Probabilistic Program Analysis

Matthew B. Dwyer, Antonio Filieri, Jaco Geldenhuys, Mitchell Gerrard,
Corina Pasareanu, and Willem Visser

No Institute Given

Abstract. The abstract should summarize the contents of the paper and should contain at least 70 and at most 150 words. It should be written using the *abstract* environment.

Keywords:

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 since the 1960s [?]. Initially developed to allow compilers to generate more efficient output programs by the mid-1970s [?] researchers had understood that such program analyses could be applied to fault detection – and verification of the absence of specific classes of faults.

There are extremely well-developed frameworks for defining and implementing such analyses. In this paper we focus on three such frameworks: data flow analysis, model checking, and symbolic execution.

The power of these analysis techniques, and what distinguishes them from simply running a program and observing its behavior, is in 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 with associated logical constraints are used, in symbolic execution, to define abstract domains in data flow analysis, and for predicate abstraction in model checking, to capture sets of data values that may be input to a program. Non-deterministic choice is another widely used approach for modeling uncertainty – for instance in modeling uncertain branch decisions in data flow analysis and in 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 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 it varies according to a generalized beta distribution [?]. Can this type of information be exploited in a program analysis to speed analysis, to yield more useful analysis results, or to reason about new types of program properties?

For decades there has been a growing awareness of the value of incorporating more precise forms of uncertainty into program behavior. The field of randomized algorithms has studied how to incorporate randomness, as an additional program input, as a means of achieving good average case performance – and consequently as a defense against intolerable worst case performance. Programming such algorithms requires that primitives be available to draw values from probability distributions and there are many languages that provide such primitives [?,?].

Regardless of whether information about the distribution of values is embedded within a program or stated as an input assumption, the semantics of these probabilistic programs is well-understood – and has long been studied [?,?] [1]. What has lagged behind is work developing frameworks for defining and implementing static analysis techniques for such programs.

In this paper, we survey work on adapting model checking, data flow analysis, and symbolic execution, to consider probabilistic information. We begin with a brief background that provides basic definitions related to static analysis and probabilities. Section 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. The following three sections, Section 5-6, survey work on probabilistic model checking, probabilistic data flow analysis, and probabilistic symbolic execution. These sections seek to expose similarities and differences among analysis approaches with respect to issues such as the `accuracy` of analysis results, i.e., how results are related to executable program behavior, the aspects of behavior that are governed by probabilities, and the treatment of non-determinism – which is a standard modeling approach in static-analysis. Section 7 surveys approaches to the cross-cutting issue of how probabilities can be specified or encoded for use in analyses. Finally, we conclude with the discussion of a series of open questions and research challenges that we believe are worth pursuing.

2 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

[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 [?] for discussion of these more general programs.

drawn from given distributions such as those provided by the C++ `<random>` library. *We should pull an example from one of the later sections back here to illustrate the idea.*

While researchers have developed analyses that consider a wide range of program properties, in this survey we restrict our attention to program properties that can be encoded as boolean predicates that can be embedded in the program, e.g., `assert` statements, to simplify the explanation of how probabilities are incorporated into the analyses. These are refered to as *invariant properties* since they are intended to hold at every state reached in all program program executions

2.1 Programs and Program Analyses

A program defines a set of execution *traces* each of which is a sequence of *concrete states*, i.e., the current program counter and a map from memory locations to values. A program *satisfies* an invariant property if in all states in all traces the predicate evaluates to true, otherwise the program *falsifies* the property.

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 formula or binary decision diagrams [**?**].

Analyses that seek to prove the satisfaction of properties generally define abstractions that *overapproximate* the set of program states, whereas 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 which are said to be defined over the *concrete domain*, $\mathcal{A}$. For any reachable state of a program a pair of abstraction and concretization functions, $\alpha : \mathcal{C} \mapsto \mathcal{A}$ and $\gamma : \mathcal{A} \mapsto 2^{\mathcal{C}}$, serve to relate the concrete and abstract domains such that $\forall c \in \mathcal{C} : \alpha(c) \in \mathcal{A}$ and $c \in \gamma(\alpha(c))$. Such an abstract domain is typically partially ordered, $\sqsupseteq$, such that $\forall a, a' \in \mathcal{A} : a \sqsupseteq a' \implies \gamma(a) \supseteq \gamma(a')$.

Data flow analysis (and abstract interpretation) can be viewed as a non-standard interpretation of program executions over an abstract domain. The semantics of program statements is lifted to operate on a set of states, encoded as an element of the abstract domain, rather than a single concrete state. For a program statement τ, $\tau^{\#}$ defines its abstract semantics such that $\forall c, c' \in \mathcal{C} : \tau(c) = c' \implies \tau^{\#}(\alpha(c)) \sqsupseteq \alpha(c')$. This implies the classic overapproximating correctness relation for abstracted program statements: $\tau^{\#} \sqsupseteq \alpha \circ \tau \circ \gamma$.

Matt: add definitions of underapproximation and whatever else symbolic execution needs

Matt: develop a figure which shows pseudo code for model checking, data flow analysis, and symbolic execution side by side

Matt: walk through an explanation of that figure which is where we will define what a fixpoint is

2.2 Probabilities and Probabilistic Models

Define and explain the following

- probability measure
- rewards/costs
- discrete time Markov chain
- discrete time Markov decision process

3 Computing Program Probabilities

4 Probabilistic Data Flow Analysis

As we will see in all sections, probabilistic data flow analysis moves from the true/false nature of its classical counterpart to the probably-true/probably-false nature of a static analysis extended with probabilities. The shift from the qualitative to the quantitative allows you to incorporate probabilistic information into the analysis in different ways.

4.1 Probabilities on Control Structure

Initial work in extending data flow analysis techniques with probabilities did not consider the semantics of the program; instead, already-given probabilities were attached to nodes in that program's control flow graph. The goal was to predict the probability of an expression evaluating to some value or type at runtime, which could allow you to perform useful program optimizations.

This approach begins with a control flow graph where each edge e is mapped to the probability that e 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.

(insert figure to make explanation more concise)

Within this bag of execution traces which reach point u, we want to find the portion of traces which satisfy some data flow fact d. The ratio of the satisfying portion to the size of the trace bags gives us the probability of fact d holding at point u.

Fragments.

Kildall's general framework for computing fixpoints over this annotated flow graph.

This assumes execution history does not matter (analysis is path insensitive). Later work (Mehofer) adds some path sensitivity, but as Ramalingam's framework deals with exploded control flow graphs, a fully path-sensitive approach is not tractable.

4.2 Probabilities on Data Structure

Within the last 15 years, research in probabilistic data flow analysis began incorporating probabilistic information directly into the semantics of a program. This is typically done using a variation on Kozen's probabilistic semantics alongside traditional data flow techniques. Embedding probabilities into the semantics allows probabilistic information to influence how both control *and* data structure probabilities are computed during the analysis.

The remaining approaches in this section permit probabilistic information to be defined as either an environment property (i.e., distribution given for an input) or in the type of an expression (i.e., rand call). Many of these techniques use the framework of abstract interpretation.

We cannot go over the details of probabilistic semantics here, but there are a few modifications to the abstract semantics of a traditional imperitive program (see Hankin's WHILE language) which we will point out. One is the addition of a random number generator primitive; it is possible and straightforward to approximate a safe upper bound on this generator. Approximations on loop semantics are dealt with in a safe way using "suitable" widening operators instead of fixed points. We explain how conditionals are treated in the following graphical example.

(insert example)

How do the classical abstract domains work in a probabilistic setting? We will focus on one technique developed by Monniaux that requires little change to the classical domain. The only difference between the classical and the probabilistic case is that in the probabilistic case, an abstract domain has a weight attached to any of its subsets.

The valuation of any element in the concrete domain then becomes the additive composition of weights of points in the concrete domain which are represented by elements in the abstract domain. The concretization function maps from a set of abstract values to a weight function.

An example will be helpful. Consider the abstract domain of intervals (in one dimension).

This correctness criterion is given as an upper bound on the probability of some outcome in the program, e.g. *the probability of violation ϕ is less than* 0.0001%. Dually, other approaches have used lower probability bounds as their correctness criterion.

4.3 Estimating Property Probabilities

Di Pierro et al. have a different approach to abstract interpretation in the probabilistic setting. Instead of defining the abstract domain over a lattice, they define the domain over vector spaces. The dataflow information is now collected using the Moore-Penrose pseudo-inverse instead of the usual fixed point. This is a way to measure property probabilities by estimating some "tight" approximation, e.g. *this variable will be an even integer at this program point* 67% *of the time*. The matrices quickly become large; it is not clear how this approach scales with bigger programs.

4.4 Modeling Nondeterminism

Monniaux recognizes that there are variations of uncertainty, and that not every program property should be modeled by a probability distribution. For instance, a user may exploit an unlikely control sequence in a vending machine to get free candy bars. If word gets out, the probability of this exploited behavior occurring is poorly modeled by a uniform random distribution. It is better to treat this kind of input nondeterminisitically.

In one of Monniaux's semantics, variables that can be tied to a known distribution are cleanly separated from those that cannot. Assignments to variables drawn from a distribution are reasoned about through numerical sampling (Monte Carlo), while nondeterminisic choices are explored using abstract interpretation. This combination of methods provides upper bounds on the probability of outcomes, where one domain is associated with a distribution and the other (nondeterministic) domain is modeled using worse-case behavior.

(replace with DM's modelling of nondeterminism in the non-MC papers)

4.5 Expanding the Scope

4.6 Meta-comments

We have chosen to organize the work on prob. data flow analysis based on how the probabilistic information is incorporated into the analysis (e.g., probabilities on data, probabilities on control) and on the nature of approximation in the analysis, i.e., underapproximation, overapproximation, or "tight" approximation.

We plan a separate discussion of how non-deterministic choice is handled in data flow analysis.

Finally, we plan a brief mention of work that does not fit into "basic probabilistic program" category, i.e., programs that use conditioning.

We would be interested in exposing other dimensions ASAP. Specifically, are there different dimensions that might arise due to thinking about model checking or symbolic execution?

4.7 Required Terminology

The following terms/concepts should be defined earlier in the paper since we will need them in this section.

1. probabilistic program
2. concrete domain
3. abstract domain
4. fixpoint
5. abstract interpretation/data flow analysis
6. program trace
7. path

8. conditioned distribution
9. Bayesian inference

We expect that this will be done in the intro and background section. With regards to that section it would be ideal if we could have a compact explanation of non-probabilistic data flow analyis/abstract interpretation, model checking, and symbolic execution with the attendant concepts. That will cover most of the above and then we can have a separate subsection of the background covering the probabilistic concepts/terms/definitions.

4.8 The Outline

Probabilistic Data Flow Analysis Outline

We are considering approaches that start from classical abstract domains. and characterize the probability of properties expressed as subsets of those domains holding at program points. - this is equivalent to reasoning about the probability of assertions holding or not (in prob sym exe) or probabilistic universal properties (in prob model checking)

1. Probabilities on control structure
   (a) probabilistic information explicitly annotates the control flow structure of the program
   (b) Frequency analysis
      i. Ramalingam
      ii. See if we can tie this approach to the linear operators representing transfer functions which both Monniaux and Di Pierro use
2. Probabilities on data structure
   (a) probabilistic information annotates the data structure of the program and its influence on control and data is computed through the analysis
   (b) the rest of the approaches permit probabilistic information to be defined as either an environment property (i.e., distribution given for an input) or in the type of an expression (i.e., rand call)
3. Bounding property probabilities
   (a) Monniaux
      i. From above
      ii. From below
4. Estimating property probabilities
   (a) PAI
      i. Di Pierro
      ii. "Tight" in a least-square sense
5. Treating uncertainty expressed as non-determinism
   (a) Monniaux can do this
6. Expanding the scope of analysis
   (a) ... to different probabilistic properties
      i. Chakarov
      ii. Fixed Points
      iii. Martingales
   (b) ... to more general probabilistic programs
      i. Bayesian inference
         A. Nori
         B. Modern Probabilistic Programming Languages

5 Probabilistic Model Checking

Matt: do we want to cover probabilistic explicit state model checking? It is in some sense easier to explain especially in relation to data flow analysis and symbolic execution. PRISM actually has this as does MRMC and some other work.

All based on checking universal properties.

1. Probabilities on control structure
   (a) probabilistic information explicitly annotates the modeled transition system
   (b) discrete time Markov chains
   (c) sources of these annotations
      i. operational profiles
      ii. developer guesswork
2. Bounding property probabilities
   (a) computing fixed points over some formula
   (b) from above/below
   (c) extensions with rewards/costs
3. Estimating property probabilities (via sampling)
   (a) difference between probabilistic and statistical model checking ("simulating" the model is how we obtain the estimates of this section)
   (b) confidence intervals
   (c) approximate probabilistic model checking (Herault)
   (d) sequential probability ratio test
4. Treating uncertainty expressed as nondeterminism
   (a) Markov decision processes as extension of DTMCs
   (b) minimum/maximum prob. values relating to nondeterminist constructs in sym. ex. and in Monniaux's data flow
5. Expanding the scope of analysis
   (a) probabilities on data structures?
      i. extract model from actual program and decorate data values with probabilities
   (b) run-time probabilistic model checking (Anto)

6 Probabilistic Symbolic Execution

7 Specifying and Inferring Probability Distributions

8 Open Questions and Future Directions

Some possible things to discuss:

1. tools: tools support for prob. data flow analysis is really lacking
2. building a robust and flexible counting tool: that mixes methods, that can-compute prob. estimates with bounds, etc.

3. hybrid approaches: could we identify portions of a state space that could be analyzed exactly through symex and then fold that information back into mc/dfa approaches to boost their precision
4. inference: while we don't cover it in this paper, a key feature of the most recent prob. programming languages is conditioning in the form of an "observe" statement. This functions exactly like an "assume" statement and could be used to drive backtracking in mc/symex. Work would have to be done to renormalize the prob. measures as in existing approaches.
5. ... add more here ...