Probabilistic Program Analysis

Matthew B. Dwyer[1], Antonio Filieri[2], Jaco Geldenhuys[4], Mitchell Gerrard[1], Corina Pasareanu[3], and Willem Visser[4]

[1] University of Nebraska – Lincoln
[2] Imperial College London
[3] NASA Ames Research Center
[4] University of Stellenbosch

Abstract. This paper provides a survey of recent work on adapting techniques for program analysis to compute probabilistic characterizations of program behavior. We study how the frameworks of data flow analysis and symbolic execution have incorporated information about input probability distributions to quantify the likelihood of properties of program states. We identify cross-cutting themes that relate and distinguish the variety of techniques that have been developed over the past 15 years in this area. In doing so, we point out opportunities for future research that builds on the strengths of different techniques.

Keywords: data flow analysis, symbolic execution, abstract interpretation, model checking, probabilistic program, MDP

1 Introduction

Static program analyses aim to calculate properties of the possible executions of a program without ever running the program, and have been an active topic of study for over five decades. Initially developed to allow compilers to generate more efficient output programs, by the mid-1970s [29] researchers had understood that such program analyses could be applied to fault detection and verification of the absence of specific classes of faults.

The power of these analysis techniques, and what distinguishes them from simply running a program and observing its behavior, is their ability to reason about program behavior without knowing all of the exact details of program execution (e.g., the specific input values provided to the program, the set of operating system thread scheduler decisions). This tolerance of uncertainty allows analyses to provide useful information when users don't know exactly how a program will be used (e.g., when a program is first released, when embedded systems read sensor inputs from the physical world, or when it is ported to an operating system with a different scheduler).

Static analyses model uncertainty in program behavior through the use of various forms of abstraction and symbolic representation. For example, symbolic expressions are used to encode logical constraints in symbolic execution [46], to define abstract domains in data flow analysis [45, 19], and to capture sets of data

values that constitute reachable states via predicate abstraction [36]. Nondeterministic choice is another widely used approach for modeling uncertainty—for instance, in modeling uncertain branch decisions in data flow analysis, or scheduler decisions in model checking. While undeniably effective, these approaches sacrifice potentially important distinctions in program behavior.

Consider a program that accepts an integer input representing a person's income. A static analysis might reason about the program by allowing any integer value, or, perhaps, by applying some simple assumption, i.e., that income must be non-negative. Domain experts have studied income distributions and find that incomes vary according to a generalized beta distribution [55, 77]. Can this type of information be exploited to yield useful analysis results when classic analyses fail, or to reason about new types of program properties?

Whether information about the distribution of values is embedded within a program or stated as an input assumption, the semantics of such probabilistic programs is well-understood—and has long been studied [47, 48, 43, 61] [1]. What has lagged behind is the development of frameworks for defining and implementing static analysis techniques for such programs.

What would such analyses have to offer? They can, of course, provide a means of analyzing programs that compute with values chosen from probability distributions, but they offer much more. For example, researchers have explored the use of probabilistic analysis results to assess the security of software components [54], to assess program reliability [27], to measure program similarity [31], to characterize the coverage achieved by an analysis technique [22], and to provide information about program properties when a classic analysis would fail, e.g., by running out of memory, time, or due to excessive approximation.

In this paper, we survey work on adapting data flow analysis and symbolic execution to use probabilistic information. We begin with background that provides basic definitions related to static analysis and probabilistic models. Section 2.2 attempts to expose some of the key intuitions and concepts that cross-cut the work in this area. The following two sections, 3.1-3.2, survey work on probabilistic data flow analysis and probabilistic symbolic execution. Section 3.3 discusses approaches that have been developed to reason about the probability of program-related events, e.g., executing a path, taking a branch, or reaching a state. We conclude with a set of open questions and research challenges that we believe are worth pursuing.

```
m(int x) {
 if(drawBernoulli(0.5) == 1) {
  if(drawBernoulli(0.5) == 1) {
   if(x <= 60)
    ...
   else
    assert false
  } else {
   if(x <= 30)
    ...
   else
    assert false
  }
 } else {
  if(x <= 55)
   ...
  else
   assert false
 }
}
```

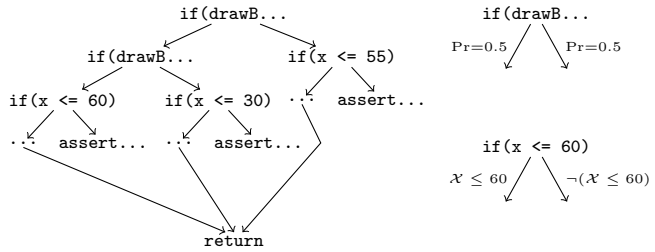

Fig. 1. Example

2 Overview

2.1 Scope and Background

We focus in this paper on programs that draw input variables from given probability distributions, or, equivalently, that make calls on functions returning values drawn from given distributions. The left side of Figure 1 shows a method, `m`, that we will use to illustrate concepts in this paper. It takes an integer variable, `x`, as input, then based on the results of drawing values from a Bernoulli distribution, it either performs its computation (which is unspecified and denoted with `...`) or triggers an assertion.

While researchers have developed analyses that consider a wide range of program properties, here we restrict our attention to program properties that can be encoded as boolean predicates embedded in the program, e.g., `assert` statements, to simplify the explanation of how probabilities are incorporated into the analyses. For the example, we are interested in reasoning about assertion freedom, i.e., that the program does not reach a false assertion. In our example this can happen by reaching the implicit return from `m`, thereby avoiding the three `assert` statements.

Programs and Program Analyses There are many different ways to represent the execution behavior of a program to facilitate analysis. Immediately to the right of the code in Figure 1, we show the control flow graph (CFG), which

[1] In recent years, the term probabilistic program has been generalized beyond drawing inputs from probability distributions, which we consider here, to programs that can condition program behavior—by rejecting certain program runs—and thereby be viewed as computations over probability distributions. We refer the reader to the recent paper by Gordon, Henzinger, Nori and Rajamani [35] for discussion of the analysis of these more general probabilistic programs.

explicitly represents control successor relationships between statements. A CFG models choice among successors as nondeterministic choice – depicted by the lack of labels on the edges. We will also consider models that include probabilistic choice, e.g., defining the probability that a branch is taken; the upper right fragment of Figure 1 shows probabilities that reflect the outcome of the Bernoulli draw. In addition, we will study models where the choice of successor is defined based on the semantics of program state, e.g., defining a condition under which the branch is taken; the lower right fragment of Figure 1 shows logical conditions that reflect the fact that the value of parameter x must be less than or equal to 60 for control to traverse the branch.

A key concept in the program analysis frameworks we survey is *symbolic abstraction*. A symbolic abstraction is a representation of a set of states. Abstractions can be encoded in a variety of forms, e.g., logical formulae [74], binary decision diagrams [10], or custom representations [3]. For example, the set of integer values that are less than 0 can be defined, through its characteristic function $lt0 \equiv \lambda x.x < 0$ which returns true for all values in the set.

While abstractions encode sets of states *abstract transformers* compute the effect of a program statement on a set of states. For example, the fact that the sum of any pair of negative values is negative is encoded as $lt0 +^{\#} lt0 = lt0$, where $\#$ denotes an abstract transformer for $+$ that operates on symbolic encodings of sets.

Analyses that seek to prove the satisfaction of properties generally define abstractions that *overapproximate* the set of program states, whereas those that seek to falsify properties generally define abstractions that *underapproximate* the set of program states. With overapproximating analyses, it is common to define an *abstract domain*, $\mathcal{A}$, which symbolically represents a set of concrete states that are said to be defined over the *concrete domain*, $\mathcal{C}$.

Data Flow Analysis Data flow analyses [45] provides a framework for computing properties shared by sets of program traces reaching a program state, or set of states. It is common for such analyses to group together the states that share a common control location; the computed properties attempt to characterize the invariants over those states.

Data flow analyses are solved using a fixpoint computation which allows properties of all program paths to be safely approximated. Model checking [17] also relies on an underlying fixpoint computation. Moreover, data flow analyses operate on symbolic abstractions of program states that can be defined by abstract interpretation [19]. In fact, it is now well-understood that data flow analysis can be viewed as model checking of abstract interpretations [70].

An abstract interpretation is a non-standard interpretation of program executions over an abstract domain. The semantics of program statements are lifted to operate on a set of states, encoded as an element of the abstract domain, rather than on a single concrete state. Generating the set of traces for non-trivial programs is impractical; instead, abstract states can be combined, via a meet operation, wherever traces merge in the control flow, and loops are processed repeatedly to compute the maximum fix point (MFP).

While not traditionally a component of a data flow analysis, a property, ϕ, can be checked by relating it to abstract states. Specifically, if some abstract state at location l implies ϕ, then ϕ is verified to hold whenever location l is reached. However, it may be the case that ϕ holds at some location l' but the abstract state at l' does not imply ϕ; this is referred to as a *false error* report and is due to the overapproximating nature of abstract interpretation.

Data flow analysis tools and toolkits exist for many popular languages [79, 28] and have been used primarily for program optimization and verifying program conformance with (implicit and explicit) assertional specifications.

Symbolic Execution Like data flow analysis, symbolic execution [46, 18] performs a non-standard interpretation of program executions using a symbolic abstraction of program states. Symbolic execution records symbolic expressions encoding the values of program memory for each program location. A *path condition* accumulates symbolic expressions that encode branch constraints taken along a trace. The analysis begins at the first program location with the path condition set to *true*.

Sequences of program statements are interpreted by applying the operation at each program location to update the values of program variables with expressions defined over symbolic variables. An operation that reads from an input generates a fresh symbolic variable which represents the set of possible input values. When a branching statement is encountered, the symbolic expression, c, encoding the branch condition is computed and a check is performed to determine whether the current trace—encoded by the path condition—can be extended with c or c's negation. This is done by formulating the constraints as a satisfiability query; if the formula encoding branch constraints is satisfiable, then there must exist an input that will follow the trace. The trace is extended following the feasible branch outcomes, usually in a depth-first manner. In the example of Figure 1, on the leftmost trace through the control flow graph, when symbolic execution reaches the condition shown in the lower right, $c \equiv \mathcal{X} \leq 60$. That condition is conjoined to the path condition, $PC \wedge \mathcal{X} \leq 60$, and then confirmed to be satisfiable since the previous branches do not constrain the values of $\mathcal{X}$—the symbolic variable representing the value of input x.

While symbolic execution is capable of computing an *exact* symbolic approximation of the set of program states on a trace reaching some location, in practice symbolic execution computes an underapproximation. Programs with looping behavior that is determined by input values may result in an infinite symbolic execution tree. For this reason, symbolic execution is typically run with a (user-specified) bound on the search depth, thus some paths may be unexplored. Moreover, there may be path constraints for which efficient satisfiable checking is not possible. Variants of symbolic execution [34, 71, 73] address this by replacing problematic path condition constraints with equality constraints based on values collected while executing the program along the trace.

It is straightforward to integrate the checking of a property, ϕ, into symbolic execution. Specifically, if a symbolic expression reaching location l implies $\neg\phi$, then ϕ is falsified when l is reached. This does not mean that ϕ fails to hold

whenever l is reached, but due to the underapproximation of symbolic execution there is guaranteed to be an execution for which it fails to hold. However, it may be the case that ϕ fails to hold on some trace reaching location l, but this may be missed due to the underapproximating nature of symbolic execution. In other words, if symbolic execution fails to find an error, then one cannot conclude the lack of errror.

Symbolic execution tools and toolkits exist for many popular languages [62, 34, 40, 11] and have been used primarily for test generation and fault detection.

Probabilities and Probabilistic Models There is an enormous literature on probability and statistics that can be brought to bear in program analysis. In this paper, we consider two types of discrete time models: Markov chains and Markov decision processes (MDP).

Both models rely on the concept of a probability distribution. A *probability distribution* over a set is given by a function which maps each of the set's subsets to some real number between 0 and 1; this number represents the probability of a given subset being the outcome of a random experiment. The sum of the probabilities of all subsets is 1.

A Markov chain is a labeled transition system which, given some state, defines the probability of moving to another state. The probability of executing a sequence of states is then the product of the transition probabilities between each state. The model fragment in the upper right corner of Figure 1 depicts a Markov chain fragment. It defines, for the set of states that are at the first line in the program, a 0.5 probability of transitioning to the state representing the beginning of the then block, and similarly for the beginning of the else block. The distribution indicates a 0 probability of moving to any other state.

For this small example, if we were to assume a probability distribution on the input x, then it would be possible to compute the probability of taking every edge in the CFG. This would be a Markov chain model of `m` and it would replace all nondeterministic choice in the CFG with probabilistic choice.

There are many situations where the removal of nondeterministic choice is impractical or undesirable. For example, if the input distribution of x is unknown, then retaining nondeterministic choices for the conditionals which test that value would yield a faithful program model. In addition, it may be desirable, for efficiency of analysis, to abstract program behavior, and that abstraction may make it impossible to accurately compute the probability of a transition. For example, to reason about the behavior of a multi-threaded program, nondeterministic choice is used to model the scheduler policy.

When nondeterminism is included in a probabilistic state transition model, it results in an MDP. An MDP adds an additional structure, A, that defines a set of (internal) actions which are used to model the selection among a set of possible next-state probability distributions. When traversing a path in an MDP, in each state, a choice from A must be made in order to determine how to transition, probabilistically, to a next state. That sequence of choices is termed a *schedule* (or policy or strategy) for the MDP. Given a fixed schedule, an MDP

reduces to a Markov chain, but the value of an MDP lies in its ability to provide upper and lower bounds on the behavior of the modeled system (discussed in the following section).

2.2 Extending Program Analyses with Probabilities

The literature on incorporating probabilistic techniques into program analysis is large and growing, technically deep, and quite varied. In this paper we cannot hope to cover all of it, but our intention is to expose key similarities and differences between families of approaches and, in doing so, provide the reader with intuitions that are often missing in the detailed presentation of techniques.

Where do the probabilities come from? There are two perspectives adopted in the literature. Programs are *implicitly* probabilistic because the distributions from which input values are drawn are not specified in the program, but are characteristics of the execution environment. Alternately, programs are *explicitly* probabilistic in that the statements within the program define the input probability distributions. More generally, a program might combine both implicit and explicit probabilistic calls. Method `m` in Figure 1 is an example of such a program.

It is possible to transform explicit probabilistic constructs into implicit ones by introducing auxiliary input variables and then specifying their distributions. For the example, this would result in the addition of two integer input variables

```
m(int x, int b1, int b2) {...
```

where the two instances of `drawBernoulli(0.5)` expressions would be replaced by `b1` and `b2`, respectively. The input distribution for these auxiliary inputs would then be specified as a set of pairs,

$$\{(\lambda x.x = 0, 0.5), (\lambda x.x \neq 0 \vee x \neq 1, 0), (\lambda x.x = 1, 0.5)\}$$

where the first component defines the characteristic function of a set of values and the second component defines the probability of a value in that set.

Section 3.1 discusses approaches where probabilities governing specific branch outcomes, as opposed to input values, are built into the program model from knowledge the developer has at hand, while Sections 3.2 and 3.3 describe techniques for computing such probabilities from information about the program semantics and input distribution.

What does the analysis compute? There are again two perspectives adopted in the literature. One can view a probabilistic program as a transformer on probability distributions; the analysis computes the probability distribution over the concrete domain which holds at a program state. Alternatively, one can view a probabilistic program as a program whose inputs happen to vary in some principled way; the analysis computes program properties—properties of

sets of concrete domain elements—along with a characterization of how these properties vary with varying input. Within these approaches, there are different types of approximations computed for probabilities. It is common to compute upper bounds on probabilities for program properties, but lower bounds can be computed as well. In addition, it is possible to estimate the probability within some margin of error—an approach that several techniques explore—and it is even possible to compute the probability *exactly*, if certain restrictions hold on the program and its distributions.

Conceptually, there are two pieces of information that are necessary to reason probabilisticaly about a set of concrete values: a quantity that approximates the probability of each value in the set and the number of values in the set. Many of the earlier probabilistic static analysis techniques did not explicitly capture this latter quantity, but more recent work discussed in Section 3.2, using the techniques of Section 3.3, does capture this quantity, as do other recent approaches [54].

Mixing abstraction with probabilities Any analysis that hopes to scale will have to approximate behavior. As explained earlier, in static analyses it is common to model such overapproximation using nondeterministic choice. Across all of the analysis techniques we have surveyed, MDPs have been used when there is a need to mix probabilistic and nondeterministic choice. An important consequence of using MDPs is that it is no longer possible to compute a single probabilistic characterization of a property. Instead, analyses can compute, across the set of all possible sequences of nondeterministic choice outcomes, the minimal and maximal probabilities for a property to hold.

If the minimal probability for a property of interest lies above a desired probability threshold, then regardless of how the nondeterminism is resolved, the property is guaranteed to hold within the desired threshold—the probabilistic property *must* hold. If that is not the case, but the maximal probability for a property of interest lies above a desired threshold, then there is a schedule to resolve the nondeterminism that satisfies the property with at least the desired probability—the probabilistic property *may* hold.

3 Survey of Probabilistic Approaches to Program Analysis

3.1 Probabilistic Data Flow Analysis

The key challenge in probabilistic data flow analysis is determining how probabilities are incorporated into the control and data abstractions that form its foundation. As in the classical case, we will see that probabilistic data flow analysis can be thought of as model checking of probabilistic abstract interpretations, and in some cases, the model checking itself must be made probabilistic.

Control Flow Probabilities Early work in extending data flow analysis techniques with probabilities did not consider the influence of control and data flow on probabilities. Instead, user-defined probabilities were attached to nodes in the program's control flow graph. This allowed the analysis to estimate the probability of an expression evaluating to some value or type at runtime, which was used to enable program optimization.

This approach begins with a control flow graph where each edge is mapped to the probability that it is taken during execution. The sum of all probabilities leaving any control flow node must be 1 (excepting the exit node). These probabilities may be obtained through heuristics, profiling, or some static analysis. Imagine an execution trace following some path along the edges of the control flow graph. The probability of executing that trace is expressed as the product of edge probabilities along this path. So the probability of executing some program point can be seen as the summation of the probabilities of traces which can reach that program point.

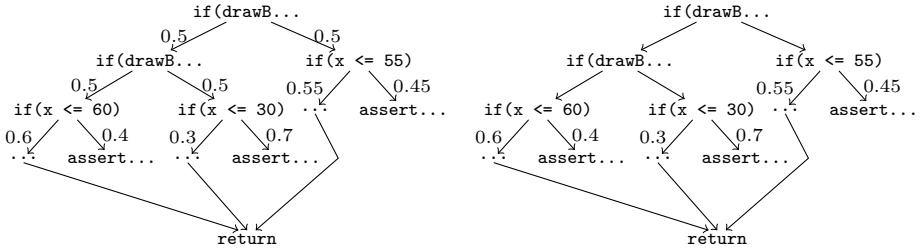

Fig. 2. Probabilistic CFG (left) and MDP (right)

Figure 2 shows the probabilistic CFG for the example from Figure 1, given that input x is uniformly distributed in the range $[1, 100]$.

To compute the probability of a data flow fact holding at a program point, Ramalingam uses a slightly modified version of Kildall's dataflow analysis framework [64]. Instead of the usual semilattice with an idempotent meet operation, a non-idempotent addition operator is used. The restricted properties of the meet operation can be relaxed, because instead of computing an invariant dataflow fact, we only want the summation of probabilities of all traces reaching a certain point. The expected frequencies may now be computed as the least fixpoint using the traditional iterative data flow algorithm; the quantity becomes a *sum-over-all-paths* instead of a *meet-over-all-paths*.

Ramalingam's work assumes execution history does not matter—the analysis is path insensitive. Later work adds some path sensitivity [57], but as both frameworks operate over exploded control flow graphs, a fully path-sensitive approach is not tractable.

Ramalingam's sum-over-all-paths approach is reminiscent of the approach taken in probabilistic model checking of DTMCs. In that approach, a system

of linear equations is formulated whose solution computes the probability with which a property holds—so-called *quantitative* properties in PRISM [49]. Ramalingam formulates an equivalent system of linear equations.

Both of these techniques rely on being able to annotate branch decisions in the program (or model, in PRISM's case) with probabilities. When those decisions are governed by computed conditions over input variables, the calculation of branch probabilities quickly becomes challenging—Section 3.2 and Section 3.3 describe approaches to address this problem.

Abstract Data Probabilities Within the last 15 years, researchers began incorporating probabilistic information directly into the semantics of a program and then abstracting over those semantics [58, 72, 20] to enable data flow analysis. This is typically done using a variation on Kozen's probabilistic semantics [47] alongside abstract interpretation and data flow techniques. Embedding probabilities into the semantics allows the expression of both control flow and data values to influence the property probabilities computed during the analysis.

Abstracting Probability Distributions The pioneering work in this area, by Monniaux [58, 59], developed the key insights that other work has built on. The goal is to exploit the rich body of work on developing abstract domains and associated transformers, and to extend this work so as to record bounds on probability measures for the concrete values described by domain elements.

Monniaux's work takes the view that probabilistic programs effectively transform an input distribution into an output distribution. More generally, probabilistic programs compute a distribution that characterizes each state in the program. He develops a probabilistic abstract domain, $\mathcal{A}_p$, as an (indexed) collection of pairs, $\mathcal{A} \times [0, 1]$. The intuition is that a classic abstract domain is paired with a *bounding probability weight* that is used to compute an upper bound on the concrete elements mapped by that domain. Given a concrete value c, an upper approximation of its probability for a probabilistic abstract domain element $pa = \{(a_1, w_1), ..., (a_n, w_n)\}$ is given by

$$\sum_{j \in \{i | (a_i, w_i) \in pa \wedge c \in \gamma(a_i)\}} w_j$$

where γ is the function that maps a symbolic abstraction to the set of concrete values it describes.

In Monniaux's work, multiple abstract domain elements can map onto a given concrete value; each of these abstract domain elements' weights must be totalled to bound the probability of the given concrete value. As an example, let $\mathcal{A}$ be the interval abstraction applied to a single integer value and let $pa = \{([1, 5], 0.1), ([3, 7], 0.1), \ldots\}$. For a value of 2, only the first pair would apply, since $2 \notin [3, 7]$, contributing 0.1 to the bound on $Pr(2)$. For a value of 3, both pairs would apply and contribute their sum of 0.2 to the bound on $Pr(3)$.

To clarify, these weight components are *not* bounds on the probability of the abstract domain as a whole, but rather are bounds on the probability of

each concrete element represented by the abstract domain. This simplifies the formulation of the probabilistic abstract transformers, i.e., the extension of $\tau^{\#}$, the non-probabilistic abstract transformer, to account for $\mathcal{A}_p$, but it means that additional work is required to compute the probability of a property holding. Fundamentally, that requires estimating the size of the concretization of the abstract domain element and then multiplying by the computed bound for each concrete value.

It is important to note that an upper or lower bound on a probability distribution is not itself a distribution, since the sum across the domain may be greater than 1. This poses challenges for modular probabilistic data flow analyses.

We will see that the techniques from Section 3.3 can be applied to the problem of counting the concretization of an abstract domain element that is encoded as a logical formula. This may offer a potential connection between data flow analyses formulated over distributions and those formulated over abstract states—which we discuss below.

The design of probabilistic abstract transformers can be subtle. For statements that generate variables drawn from a probability distribution, an upper approximation of the distribution for regions of the abstract domain is required. The literature has constructed these using ad-hoc techniques, but we believe the methods of Section 3.3 might be applied to achieve an upper approximation. For sequential statements, weight components are propogated and abstract domain elements are updated by the underlying transformer.

For conditionals, the transformer can be understood as filtering the abstract domain between those execution environments which satisfy the conditional and those which falsify the conditional. The difference in the probabilistic abstract environment with weights is that the filter is only applied to the first component of the tuple (the traditional elements of an abstract domain), and leaves the weight unchanged. For instance, consider the abstract domain of an interval of integers defined by the tuple, $([-5, 5], 0.1)$. If this domain holds before a conditional of `if(x<0){...}`, after applying the filter on the true branch, we get $([-5, -1], 0.1)$. After applying the filter on the false branch, we get $([0, 5], 0.1)$. The space is reduced; the weights remain the same.

Finally, reaching fixpoints for rich probabilistic abstract domains appears to require widening [58, 23] to be cost-effective.

Probability for Abstract States Computing bounds on the probability of a state property has been well-studied. Di Pierro et al. [21] develop analyses to estimate the probability of an abstract state, rather than bound it or its probability distribution. They formulate their analysis using an abstract domain over vector spaces, instead of lattices, and use the Moore-Penrose pseudo-inverse instead of the usual fixpoint calculation.

Abstract states encode variable domains as matrices, e.g., a 100 by 100 matrix would be needed to encode the input x for the example in Figure 1. While very efficient matrix algorithms can be employed, the space consumed by this representation can be significant for large concrete domains. Transfer functions operate on these matrices to filter values and update probabilities along branches

and, as in Ramalingam's work, weighted sums are used to accumulate probabilities at control flow merge points. Di Pierro et al.'s early work was limited to very small programs, but more recent work suggests approaches for abstracting the matrices to significantly reduce time and space complexity.

Handling nondeterminism When abstraction of program choice is required or when there is no basis for defining an input distribution programs, it is natural to use nondeterminism to account for the uncertainty in program behavior.

In Monniaux's semantics [60], choices that can be tied to a known distribution are cleanly separated from those that cannot. A nondeterministic choice allows for independent outcomes, and this is modeled by lifting the singleton outcomes of deterministic semantics to powersets of outcomes. In the probabilistic setting, the elements of this powerset are tuples of the abstract domain and the associated weight, defined above. So for any nondeterminstic choice, the resulting computation is safely modeled by one of these tuples. The challenge in the analysis is to select from among those tuples to compute a useful probability estimate.

More recent work on abstraction in probabilistic data flow analysis, as well as in model checking, takes a different approach. A trace in an MDP can be viewed as an alternation of probabilistic and nondeterministic choices. For instance, the leftmost trace in the MDP on the right side of Figure 2 can be read as $\mathit{left}; \_; \mathit{left}; 0.6$, where $\_$ denotes an empty choice—in this case a probabilistic choice. Probabilistic model checkers such as PRISM and PASS exploit this 2-phase structure to formulate MDP model checking as a 2-player game. One player resolves the nondeterminism, thereby determining the schedule, and the other resolves the probabilistic choice.

Abstract interpretation can be applied to the data states in such approaches [50, 81, 23] to improve efficiency. Note, however, that these abstractions are independent of the probabilistic choices implicit in the semantics, and must be specified by the developer in some way—as in the case of Ramalingam's work.

More and varied probabilistic data flow analyses Recent years have seen several varied lines of work draw on the lessons learned from the early research on probabilistic data flow analysis.

Researchers have developed rich customized abstract domains that incorporate probability bounds [54, 1] and permit analyses of probabilistic properties of real software systems.

More recent work has explored computing an alternative probabilistic property called an *expectation invariant* [12]. This approach uses an iterative data flow analysis to compute a bound on the long-run expected value of some program expression, e.g. $E[f(\mathit{uniform}(0, 10))] < 7$ states that, over a sufficiently large number of runs, when f is called with a uniformly distributed number in the range $[0, 10]$, it will return an average value which is less than 7.

The idea of a "probabilistic program" has been generalized from Kozen's original semantics to include conditioning on program observations [35]. In this

setting, the program implicitly specifies a probability distribution conditioned on these stated observations. Data flow analyses have recently been adapted to perform Bayesian inference on this new class of probabilistic programs [16].

3.2 Probabilistic Symbolic Execution

Symbolic execution produces path conditions (PCs), i.e., constraints on the inputs, that characterize how a certain target property can be reached. During the process of symbolic execution, the most important question to answer about each PC is whether it is satisfiable or not. If not, then the corresponding path does not need to be analyzed any further. However, now we are additionally interested in the *probability* of a target property being satisfied.

For simplicity, we assume we are working with a uniform usage profile for the program under analysis. In other words, all input values are equally likely. See [27] for a detailed discussion of usage profiles within symbolic execution.

For example, in Figure 3, we introduce variables b_0 and b_1 to model the two `drawBernoulli` distributions from Figure 1; the domains of those variables consists of 10 values and the tests check for half of the domain, which corresponds to the 0.5 parameter in the Bernoulli distribution. Note that this program now has 3 inputs and an input domain size of $10 \times 10 \times 100 = 10000$. The domain of variables, which is finite and discrete, is denoted by D.

Approach We outline a general approach to probabilistic symbolic execution that accomodates many of the advances in the recent literature. Algorithms 1 and 2 are modifications of the traditional symbolic execution algorithm to sample and process symbolic paths one at a time. This processing includes the calculation of probabilities as described in the next section. The interested reader is referred to [25] for a more detailed discussion of the precise algorithm.

When considering the example in Figure 3, each of the six paths from the root to the leaves can be sampled and the path condition describing that path will be the conjunct of the constraints along the path; for example the leftmost path will have $b_0 < 5 \wedge b_1 < 5 \wedge x \leq 60$ as its path condition.

Below we discuss *stoppingPath*, *selectBranch*, *stoppingSearch* and *processPath* of Algorithms 1 and 2. At a high level, `symsample` is called from the initial state of the program with $pc = \texttt{true}$ as the current path condition; it returns a single path which is then processed. This is followed by a check to see if the analysis is complete and can stop. Within `symsample` we first check to see if the search needs to be stopped; otherwise, we symbolically execute the program up to the next branching point and decide which of the next branching statements must be taken. Note that now only one branch is taken, unlike in traditional symbolic execution, where both branches could be taken if they were both satisfiable.

```
▷  b0, b1 ∈ {0, ..., 9}
if b0 < 5 then
  if b1 < 5 then
    if x ≤ 60 then
      ... A
    else
      assert false
    end if
  else
    if x ≤ 30 then
      ... B
    else
      assert false
    end if
  end if
else
  if x ≤ 55 then
    ... C
  else
    assert false
  end if
end if
```

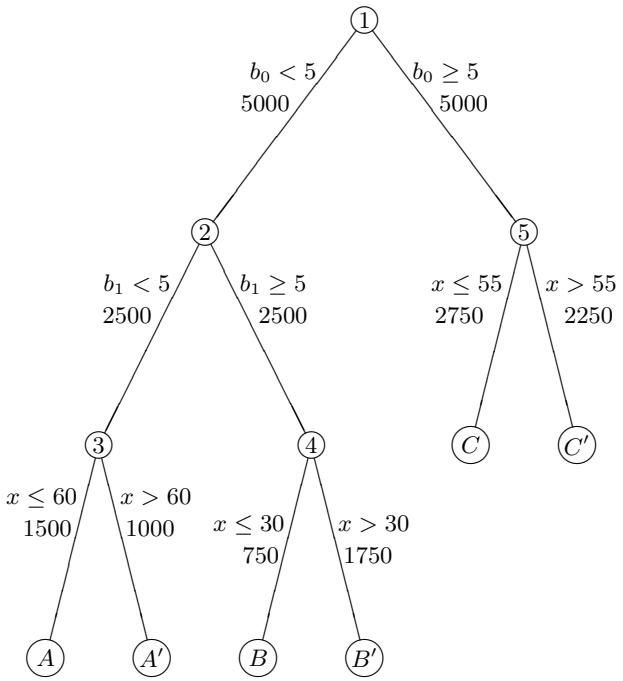

Fig. 3. Illustration of symbolic execution

Alg. 1 `pse`(l, m, pc)

```
repeat
  p ← symsample(l0, m0, true)
  processPath(p)
until stoppingSearch(p)
```

Alg. 2 `symsample`(l, m, pc)

```
if stoppingPath(pc) then
  return pc
end if
while ¬branch(l) do
  m ← op(l)(m)
  l ← succ(l)
  check(l, m, φ)
end while
c ← cond(l)(m)
if selectBranch(c, pc) then
  return symsample(succ_t(l), m, pc ∧ c)
else
  return symsample(succ_f(l), m, pc ∧ ¬c)
end if
```

stoppingPath is the same as in the original symbolic execution algorithm and is there to handle loops conditioned on input variables, since these can cause

infinite executions. However, since some paths might now be truncated before reaching the target property, we introduce three types of paths, (1) success paths, which reach and satisfy the target property, (2) failure paths, which reaches and falsify the property, and (3) grey paths, which are truncated before reaching the property. We consider these as three disjoint sets of paths and calculate the cumulative probability of success (i.e., the reliability of the code), failure and grey paths. Grey paths can be handled optimistically (grouped with the success paths), pessimistically (grouped with the failure paths) or kept separate and be used as a measure for how confident we are in our estimates (for example, if the grey paths probability is very low, we are more confident).

selectBranch In the context of symbolic execution, we define a sample as the simulation of one symbolic path. Whenever a branch is encountered during such simulation, the decision to proceed along either of the alternative branches has to be taken according to the probability of satisfying the corresponding branch conditions. To calculate these, we calculate the number of solutions for each path condition as described at the beginning of this section—techniques for calculating this number are detailed in Section 3.3.

At each branching point, we have the count for the PC that reached the branching point ($\sharp(pc)$) and the counts for the path condition for both branches ($\sharp(pc \wedge c)$ and $\sharp(pc \wedge \neg c)$). The probability for the true branch is thus $Pr(succ_t(l)) = \sharp(pc \wedge c)/\sharp(pc)$ and for the false branch it is $Pr(succ_f(l)) = \sharp(pc \wedge \neg c)/\sharp(pc)$.

In the example from Figure 3, when we sample at node 3, we have that the path condition at the node is $pc = b_0 < 5 \wedge b_1 < 5$ and $\sharp(pc) = 2500$. The true branch ($b_0 < 5 \wedge b_1 < 5 \wedge x \leq 60$ with a count of 1500) will thus be taken with probability $1500/2500 = 0.6$ and the false branch will be taken with probability 0.4.

processPath calculates the probability for the path being processed and checks whether the path falls into the success, failure or grey set. Note that many of these calculations have already been performed during the `selectBranch`, and caching is used to eliminate duplication.

Again in the example from Figure 3, the paths ending at the labels A', B' and C' indicate assertion failures, and thus the probability of failure will be $1500/10000 + 1750/10000 + 2250/10000 = 0.55$. Since there are no loops in the example the rest of the paths indicate success, which will have probability 0.45.

In addition to the probability calculations, another import task performed here is to handle sampling without replacement. More specifically, how to ensure an exhaustive analysis can be done even when certain behaviors have very small probability (and thus would be hard to sample in a purely Monte Carlo fashion). We leverage the counts we store for each path condition to ensure no path is sampled twice (when we don't want replacement). Whenever a path is finished being explored, we subtract the final PC's count from all the PC counts along the current path back to the root. Note that these counts are being used by *selectBranch* to calculate the conditional probabilities at a branch, and thus it changes the distribution of the probabilities. More importantly, if a count

becomes zero, it will never be sampled. As more of the paths of the program are analyzed, counts are being propagated up the tree until the root node's count becomes zero, at which point all paths have been explored.

stoppingSearch uses either a measure of confidence based on the percentage of the input domain that has been explored, or it uses a statistical measure of confidence.

Enough confidence exists about the portion of the input domain that has been analyzed when $1 - Pr(success) + Pr(failure) < \epsilon$. If we treat grey paths separately, this means $Pr(grey) < \epsilon$. The parameter ϵ is provided by the user, and is typically very small. Note that although it is shown that *stoppingSearch* takes the path as input, in reality it just reuses the results computed during the *processPath*.

Handling nondeterminism Handling nondeterminism within the systems being analyzed has been studied in [27] in the context of scheduling choices in concurrent programs. The approach was to determine the schedule giving the highest (or lowest) reliability. More recently, an approach based on value iteration learning was presented in [52] to handle the problem in a more general fashion.

More and varied probabilistic symbolic execution Probabilistic symbolic execution was introduced in [31], where the approach was to do model counting (using LattE) on-the-fly during symbolic execution. This work was extended in [27] to collect path conditions from symbolic execution and then calculate the reliability of the code. This work also showed how usage profiles can be handled. In both of these works it is important to observe that *all* paths of the program are analyzed, and thus the probability calculations are precise.

The work in [69] was the first to apply sampling of paths during static analysis to provide a probability calculation with a certain confidence bound. They applied the approach to calculate bounds on the probability of assertions holding in the code. In [25], an approach similar to Algorithm 1 was used to also sample paths and then use Bayesian estimation and hypothesis testing. This paper also introduced the subtraction of the counts to ensure rare events can be sampled (see `processPath` above).

3.3 Computing Program Probabilities

Computing probabilities for probabilistic program analysis can usually be reduced to computing the probability of satisfying a boolean constraint over the program variables. In this section we will introduce some of the basic techniques currently used in program analysis.

To simplify the notation, we will focus on implicit probabilistic constructs, assuming the program under analysis has input variables $V = \{v_1, v_2, \ldots, v_n\}$, where v_i has domain d_i and comes with a probability distribution $\mathcal{P}_i : d_i \to [0, 1]$.

The input domain D is defined as the cartesian product of the domains d_i, while the input distribution $\mathcal{P}$ is defined as the joint distribution over all the input variables $\prod_i \mathcal{P}_i(\bar{v_i})$. Given a constraint $\phi : V \to \{true, false\}$, our goal is to compute the probability $Pr(\phi)$ of satisfying ϕ given the input domains and probability distributions. This problem is usually referred to as model counting, when the input domains are countable, or solution space quantification, when the input domains are modeled as uncountable (e.g., abstracting floating-point numbers as reals).

Exact and numeric computation

Finite domains. If the input domain is finite, the classical formulation of probability can be used to reduce the computation of $Pr(\phi)$ to a counting problem (here we assume a uniform distribution over all the possible input values, i.e., each valid input has the same probability):

$$Pr(\phi) = \frac{\sharp(\phi \wedge D)}{\sharp(D)} \tag{1}$$

where $\sharp(\cdot)$ counts the number of inputs satisfying the argument constraint; D has been overloaded to represent the finite domain as a constraint ($\sharp(D)$ is a short form for the size of the domain)[2]. For example, considering a single integer input variable x taking values between 1 and 10 uniformly, $\sharp(D) = 10$ and $\sharp(x \leq 5 \wedge D) = 5$, leading to a 0.5 probability of satisfying the constraint.

Efficient implementations of $\sharp(\cdot)$ are available for several classes of model counting problems, we focus here on linear integer arithmetic (LIA). An LIA constraint—the conjunction of linear constraints over a finite integer domain—geometrically defines a multi-dimensional lattice bounded by a convex polytope [5]. To count the number of points composing this structure, an efficient solution has been proposed by Barvinok in [4]. This algorithm is grounded on the construction of generating functions suitable for solving the counting problem in polynomial time, with respect to the number of variables and the number of constraints. Notably, besides the number of bits required to represent the numerical values, the complexity of this algorithm does not depend on the actual size of the variable domains. This makes the computation feasible for very large input domains, allowing its application to probabilistic program analysis. Several implementations of this algorithm are available, the two most popular being LattE [78] and Barvinok [80]. When disjunctions appear in the constraint, these have to be preprocessed before applying Barvinok's algorithm (e.g., using the Omega library [44]). Though this normalization increases the overall complexity of model counting, several optimizations can mitigate the computational effort required. Barvinok's algorithm has been used for probabilistic program analysis in [31, 27, 24].

[2] More precisely, Equation (1) represents the probability of satisfying the constraint ϕ conditioned on the fact that the input is within the prescribed domain D.

The counting function $\sharp(\cdot)$ can often be stated as a boolean satisfiability problem. The problem of counting the number of distinct truth assignments for a propositional formula is called #SAT, or propositional model counting. There are a number of tools that can efficiently solve many cases of #SAT, including sharpSAT [76] and Cachet [68].

Other finite domains, such as bounded data structures [24] and regular languages [53, 2], are active topics of study in applied model counting.

Floating-point numbers. Floating-point numbers are usually abstracted as real numbers for analysis purposes. Computing the probability of satisfying a constraint over reals requires refining Equation 1 to cope with the density of the domain. In particular, the counting function $\sharp(\phi)$ has to be replaced by the integration of an indicator function on ϕ, i.e., a function returning 1 for all the inputs satisfying ϕ [9]. This integration can be performed exactly only for a limited number of cases—those where symbolic integration is possible. For all the other cases, only numerical integration is possible. A number of commercial and open-source tools can be used for this purpose, however, 1) numerical computations are accurate only up to a certain bound, and 2) they do not scale when the cardinality of the input domain grows. In the latter case, sampling-based methods are preferable.

Handling input distributions. For finite domains, we assume, without loss of generality, the input distribution to be specified on a finite partition $D^1, D^2, \ldots, D^n$ of the input domain D (i.e., $\cup_i D^i \equiv D$ and $D^i \cap D^j \neq \emptyset \implies i = j$) via the probability function $Pr(D^i)$. We assume elements within the same set D^i to have the same probability. The case of uniform distribution described so far corresponds to the partition with cardinality 1, i.e., the whole domain.

Since the elements of the partition are disjoint by construction, we can exploit the law of total probability to extend Equation (1) to include the information about the input distribution:

$$Pr(\phi) = \sum_i \frac{\sharp(\phi \wedge D^i)}{\sharp(D^i)} \cdot Pr(D^i) \tag{2}$$

where D^i has again been overloaded to represent the constraint of an element belonging to D^i.

Formalizing the input distribution on a finite partition of the input domain is general enough to capture every valid distribution on the inputs, including possible correlations or functional dependencies among the input variables. However, the finer the specification of the input distribution, the more complex the computation of Equation (2), which, in the worst case, may require the computation of $|D|$ summands [9]. While this worst case is unlikely to occur (partially due to the optimization strategies described later), more efficient probability computations are possible which employ distribution-aware sampling-based methods, described in the next section.

Sampling-based methods Exact methods can suffer from two main limitations: 1) generality with respect to input domains and constraint classes and 2) scalability, either due to the intrinsic complexity of the algorithm used or to the discretization of the input distributions. In many cases, sampling-based methods may be used to leverage both limitations.

In this section we will present sampling-based methods for quantifying the probability of satisfying arbitrarily complex floating-point constraints. We will briefly discuss how to generalize to different domains at the end of the section.

Sampling-based methods estimate the probability of satisfying a given constraint using a Monte Carlo approach [66]. For simplicity, we will focus on the simplest, though general, method suitable for our purpose: hit-or-miss Monte Carlo. We reference more advanced methods later in the section.

Hit-or-miss Monte Carlo. For the sake of explanation, we will assume a uniform input distribution over bounded, real domains. Consider for example the constraint $x \leq -y \wedge y \leq x$, where both $x, y \in [-1, 1] \cap \mathbb{R}$. Given an input x, y, the constraint would either be satisfied or violated. In probabilistic terms, this can be seen as a Bernoulli experiment, i.e., an experiment having only two mutually exclusive outcomes, *true* or *false*, where the probability of the *true* outcome is the parameter p of a Bernoulli distribution [63] (the probability of the *false* outcome is in turn $1 - p$). Our goal is to estimate the parameter p, from n random samples over the input domain.

Figure 4 plots the solution space for the example constraint (x and y on the x- and y-axis, respectively); the value p we aim to estimate is the ratio between the shadowed area, enclosing all the points satisfying the constraint, an the input domain (i.e., the outer box).

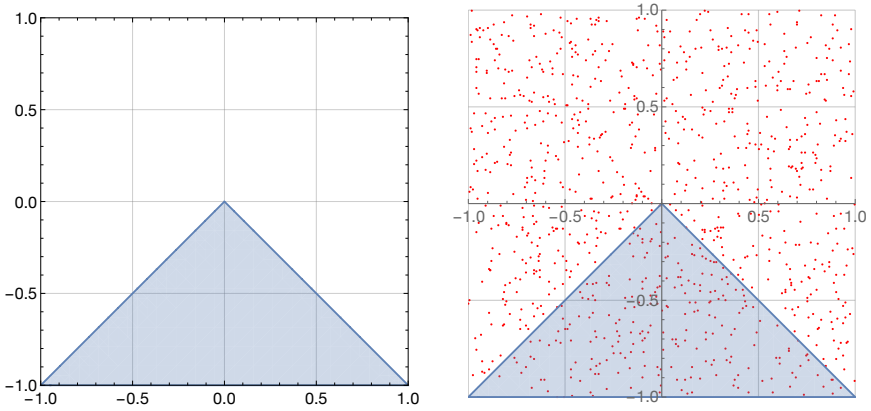

Fig. 4. Sampling-based solution space quantification for $x \leq -y \wedge y \leq x$, $x, y \in [-1, 1]$.

The hit-or-miss Monte Carlo method consists in taking n independent random samples uniformly within the domain; if a sample s_i satisfies the constraint,

we assign $s_i = 1$, otherwise, $s_i = 0$. This process is formally called a Binomial experiment with n samples. The maximum likelihood estimate for p is then $\hat{p}$ [63]:

$$\hat{p} = \frac{\sum_{i=1}^{n} s_i}{n} \qquad \sigma(\hat{p}) = \sqrt{\frac{\hat{p} \cdot (1 - \hat{p})}{n}} \tag{3}$$

The right part of Equation (3) shows the standard deviation σ of $\hat{p}$ [63]. The standard deviation is an index of the convergence of the estimate. Notably, it decreases with the square root of the number of samples; when the number of samples grows to infinity, the standard deviation goes to 0, making the estimation converge to the actual value of p.

Despite the convergence of $\hat{p}$ to p can be proved only in the limit, given the value of $\hat{p}$, its standard deviation σ, and a desired confidence level $0 < \alpha < 1$, it is possible to define a confidence interval for the unknown value p. In particular:

$$Pr\Big(\hat{p} - z_{\frac{\alpha}{2}} \cdot \sqrt{\frac{\hat{p} \cdot (1 - \hat{p})}{n}} \leq p \leq \hat{p} + z_{\frac{\alpha}{2}} \cdot \sqrt{\frac{\hat{p} \cdot (1 - \hat{p})}{n}}\Big) = 1 - \frac{\alpha}{2} \tag{4}$$

where $z_{\frac{\alpha}{2}}$ is the $1 - \frac{\alpha}{2}$ quantile of the standard Gaussian distribution [63].

Equation (4) is constructed using the central limit theorem, under the assumption that a large number of samples n have been collected (as a rule of thumb, hundreds of samples or more are almost surely a good fit for this assumption). The width of the interval, which is an index of the accuracy of the estimate, can be arbitrarily reduced by increasing the number of samples n; thus, Equation (4) can be used as stopping criteria for the estimation process.

In our example, in a run with $n = 10000$ samples, we obtained $\hat{p} = 0.2512$ with standard deviation $\sigma(\hat{p}) = 0.00433703$; thus, with 99% confidence, we can conclude $p \in [0.248126, 0.254274]$. From Figure 4, it is immediate to calculate that $p = 0.25$, which falls within the computed interval.

It can be noted that hit-or-miss methods may require a large number of samples to converge to a high accuracy (small interval). This is even worse when the actual value of p close to its extremes (0 or 1). Significant improvements on the convergence rates can be achieved with more complex sampling procedures, including the use of quasi-Monte Carlo sampling [66], or, when p is close to its extremes, importance sampling, Markov Chain Monte Carlo, or slice sampling [7]; some of these methods have successfully been used for similar problems in probabilistic model checking [41, 42, 51]. More accurate confidence intervals can also be used as stopping criteria. The interval in Equation (4) is indeed conservative and does not exploit all the information in the estimator because of the approximation via the central limit theorem. More precise intervals have been proposed in statistics [63]; in probabilistic model checking, the most commonly used is Chernoff-Hoeffding's bound [38, 39, 51]. Bayesian estimators can also be used, allowing for the inclusion of prior knowledge on the expected result (when available) [65, 32]; Bayesian methods demonstrated a faster convergence rate in many probabilistic verification problems [82]. Finally, a hybrid approach exploiting interval constraint propagation for a compositional solution of the estimation problem has been proposed for probabilistic program analysis in [9, 8].

Distribution-aware sampling. The hit-or-miss Monte Carlo method we described offers a straightforward way to handle input distributions. Assuming the distribution of the inputs is known, the samples for the Binomial experiment can be generated from it.

Efficient sampling algorithms exist for the most common continuous and discrete distributions, with off-the-shelf implementations for several programming languages (e.g., [75] for Java). A comprehensive survey of random number generation is beyond the scope of this paper (the interested reader can refer, e.g., to [33]). We will instead focus on one of the simplest and most general techniques for this task: *inverse CDF sampling*.

Assume our goal is to take a sample from a distribution D, e.g., a Gaussian distribution describing the inputs received by a temperature sensor. This distribution has a cumulative distribution function $CDF_D(x)$ representing the probability of observing a value less than or equal than x [63]. The value of the CDF is bounded between 0 and 1, for $x \to -\infty$ and $x \to \infty$, respectively. Furthermore, assuming every possible outcome has a strictly positive probability, as it is the case for most distributions used in practice, the CDF is strictly monotonic and invertible; let us denote its inverse $CDF_D^{-1}(\cdot)$.

Inverse CDF sampling reduces sampling from D to sampling from a Uniform distribution via the following three steps:

1. generate a random sample u from the Uniform distribution on $[0, 1]$
2. find the value x such that $CDF_D(x) = u$, i.e., $CDF_D^{-1}(u)$
3. return x as the sample from D

For example, to generate a sample from a Gaussian distribution $\mathcal{N}(10, 3)$, we first generate a sample u from the uniform distribution in $[0, 1]$, let's say 0.83; then, we compute $CDF^{-}1_{\mathcal{N}(10,3)}(0.83) = 12.8625$, which is our sample input.

The computation of the CDF and its inverse is efficient for most univariate distributions used in practice, and implementations are available for all common programming languages. Multivariate distributions are usually more challenging, with only a few cases allowing direct solutions. Nonetheless, more complex computation methods exist (e.g., Gibbs sampling [67]), covering most of the practically useful distributions. Multivariate distributions are indeed useful to capture statistical dependence among input variables, whether this is know in the application domain or inferred from the data. Finally, the discretization method described in Section 3.3 remains a viable general, approximate solution; however, distribution-aware sampling provide a significantly better scalability, especially when high accuracy is required [8].

Beyond numerical domains. Sampling-based methods are theoretically applicable for any input domain, provided a procedure for generating unbiased samples (according to the input distribution) is available. Solutions have been proposed for model counting of SAT problems (e.g., in [14, 6, 56], also with distribution-aware approaches [13]) and SMT problems (e.g., in [15]), while stochastic grammars can be used to generate random strings according to specified distributions [30].

4 Future Directions

In this paper we have provided a survey on work to adapt two powerful program analysis frameworks, data flow analysis and symbolic execution, to incorporate probabilistic reasoning. This work has already motivated exciting advances in model counting and solution space quantification as discussed in Section 3.3.

As in other areas of program analysis a mutually reinforcing cycle of algorithm and tool development coupled with applications of analyses is poised to spur further advances. We believe that efforts to focus probabilistic program analyses techniques on applications will reveal new opportunities for adapting algorithms to be more efficient and effective and that this will, in turn, inspire researchers to identify additional applications of these techniques. Towards this end we describe several areas where application of probabilistic program analyses has potential and identify opportunities for cross-fertilization among probabilistic analysis techniques.

1. Program understanding has been touched on in [31] and [24] where errors are found by observing unexpected probabilities for certain behaviors. This provides a means of quantifying the notion of "bugs as deviant behavior" that underlies much work on fault detection. While numeric characterizations of distributions may be difficult for developers to interpret, visualizations of those distributions might allow them to spot unexpected patterns to focus their attention on.
2. Probabilistic symbolic execution is particularly well-suited for quantifying the difference between two versions of a program [26]. This makes it an ideal approach to rank how close a program is to a given oracle program, which has applications in mutation analysis, program repair, approximate computing or even in marking student assignments. Note that this provides a route to a semantic ranking of programs as opposed to more syntactic rankings, e.g., by measuring the shared syntactic structure.
3. Probabilistic programming is becoming very popular [35], but the current approaches mainly focus on sampling, whereas a more accurate approach would be to use probabilistic symbolic execution. To achieve this, symbolic execution must be extended to support `observe(e)` statements which condition input on a boolean expression, e, by aborting the path if the expression is false and renormalizing the output distribution. Most existing symbolic execution frameworks already support `assume` and `assert` statements to check and enforce predicates at program points. Extending this to support `observe` requires that the probability estimates of aborted paths be accumulated to permit renormalization at the end of the symbolic execution. We note that relative to existing probabilistic symbolic execution approaches this adds negligable overhead.
4. Probabilistic programs, in the sense of [35], can be used to define a distribution. This means that they could be a useful means of summarizing probability information for modular probabilistic program analysis. A program could define the input distribution, and the output distribution could then

be converted to a program—the pair would form a probabilistic contract of sorts.

5. It would be interesting to explore the extent to which the computation of branch probabilities—which annotate models in tools like like PRISM [50] and PASS [37]—could be achieved, in part, by using path condition calculation and solution space quantification techniques drawn from probabilistic symbolic execution.
6. Hybrid approaches that mix probabilistic symbolic execution and data flow seem promising. The unanalyzed portion of a program's symbolic execution tree defines a "residual" program. If that program can be extracted, via techniques like slicing, then it could be encoded for analysis with data flow techniques. The results of the precise-but-slow, and faster-but-less-precise, analysis, could then be combined.

References

1. Adje, A., Bouissou, O., Goubault-Larrecq, J., Goubault, E., Putot, S.: Static analysis of programs with imprecise probabilistic inputs. In: Verified Software: Theories, Tools, Experiments, pp. 22–47. Springer (2014)
2. Aydin, A., Bang, L., Bultan, T.: Automata-based model counting for string constraints. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. pp. 255–272 (2015)
3. Bagnara, R., Hill, P.M., Zaffanella, E.: The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming 72(1), 3–21 (2008)
4. Barvinok, A.I.: A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. Mathematics of Operations Research 19(4), 769–779 (1994)
5. de Berg, M.: Computational Geometry: Algorithms and Applications. Springer (2008)
6. Biere, A., van Maaren, H.: Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, IOS Press (2009)
7. Bishop, C.: Pattern Recognition and Machine Learning. Information Science and Statistics, Springer (2006)
8. Borges, M., Filieri, A., d'Amorim, M., Păsăreanu, C.S.: Iterative distribution-aware sampling for probabilistic symbolic execution. In: Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ESEC/FSE 2015, ACM (2015)
9. Borges, M., Filieri, A., d'Amorim, M., Păsăreanu, C.S., Visser, W.: Compositional solution space quantification for probabilistic software analysis. SIGPLAN Not. 49(6), 123–132 (Jun 2014), http://doi.acm.org/10.1145/2666356.2594329
10. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys (CSUR) 24(3), 293–318 (1992)
11. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)

12. Chakarov, A., Sankaranarayanan, S.: Expectation invariants for probabilistic program loops as fixed points. In: Static Analysis, pp. 85–100. Springer (2014)
13. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: Distribution-aware sampling and weighted model counting for sat. In: Twenty-Eighth AAAI Conference on Artificial Intelligence (2014)
14. Chakraborty, S., Meel, K., Vardi, M.: A scalable approximate model counter. In: Schulte, C. (ed.) Principles and Practice of Constraint Programming, Lecture Notes in Computer Science, vol. 8124, pp. 200–216. Springer Berlin Heidelberg (2013)
15. Chistikov, D., Dimitrova, R., Majumdar, R.: Approximate counting in smt and value estimation for probabilistic programs. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 9035, pp. 320–334. Springer Berlin Heidelberg (2015)
16. Claret, G., Rajamani, S.K., Nori, A.V., Gordon, A.D., Borgström, J.: Bayesian inference using data flow analysis. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 92–102. ACM (2013)
17. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT press (1999)
18. Clarke, L., et al.: A system to generate test data and symbolically execute programs. Software Engineering, IEEE Transactions on (3), 215–222 (1976)
19. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 238–252. ACM (1977)
20. Cousot, P., Monerau, M.: Probabilistic abstract interpretation. In: Programming Languages and Systems, pp. 169–193. Springer (2012)
21. Di Pierro, A., Wiklicky, H.: Probabilistic data flow analysis: a linear equational approach. arXiv preprint arXiv:1307.4474 (2013)
22. Dwyer, M.B.: Unifying testing and analysis through behavioral coverage. In: Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on. pp. 2–2. IEEE (2011)
23. Esparza, J., Gaiser, A.: Probabilistic abstractions with arbitrary domains. In: Static Analysis, pp. 334–350. Springer (2011)
24. Filieri, A., Frias, M., Păsăreanu, C., Visser, W.: Model counting for complex data structures. In: Proceedings of the 2015 International SPIN Symposium on Model Checking of Software. ACM (2015)
25. Filieri, A., Păsăreanu, C.S., Visser, W., Geldenhuys, J.: Statistical symbolic execution with informed sampling. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 437–448. ACM (2014)
26. Filieri, A., Păsăreanu, C.S., Yang, G.: Quantification of software changes through probabilistic symbolic execution. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) - Short Paper (November 2015)
27. Filieri, A., Păsăreanu, C.S., Visser, W.: Reliability analysis in symbolic pathfinder. In: Proceedings of the 2013 International Conference on Software Engineering. pp. 622–631. ICSE '13, IEEE Press, Piscataway, NJ, USA (2013), http://dl.acm.org/citation.cfm?id=2486788.2486870
28. Fink, S., Dolby, J.: Wala–the tj watson libraries for analysis (2012)
29. Fosdick, L.D., Osterweil, L.J.: Data flow analysis in software reliability. ACM Computing Surveys (CSUR) 8(3), 305–330 (1976)
30. Fu, K., Huang, T.: Stochastic grammars and languages. International Journal of Computer and Information Sciences 1(2), 135–170 (1972)

31. Geldenhuys, J., Dwyer, M.B., Visser, W.: Probabilistic symbolic execution. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. pp. 166–176. ISSTA 2012, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2338965.2336773
32. Gelman, A., Carlin, J., Stern, H., Dunson, D., Vehtari, A., Rubin, D.: Bayesian Data Analysis, Third Edition. Chapman & Hall/CRC Texts in Statistical Science, Taylor & Francis (2013)
33. Gentle, J.: Random Number Generation and Monte Carlo Methods. Statistics and Computing, Springer New York (2013)
34. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: ACM Sigplan Notices. vol. 40, pp. 213–223. ACM (2005)
35. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: Proceedings of the on Future of Software Engineering. pp. 167–181. FOSE 2014, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2593882.2593900
36. Graf, S., Saidi, H.: Computer Aided Verification: 9th International Conference, CAV'97 Haifa, Israel, June 22–25, 1997 Proceedings, chap. Construction of abstract state graphs with PVS, pp. 72–83. Springer Berlin Heidelberg (1997)
37. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: Pass: Abstraction refinement for infinite probabilistic models. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 353–357. Springer (2010)
38. Hoeffding, W.: Probability inequalities for sums of bounded random variables. Journal of the American statistical association 58(301), 13–30 (1963)
39. Hrault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate probabilistic model checking. In: Steffen, B., Levi, G. (eds.) Verification, Model Checking, and Abstract Interpretation, Lecture Notes in Computer Science, vol. 2937, pp. 73–84. Springer Berlin Heidelberg (2004)
40. Jamrozik, K., Fraser, G., Tillman, N., De Halleux, J.: Generating test suites with augmented dynamic symbolic execution. In: Tests and Proofs, pp. 152–167. Springer (2013)
41. Jegourel, C., Legay, A., Sedwards, S.: Cross-entropy optimisation of importance sampling parameters for statistical model checking. In: Madhusudan, P., Seshia, S. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 7358, pp. 327–342. Springer Berlin Heidelberg (2012)
42. Jegourel, C., Legay, A., Sedwards, S.: Importance splitting for statistical model checking rare properties. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 8044, pp. 576–591. Springer Berlin Heidelberg (2013)
43. Jones, C.: Probabilistic non-determinism (1990)
44. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The omega calculator and library. College Park, MD 20742, 18 (1996)
45. Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 194–206. ACM (1973)
46. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (1976)
47. Kozen, D.: Semantics of probabilistic programs. Journal of Computer and System Sciences 22(3), 328–350 (1981)
48. Kozen, D.: A probabilistic pdl. In: Proceedings of the fifteenth annual ACM symposium on Theory of computing. pp. 291–297. ACM (1983)

49. Kwiatkowska, M., Norman, G., Parker, D.: Advances and challenges of probabilistic model checking. In: 2010 48th Annual Allerton Conference on Communication, Control, and Computing (Allerton)
50. Kwiatkowska, M., Norman, G., Parker, D.: Prism 4.0: Verification of probabilistic real-time systems. In: Computer aided verification. pp. 585–591. Springer (2011)
51. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Rou, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification, Lecture Notes in Computer Science, vol. 6418, pp. 122–135. Springer Berlin Heidelberg (2010)
52. Luckow, K., Păsăreanu, C.S., Dwyer, M.B., Filieri, A., Visser, W.: Exact and approximate probabilistic symbolic execution for nondeterministic programs. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. pp. 575–586. ACM (2014)
53. Luu, L., Shinde, S., Saxena, P., Demsky, B.: A model counter for constraints over unbounded strings. SIGPLAN Not. 49(6), 565–576 (Jun 2014), http://doi.acm.org/10.1145/2666356.2594331
54. Mardziel, P., Magill, S., Hicks, M., Srivatsa, M.: Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation. Journal of Computer Security 21(4), 463–532 (2013)
55. McDonald, J.B.: Some generalized functions for the size distribution of income. Econometrica: Journal of the Econometric Society pp. 647–663 (1984)
56. Meel, K.S.: Sampling techniques for boolean satisfiability. CoRR abs/1404.6682 (2014), http://arxiv.org/abs/1404.6682
57. Mehofer, E., Scholz, B.: A novel probabilistic data flow framework. In: Compiler Construction. pp. 37–51. Springer (2001)
58. Monniaux, D.: Abstract interpretation of probabilistic semantics. In: Static Analysis, pp. 322–339. Springer (2000)
59. Monniaux, D.: Backwards abstract interpretation of probabilistic programs. In: Programming Languages and Systems, pp. 367–382. Springer (2001)
60. Monniaux, D.: Abstract interpretation of programs as markov decision processes. Science of Computer Programming 58(1), 179–205 (2005)
61. Morgan, C., McIver, A., Seidel, K.: Probabilistic predicate transformers. ACM Transactions on Programming Languages and Systems (TOPLAS) 18(3), 325–353 (1996)
62. Păsăreanu, C.S., Rungta, N.: Symbolic pathfinder: symbolic execution of java bytecode. In: Proceedings of the IEEE/ACM international conference on Automated software engineering. pp. 179–180. ACM (2010)
63. Pestman, W.R.: Mathematical statistics: an introduction, vol. 1. Walter de Gruyter (1998)
64. Ramalingam, G.: Data flow frequency analysis. In: ACM SIGPLAN Notices. vol. 31, pp. 267–277. ACM (1996)
65. Robert, C.: The Bayesian Choice: From Decision-Theoretic Foundations to Computational Implementation. Springer Texts in Statistics, Springer (2007)
66. Robert, C., Casella, G.: Monte Carlo statistical methods. Springer Science & Business Media (2013)
67. Robert, C.P., Casella, G.: Monte Carlo Statistical Methods. Springer-Verlag New York, Inc. (2005)
68. Sang, T., Beame, P., Kautz, H.: Heuristics for fast exact model counting. In: Theory and Applications of Satisfiability Testing. pp. 226–240. Springer (2005)

69. Sankaranarayanan, S., Chakarov, A., Gulwani, S.: Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 447–458. PLDI '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2491956.2462179
70. Schmidt, D.A.: Data flow analysis is model checking of abstract interpretations. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 38–48. ACM (1998)
71. Sen, K., Marinov, D., Agha, G.: Cute: A concolic unit testing engine for c (2005)
72. Smith, M.J.: Probabilistic abstract interpretation of imperative programs using truncated normal distributions. Electronic Notes in Theoretical Computer Science 220(3), 43–59 (2008)
73. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: Bitblaze: A new approach to computer security via binary analysis. In: Information systems security, pp. 1–25. Springer (2008)
74. Thakur, A., Elder, M., Reps, T.: Bilateral algorithms for symbolic abstraction. In: Static Analysis, pp. 111–128. Springer (2012)
75. The Apache Software Foundation: Commons math. http://commons.apache.org/proper/commons-math/, accessed: 2014-12-16
76. Thurley, M.: sharpsat–counting models with advanced component caching and implicit bcp. In: Theory and Applications of Satisfiability Testing-SAT 2006, pp. 424–429. Springer (2006)
77. Thurow, L.C.: Analyzing the american income distribution. The American Economic Review pp. 261–269 (1970)
78. UC Davis, Mathematics: Latte. —http://www.math.ucdavis.edu/ latte—
79. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot-a java bytecode optimization framework. In: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research. p. 13. IBM Press (1999)
80. Verdoolaege, S.: Software package barvinok. 2004. Electronically available at http://freshmeat.net/projects/barvinok
81. Wachter, B., Zhang, L.: Best probabilistic transformers. In: Verification, Model Checking, and Abstract Interpretation. pp. 362–379. Springer (2010)
82. Zuliani, P., Platzer, A., Clarke, E.M.: Bayesian statistical model checking with application to simulink/stateflow verification. In: Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control. pp. 243–252. HSCC '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1755952.1755987