Probabilistic Program Analysis

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

[1] University of Nebraska – Lincoln
[2] Imperial College London
[3] Carnegie Mellon Silicon Valley and 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 survey 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 themes that relate and distinguish a 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, Markov Decision Processes

1 Introduction

Static program analyses 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 understood that 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 details of program execution (e.g., the specific input values provided to the program). This tolerance of uncertainty allows analyses to provide useful information when users don't know exactly how a program will be used.

Static analyses model uncertainty 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, 18], and to capture sets of data values that constitute reachable states via predicate abstraction [36]. Nondeterministic choice is another widely used approach, for instance, in modeling branch decisions in data flow analysis. 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 [57, 82]. With such a distribution the program can now be viewed as a *probabilistic program* and, beginning with Kozen's seminal work in the early 1980s, the semantics of such programs has long been studied [47, 48, 44, 62].

For non-probabilistic programs, it was just over six years from Floyd's foundational work on program semantics [28] to Kildall's widely-applicable static analysis framework [45]. Sophisticated extensions of Kildall's work are prevalent today, e.g., [51, 52], and form the basis for modern software development environments. For probabilistic programs, however, the development of static analysis frameworks has taken decades and they have not yet reached the level of applicability of their non-probabilistic counterparts.

What would such analyses have to offer? Researchers have explored the use of probabilistic analysis results to assess the security of software components [56] and to measure side-channel leakage [65, 3], to assess program reliability [26], to measure program similarity [31], to characterize fault propagation [63], and to characterize the coverage achieved by an analysis technique [21]. We believe that there are many more applications for cost-effective and widely-applicable static analysis frameworks for probabilistic programs.

In recent years, the term "probabilistic program" has been generalized beyond Kozen's definition in which programs draw inputs from probability distributions. This more general setting permits the conditioning of program behavior by allowning certain program runs to be rejected. These programs can be viewed as expressing computations over probability distributions rather than inputs drawn from a distribution. While recent work has just begun to explore the foundations of analysis for this more general setting [15, 35], in this paper we consider Kozen's original definition and analysis frameworks targetting such programs.

More specifically, we survey work on adapting data flow analysis and symbolic execution to use information about input distributions. We begin with background that provides basic definitions related to static analysis and probabilistic models. Section 2.2 exposes 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. While we focus on analysis of imperative programs, we note that principles exposed in our survey apply to analysis frameworks for functional programs as well. 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.

2 Overview

2.1 Scope and Background

This paper focuses 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. For the example, we might be interested in reasoning about the lack of assertion violations.

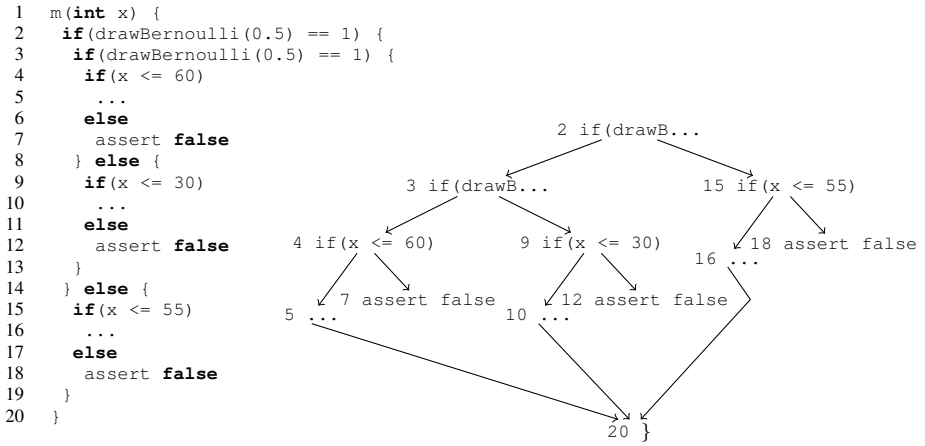

Fig. 1. Example: Source Code (left) and Control Flow Graph (right)

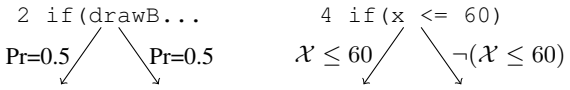

Fig. 2. Probabilistic Choice (left) and Symbolic Choice (right)

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 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 left side of Figure 2 shows edge probabilities that reflect the outcome of the Bernoulli draw on line 2. Thus the probability of taking the *then* branch is 0.5; the probability of taking the *else* branch is also 0.5. In addition, we will consider models where the choice of successor is defined by the semantics of the branch condition. The right side of Figure 2 shows a logical condition that reflects the fact that the value of parameter x must be less than or equal to 60 for control to traverse the true branch at line 4.

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 [79], binary decision diagrams [10], or custom representations [2]. For example, the set of negative integer values can be defined by a predicate $lt0 \equiv \lambda x.x < 0$ which returns true for all values in the set. Logical combinations of such predicates can be used to define an abstract domain, $\mathcal{A}$, whose elements describe sets of possible states of the program.

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.

Data Flow Analysis Data flow analysis [45] provides a framework for computing properties shared by sets of program traces reaching a program state. 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 [16] is a popular verification technique which 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 [18]. In fact, it is now well-understood that data flow analysis can be viewed as model checking of abstract interpretations [75].

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).

Data flow analysis tools and toolkits exist for popular languages, e.g., [84, 27, 52], and have been used primarily for program optimization and verifying program conformance with assertional specifications.

Symbolic Execution Like data flow analysis, symbolic execution [46, 17] 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 an execution.

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 path through the control flow graph, when symbolic execution reaches the final branch it reords the condition $\mathcal{X} \leq 60$, where $\mathcal{X}$ models the unknown value of input x. This condition describes the set of input values that trigger execution of this path—as long as both Bernoulli trials yield a value of 1.

In practice symbolic execution computes an underapproximation of program behavior. Programs with looping behavior that is determined by input values may result in an

infinite symbolic execution. 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, 76, 78], called concolic execution, address this problem by replacing problematic constraints with equality constraints between variables and values collected while executing the program along the trace.

Symbolic execution tools and toolkits exist for many popular languages [66, 34, 41, 11] and have been used primarily for test generation and fault detection.

Probabilities and Probabilistic Models There is an enormous literature on probabilistic reasoning and statistics that can be brought to bear in program analysis. In this paper, we consider two types of discrete time probabilistic models: Markov chains and Markov decision processes [69].

Both models rely on the concept of a probability distribution. A *probability distribution* is a function that provides the probability of occurrence of different possible outcomes in an experiment. The sum of the probabilities for all outcomes is 1.

A Markov chain is a labeled transition system that, given some state, defines the probability of moving to another state, according to a probability distribution. The probability of executing a sequence of states is then the product of the transition probabilities between the states. 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 $\mathtt{m}$ and it would replace all nondeterministic choices in the CFG with probabilistic choices.

There are many situations where the removal of nondeterministic choices 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.

Including nondeterminism in a probabilistic state transition model yields a Markov decision process (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 *policy* for the MDP. Given a policy, an MDP reduces to a Markov chain.

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, our intention is to expose key similarities and differences between families of approaches and, in so doing, 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.

It is possible to transform explicit probabilistic constructs 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 each auxiliary input would then be specified as a set of pairs, $\{\ldots, (-1, 0), (0, 0.5), (1, 0.5), (1, 0), \ldots\}$ where the first component defines a value and the second defines its probability.

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 values: a quantity that approximates the probability of each value in the set and the number of values in the set. Early 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 [56].

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 survey, 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.

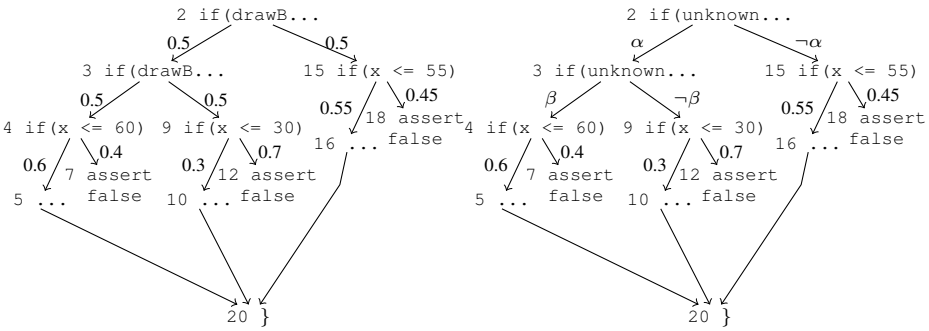

Fig. 3. Probabilistic CFG (left) and MDP (right)

3 Probabilistic Approaches to Program Analysis

This section introduces probabilistic variants of data flow analysis and symbolic execution. Several of these variants either exploit, or could be adapted to exploit, recent advances in methods for quantifying or estimating constraint solution spaces which we discuss at the end of this section.

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 upon which it is based.

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 left side of Figure 3 shows the probabilistic CFG for the example of Figure 1, given that input x is uniformly distributed in the range $[1, 100]$. This program is simple enough that the branch probabilities can be easily computed—because the probabilities for each branch along a path are independent.

The sum of all probabilities leaving any control flow node must be 1 (except for the exit node). The probability of executing a path is the product of edge probabilities along the path. Thus, the probability of reaching a program state is the sum of the probabilities of traces that reach that state.

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 [70]. Instead of the usual semilattice with an idempotent meet operation, a non-idempotent addition operator is used. The usual 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 sum-over-all-paths approach is reminiscent of the approach taken in probabilistic model checking of Markov chains. In the latter approach, a system of equations is formulated whose solution yields the probability of some property holding—so-called *quantitative* properties in PRISM [49]. Ramalingam's analysis effectively solves an equivalent system of equations.

These techniques rely on being able to annotate branch decisions in the program with probabilities. When the input distributions governing branches is unknown or when the branches along a path are dependent the techniques described above cannot be applied. The discussions below on non-determinism and in Sections 3.2 and 3.3 describe ways to address this problem.

Abstract Data Probabilities Researchers have incorporated probabilistic information directly into the semantics of a program and then abstracted over those semantics [59, 77, 19] to construct probabilistic data flow analyses. 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 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 [59, 60], 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 a 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 values mapped by that domain. Consider a probabilistic abstract state, $pa \in \mathcal{A} \times [0, 1]$, an upper approximation of the probability of a value v in that state is given by $Pr(v) \leq \sum_{(a,w) \in pa \wedge c \in \gamma(a)} w$, where γ is maps a symbolic abstraction to the set of 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, e.g., the extension of $+^{\#}$ to account for $\mathcal{A}_p$, but it means that additional work is required to compute the probability of a property holding. In essence, this 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. 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 probabilistic abstract transformer need only apply that filter to the first component of the tuple (the elements of the underlying abstract domain), leaving 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 `if(x<0){...}`, then applying the filter on the true branch results in $([-5, -1], 0.1)$ and applying the filter on the false branch results in $([0, 5], 0.1)$. The space is reduced; the weights remain the same.

Finally, reaching fixpoints for rich probabilistic abstract domains appears to require widening [59, 22] to be cost-effective. These can be challenging to define and, generally, lead to a loss in precision.

Probability for Abstract States Computing bounds on the probability of a state property has been well-studied. Di Pierro et al. [20] 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 when scaling to real programs. 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, it is natural to use nondeterminism to account for the uncertainty in program behavior.

In Monniaux's semantics [61] 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. In the MDP on the right side of Figure 3 α and β are used to denote the outcomes of nondeterministic choices—modeling for instance unknown branch conditions—and their values comprise the MDP policy. There are three policies for this example: (α, β), $(\alpha, \neg\beta)$, and $(\neg\alpha)$. For a given program state, we can formulate an alternating game that seeks to determine values for α and β which maximize (or minimize) the probability of reaching that state. Probabilistic model checkers such as PRISM and PASS use this approach to formulate MDP-based analyses.

The reachability of line 5 in the MDP is only possible under the policy (α, β). Thus, that policy maximizes the probability of reaching that state at 0.6—the product of the probabilities along the path. Any other policy will minimize the probability of reaching line 5 at 0. Bounding the probability of violating an assertion in a call to m requires considering all three policies. The maximal probability is 0.7 under policy $(\alpha, \neg\beta)$, whereas the minimal probability is 0.4 under policy (α, β).

Abstract interpretation can be applied to the data states in such approaches [50, 86, 22] to improve efficiency. These abstractions are, however, 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.

Theoretical advances in the analysis of stochastic processes [68] and coalgebraic semantics [38, 64] may provide new pathways towards the definition of more advanced analysis methods that combine nondeterministic and probabilistic choice.

3.2 Probabilistic Symbolic Execution

Probabilistic symbolic execution extends traditional symbolic execution with the ability of computing probabilities of reaching certain traget states in a program. The computation is based on *quantifying* the solution spaces of the path conditions computed by symbolic execution.

We illustrate probabilistic symbolic execution using the example in Figure 4, where we introduce variables b_0 and b_1 to model the two `drawBernoulli` distributions from Figure 1; the domains of those variables consist of 10 values and the tests check for half of the domain, corresponding 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. Figure 4 illustrates the six symbolic paths generated by a symbolic execution of the example program. The path condition describing each path is the conjunction 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.

Algorithm *pse* (Algorithm 1) illustrates probabilistic symbolic execution; it is a modification of traditional symbolic execution to process symbolic paths one at a time using procedure *symsample*. The selection of each path can be done systematically (e.g. using depth-first search as in traditional symbolic execution) or statistically, guided by branch probabilities as in [24]. Our description accomodates many of the advances in

```
// b0, b1 in 0..9
m(int x, int b0, int b1) {
 if(b0<5) {
  if(b1<5) {
   if(x <= 60)
    ...A
   else
    assert false
  } else {
   if(x <= 30)
    ...B
   else
    assert false
  }
 } else {
  if(x <= 55)
   ...C
  else
   assert false
 }
}
```

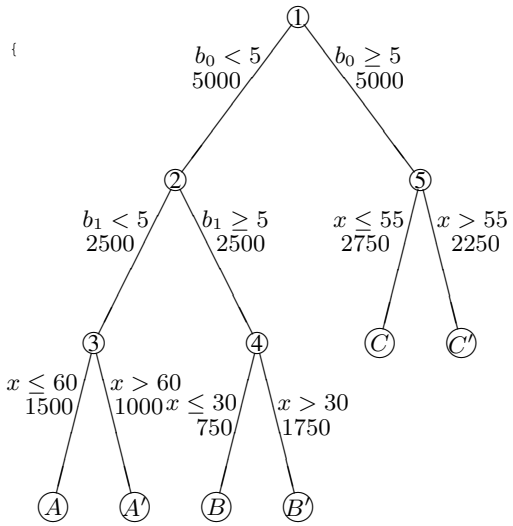

Fig. 4. Illustration of probabilistic symbolic execution

the recent literature [24, 26]. The processing of each path involves the calculation of probabilities as described in the next section.

At a high level, procedure *symsample* is called from the initial state of the program; it returns a single path which is then processed. After each path, procedure *stoppingSearch* is called to check if the analysis is complete or some other termination criterion is met, and the analysis can stop. Within procedure *symsample* we first check if the search for a path 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.

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

repeat
  $p \leftarrow symsample(l_0, m_0, true)$
  $processPath(p)$
until $stoppingSearch(p)$

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

if $stoppingPath(pc)$ **then**
  return pc
end if
while $\neg branch(l)$ **do**
  $m \leftarrow op(l)(m)$
  $l \leftarrow succ(l)$
end while
$c \leftarrow cond(l)(m)$
if $selectBranch(c, pc)$ **then**
  return $symsample(succ_t(l), m, pc \wedge c)$
else
  return $symsample(succ_f(l), m, pc \wedge \neg c)$
end if

Procedure stoppingPath uses a stopping criterion (limit on search depth) to avoid exploration of infinite or very long paths, that are due to loops conditioned on input variables. Since some paths might now be truncated before reaching a target property, we introduce three types of paths, (1) *success* paths, which reach and satisfy the target property, (2) *failure* paths, which reach and falsify the property, and (3) *grey* paths, which are truncated before reaching the property. These paths form three disjoint sets; we calculate the cumulative probability of success $Pr(success)$ (i.e., the *reliability* of the code), failure $Pr(failure)$ and grey paths $Pr(grey)$. 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).

Procedure selectBranch selects which branch to execute next; this can be done either systematically or probabilistically, according to the probability of satisfying the corresponding branch conditions. This is computed based on the number of solutions for each path condition as follows. At each branching point, we count the number of solutions for the path condition at that branching point ($\sharp(pc)$) and the number of solutions for the path condition for both branches ($\sharp(pc \wedge c)$ and $\sharp(pc \wedge \neg c)$). Assuming a uniform distribution of the inputs, the probability for the true branch is then simply $Pr(succ_t(l)) = \sharp(pc \wedge c)/\sharp(pc)$; similarly for the false branch, $Pr(succ_f(l)) = \sharp(pc \wedge \neg c)/\sharp(pc)$. Techniques for counting the number of solutions are discussed in Section 3.3.

In the example of Figure 4, when we sample probabilistically 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.

Procedure 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 can be used to eliminate redundant work.

Again in the example of Figure 4, 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.

Furthermore, the procedure can handle sampling without replacement—to guarantee an exhaustive analysis even when certain behaviors have very small probability. In [24] we describe how we leverage the counts we store for each path condition to ensure no path is sampled twice. Whenever a path has been explored completely, we subtract the final path condition count from all the counts along the current path back up to the root. Note that these counts are being used by $selectBranch$ to calculate the conditional probabilities at a branch, and thus they change with each sample. If a count becomes zero, the corresponding branch will no longer be selected. The more paths of the program are analyzed, the more counts propagate up the tree until the root node's count becomes zero, at which point all paths have been explored.

Procedure 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 we show, for simplicity, that procedure $stoppingSearch$ takes the path as input, in practice it just reuses the results computed by procedure $processPath$.

Handling nondeterminism Handling nondeterminism within the systems being analyzed has been studied in previous work [26] 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 [54] to handle the problem in a more general fashion.

3.3 Computing Program Probabilities

Computing probabilities for probabilistic symbolic execution and other program analyses reduces to computing the probability of satisfying a boolean constraint over the program variables. This operation is performed within the $selectBranch$ function. Given the path condition PC reaching a branching point and the condition c of the conditional statement, the goal is to compute the probability of satisfying $PC \wedge c$ and, consequently, $PC \wedge \neg c$. Depending on the theory the constraints are expressed in, different techniques can be used to quantify the solution space of the constraints and, in turn, their probability of being satisfied. In this section introduce the basics of some of these techniques.

Assume 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 the Cartesian product of the domains d_i, while the input distribution $\mathcal{P}$ is the joint distribution over all the input variables $\prod_i \mathcal{P}_i(\bar{v_i})$. For a constraint $\phi : V \to \{true, false\}$, the goal is to compute the probability $Pr(\phi)$ of satisfying ϕ given the input domains and probability distributions.

Exact and numeric computation

Finite domains. If the input domain is finite, the computation of $Pr(\phi)$ reduces to a counting problem as already mentioned (assume for now all inputs are uniformly distributed, i.e. they are equally likely):

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

Here $\sharp(\cdot)$ 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[5]. 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.

The computation of $\sharp(\cdot)$ can be performed efficiently for linear integer arithmetic (LIA) constraints. A LIA constraint 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 [4]. This algorithm uses generating functions

[5] 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.

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 [31, 26, 23]. Several implementations of this algorithm are available, the most popular being LattE [83] and Barvinok [85].

Other finite domains, such as bounded data structures [23] and regular languages [55, 1], are active topics of study in applied model counting. 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 [81] and Cachet [74].

Handling input distributions. For finite domains, 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 $Pr(D^i)$. 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:

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

Here 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).

Floating-point numbers. Floating-point numbers are often 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 $\sharp(\phi)$ is 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 when symbolic integration is possible, but in general only numerical integration is possible. A number of commercial and open-source tools can be used for this purpose. However, numerical computations are accurate only up to a certain bound, and they do not scale to large cardinalities. In the latter case, sampling-based methods are preferable.

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. Sampling-based methods may be used to address these limitations.

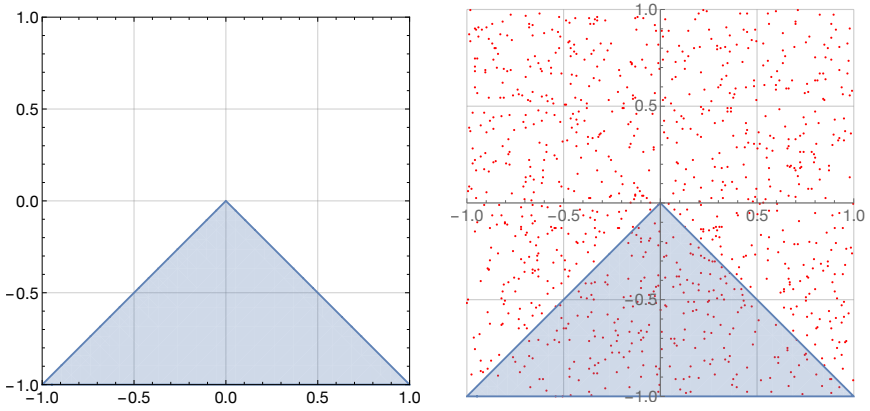

Fig. 5. Sampling-based solution space quantification for $x \leq -y \wedge y \leq x$, $x, y \in [-1, 1]$.

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 other domains at the end of the section.

Sampling-based methods estimate the probability of satisfying a given constraint using a Monte Carlo approach [72]. For simplicity, we will focus on the simplest, though general, method suitable for our purpose: hit-or-miss Monte Carlo.

Hit-or-miss Monte Carlo. We assume first a uniform input distribution over bounded, real domains (we relax this assumption later). Consider the constraint $x \leq -y \wedge y \leq x$, where both $x, y \in [-1, 1] \cap \mathbb{R}$. 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 [67] (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 5 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, and the input domain (i.e., the outer box).

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 called a Binomial experiment with n samples. The maximum likelihood estimate for p is then $\hat{p}$ [67]:

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

The right part of Equation (3) shows the standard deviation σ of $\hat{p}$ [67]. 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:

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

where $z_{\frac{\alpha}{2}}$ is the $1 - \frac{\alpha}{2}$ quantile of the standard Gaussian distribution [67].

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 5, it is easy to see that $p = 0.25$, which falls within the computed interval.

Note 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 is close to its extremes (0 or 1). Improvements on the convergence rates can be achieved using more complex sampling procedures such as quasi-Monte Carlo sampling [72], or importance sampling, Markov Chain Monte Carlo, or slice sampling [7]; some of these methods have been used in probabilistic model checking [42, 43, 53].

Further more accurate confidence intervals can also be used as stopping criteria [67]; in probabilistic model checking, the most commonly used is the Chernoff-Hoeffding's bound [40, 39, 53]. Bayesian estimators can also be used, allowing for the inclusion of prior knowledge on the expected result (when available) [71, 32]; the use of Bayesian methods led to faster convergence rate in many probabilistic verification problems [87]. Finally, a hybrid approach, that exploits interval constraint propagation and compositional solving 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: the samples for the Binomial experiment can be simply drawn from the known distribution. Efficient sampling algorithms exist for the most common continuous and discrete distributions, with off-the-shelf implementations for several programming languages (e.g., [80] for Java). A comprehensive survey of random number generation is beyond the scope of this paper (see e.g. [33]). We describe here 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 [67]. 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 [73]), 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 scales significantly better, 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 [13, 6, 58], also with distribution-aware approaches [12]) and SMT problems (e.g., in [14]), while stochastic grammars can be used to generate random strings according to specified distributions [30].

4 Conclusions and 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 developments in algorithms, tools, and applications 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. 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 [23] 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 [25]. 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. 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.
4. 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.
5. Probabilistic symbolic execution could be extended to support the more general notion of probabilistic program treated by Gordon et al. [35]. The semantics of `observe(e)` statements 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.
6. Probabilistic program analysis has many promising applications in the security domain; for example it can be used in quantitative information flow analysis [3], where the goal is to detect vulnerabilities and compute the leakage (in number of bits of the secret) using information theory metrics. A program can be viewed as a probabilistic function that maps a high security input to an *observable* output. An adversary tries to guess the secret by observing the output. The leakage of a (deterministic) program can be expressed as classical Shannon entropy: $Leakage(P) = -\sum_{i=1,n} p(o_i) \log_2 p(o_i)$, where $p(o_i)$ denotes the probability of observing o_i and can be computed with the techniques discussed in this article.

References

1. 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)
2. 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)
3. Bang, L., Aydin, A., Phan, Q., Pasareanu, C.S., Bultan, T.: String analysis for side channels with segmented oracles. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016. pp. 193–204 (2016)

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. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 123–132. ACM (2014)
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. 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)
13. 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)
14. 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, LNCS, vol. 9035, pp. 320–334. Springer (2015)
15. 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)
16. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT press (1999)
17. Clarke, L., et al.: A system to generate test data and symbolically execute programs. Software Engineering, IEEE Transactions on (3), 215–222 (1976)
18. 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)
19. Cousot, P., Monerau, M.: Probabilistic abstract interpretation. In: Programming Languages and Systems, pp. 169–193. Springer (2012)
20. Di Pierro, A., Wiklicky, H.: Probabilistic data flow analysis: a linear equational approach. arXiv preprint arXiv:1307.4474 (2013)
21. 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)
22. Esparza, J., Gaiser, A.: Probabilistic abstractions with arbitrary domains. In: Static Analysis, pp. 334–350. Springer (2011)
23. 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)

24. 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)
25. 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)
26. 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. IEEE Press (2013)
27. Fink, S., Dolby, J.: Wala–the tj watson libraries for analysis (2012)
28. Floyd, R.W.: Assigning meanings to programs. In: Mathematical Aspects of Computer Science. pp. 19–32 (1967)
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. ACM (2012)
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. ACM (2014)
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. Hasuo, I., Jacobs, B., Sokolova, A.: Generic trace theory. Electronic Notes in Theoretical Computer Science 164(1), 47 – 65 (2006)
39. Herault, 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 (2004)
40. Hoeffding, W.: Probability inequalities for sums of bounded random variables. Journal of the American statistical association 58(301), 13–30 (1963)
41. 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)
42. 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)
43. 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)
44. Jones, C.: Probabilistic non-determinism (1990)

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. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The Soot framework for Java program analysis: a retrospective. In: Cetus Users and Compiler Infrastructure Workshop. Galveston Island, TX (October 2011)
52. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (2004)
53. 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)
54. 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)
55. Luu, L., Shinde, S., Saxena, P., Demsky, B.: A model counter for constraints over unbounded strings. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 565–576. ACM (2014)
56. 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)
57. McDonald, J.B.: Some generalized functions for the size distribution of income. Econometrica: Journal of the Econometric Society pp. 647–663 (1984)
58. Meel, K.S.: Sampling techniques for boolean satisfiability. CoRR abs/1404.6682 (2014), http://arxiv.org/abs/1404.6682
59. Monniaux, D.: Abstract interpretation of probabilistic semantics. In: Static Analysis, pp. 322–339. Springer (2000)
60. Monniaux, D.: Backwards abstract interpretation of probabilistic programs. In: Programming Languages and Systems, pp. 367–382. Springer (2001)
61. Monniaux, D.: Abstract interpretation of programs as markov decision processes. Science of Computer Programming 58(1), 179–205 (2005)
62. Morgan, C., McIver, A., Seidel, K.: Probabilistic predicate transformers. ACM Transactions on Programming Languages and Systems (TOPLAS) 18(3), 325–353 (1996)
63. Murta, D., Oliveira, J.N.: A study of risk-aware program transformation. Sci. Comput. Program. 110(C), 51–77 (Oct 2015)
64. Oliveira, J.N., Miraldo, V.C.: keep definition, change category a practical approach to state-based system calculi. Journal of Logical and Algebraic Methods in Programming 85(4), 449 – 474 (2016)

65. Pasareanu, C.S., Phan, Q., Malacaria, P.: Multi-run side-channel analysis using symbolic execution and max-smt. In: IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016. pp. 387–400 (2016)
66. 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)
67. Pestman, W.R.: Mathematical statistics: an introduction, vol. 1. Walter de Gruyter (1998)
68. Puggelli, A., Li, W., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Polynomial-time verification of pctl properties of mdps with convex uncertainties. In: Proceedings of the 25th International Conference on Computer Aided Verification. pp. 527–542 (2013)
69. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons, Inc., New York, NY, USA (1994)
70. Ramalingam, G.: Data flow frequency analysis. In: ACM SIGPLAN Notices. vol. 31, pp. 267–277. ACM (1996)
71. Robert, C.: The Bayesian Choice: From Decision-Theoretic Foundations to Computational Implementation. Springer Texts in Statistics, Springer (2007)
72. Robert, C., Casella, G.: Monte Carlo statistical methods. Springer (2013)
73. Robert, C.P., Casella, G.: Monte Carlo Statistical Methods. Springer-Verlag (2005)
74. Sang, T., Beame, P., Kautz, H.: Heuristics for fast exact model counting. In: Theory and Applications of Satisfiability Testing. pp. 226–240. Springer (2005)
75. 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)
76. Sen, K., Marinov, D., Agha, G.: Cute: A concolic unit testing engine for c (2005)
77. Smith, M.J.: Probabilistic abstract interpretation of imperative programs using truncated normal distributions. Electronic Notes in Theoretical Computer Science 220(3), 43–59 (2008)
78. 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)
79. Thakur, A., Elder, M., Reps, T.: Bilateral algorithms for symbolic abstraction. In: Static Analysis, pp. 111–128. Springer (2012)
80. The Apache Software Foundation: Commons math. http://commons.apache.org/proper/commons-math/, accessed: 2014-12-16
81. 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)
82. Thurow, L.C.: Analyzing the american income distribution. The American Economic Review pp. 261–269 (1970)
83. UC Davis, Mathematics: Latte. —http://www.math.ucdavis.edu/ latte—
84. 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)
85. Verdoolaege, S.: Software package barvinok. 2004. Electronically available at http://freshmeat.net/projects/barvinok
86. Wachter, B., Zhang, L.: Best probabilistic transformers. In: Verification, Model Checking, and Abstract Interpretation. pp. 362–379. Springer (2010)
87. 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. ACM (2010)