a conditional probability distribution over the values of random variables in $X^{(t)}$. One example of a temporal model is a dynamic Bayesian network. A dynamic Bayesian network consists of a pair $\langle \mathscr{B}_0, \mathscr{B}_\rightarrow \rangle$, where $\mathscr{B}_0$ is a Bayesian network over X that gives the initial probability distribution and $\mathscr{B}_\rightarrow$ is a Bayesian network over $X \cup X'$, such that only variables in X' have incoming edges (or conditional probability distributions associated with them). Here, X' denotes a fresh copy of variables in X. The network $\mathscr{B}_\rightarrow$ defines the distribution of variables in X' given values of variables in X. The distribution of $X^{(t+1)}$ is obtained from $X^{(t)}$ according to $\mathscr{B}_\rightarrow$. Given a time horizon T, a dynamic Bayesian network is unrolled for T steps in the obvious way: by first running $\mathscr{B}_0$ and running T copies of $\mathscr{B}_\rightarrow$ in sequence. Again, for any T, such an unrolling can be expressed by a probabilistic program. Dynamic Bayesian networks subsume several other models, such as hidden Markov models and linear-Gaussian dynamical systems.

Influence diagrams [38]. Influence diagrams are a common model to study decision making under uncertainty. They extend Bayesian networks with nondeterministic variables under the

control of an agent. An influence diagram is a directed acyclic graph $G = (V, E)$, where the nodes are partitioned into random variables V_R, decision variables V_D, and utility variables V_U. Each variable in $V_R \cup V_D$ has a finite domain. The incoming edges to variables in V_R model direct dependencies as in a Bayesian network, and the distribution of a random variable is given by a distribution conditioned on the values of all incoming variables. Decision variables are chosen by an adversary. Utility variables have no outgoing edges and model the utility derived by an agent under a given scenario and choice of decisions. The value of a utility variable is derived as a deterministic function of values of incoming edges. For a given scenario of random variables and choice of decision variables, the value of the diagram is the sum of all utility variables. By comparing the utility to a constant, we can reduce computing a bound on the utility to the value problem. Influence diagrams subsume models such as Markov decision processes with adversarial nondeterminism.[5] The Monty Hall problem in Sect. 4 represents an example of an influence diagram.

Probabilistic guarded command languages (pGCL) [45]. pGCLs extend Dijkstra's guarded command language with a probabilistic choice operation. They have been used to model communication protocols involving randomization. Our programs can model bounded unrollings of pGCLs, and the value problem can be used to check probabilistic assertions of loop-free pGCL code. This is the core problem in the deductive verification of pGCLs [35].

5.5 From value estimation to model counting

We show a reduction from the value problem for a probabilistic program to a model counting problem. First, we define a *symbolic semantics* of programs.

Let $\mathscr{P} = (\mathscr{X}, V, E, \mathsf{init}, \mathsf{acc}, \mathsf{rej})$ be a program in SSA form. Let $R = \{x \in \mathscr{X} \mid x \sim \mathsf{Uniform}(a, b)$ is a statement in $\mathscr{P}\}$. For each variable $r \in R$, we write $\mathsf{dist}(r)$ for the (unique) distribution $\mathsf{Uniform}(a, b)$ such that $r \sim \mathsf{Uniform}(a, b)$ appears in the program.

Let $B_V = \{b_v \mid v \in V\}$ be a set of fresh Boolean variables. We associate the following *verification condition* $\mathsf{vc}(\mathscr{P})$ with the program $\mathscr{P}$:

$$\bigwedge_{v \in V} \left[b_v \Rightarrow \left(\bigvee_{(v', s, v) \in E} b_{v'} \wedge \Psi(s) \right) \right] \wedge b_{\mathsf{init}}$$

where $\Psi(s)$ is defined as follows: $\Psi(\mathsf{skip})$ is *true*, $\Psi(x := e)$ is $x = e$, $\Psi(x \sim \mathsf{Uniform}(a, b))$ is *true*, and $\Psi(\mathsf{assume}(\varphi))$ is φ.

Intuitively, the variable b_v encodes "node v is visited along the current execution." The constraints encode that in order for v to be visited, the execution must traverse an edge (v', s, v) and update the state according to s. The predicate $\Psi(s)$ describes the effect of the execution on the state.

The predicates $\Psi(s)$ do not add an additional constraint for probabilistic assignments because we account for such assignments separately as follows. Define formulas

$$\varphi_{\mathsf{acc}} = \exists B_V \, \exists \mathscr{X} \backslash R \, . \, \mathsf{vc}(\mathscr{P}) \wedge b_{\mathsf{acc}}, \quad \text{and}$$
$$\varphi_{\mathsf{term}} = \exists B_V \, \exists \mathscr{X} \backslash R \, . \, \mathsf{vc}(\mathscr{P}) \wedge (b_{\mathsf{acc}} \vee b_{\mathsf{rej}}).$$

---

[5] Strictly speaking, MDPs and influence diagrams, where nondeterminism is resolved adversarially, are modeled by the *duals* of our programs (as defined in Sect. 5.3). Thus, the verification problem asks what is the worst case effect of the environment.

Note that φ_{acc} and φ_{term} are over the free variables R; if the program $\mathscr{P}$ is *over a measured theory* $\mathscr{T}$, i.e., its expressions and predicates come from $\mathscr{T}$, then φ_{acc} and φ_{term} are formulas in $\mathscr{T}$.

Theorem 3 *(cf. Proposition 2) For a program $\mathscr{P}$, we have* $\mathsf{Accept} = [\![\varphi_{\text{acc}}]\!]$ *and* $\mathsf{Term} = [\![\varphi_{\text{term}}]\!]$, *where each free variable $r \in R$ is interpreted over its domain with measure* $\mathsf{dist}(r)$. *Thus,* $\mathsf{val}(\mathscr{P}) = \mathsf{mc}(\varphi_{\text{acc}})/\mathsf{mc}(\varphi_{\text{term}})$.

Theorem 3 reduces the value estimation question to model counting. Note that our reasoning is program-level as opposed to path-level: in contrast to other techniques (see, e.g., [23,52]), our analysis makes only two #SMT queries and not one query per path through the program. While this results in more complex satisfiability queries, the burden of path enumeration is shifted from the analysis procedure to the underlying SMT solver.

For the theories of integer and linear real arithmetic, Theorem 3 gives us a #**P** upper bound on the complexity of the value problem. On the other hand, the value problem is #**P**-hard, as it easily encodes #SAT. Indeed, given an instance of #SAT (a Boolean formula in conjunctive normal form), consider a program that picks the Boolean variables uniformly at random, and accepts iff all the clauses are satisfied. The number of satisfying assignments to the formula is obtained from the probability of reaching the accept vertex. Finally, since the model counting problem can be approximated using a polynomial-time randomized algorithm with an SMT oracle, we also get an algorithm for approximate value estimation.

Theorem 4 *(complexity of the value problem)*

1. *The value problem for loop-free probabilistic programs (over* IA *and* RA*) is #**P**-complete. The problem is #**P**-hard even for programs with only Boolean variables.*
2. *The value problem for loop-free probabilistic programs over* IA *can be approximated with a multiplicative error by a polynomial-time randomized algorithm that has oracle access to satisfiability of formulas in* IA.
3. *The value problem for loop-free probabilistic programs over* RA *can be approximated with an additive error by a polynomial-time randomized algorithm that has oracle access to satisfiability of formulas in* IA + RA.

Remark The core of our value estimation algorithms is a procedure to estimate the number of models of a formula in a given theory (approximate #SMT). An alternative approach to the value problem—and, similarly, to model counting—would perform Monte Carlo simulation. It can easily handle complicated probability distributions for which there is limited symbolic reasoning available. However, to achieve good performance, Monte Carlo often depends on heuristics that sacrifice theoretical guarantees. In contrast, while using "for free" successful heuristics that are already implemented in off-the-shelf SMT solvers to search the state space, our approach still preserves the theoretical guarantees.

There are simple instances in which Monte Carlo simulation must be run for an exponential number of steps before providing a non-trivial answer [33]. Consider the case when the probability in question, p, is very low and the required precision is a constant multiple of p. In such a case, model counts are small and so there are only a few queries to the SMT solver. On the other hand, for Monte Carlo simulation, Chernoff bound arguments would suggest running the program $\Omega(\frac{1}{p})$ times.

While our SMT-based techniques can also require exponential time within the SMT solver in the worst case, experience with SMT-based verification of deterministic programs suggests that SMT solvers can be quite effective in symbolically searching large state spaces in

reasonable time. An illustrative analogy is that the relation between Monte Carlo techniques and SMT-based techniques resembles that between enumerative techniques and symbolic techniques in deterministic model checking: while in the worst case, both must enumerate all potential behaviors, symbolic search often empirically scales to larger state spaces.

In conclusion, Monte Carlo sampling will easily outperform hashing techniques in a host of "regular" settings, i.e., where the probability of termination is non-vanishing. "Singular" settings where this probability is close to zero—as, for instance, the formula from Example 4 in Sect. 3.2— will be beyond the reach of Monte Carlo even for generating a single positive sample (path), let alone for providing a confidence interval sufficient for multiplicative approximation of the value of the program. Indeed, since the success probability decreases exponentially with the number of bits, the number of Monte Carlo simulations required increases exponentially. The hashing approach that we explore deals with such settings easily, so the two techniques are, in fact, complementary to each other.

5.6 Evaluation

We have implemented the algorithm from Sect. 3.2 in C++ on top of the SMT solver Z3 [17][6]. The SMT solver is used unmodified, with default settings.

Examples

We evaluate our techniques on five examples. The first two are probabilistic programs that use nondeterminism. The remaining examples are Bayesian networks encoded in our language.

The Monty Hall problem [53] For the example from Sect. 4 we compute the probability of success of the switching strategy.

The three prisoners problem. Our second example is a problem that appeared in Martin Gardner's "Mathematical Games" column in the Scientific American in 1959. There, one of three prisoners (1, 2, and 3), who are sentenced to death, is randomly pardoned. The guard gives prisoner 1 the following information: If 2 is pardoned, he gives 1 the name of 3. If 3 is pardoned, he gives him the name of 2. If 1 is pardoned, he flips a coin to decide whether to name 2 or 3. Provided that the guard tells prisoner 1 that prisoner 2 is to be executed, determine what is prisoner 1's chance to be pardoned?

Pearl's burglar alarm; grass model. These two examples are classical Bayesian networks from the literature. Pearl's burglar alarm example is as given in [29, Figure 15]; the grass model is taken from [36, Figure 1].

Kidney disease eGFR sensitivity estimation. The last example is a probabilistic model of a medical diagnostics system with noisy inputs. We considered the program given in [29, Figure 11] using a simplified model of the input distributions. In our setting, we approximate the original lognormal distribution (the logarithm of the patient's creatinine level) by drawing its value uniformly from the set $\{-0.16, -0.09, -0.08, 0, 0.08, 0.09, 0.16, 0.17\}$, regardless of the patient's gender, and we draw the patient's age uniformly from the interval [30, 80]. The patient's gender and ethnicity are distributed in the same way as described in [52].

---

[6] More specifically, using version Z3 4.4.0.

Table 4 Input and runtime parameters

Example	Free	Atoms	ε	α	a	k'
Monty Hall (1)	1	5	0.2	0.01	13	10
Three prisoners (2)	2	6	0.2	0.01	27	12
Alarm (3)	4	8	0.5	0.1	19	56
Grass model (4)	6	8	0.5	0.1	19	48
Sensitivity est. (5)	8	63	0.5	0.1	19	66

Free, number of free (probabilistic) variables in the input formula; Atoms, number of atomic arithmetic predicates in the input formula; ε, parameter in the multiplicative approximation factor $(1 + \varepsilon)$; α, maximum error probability; a, the SMT enumeration threshold (number of models the SMT solver checks for); k', number of binary variables in the formula given to the solver

Table 5 Running time of the tool

Example	m_{acc}	m_{term}	Time (s) for φ_{acc}	Time (s) for φ_{term}
Monty Hall (1)	2	5	0.27	0.89
Three prisoners (2)	0	2	0.01	0.73
Alarm (3)	36	49	121.94	76.34
Grass model (4)	34	35	54.86	50.61
Sensitivity est. (5)	56	57	250.69	223.56

$m_{\text{acc}}, m_{\text{term}}$, maximal hash sizes for $\varphi_{\text{acc}}, \varphi_{\text{term}}$, respectively

Results

For each program $\mathscr{P}$, we used our tool to estimate the model count of the formulas φ_{acc} and φ_{term}; the value $\mathsf{val}(\mathscr{P})$ of the program is approximated by $v_{\text{acc}}/v_{\text{term}}$, where v_{acc} and v_{term} are the approximate model counts computed by our tool. Table 4 shows input and runtime parameters for the considered examples. The approximation factor ε, the bound α on the error probability, and the enumeration limit a for the SMT solver are provided by the user. For examples (1) and (2), we choose ε to be 0.2, while for the remaining examples we take 0.5. The chosen value of ε has impact on the number of copies q of the formula that we construct, an thus on the number k' of binary variables in the formula given to the solver. Furthermore, the more satisfying assignments a formula has, the larger dimension m of the hash function is reached during the run. Table 5 shows m_{acc} and m_{term}: the maximal values of m reached during the runs on φ_{acc} and φ_{term}; it also shows the time (in seconds) our tool takes to compute v_{acc} and v_{term}. It might seem strange that for examples (3), (4) and (5) the time it takes to compute v_{acc} is larger than that for v_{term}, despite that the set of paths satisfying φ_{acc} is a subset of φ_{term}. While, as expected, we have $m_{\text{acc}} < m_{\text{term}}$, the calls to the SMT solver for φ_{term} take less time than those for φ_{acc}.

While our technique can solve these small instances in reasonable time, there remains much room for improvement. Although SAT solvers can scale to large instances, it is well known that even a small number of XOR constraints can quickly exceed the capabilities of state-of-the-art solvers [30,57,60]. Since for each m we add m parity constraints to the formula, we run into the SAT bottleneck: computing an approximation of $\mathsf{mc}(\varphi_{\text{acc}})$ for example (4) with $\varepsilon = 0.3$ results in running time of several hours. (At the same time, exact counting by enumerating satisfying assignments is not a feasible alternative either: for the formula φ_{acc} in example (4), which has more than 400 000 of them, performing

this task naively with Z3 also took several hours.) Our current implementation pre-solves the system of XOR constraints before passing them to Z3, which somewhat improves the performance; however, the efficiency of the hashing approach can benefit greatly from better handling of XOR constraints in the SMT solver. For example, a SAT solver that deals with XOR constraints efficiently —such as CryptoMiniSat [55,56]— can scale to over a thousand variables [8,9,28]; incorporating such a SAT solver within Z3 remains a task for the future. (Needless to say, other families of pairwise independent hash functions can be used instead of XOR constraints, but essentially all of them seem to use arithmetic modulo p for $p \geq 2$, which appears hard for theory solvers.)

The scalability needs improvement also in the continuous case, where our discretization procedure introduces a large number of discrete variables. For instance, a more realistic model of example (5) would be one in which the logarithm of the creatinine level is modeled as a continuous random variable. This would result, after discretization, in formulas with hundreds of Boolean variables, which appears to be beyond the limit of Z3's XOR reasoning.

6 Concluding remarks

Static reasoning questions for probabilistic programs [29,31,52], as well as quantitative and probabilistic analysis of software [6,23,24,42], have received a lot of recent attention. There are two predominant approaches to these questions. The first one is to perform Monte Carlo sampling of the program [6,7,42,51,52]. To improve performance, such methods use sophisticated heuristics and variance reduction techniques, such as stratified sampling in [6, 52]. The second approach is based on reduction to model counting [23,24,43,44], either using off-the-shelf #SMT solvers or developing #SMT procedures on top of existing tools. Another recent approach is based on data flow analysis [14]. Our work introduces a new dimension of approximation to this area: we reduce program analysis to #SMT, but carry out a randomized approximation procedure for the count. In contrast to previous techniques, our analysis is performed at the program level and not at the path level: the entire analysis makes only two queries to a #SMT oracle (not one query per path through the program). Analysis at the path level requires enumeration of the program-paths, whose number can be exponential in the length of the program. Our approach shifts this enumeration to the SMT oracle. It avoids the need for implementing complex heuristics for efficient path enumeration at the price of harder SMT queries, thus relying on the efficiency of SMT solvers.

By known connections between counting and uniform generation [3,34], our techniques can be adapted to generate (approximately) uniform random samples from the set of models of a formula in IA or RA. Uniform generation from Boolean formulas using hashing techniques was recently implemented and evaluated in the context of constrained random testing of hardware [8,9]. We extend this technique to the SMT setting, which was left as a future direction in [9] (previously known methods for counting integral points of polytopes [2,24] do not generalize to the nonlinear theory IA).

Further directions

Scalability. An extension of the presented techniques may be desirable to cope with larger instances of #SMT. As argued in Sect. 5.6, incorporating XOR-aware reasoning into an SMT solver can be an important step in this direction.

Theories. Similar techniques apply to theories other than IA and RA. For example, our algorithm can be extended to an appropriate fragment of the combined theory of string constraints and integer arithmetic. While SMT solvers can handle this theory (using heuristics), it would be nontrivial to design a model counting procedure using the previously known approach based on generating functions [43].

Distributions. Although the syntax of our probabilistic programs supports only Uniform, it is easy to simulate other distributions: Bernoulli, uniform with non-constant endpoints, (approximation of) normal. This, however, will not scale well, so future work may incorporate non-uniform distributions as a basic primitive. (An important special case covers weighted model counting in SAT, for which a novel extension of the hashing approach was recently proposed [8] and, by the time the present paper was submitted, also studied in the context of SMT [4].)

Applications. A natural application of the uniform generation technique in the SMT setting would be a procedure that generates program behaviors uniformly at random from the space of possible behaviors. (For the model we studied, program behaviors are trees: the branching comes from nondeterministic choice, and the random variables are sampled from their respective distributions.)

Appendix: Technical proofs

In this section we fill in the details in the proof of Theorem 1, continuing Sect. 3.2 and thus proving correctness of Algorithm 1; however, to simplify notation, we write n instead of k' to denote the total number of Boolean variables. As the entire analysis is essentially Boolean, we build on a previous exposition of the topic due to Trevisan [59]. We pay much more attention to the precise choice of parameters, though; we assume that the SMT enumeration threshold $a \geq 1$, the approximation parameter $\varepsilon > 0$, and the upper bound on the probability of bad estimate $\alpha \in (0; 1)$ are given as input. In Appendix A.2 we show how to choose:

- q, the number of copies of the formula, see equation (9), and
- v, the output value of the algorithm (during its run), see equation (10).

In Appendix A.3 we show

- p, the initial enumeration threshold (number of models that are sought before the algorithm enters the main loop), see equation (11),
- m^*, the largest possible dimension of the hash, see equation (12), and
- r, the number of calls to the "Estimate" oracle ($\mathscr{E}$), see equation (13).

Before this, in Appendix A.1, we show how to choose internal parameters (equation (6)) so as to establish key properties of the oracle $\mathscr{E}$; the choice of parameters in the following Appendix A.2 and A.3 relies on these properties.

A.1 The "estimate" oracle from Sect. 3.2

We use a simple form of the *Leftover Hash Lemma*. This lemma was originally proved by Impagliazzo, Levin, and Luby [32]; we use a formulation due to Trevisan [59]. In brief, the lemma establishes the following property: For any sufficiently large set $S \subseteq \{0, 1\}^n$, the number of elements of S that are mapped to a particular image, 0^m, by a random hash function h from an appropriate family $\mathscr{H}$ does not deviate a lot from its expected value, $|S|/2^m$.

Lemma 1 *Let $\mathscr{H}$ be a family of pairwise independent hash functions $h\colon \{0, 1\}^n \to \{0, 1\}^m$. Let $S \subseteq \{0, 1\}^n$ be such that $|S| \geq 4/\rho^2 \cdot 2^m$. For $h \in \mathscr{H}$, let ξ be the cardinality of the set $\{w \in S\colon h(w) = 0^m\}$. Then*

$$\Pr\left[\left|\xi - \frac{|S|}{2^m}\right| \geq \rho \cdot \frac{|S|}{2^m}\right] \leq \frac{1}{4}.$$

We now show how to implement the "Estimate" oracle $\mathscr{E}$. Recall that its goal, roughly speaking, is to answer questions of the form

$$\text{Does the formula } \Psi \text{ have at least } N = 2^m \text{ models?} \tag{3}$$

Let a be a positive integer parameter, to be chosen arbitrarily. Our oracle $\mathscr{E}$ will rely, in turn, on an SMT oracle (solver) for the underlying theory (IA) and will post queries of the form

$$\text{Does the formula } \Psi_h := (\Psi \wedge (h = 0)) \text{ have at least } a \text{ models?} \tag{4}$$

where h is a hash function $h\colon \{0, 1\}^n \to \{0, 1\}^m$ with m chosen in an appropriate way. Instead of answering questions (3) exactly, our oracle $\mathscr{E}$ will have a blind spot. Let $\mathsf{mc}(\Psi)$ be the number of models of Ψ; for some parameters $g < G$ and for any sufficiently large m, we ensure that the following properties hold: (I) if $\mathsf{mc}(\Psi) < g \cdot 2^m$, then $\mathscr{E}$ returns "no" with high probability; (II) if $\mathsf{mc}(\Psi) > G \cdot 2^m$, then $\mathscr{E}$ returns "yes" with high probability. The blind spot is the intermediate case, $g \cdot 2^m \leq \mathsf{mc}(\Psi) \leq G \cdot 2^m$: the oracle $\mathscr{E}$ can answer "yes" or "no" in an arbitrary way. The entire implementation of the oracle will be very simple: it will pick h at random from $\mathscr{H}$, ask the question (4) for the obtained formula $\Psi_h = (\Psi \wedge (h = 0))$ and repeat the answer—yes or no—of the underlying SMT oracle.

Let us now proceed to proofs of properties (I) and (II).

Claim 1 *Let $x > 0$ be a real number such that $g = 4/x^2$ and $a = (1 + x)\, g$. Suppose $\mathsf{mc}(\Psi) \leq g \cdot 2^m$; then $\Pr[\mathscr{E} = \text{"no"}] \geq 3/4$.*

Proof Denote $\theta = \mathsf{mc}(\Psi)/2^m \leq g$ and pick any formula $\overline{\Psi}$ such that, first, $[\![\Psi]\!] \subseteq [\![\overline{\Psi}]\!]$ and, second, $\mathsf{mc}(\overline{\Psi}) = g \cdot 2^m$. Write $S = [\![\overline{\Psi}]\!]$ and, as above, let ξ be the cardinality of the set $\{w \in S\colon h(w) = 0^m\}$; note that $\xi = \mathsf{mc}(\overline{\Psi}_h)$ where $\overline{\Psi}_h = \overline{\Psi} \wedge (h = 0)$. Observe that $\Pr[\mathscr{E} = \text{"yes"}]$ is equal to

$$\Pr[\mathsf{mc}(\Psi_h) - g \geq a - g] \leq \Pr[\mathsf{mc}(\overline{\Psi}_h) - g \geq a - g]$$
$$= \Pr[\xi - g \geq x \cdot g] \leq \Pr[|\xi - g| \geq x \cdot g] \leq 1/4,$$

where the last inequality follows from Lemma 1 with $\rho = x$, since $|S| = \mathsf{mc}(\overline{\Psi}) = g \cdot 2^m = 4/x^2 \cdot 2^m$. □

Claim 2 *Let $y > 0$ be a real number such that $G = 4/y^2$ and $a = (1 - y)\, G$. Suppose $\mathsf{mc}(\Psi) \geq G \cdot 2^m$; then $\Pr[\mathscr{E} = \text{"yes"}] \geq 3/4$.*

Proof As in Claim 1, denote $\theta = \mathsf{mc}(\Psi)/2^m \geq G$. Now pick $S = [\![\Psi]\!]$ and let ξ again be the cardinality of the set $\{w \in S\colon h(w) = 0^m\}$; we have $\xi = \mathsf{mc}(\Psi_h)$. Observe that $\Pr[\mathscr{E} = \text{"yes"}]$ is equal to

$$\Pr[\mathsf{mc}(\Psi_h) - \theta \geq a - \theta] \geq \Pr[\mathsf{mc}(\Psi_h) - \theta \geq (1 - y) \cdot G - G] = \Pr[\xi - \theta \geq -y \cdot G] \geq$$
$$\Pr[|\xi - \theta| \leq y \cdot G] \geq 3/4,$$

where the last inequality again follows from Lemma 1, now with $\rho = y$ and $|S| = \mathsf{mc}(\Psi) = \theta \cdot 2^m \geq G \cdot 2^m = 4/y^2 \cdot 2^m$. □

Let us match the parameter settings from Claims 1 and 2. We have

$$g(x) = 4/x^2, \qquad\qquad G(y) = 4/y^2,$$
$$a(x) = (1 + x) \cdot 4/x^2, \qquad\qquad a(y) = (1 - y) \cdot 4/y^2.$$

Needless to say, the following equality needs to be satisfied:

$$a = a(x) = a(y). \tag{5}$$

The multiplicative gap between G and g is $B = G(y)/g(x) = (x/y)^2$, i.e., $B = \lambda^2$ for $\lambda = x/y$. Suppose $\lambda \geq 1$ is fixed; then Eq. (5) gives us

$$\frac{1 + \lambda y}{\lambda^2 y^2} = \frac{1 - y}{y^2},$$
$$1 + \lambda y = \lambda^2 - \lambda^2 y,$$
$$(\lambda^2 + \lambda)y = (\lambda^2 - 1),$$
$$y = \frac{\lambda - 1}{\lambda} = 1 - \frac{1}{\lambda} \quad \text{and}$$
$$a = a(\lambda) = \frac{1}{\lambda} \cdot 4 \cdot \frac{\lambda^2}{(\lambda - 1)^2} = \frac{4\lambda}{(\lambda - 1)^2}.$$

Given an integer $a \geq 1$, how big a gap $B(a)$ does it correspond to? Rewrite the equation $a(\lambda) = a$ as $(\lambda - 1)^2 \cdot a = 4\lambda$ and further as

$$\lambda^2 \cdot a - \lambda \cdot (2a + 4) + a = 0.$$

Both roots of this quadratic equation are real, but only the greater one is ≥ 1; it is given by the formula

$$\lambda(a) = \frac{a + 2 + 2\sqrt{a + 1}}{a} = \frac{(\sqrt{a + 1} + 1)^2}{a} \quad \text{and corresponds to}$$
$$y(a) = \frac{\lambda(a) - 1}{\lambda(a)} = \frac{2 + 2\sqrt{a + 1}}{a + 2 + 2\sqrt{a + 1}} = \frac{2}{\sqrt{a + 1} + 1} \quad \text{and}$$
$$x(a) = \lambda(a) \cdot y(a) = \lambda(a) - 1 = \frac{2(\sqrt{a + 1} + 1)}{a}.$$

Finally,

$$g = g(x) = \frac{a^2}{(\sqrt{a + 1} + 1)^2} = (\sqrt{a + 1} - 1)^2,$$
$$G = G(y) = (\sqrt{a + 1} + 1)^2, \quad \text{and} \tag{6}$$
$$B = \lambda^2 = \frac{(\sqrt{a + 1} + 1)^4}{a^2} = \left(\frac{\sqrt{a + 1} + 1}{\sqrt{a + 1} - 1}\right)^2.$$

To sum up, fixing $a \geq 1$ for SMT queries (4) leads to the multiplicative blind spot of "size" B and constants g and G defined in Equation (6); we will use these parameters in the following subsections.

A.2 Copies of the formula and return value of the algorithm

Recall that the formula Ψ that we use throughout the algorithm is of the following form:

$$\Psi = \psi_q = \psi^{(1)} \wedge \psi^{(2)} \wedge \cdots \wedge \psi^{(q)}$$

where $q \geq 1$ is a natural parameter and formulas $\psi^{(i)}$, $1 \leq i \leq q$, are copies of ψ where all variables are replaced by fresh copies. In total, Ψ has q times as many variables as ψ, and $\mathsf{mc}(\Psi) = \mathsf{mc}(\psi)^q$.

We now describe how the parameter q is chosen. Recall that Algorithm 1 calls the "Estimate" oracle $\mathscr{E}$ with $m = 1, 2, \ldots$ (we ignore the majority vote machinery for now; the reader can safely assume $r = 1$ as the reasoning in the general case is the same). Suppose first several calls to $\mathscr{E}$ result in "yes" answers, and let m be the first dimension of the hash that corresponds to a "no". We can now rule out (here and below—with high probability) the case that $\mathsf{mc}(\Psi) \geq G \cdot 2^m$; similarly, it is unlikely that $\mathsf{mc}(\Psi) \leq g \cdot 2^{m-1}$. We should conclude, therefore, that

$$g/2 \cdot 2^m < \mathsf{mc}(\Psi) < G \cdot 2^m \tag{7}$$

with high probability. Now, the task of the overall algorithm is to return a value v that lies in the segment $((1+\varepsilon)^{-1} \cdot \mathsf{mc}(\psi); (1+\varepsilon) \cdot \mathsf{mc}(\psi))$; in other words, v^q—which is an estimate of $\mathsf{mc}(\Psi)$—should satisfy the condition

$$(1+\varepsilon)^{-q} \cdot \mathsf{mc}(\Psi) \leq v^q \leq (1+\varepsilon)^q \cdot \mathsf{mc}(\Psi). \tag{8}$$

We now align the segments defined in equations (7) and (8) above. First, observe that the ratios of the right and left endpoints for each of these segments are $2G/g = 2B$ and $(1+\varepsilon)^{2q}$, respectively; recall that $B = G/g$ is given by equation (6). As our goal is thus to ensure that $(1+\varepsilon)^{2q} \geq 2B$, we choose

$$q = \left\lceil \frac{1 + \log B}{2 \log(1+\varepsilon)} \right\rceil. \tag{9}$$

Second, our best estimate for $\mathsf{mc}(\Psi)$ is, accordingly, the geometric mean of the left and right endpoints of the segment in (7); in other words, the best estimate for $\log \mathsf{mc}(\Psi)$ is the arithmetic mean of their logarithms:

$$\log(v^q) = m + \frac{\log g - 1 + \log G}{2} = m + \frac{\log(g \cdot G) - 1}{2} =$$
$$m + \frac{\log(a^2) - 1}{2} = m + \log a - \frac{1}{2},$$

and thus the return value of Algorithm 1 is

$$v = \sqrt[q]{a \cdot 2^{m-0.5}}. \tag{10}$$

A.3 Majority vote and confidence level

It remains to choose the parameter r that determines how many times the "Estimate" oracle $\mathscr{E}$ is called for each value of m. The choice of r depends primarily on $\alpha \in (0; 1)$, a number provided as part of the input: the probability that the algorithm returns a value v that is not within a $(1+\varepsilon)$-factor of $\mathsf{mc}(\psi)$ should be at most α. For this choice, however, we also take into account the smallest and largest values of m that can be reached during the run of the algorithm.

We first look into the smallest m on the run. To make the algorithm simple, we start from $m = 1$; to achieve the same quality of the produced values as in the previous subsection, we need to ensure that the maximum possible gap that arises if $\mathscr{E}$ returns "no" for $m = 1$ is (at most) of the same size. For larger values of m, the algorithm would conclude that Eq. (7) is satisfied. For $m = 1$, this means that the case $\mathsf{mc}(\Psi) \leq g$ should be ruled out. Therefore,

the *initial* enumeration threshold for $\mathsf{mc}(\Psi)$ should be set to g (note that, in fact, $g \leq a$); if the enumeration is done on the original formula ψ instead, the threshold is chosen as

$$p = \lceil g^{1/q} \rceil = \left\lceil (\sqrt{a+1} - 1)^{2/q} \right\rceil \tag{11}$$

where q is as in Eq. (9).

Let us now look into the largest m on the run. Here the purpose of the call to $\mathscr{E}$ is essentially to rule out the case $\mathsf{mc}(\Psi) \geq G \cdot 2^m$; this is needed unless $G \cdot 2^m > 2^n$ where n is the total number of Boolean variables. Hence, the last call to $\mathscr{E}$ should have $m \leq m^*$ where m^* is defined as

$$m^* = \lfloor n - \log G \rfloor = \lfloor n - 2 \log(\sqrt{a+1} + 1) \rfloor. \tag{12}$$

Finally, let us proceed to the choice of r. Suppose that m is fixed, and let X_i denote the Bernoulli random variable that is equal to 1 if the ith call to $\mathscr{E}$ returns the *less likely* answer and to 0 otherwise (recall that, by Claims 1 and 2, the *less likely* answer is "no" if $\mathsf{mc}(\Psi) \geq G \cdot 2^m$ and "yes" if $\mathsf{mc}(\Psi) \leq g \cdot 2^m$; these answers correspond to $\mathscr{E}$ being "wrong"). Let E denote the expectation; we have $\mathsf{E} X_i \leq 1/4$ for $1 \leq i \leq r$ by the choice of parameters in Appendix A.1. Denote by A_m the event that, in the presence of r "voters", the majority will vote "in the wrong way"; this event is captured by the inequality $\sum_{i=1}^{r} X_i \geq r/2$. If $\overline{X} = \sum_{i=1}^{r} X_i / r$, then $A_m = (\overline{X} \geq 1/2)$, while the expectation satisfies $\mathsf{E} \overline{X} \leq 1/4$. We have

$$\Pr[A_m] = \Pr[\overline{X} \geq 1/2] \leq \Pr[\overline{X} - \mathsf{E}\overline{X} \geq 1/4] \leq \exp\left\{-2 \cdot \left(\frac{1}{4}\right)^2 \cdot r\right\} = e^{-r/8},$$

where the last inequality is the one-sided Chernoff bound (see, e.g., [26, Section D.1.2.3]). Therefore, the probability of Algorithm 1 giving a bad estimate is upper-bounded by

$$\Pr[A_1 \cup \cdots \cup A_{m^*}] \leq m^* \cdot e^{-r/8};$$

to ensure that it is at most α, we pick the smallest r such that the right-hand side does not exceed α:

$$r = \left\lceil 8 \cdot \ln\left(\frac{1}{\alpha} \cdot \lfloor n - \log G \rfloor\right) \right\rceil = \left\lceil 8 \cdot \ln\left(\frac{1}{\alpha} \cdot \lfloor n - 2\log(\sqrt{a+1} + 1) \rfloor\right) \right\rceil. \tag{13}$$

This completes our analysis.

References

1. Allouche, D., de Givry, S., Schiex, T.: Toulbar2, an open source exact cost function network solver. Technical report, INRIA (2010)
2. Barvinok, A.: A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. In: FOCS 93. ACM (1993)
3. Bellare, M., Goldreich, O., Petrank, E.: Uniform generation of NP-witnesses using an NP-oracle. Inf. Comput. **163**(2), 510–526 (2000)
4. Belle, V., Van den Broeck, G., Passerini, A.: Hashing-based approximate probabilistic inference in hybrid domains. In: Proceedings of the 31st Conference on Uncertainty in Artificial Intelligence (UAI), Amsterdam (2015)
5. Belle, V., Passerini, A., Van den Broeck, G.: Probabilistic inference in hybrid domains by weighted model integration. In: Q. Yang, M. Wooldridge (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25–31, 2015, pp. 2770–2776. AAAI Press (2015). http://ijcai.org/papers15/Abstracts/IJCAI15-392.html
6. Borges, M., Filieri, A., d'Amorim, M., Pasareanu, C., Visser, W.: Compositional solution space quantification for probabilistic software analysis. In: PLDI, p. 15. ACM (2014)

7. Chaganty, A., Nori, A., Rajamani, S.: Efficiently sampling probabilistic programs via program analysis. In: AISTATS, JMLR Proceedings, vol. 31, pp. 153–160. JMLR.org (2013)

8. Chakraborty, S., Fremont, D., Meel, K., Seshia, S., Vardi, M.: Distribution-aware sampling and weighted model counting for SAT. In: AAAI'14, pp. 1722–1730 (2014). http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8364

9. Chakraborty, S., Meel, K., Vardi, M.: A scalable and nearly uniform generator of SAT witnesses. CAV, LNCS vol. 8044, pp. 608–623 (2013)

10. Chakraborty, S., Meel, K., Vardi, M.: A scalable approximate model counter. In: CP: Constraint Programming, LNCS, vol. 8124, pp. 200–216 (2013)

11. Chakraborty, S., Meel, K.S., Mistry, R., Vardi, M.Y.: Approximate probabilistic inference via word-level counting. In: Schuurmans, D., Wellman, M.P. (eds.) Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12–17, 2016, Phoenix, pp. 3218–3224. AAAI Press (2016)

12. Chatzikokolakis, K., Palamidessi, C.: Making random choices invisible to the scheduler. Inf. Comput. **208**(6), 694–715 (2010). doi:10.1016/j.ic.2009.06.006

13. Chistikov, D., Dimitrova, R., Majumdar, R.: Approximate counting in SMT and value estimation for probabilistic programs. In: Tools and Algorithms for the Construction and Analysis of Systems—21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, April 11–18, 2015. pp. 320–334 (2015)

14. Claret, G., Rajamani, S.K., Nori, A.V., Gordon, A.D., Borgström, J.: Bayesian inference using data flow analysis. In: ESEC/FSE'13, pp. 92–102 (2013). doi:10.1145/2491411.2491423

15. Cousot, P., Monerau, M.: Probabilistic abstract interpretation. In: ESOP, LNCS 7211, pp. 169–193. Springer (2012)

16. Darwiche, A.: Modeling and Reasoning with Bayesian Networks. Cambridge University Press, Cambridge (2009)

17. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS, TACAS'08/ETAPS'08, pp. 337–340. Springer (2008)

18. Durrett, R.: Probability: Theory and Examples, 4th edn. Cambridge University Press, Cambridge (2010)

19. Dyer, M., Frieze, A.: On the complexity of computing the volume of a polyhedron. SIAM J. Comput. **17**(5), 967–974 (1988)

20. Dyer, M., Frieze, A., Kannan, R.: A random polynomial time algorithm for approximating the volume of convex bodies. J. ACM **38**(1), 1–17 (1991)

21. Ermon, S., Gomes, C., Sabharwal, A., Selman, B.: Taming the curse of dimensionality: discrete integration by hashing and optimization. ICML **2**, 334–342 (2013)

22. Filar, J., Vrieze, K.: Competitive Markov Decision Processes. Springer, Berlin (1997)

23. Filieri, A., Pasareanu, C., Visser, W.: Reliability analysis in symbolic pathfinder. In: ICSE, pp. 622–631 (2013)

24. Fredrikson, M., Jha, S.: Satisfiability modulo counting: a new approach for analyzing privacy properties. In: CSL-LICS, pp. 42:1–42:10. ACM (2014)

25. Gilks, W., Thomas, A., Spiegelhalter, D.: A language and program for complex Bayesian modelling. Statistician **43**(1), 169–177 (1994)

26. Goldreich, O.: Computational Complexity: A Conceptual Perspective. Cambridge University Press, Cambridge (2008)

27. Gomes, C., Hoffmann, J., Sabharwal, A., Selman, B.: From sampling to model counting. In: IJCAI, pp. 2293–2299 (2007)

28. Gomes, C., Sabharwal, A., Selman, B.: Model counting. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 633–654. IOS Press (2009)

29. Gordon, A., Henzinger, T., Nori, A., Rajamani, S., Samuel, S.: Probabilistic programming. In: FOSE 14, pp. 167–181. ACM (2014)

30. Han, C., Jiang, J.R.: When Boolean satisfiability meets Gaussian elimination in a simplex way. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification—24th International Conference, CAV 2012, Berkeley, CA, USA, July 7–13, 2012 Proceedings, Lecture Notes in Computer Science, vol. 7358, pp. 410–426. Springer (2012). doi:10.1007/978-3-642-31424-7_31

31. Hur, C.K., Nori, A., Rajamani, S., Samuel, S.: Slicing probabilistic programs. In: PLDI, p. 16. ACM (2014)

32. Impagliazzo, R., Levin, L.A., Luby, M.: Pseudo-random generation from one-way functions (extended abstract). In: Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14–17, 1989, Seattle, Washigton, USA, pp. 12–24 (1989)

33. Jerrum, M., Sinclair, A.: The Markov chain Monte Carlo method: an approach to approximate counting and integration. Approximation algorithms for NP-hard problems pp. 482–520 (1996)

34. Jerrum, M., Valiant, L., Vazirani, V.: Random generation of combinatorial structures from a uniform distribution. TCS **43**, 169–188 (1986)
35. Katoen, J.P., McIver, A., Meinicke, L., Morgan, C.: Linear-invariant generation for probabilistic programs: automated support for proof-based methods. In: SAS, LNCS 6337, pp. 390–406. Springer (2010)
36. Kiselyov, O., Shan, C.C.: Monolingual probabilistic programming using generalized coroutines. In: UAI, pp. 285–292. AUAI Press (2009)
37. Klee, V.: Can the measure of $\cup[a_i, b_i]$ be computed in less than $O(n \log n)$ steps? Am. Math. Mon. **84**, 284–285 (1977)
38. Koller, D., Friedman, N.: Probabilistic graphical models: principles and techniques. MIT Press, Cambridge (2009)
39. Kozen, D.: Semantics of probabilistic programs. JCSS **22**, 328–350 (1981)
40. LattE tool. https://www.math.ucdavis.edu/~latte
41. Lawrence, J.: Polytope volume computation. Math. Comput. **57**(195), 259–271 (1991)
42. Luckow, K.S., Pasareanu, C.S., Dwyer, M.B., Filieri, A., Visser, W.: Exact and approximate probabilistic symbolic execution for nondeterministic programs. In: ASE'14, pp. 575–586 (2014). doi:10.1145/2642937.2643011
43. Luu, L., Shinde, S., Saxena, P., Demsky, B.: A model counter for constraints over unbounded strings. In: PLDI, p. 57. ACM (2014)
44. Ma, F., Liu, S., Zhang, J.: Volume computation for Boolean combination of linear arithmetic constraints. In: CADE-22, LNCS 5663, pp. 453–468. Springer (2009)
45. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Springer, Berlin (2005)
46. Minka, T., Winn, J., Guiver, J., Kannan, A.: Infer.NET 2.3 (2009)
47. Monniaux, D.: Abstract interpretation of programs as Markov decision processes. Sci. Comput. Progr. **58**, 179–205 (2005)
48. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan-Kaufman, Burlington (1997)
49. Papadimitriou, C.: Games against nature. JCSS **31**(2), 288–301 (1985)
50. Monty Hall problem: http://en.wikipedia.org/wiki/Monty_Hall_problem
51. Sampson, A., Panchekha, P., Mytkowicz, T., McKinley, K., Grossman, D., Ceze, L.: Expressing and verifying probabilistic assertions. In: PLDI, p. 14. ACM (2014)
52. Sankaranarayanan, S., Chakarov, A., Gulwani, S.: Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In: PLDI, pp. 447–458. ACM (2013)
53. Selvin, S.: A problem in probability. Am. Stat. **29**(1), 67 (1975)
54. Sipser, M.: A complexity-theoretic approach to randomness. In: STOC, pp. 330–335. ACM (1983)
55. Soos, M.: CryptoMiniSat—a SAT solver for cryptographic problems. http://www.msoos.org/cryptominisat4/
56. Soos, M.: Enhanced Gaussian elimination in DPLL-based SAT solvers. In: POS-10. Pragmatics of SAT, Edinburgh, UK, July 10, 2010, pp. 2–14 (2010). http://www.easychair.org/publications/?page=1319113489
57. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Theory and Applications of Satisfiability Testing—SAT 2009, 12th International Conference, SAT 2009, Swansea, June 30–July 3, 2009. Proceedings, pp. 244–257 (2009)
58. Stockmeyer, L.: On approximation algorithms for $\#P$. SIAM J. Comput. **14**, 849–861 (1985)
59. Trevisan, L.: Computational complexity (CS 254), lecture 8 (2010). http://www.cs.stanford.edu/~trevisan/cs254-10/lecture08.pdf
60. Urquhart, A.: Hard examples for resolution. J. ACM **34**(1), 209–219 (1987)
61. Valiant, L.: The complexity of computing the permanent. Theor. Comput. Sci. **9**, 189–201 (1979)
62. Valiant, L., Vazirani, V.: NP is as easy as detecting unique solutions. Theor. Comput. Sci. **47**, 85–93 (1986)
63. Zhou, M., He, F., Song, X., He, S., Chen, G., Gu, M.: Estimating the volume of solution space for satisfiability modulo linear real arithmetic. Theory Comput. Syst. **56**(2), 347–371 (2014). doi:10.1007/s00224-014-9553-9