Probabilistic Symbolic Execution

Jaco Geldenhuys
Stellenbosch University
Stellenbosch, South Africa
jaco@cs.sun.ac.za

Matthew B. Dwyer
University of Nebraska - Lincoln
Lincoln, NE, USA
dwyer@cse.unl.edu

Willem Visser
Stellenbosch University
Stellenbosch, South Africa
wvisser@cs.sun.ac.za

ABSTRACT

The continued development of efficient automated decision procedures has spurred the resurgence of research on symbolic execution over the past decade. Researchers have applied symbolic execution to a wide range of software analysis problems including: checking programs against contract specifications, inferring bounds on worst-case execution performance, and generating path-adequate test suites for widely used library code.

In this paper, we explore the adaptation of symbolic execution to perform a more quantitative type of reasoning — the calculation of estimates of the probability of executing portions of a program. We present an extension of the widely used Symbolic PathFinder symbolic execution system that calculates path probabilities. We exploit state-of-the-art computational algebra techniques to count the number of solutions to path conditions, yielding exact results for path probabilities. To mitigate the cost of using these techniques, we present two optimizations, PC slicing and count memoization, that significantly reduce the cost of probabilistic symbolic execution. Finally, we present the results of an empirical evaluation applying our technique to challenging library container implementations and illustrate the benefits that adding probabilities to program analyses may offer.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification, Experimentation

Keywords

Symbolic execution, probabilistic analysis, model counting, testing

ISSTA '12, July 15–20, 2012, Minneapolis, MN, USA

1. INTRODUCTION

Understanding the behavior of a program is central to the process of testing it. Most of the time we are happy with just determining whether a behavior can, or can't, happen, but we believe we can understand program behavior even better with more fine-grained information that allows one to know how *likely* a behavior is to occur. We show that with a combination of symbolic execution and model counting one can do *probabilistic symbolic execution* which allows us to assign probabilities to program paths.

Symbolic execution explores the execution tree of a program using symbolic values (instead of actual values) for the inputs. Each execution path is described by a conjunction of constraints on the inputs, known as a *path condition*. The paths can be explored in various ways — depth-first order is arguably the simplest and most common — and are constructed incrementally. As each vertex of the execution tree is reached, the (partial) path condition that describes the path from the root to the vertex is evaluated: if it is satisfiable, the search continues. If not, that branch of the tree is known to be unreachable. Satisfiability checking has a 0–1 outcome, but what about all of the real values in between? The values 0 and 1 are in fact coarse approximations of the probability that a given path condition is true. This paper deals with finer approximations of the probability, and, in the extreme, its exact calculation. We discuss the theory, implementation and potential applications of this idea and present empirical evidence to demonstrate its feasibility.

Calculating the probability involves counting the number of solutions to a path condition (known as model counting) and dividing it by the total space of values of the inputs (the latter is simply the product of all the input domain sizes). This only works if the inputs are uniformly distributed within their domain, which is an assumption we use here. For the generation of path conditions we use Symbolic PathFinder (SPF) [27] the symbolic execution extension to Java PathFinder and for counting the solutions we use the LattE [31] tool. We restrict ourselves to the domain of linear integer arithmetic, since this is supported by LattE and SPF. The main contributions of this work are:

- A demonstration of how to add path probabilities to symbolic analysis using model counting;
- optimizations to make model counting scale;
- applications of our approach, including discovering a previously unknown error; and
- open-source implementation as part of SPF.

```
1  int classify(int a, int b, int c) {
2    if (a<=0 || b<=0 || c<=0) return 4;
3    int type=0;
4    if (a==b) type+=1;
5    if (a==c) type+=2;
6    if (b==c) type+=3;
7    if (type==0) {
8      if (a+b<=c || b+c<=a || a+c>=b) type=4;
9      else type=1;
10     return type;
11   }
12   if (type>3) type=3;
13   else if (type==1 && a+b>c) type=2;
14   else if (type==2 && a+c>b) type=2;
15   else if (type==3 && b+c>a) type=2;
16   else type=4;
17   return type;
18 }
```

Figure 1: Solution for Myers's triangle problem

2. MOTIVATING EXAMPLE

Consider the function shown in Figure 1; this is a solution by DeMillo and Offutt [6, Figure 6] for the classic triangle classification problem of Myers [24]. Its inputs are the three side lengths of a triangle, and it returns 1 if the triangle is scalene, 2 if isosceles, 3 if equilateral, and 4 if it is not a triangle at all. Suppose for now that $\mathtt{a}, \mathtt{b}, \mathtt{c} \in [-1000, 1000]$, and that the arguments are independently and uniformly distributed across this range.

How do we go about verifying the correctness of this code? Typically a hand-crafted or random test suite would be used for this function. However, apart from the effort of determining the correct output, there is no guarantee that all errors will be found, and this problem is notoriously difficult to test thoroughly.

While not a cure-all, the use of probabilistic symbolic execution can provide valuable insights about the behavior of the function. For example, what is the probability that the function classifies a set of inputs as equilateral? Each variable can take one of 2001 values, so the size of the input space is 2001^3. Of these, there are 1000 triangles with three equal sides: $(1,1,1)$, $(2,2,2)$, ..., $(1000,1000,1000)$. The probability that an input forms an equilateral triangle is therefore $1000/2001^3 = 1.25\times10^{-7}$. This is exactly the answer computed by the system described in this paper. Here are the classifications:

Classification	Probability
scalene	2.07×10^{-2}
isosceles	2.80×10^{-4}
equilateral	1.25×10^{-7}
illegal	9.79×10^{-1}

The low probability of an equilateral triangle means that we would have to execute a large number of random tests before detecting any potential error. In this simple example it was still possible to calculate the expected probability by hand. There was, however, no guarantee that it would agree with the code, and if our function contains a bug, we would have noticed the discrepancy between the calculated and expected probability.

Even — especially — when it is impossible to calculate the expected probability of a behavior by hand, this approach can still guide verification. For example, the probability that the assignment in line 12 is executed, is also 1.25×10^{-7}. This strongly suggests that the assignment happens if and only if the input forms an equilateral triangle. From the code it is clear that illegal triangles are detected in lines 2, 8, and 16, and the probability that these lines are triggered is 0.875, 0.104, and 9.36×10^{-5}, respectively. In other words, 89.395% of illegal triangles are detected in line 2, 10.595% in line 8, and only a very small percentage in line 16 (fewer than 0.00956%). In this case, line 16 is still indispensable, but in other situations this kind of observation may suggest ways to optimize the performance of code.

As it turns out, this example contains a bug that shows up when we increase the range to Java's full integer range. The additions in the code can result in arithmetic overflow and the input $\mathtt{a} = x$, $\mathtt{b} = y$, $\mathtt{c} = 1$ is incorrectly classified as an illegal triangle whenever $x + y \geq 2^{31}$. This is a hard error to notice unless you know what you are looking for. However, there is another way. We have taken three alternative solutions (two correct, one incorrect) to the same problem from [34]. The variable values $\mathtt{a}, \mathtt{b}, \mathtt{c} \in [0, 2^{30}]$ produce the following probabilities:

	scalene	isosceles	equilateral	illegal
A	4.99×10^{-1}	2.10×10^{-9}	8.67×10^{-19}	5.00×10^{-1}
B	4.99×10^{-1}	2.79×10^{-9}	8.67×10^{-19}	5.00×10^{-1}
C	4.99×10^{-1}	2.10×10^{-9}	8.67×10^{-19}	5.00×10^{-1}
D	1.67×10^{-1}	2.10×10^{-9}	8.67×10^{-19}	8.33×10^{-1}

The correct solutions A and C agree on all classifications. Solution B produces too many isosceles classifications (and too few illegal triangles, although the discrepancy is not visible in the table) and our solution D produces too few scalene triangles and too many illegal triangles. This illustrates that even if we do not know that A and C are test oracles, our method can be useful to compare independent programs and, with more detailed analysis, can help to pinpoint the exact differences. The analysis of A, B, C, and D takes 3, 10, 8, and 4 seconds to complete.

3. ADDING PROBABILITIES TO SYMBOLIC EXECUTION

We begin with a short review of how symbolic execution is performed then discuss our adaptation to calculate path probabilities. Our algorithm for probabilistic symbolic execution permits, in principle, a variety of different approaches for calculating or estimating probabilities from path conditions. In this section, we discuss our use of algorithms for precise model counting, developed for counting lattice points and the volume of convex polytopes, and how we have optimized the mapping of path conditions onto those algorithms.

3.1 Background

For simplicity, we consider a simple three-address language where statements are identified by their location, l. There are two classes of instruction: branches, identified by predicate $\mathit{branch}(l)$, whose branch condition is $\mathit{cond}(l)$, and non-branching instructions of the form $v = e$. We illustrate the algorithms for variable operands, but constants are of course permitted and their handling is straightforward. For branches, the target location is $\mathit{target}(l)$ and for all statements the next location is $\mathit{next}(l)$.

Symbolic execution is a non-standard interpretation of a program which represents the program's execution state

symbolically. The state consists of two parts: a path condition (PC) which is a conjunctive formula, ϕ, encoding the branch decisions which must be true to reach a program location, and a map from program variables to symbolic expressions over constants and free variables. The expression associated with variable v is accessed from the map as $m[v]$ and a map update, with a new expression e, is written $m\langle v, e\rangle$. We extend this notation to expressions: if e is a constant or free variable, then $m[e] = e$. Otherwise, if $e = e_1 \odot e_2$ for some operator $\odot$, then $m[e] = m[e_1] \odot m[e_2]$.

Algorithm 1 `symbolicExecute`(l, ϕ, m)

```
while ¬branch(l) do
  m ← m⟨v, e⟩
  l ← next(l)
end while
c ← m[cond(l)]
if SAT(φ ∧ c) then
  symbolicExecute(target(l), φ ∧ c, m)
end if
if SAT(φ ∧ ¬c) then
  symbolicExecute(next(l), φ ∧ ¬c, m)
end if
```

Algorithm 1 sketches the key elements of symbolic execution. The initial call is `symbolicExecute`$(l_0, \mathit{true}, m_0)$ where l_0 is the initial location in the program, the initial PC is *true*, and m_0 is the initial map. The initial map is formed by introducing a new free variable f_v for each of the program's input variables *Input*. In other words, $\forall v \in \mathit{Input} : m_0[v] = f_v$.

Conceptually, symbolic execution can be decomposed into the processing of regions that lie between branch statements. At the beginning of such a region all non-branching statements are processed by updating the map component of the state and advancing the location. Then each branch outcome is considered. For the positive branch outcome, a formula conjoining the current PC and the branch condition is formed. If that formula is satisfiable, then it becomes the new path condition and the region of code rooted at the branch target is processed. The negative outcome is processed identically, except that the branch condition is negated and the next region processed begins at the next location of the branch.

The simple algorithm sketch shown here does not include a number of features found in modern symbolic execution frameworks. For instance, this algorithm makes no attempt to bound the search.

The resurgence of interest in symbolic execution in recent years owes much to advances in automated decision procedures. Solvers such as Z3 [5] include numerous optimizations that allow them to handle a wide range of SAT queries efficiently. For example, incremental solving optimizes the processing of a sequence of SAT calls where the formulae in the sequence are extended by adding additional conjuncts. As one can see from the recursive nature of Algorithm 1, this is precisely the nature of SAT calls in symbolic execution.

3.2 Probabilistic Symbolic Execution

Algorithm 2 sketches the key elements of our extension of symbolic execution to compute path probabilities. We enrich the state with a third component — the path probability — which is initially set to 1. The algorithm proceeds identically through non-branching statements.

Algorithm 2 `probSymbolicExecute`(l, ϕ, m, p)

```
while ¬branch(l) do
  m ← m⟨v, e⟩
  l ← next(l)
end while
c ← m[cond(l)]
φ' ← slice(φ, c)
p_c ← prob(φ' ∧ c)/prob(φ')
if SAT(φ' ∧ c) then
  probSymbolicExecute(target(l), φ ∧ c, m, p * p_c)
end if
if SAT(φ' ∧ ¬c) then
  probSymbolicExecute(next(l), φ ∧ ¬c, m, p * (1 − p_c))
end if
```

To calculate the probabilities associated with branch outcomes, our algorithm calculates two quantities. $\mathit{slice}(\phi, c)$ computes the closure of the data dependence relation over PC conjuncts beginning with c. In other words, it returns the smallest subset of the conjuncts of ϕ that depend on c. For example,

$$\mathit{slice}(x > 10 \wedge y > 10 \wedge z > x, z = 5)$$

returns $x > 10 \wedge z > x$ but omits $y > 10$ because it alone is independent of $z = 5$. The result is the *independent* sub-formula of the PC that determines executability of the branching statement l.

Next the conditional probability p_c of the branch condition given the PC slice ϕ' is calculated. The conditional probability is then used to incrementally calculate the path probability by multiplying it to the prior probability ($p * p_c$). The use of PC slicing and conditional probability allows us to calculate probabilities for significantly smaller formula and to reuse the results of those calculations more readily and, in fact, only the first call to $\mathit{prob}(\cdot)$ is required. For the second branch outcome, we exploit the fact that the sum of the conditional probabilities of the branch outcomes is 1 — this allows us to avoid further calls to $\mathit{prob}(\cdot)$.

Issues such as exceptional behaviour in the program under analysis does not influence the calculation of probabilities. From SPF's point of view, exceptions are simply regarded as a special (but not unique) control flow mechanism within the program.

In the remainder of this section we discuss the implementation and optimization of $\mathit{prob}(\cdot)$

3.3 Calculating Path Condition Probabilities

Algorithm 2 allows for a variety of different approaches to be applied to estimate the probability of executing a path condition. We discuss one approach in this section that is based on *model counting*.

Model counting is the problem of determining the number of solutions of a given formula. While the complexity of model counting varies with the theories used to express the formula, the counting problem is at least as hard to solve as the decision problem. In practice, counting algorithms have not been studied and optimized as extensively as decision algorithms and, consequently, they can be significantly more expensive in practice.

While a range of model counting techniques could be used, we explore the use of the LattE [31] toolset. LattE is well-supported and implements state-of-the-art algorithms for

computing volumes, both real and integral, of convex polytopes [19, 16] as well as integrating functions over those polytopes [4]. The former can be used to compute path probabilities when input variables are drawn uniformly from their type's domain or a probability mass function is available for integral variables, and the latter when a probability density function is available for real-valued variables. Our discussion below focuses on the case of computing probabilities for linear integer arithmetic (LIA) constraints over variables whose values are uniformly distributed over their type.

LattE accepts constraints expressed as a system of linear inequalities each of which defines a hyperplane. Together, the half-spaces defined by these hyperplane define a convex polytope — an n-dimensional analog of a polyhedron. The halfspace (H) representation is concisely encoded as the matrix inequality: $Ax \leq B$, where A is an $m \times n$ matrix of coefficients and B is an $n \times 1$ column vector of constants. H-representations are a natural representation for the conjunctive fragment of LIA except that it is not possible to directly express disequality constraints, e.g., $x \neq 0$.

Most LIA constraints can easily be converted into the form $a_1x_1 + \ldots a_nx_n \leq b$. For example, $\geq$ and $>$ can be *flipped* by multiplying both sides by -1, and strict inequalities, $<$, can be converted by decrementing the constant. In LattE equalities can be expressed directly; there is no need to use a pair of overlapping inequalities.

Algorithm 3 $\texttt{prob}(\phi \equiv \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_n)$

$cSet \leftarrow \{\psi_1, \psi_2, \ldots, \psi_n\}$
$vars \leftarrow \{v \mid \exists c \in cSet : v \in c\}$
$deSet \leftarrow \{d \mid d \in cSet \wedge d \equiv \ldots \neq \ldots\}$
$ineqSet \leftarrow cSet - deSet$
$exSet \leftarrow \{ineqSet \wedge (v = e) \mid v \neq e \in deSet\}$
$count = count_\wedge(\bigwedge_{ineqSet}) - count_\vee(\bigvee_{exSet})$
return $count / \prod_{v \in vars} dom(v)$

Algorithm 3 calculates the probability for a PC by first partitioning the constraints in the conjunction ϕ into inequalities ($ineqSet$) and disequalities ($deSet$). To support disequalities, we calculate a set of constraints that encode excluded solutions ($exSet$). We explain the handling of these exclusion constraints ($count_\vee(\cdot)$) in the next subsection.

If there are no disequalities then the calculation of $exSet$ results in an empty set and the count is the result of calling LattE to count a conjunctive set of constraints ($count_\wedge(\cdot)$). The count is returned as a probability by dividing by the product of the variable domains ($dom(\cdot)$) from the original formula.

3.4 Disequality Constraints

In symbolic execution, constraints are always a conjunction of boolean valued terms. Thus, the obvious inability to encode disjunctions in an H-representation does not present a difficulty. Disequality constraints are another matter. A natural encoding of a constraint $\alpha \wedge x \neq 0$ using inequalities is

$$(\alpha \wedge x < 0) \vee (\alpha \wedge x > 0)$$

When a single disequality constraint is present, we can exploit the structure of a symbolic execution to avoid the need to handle disequality constraints at all. If a branch condition is an equality constraint, then the negated condition is a disequality, but as shown in Algorithm 2 we need not calculate the conditional probability of the negated condition — we compute it from the probability of the original condition. When the original condition is a disequality, we simply explore its negation first.

The challenge lies in treating multiple disequality constraints in a single path condition. Consider the constraint $\alpha \wedge (x \neq 0) \wedge (y \neq 0)$. These disequalities exclude solutions of α where x, y or both x and y are 0. We can express these excluded solutions as the disjunction of constraints:

$$\begin{aligned} \alpha \wedge (x = 0) \\ \alpha \wedge (y = 0) \\ \alpha \wedge (x = 0) \wedge (y = 0) \end{aligned}$$

Since we cannot count solutions for disjunctive constraints directly, the solutions of the original constraint are counted as follows:

$$\begin{aligned} &count_\wedge(\alpha) - \\ &\quad (count_\wedge(\alpha \wedge (x = 0)) + count_\wedge(\alpha \wedge (y = 0)) - \\ &\quad\quad count_\wedge(\alpha \wedge (x = 0) \wedge (y = 0))) \end{aligned}$$

The quantity on the first line is the total number of solutions. The second and third lines subtract the excluded solutions where either x or y is 0. Note that solutions where both x and y are 0 are double-counted by the second line, so the final constraint subtracts their total.

In general, we calculate the number of excluded solutions for a constraint with n disequality constraints of the form

$$\alpha \wedge (v_1 \neq e_i) \wedge \ldots \wedge (v_n \neq e_n)$$

using the following variant of the Bonferroni inequalities [8] which is adapted to compute counts instead of probabilities.

$$count_\vee(\bigvee_{i=1}^{n} \phi_i) = \sum_{I \subseteq \{1,\ldots,n\}} (-1)^{|I|-1} count_\wedge(\bigwedge_{i \in I} \phi_i) \quad (1)$$

where $\phi_i \equiv \alpha \wedge (v_i = e_i)$. In general, the combinatorics of this equation leads to 2^n calls to LattE to compute the number of solutions of the combinations of the ϕ_i, but as we discuss below this cost is amortized across the symbolic execution by reusing already computed counts.

Returning to Algorithm 3 we see that the disequality constraints are handled by formulating a set of excluding equality constraints, one for each disequality, and passing their disjunction to $count_\vee(\cdot)$. These are transformed, by equation 1, into a set of conjunctive constraints that are passed off to LattE for solution.

3.5 Optimizing Model Counting

Automated decision procedures have become highly optimized in large part because there is great demand from client applications that seek to exploit their capabilities. Providing support for software testing, analysis and verification is one area where they have enjoyed great success — symbolic execution being a prime example.

Many of these applications only require a determination of whether a formula is SAT or UNSAT, but a developing body of work on quantitative system analyses [11, 22, 18] demands support for providing richer information — model counting techniques can provide this kind of information.

In a recent paper [20], researchers enhanced an SMT algorithm for LIA to incorporate model counting. As a general SMT algorithm their approach must consider the full set of logical operators — including disjunction — which leads to significant complexity in their solution. We take a different approach that exploits the way that symbolic execution generates path conditions to optimize the use of model counters.

3.5.1 Count One, Infer the other

The nature of symbolic execution is to explore both outcomes of a conditional statement. In Algorithm 1 the satisfiability of the positive branch outcome provides no information about the satisfiability of the negative outcome. In general, two decision procedure calls are made at each branch.

For the probabilistic case, since the model counter gives us richer information we can do better. As shown in Algorithm 2 we calculate the probability of one of the branch outcomes — this may lead to several calls to the model counter — then we calculate the probability of the other outcome in terms of the first — thereby avoiding calls to the model counter. In practice, this reduces the total number of calls to the model counter during probabilistic symbolic execution by nearly a factor of 2.

3.5.2 Slice, Normalize and Memoize

In Algorithm 1 the path conditions passed to SAT are always distinct. Algorithm 2 differs in that regard due to our use of PC slicing and the calculation of probabilities.

Consider the following path conditions:

$$\alpha \wedge (x < 4) \wedge (z \geq 1 - x)$$
$$\neg\alpha \wedge (2(y-1) \leq 4) \wedge (1 - z < y - 1)$$

Clearly these are distinct PCs, since they differ in the first conjunct and they have different sets of variables.

If $\{x, z\} \cap vars(\alpha) = \emptyset$, then when symbolic execution reaches the last branch of the first PC there is no need to consider the first conjunct. This can significantly simplify the constraint for model counting. If α involves two additional variables, then PC slicing reduces the dimension of the polytope submitted to LattE by a factor of 2 — which can lead to significant performance improvements.

If $\{x, y, z\} \cap vars(\alpha) = \emptyset$, then for either of these path conditions when symbolically executing the last branch in the condition the first — involving α — can be ignored. This provides an additional opportunity for optimization. During the process of converting PCs into systems of inequalities to pass to LattE we normalize the constraints by putting them into the form: $a_1x_1 + \ldots a_nx_n \leq b$. We alphabetize the variables when performing this normalization. Thus, the resulting sub-constraints that are submitted to $count_\wedge(\cdot)$ in Algorithm 3 for the two PCs above are:

$$(x \leq 3) \wedge (-x - z \leq -1)$$
$$(y \leq 3) \wedge (-y - z \leq -1)$$

These constraints both result in the same H-representation for LattE:

```
6 3
max  1   0
min -1   0
max  0   1
min  0  -1
3    1   0
-1  -1  -1
```

where the first line indicates the matrix size: the number of inequalities by the number of variables plus one. The first four inequalities encode the max and min values for a variable based on its type — LattE is largely insensitive to these values in terms of performance. The last two inequalities express the two constraints.

Nowhere in the H-representation is the identity of a variable encoded — it is simply not needed for counting the solutions. This provides an opportunity for reusing solution counts by detecting when an identical systems of inequalities are generated. Applying slicing and normalization creates significant numbers of opportunities for reuse, and our implementation of $count_\wedge(\cdot)$ uses memoization to eliminate unnecessary calls to LattE.

Memoization is useful for dealing with the complexities introduced by disequality constraints as well. Since symbolic execution incrementally extends PCs with additional conjuncts as it moves along a path, the counts computed for prefixes are memoized for use later in the path. Consider a PC of the form:

$$\alpha \wedge deq_1 \wedge deq_2 \wedge \ldots deq_n$$

where deq_i are disequality constraints. When $prob(\cdot)$ is called on this PC all of the combinations of excluded equality constraints that were needed to count the solutions to the prefix of this without deq_n have been memoized. This eliminates the need to call LattE for half of the calls that would otherwise be needed to evaluate equation 1.

3.5.3 Further Room for Improvement

In the course of our work, we have identified several opportunities for further increasing the performance of probabilistic symbolic execution.

Model counting results can be used to determine satisfiability. As shown in Algorithm 2, we still use decision procedure calls to judge PC satisfiability. This is redundant, since a model count that is non-zero implies satisfiability. As currently architected the use of model counting is hidden from the symbolic execution algorithm, we plan to rearchitect our system to avoid the use of SMT solver calls when the theories involved in sliced path conditions permit exact model counting.

Our normalization process is effective, but there are opportunities for further equivalence reductions on systems of inequalities. A key feature of modern decision procedures is their ability to transform a given formula to a form that is simpler to solve, but preserves satisfiability — the original and resulting formula are said to be equisatisfiable. For model counting, transformations must preserve solution count which is much more challenging. We plan to explore different schemes for calculating canonical column and row orderings for the H-representations. Simply reordering matrix rows and columns is guaranteed to preserve solution count and it provides the opportunity to identify more sliced PCs as being equivalent. This will allow greater reuse of results from previous calls to $count_\wedge()$.

In the longer term, we believe that just as SMT solvers have improved their performance based on an understand-

ing of the types of constraints that client applications wish to solve, so to will model counters. We plan to share the constraints from a broad range of probabilistic symbolic execution experiments with the developers of LattE in an effort to inspire this type of client-driven optimization.

4. APPLICATIONS

We implemented the analysis in Symbolic PathFinder [27], the symbolic execution framework of JavaPathFinder [32]. The core of our system is a JPF Listener, that calculates the probabilities whenever a branch is found feasible. This listener invokes LattE to obtain the size of the current path condition using the domain size for each variable if provided and defaults otherwise. The complete system is approximately 1000 lines of code.

All examples are evaluated on a Mac Air 1.7 Ghz (Intel Core i5) with 4Gb of memory and running OSX 10.7.2. Our main aim in the evaluation is to see if the system can handle non-trivial code and secondly whether it can be an aid during testing. We therefore chose the container examples[1] from [33] which were also the focus of a number of follow-up research efforts: (a) [30] which showed that random testing performs very well for obtaining coverage in these examples and (b) [9] that showed, amongst other things, that the BinomialHeap example contains a bug. BinomialHeap might only be a few hundred lines of code, but to trigger the error a sequence of 14 API calls is required. Specifically we consider BinomialHeap, TreeMap (implementation of red-black trees), and BinaryTree in our analysis.

In the following we first show some applications of the probability analysis and then in Section 4.1 we evaluate the usefulness of the optimizations of path condition slicing and memoization.

Our experimental setup involved calculating the probability of reaching branches in the code. We used only `add(n)` and `delete(n)` (or equivalent calls) from a container where the length of the sequence of calls and the range of values for the parameter `n` can be set. For each coverage location in the code we calculate exactly all the path conditions that reach the location as well as the probability of that happening (i.e., for each location there is a map to a list of `<PC, prob>` pairs). From this list we then calculate the following probabilities for each location:

Precise – The exact probability for reaching this code which is calculated as the probability of the disjunction of the path conditions reaching the location.

Sum – The sum of the probabilities for reaching the location.

In addition we also use the data to calculate the *least likely* path(s) through the code. Note there could be more than one equally unlikely path.

During the evaluation it quickly became obvious that the *Precise* probability of reaching a location is very expensive to calculate using LattE due to the exponential blow-up required to handle disjunctions (see equation 1 in Section 3.4). If there are n paths reaching a location, then in general one needs 2^n calls to LattE to calculate the probability of reaching the location; the probability is calculated on the disjunction of the n path conditions. However, it was also apparent from our examples that most of the time $Precise = Sum$.

[1]These examples are available from `http://javapathfinder.svn.sourceforge.net/viewvc/javapathfinder/trunk/examples/issta2006/`

```
1  void bar(int x) {
2      foo(x);
3      if (x < 6) foo(x);
4  }
5
6  void foo(int x) {
7      if (x < 5) loc = 1;
8  }
```

Figure 2: Example where $Precise < Sum$

```
1  void runTest(int[] options, int limit) {
2      Container c = new Container();
3      int round = 0;
4      while (round < limit) {
5          if (options[round] == 1)
6              c.add(options[limit + round]);
7          else
8              c.delete(options[limit + round]);
9          round++;
10     }
11 }
12
13 void runTestDriver(int length) {
14     int[] values = new int[length*2];
15     int i = 0;
16     while (i < 2*length) {
17         if (i < length)
18             values[i] = makeSymInt("c" + i);
19         else
20             values[i] = makeSymInt("v" + i);
21         i++;
22     }
23    runTest(values,length);
24 }
```

Figure 3: Test Driver Code for Containers

Consider the example in Figure 2. From the point where the `bar` routine is invoked, there are two paths that reach the `loc = 1` assignment in line 7: for the first path the PC is $A = x < 5$; for the second the PC is $B = x < 6 \wedge x < 5$. If we assume that $x \in 0 \ldots 9$, then $prob(A) = prob(B) = 0.5$, but also $prob(A \wedge B) = 0.5$. Consequently, $Precise = 0.5$, whereas $Sum = prob(A) + prob(B) = 1.0$. In this case, $Precise < Sum$ because the path corresponding to B is an extension of the path corresponding to A. However, it is often the case that locations can be reached along different paths that can take either branch at a conditional (since that condition is not relevant in reaching the target location), and in those cases the probability of their conjunction is zero ($A \wedge B = false$ and $prob(A \wedge B) = 0$).

Note that when slicing the path conditions to obtain only the part relevant to the current condition (see Section 3.2) works well it means some conditions along the path don't influence the current condition, which is also when the sum is a good approximation for the precise probability. In our results we therefore don't show the precise probability, but rather the sum of the probabilities.

Figure 3 shows the code we use to run the analysis of a container. An important point is that we encode the choice

of whether to `add` or `delete` as a condition over a variable (`c`) of domain $[1, 2]$ and the values to be used with a variable (`v`) of domain $[0 \ldots n]$ where n can be varied (n is not shown in the driver code since the domain specification is given to our analysis in the JPF start-up configuration). We create the symbolic variables using recognizable names within the code (lines 18 and 20) which allows us to interpret the path conditions as a sequence after analysis.

It is important to distinguish the probability of an input sequence and the probability of a path. We work on the assumption that all input sequences (in other words a sequences of (`c`, `v`) pairs) have the same probability. For example, for the domain $[0 \ldots 9]$ and three operations, each sequence of adds/deletes has a probability of $1/(2 \times 10)^3 = 1/8000$. Of course, the sequences are not handled individually, but, because the variables are symbolic, in sets. On the other hand, generally speaking the probability of a particular path (or equivalently, a path condition) is determined by the structure of the code, the structure of the input, and the domain sizes.

Finding a Bug using the Most Unlikely Paths The first analysis we report on here is the most unlikely paths for BinaryTree for sequence length of 4 and value domain $[0 \ldots 9]$ (the calls of interest are insert and delete). The code is shown in Figure 4 and the probabilities of reaching the different branches is shown as comments. Note, the probabilities are the path probability for reaching the branch and since some of them are within `while` loops it is possible that the two sides of an `if` can add up to a probability larger than one. The following 4 sequences are returned as equally most unlikely (the analysis took $7m07s$):

1. insert(x); delete(x); insert(x); delete(x);
2. insert(x); insert(x); insert(x); delete(x);
3. insert(x); insert(x); delete(x); delete(x);
4. insert(x); delete(x); delete(x); delete(x);

The first observation is that the same element, x, is added and deleted all the time. This can be explained simply by observing that if there is any equality check in the code it is very unlikely to always have the same value in the check. Although the first sequence seems quite plausible, the second is not so obvious since once you added an element it will not be added again. However a glance at the code for insert shows a while loop checking if the current element is unequal to the one being added. The negation of this check states that the two values should be equal, thus if you insert a value it is extremely unlikely to try and insert the same value again and if you do you will take the negated branch. Note however that the last delete is less likely than trying to insert the same element again. As can be seen from the probability annotations in Figure 4 there are branches in delete that has lower probability than any in insert. A similar situation occurs in all the containers we analyzed, and will be further elaborated on in the next example below. This brings us to the two sequences (3 and 4) that are not obvious at all: one would think that once you deleted an element and the tree is empty, doing another delete will be useless. Put differently, trying to delete something from an empty tree should be a very likely action, so why is it part of the unlikely sequences? Looking at the code for delete a bug is discovered: the root node is never deleted. This bug is in the original code used

```
1  public void add(int x) {
2    Node current = root;
3
4    if (root == null) { // .9375
5      root = new Node(x);
6      return;
7    }
8
9    while (current.value != x) {
10     if (x < current.value) {
11       if (current.left == null)  // .4592
12         current.left = new Node(x);
13       else  // .5745
14         current = current.left;
15     } else {
16       if (current.right == null)  // .4592
17         current.right = new Node(x);
18       else  // .5745
19         current = current.right;
20 } } }
21
22 public boolean remove(int x) {
23   Node current = root;
24   Node parent = null;
25   boolean branch = true;
26
27   while (current != null) {
28     if (current.value == x) {
29       Node n = current;
30       while (n.left!=null||n.right!=null){
31         parent = n;
32         if (n.right != null) {// .0196
33           n = n.right;
34           branch = false;
35         } else { // .0181
36           n = n.left;
37           branch = true;
38       } }
39       // FIX if (current == root) {
40       // FIX  root = null;
41       // FIX  return true;
42       // FIX }
43       if (parent != null) {
44         if (branch)  // .0346
45           parent.left = null;
46         else // .0361
47           parent.right = null;
48       }
49       if (n != current) { // .0361
50         current.value = n.value;
51       } // else .1077
52       return true;
53     }
54     parent = current;
55     if (current.value > x) { // .5745
56       current = current.left;
57       branch = true;
58     } else { // .5745
59       current = current.right;
60       branch = false;
61   } }
62   return false;
63 }
```

Figure 4: BinaryTree with probabilities $[0..9]$

in [33], but was not detected at the time. Once fixed the following 4 sequences emerge as the most unlikely:

1. insert(x); insert(y); insert(y); delete(y); with $y < x$
2. insert(x); insert(y); insert(y); delete(y); with $y > x$
3. insert(x); insert(y); insert(y); delete(x); with $y < x$
4. insert(x); insert(y); insert(y); delete(x); with $y > x$

From this we learn simply that deleting something that you inserted before is unlikely, and if you insert the same thing twice before deleting that is even more unlikely.

Coverage Probability Next we consider how the probability of obtaining a certain degree of code coverage changes when the domain of the input variables change. For this we consider the BinomialHeap container and again use a sequence length of 4 but with increasing variable ranges $[0 \dots 9]$, $[0 \dots 49]$, $[0 \dots 99]$ and $[0 \dots 499]$. This analysis took $57s$ for each range; note the time is independent of the variable ranges. The results are presented in Table 1 and as stated before we only show the sum of the probabilities of reaching a coverage location (representing branch coverage precisely as in [33]). We only show probabilities rounded off for presentation purposes, but in the implementation we use the APfloat Java package (`http://www.apfloat.org`) with a precision of 500 digits.

The `Loc` column refers to the location in the code, `PCs` indicate with how many unique path conditions did we reach the location, the following 4 column show the (approximate) probability of reaching the location with varying variable ranges and lastly we ran random tests for the $[0 \dots 499]$ range first a 1000 times and then 10000 times to validate the probability results in the $0 \dots 499$ column (we also accumulated the results from running it a 1000 times each, i.e., the first one was actually run 10^6 times and the second one 10^7 times). The values in the two random columns are the number of times the location was reached during the random runs. Note that the gaps in the table are for locations that cannot be reached by sequences of length 4 (with the exception of location 12 which was removed since it didn't represent a branch).

For the top half of the table, until location 10, the probability of reaching the location increases as the variable ranges become larger, but for the bottom half of the table from location 13 to 21 the probability decreases as the range increases. Looking at the code it turns out that the locations numbered below 13 are part of code that gets executed during both insert and delete operations, whereas the rest is executed only during delete operations. Branches within the delete operation therefore becomes less and less likely to be executed as the data domain increases. This makes intuitive sense, since delete operations tend to involve comparisons with existing data and only deletes when an element is found that was previously inserted, which is less likely to happen if a large domain is used to insert and delete from.

The last two columns in Table 1 represents running randomly chosen sequences (and parameters from the $[0 \dots 499]$ range) of length 4 and it can be seen that in both the 1000 and $10,000$ run case we hardly ever executed any of the statements related to delete-only. Location 19 which is the least likely one to be covered, wasn't reached in either random set of runs. If it is the case that delete-related locations are less likely to be covered then one should really skew the random sequences to make deletes more likely to enable better coverage. When we skewed the last operation to be a delete with a probability above 0.9 then we cover all the locations in Table 1. This is an example where the probability results enabled a better strategy for obtaining high coverage.

Of course, to obtain better coverage one can also increase the sequence length. This observation is supported by our analysis: for example when going to sequence length 5, then all locations become more likely to be covered for the same variable domain as sequence length 4. However the same trends also hold in that deleted-related locations become less likely to be covered for larger variable domains.

Probability of bugs The BinomialHeap example contains a bug, as reported by [9]. This bug requires a sequence length of 13 inserts and then a delete (variations are possible, but this is the shortest sequence to show the bug). Our technique cannot scale to 14 operations as yet, so we hardcoded the sequence operations, but left the parameters symbolic. We also added a test after the sequence to see if the bug was triggered; if so a special coverage location was reached. The coverage results indicated that the bug was triggered with exactly the same probability as location 5 (in the `merge` helper method) being reached. On further investigation by doing random sequences based on the probability values, we discovered that in fact it seems location 5 is only reached on buggy sequences. Although we don't yet know exactly what the bug is, it seems the error is dependent on location 5 being reached. In [9] it is stated that the error is in the `extractMin` helper method[2]. The probability calculations seems therefore to also be of value in fault localization. Note also that location 5 is in the `merge` helper method and is reached when the condition is `false` to an `if` that has no `else` branch; it definitely seems like there is some *missing* code that needs to go in the `else` case.

4.1 Optimizations

In Section 3.2 and Section 3.5 slicing and memoization are introduced and in this section we will evaluate the performance improvement these two optimizations provide.

Table 2 shows the results for the optimizations on BinomialHeap and TreeMap both with a sequence length of 4. Again note that the variable domain plays no role in the performance. For each subject we show the optimizations (slicing and/or memoization) that is switched on (✔) and off (✗) in the analysis. We don't believe switching them both off is worth the effort, since is it is rather clear that would take a very long time to complete. The columns show the *Subject*, the reduction percentage in the cumulative sizes of the path conditions (*PC Red*) and the number of variables (*Var Red*) in the path conditions achieved by slicing, *Probs* indicate the number of probabilities that was calculated, *LattE* the number of times the LattE solver was invoked, *Memoized* indicates the number of times the result was already calculated and thus LattE didn't need to be invoked and lastly the time spent in the LattE solver (*LattE time*) and the *Total time* (both in seconds).

The results show that both optimizations are important for making the technique tractable, but clearly slicing plays a significantly larger role in improving the performance. Switching memoization off increases the runtime by 1.47× for Bi-

[2]From personal communications we know that they also believe it is in `merge` that is called by `extractMin`

Table 1: Probability of covering branches in BinomialHeap

Loc	PCs	0...9	0...49	0...99	0...499	Random 1000	Random 10000
1	32	6.4451×10^{-1}	7.2973×10^{-1}	7.3993×10^{-1}	7.4799×10^{-1}	817123	8213625
4	14	2.2831×10^{-1}	2.9414×10^{-1}	3.0322×10^{-1}	3.1062×10^{-1}	412440	4139439
6	30	6.0350×10^{-1}	6.7208×10^{-1}	6.7989×10^{-1}	6.8599×10^{-1}	445493	4449974
7	16	2.6932×10^{-1}	3.5179×10^{-1}	3.6326×10^{-1}	3.7263×10^{-1}	784070	7903090
8	14	2.2831×10^{-1}	2.9414×10^{-1}	3.0322×10^{-1}	3.1062×10^{-1}	412440	4139439
9	34	3.8161×10^{-1}	4.0192×10^{-1}	4.0412×10^{-1}	4.0583×10^{-1}	764453	7698128
10	32	3.0389×10^{-1}	3.8546×10^{-1}	3.9584×10^{-1}	4.0416×10^{-1}	424300	4279148
13	8	7.6499×10^{-3}	2.2814×10^{-3}	1.1945×10^{-3}	2.4775×10^{-4}	0	1
14	54	1.1863×10^{-1}	2.6729×10^{-2}	1.3556×10^{-2}	2.7422×10^{-3}	1	1
15	8	7.6499×10^{-3}	2.2814×10^{-3}	1.1945×10^{-3}	2.4775×10^{-4}	0	1
16	36	5.5518×10^{-2}	1.4164×10^{-2}	7.2893×10^{-3}	1.4915×10^{-3}	1	1
17	22	6.6206×10^{-2}	1.3669×10^{-2}	6.8555×10^{-3}	1.3742×10^{-3}	0	1
18	28	4.7868×10^{-2}	1.1882×10^{-2}	6.0947×10^{-3}	1.2437×10^{-3}	1	0
19	4	4.5562×10^{-3}	1.1764×10^{-3}	6.0643×10^{-4}	1.2425×10^{-4}	0	0
20	8	7.6499×10^{-3}	2.2814×10^{-3}	1.1945×10^{-3}	2.4775×10^{-4}	0	1
21	18	2.6099×10^{-2}	7.0089×10^{-3}	3.6261×10^{-3}	7.4500×10^{-4}	1	0

Table 2: Slicing and Memoization Optimizations

Subject	Memoization	Slicing	PC Red	Var Red	Probs	LattE	Memoized	LattE time (s)	Total time (s)
Binomial	✔	✔	55%	67%	634	518	370	35	57
	✗	✔	55%	67%	634	888	0	61	84
	✔	✗	0%	0%	634	3160	698	388	414
TreeMap	✔	✔	44%	55%	766	2264	562	118	145
	✗	✔	44%	55%	766	2826	0	150	178
	✔	✗	0%	0%	766	12108	4965	1028	1056

nomialHeap and 1.23× for TreeMap, but switching slicing off increases runtime for both examples by 7.3×!

This is due to the fact that slicing has two benefits. First, it results in smaller sliced PCs which in turn leads to smaller inequality systems, both in terms of the number of inequalities and the number of variables, that are passed to LattE. Without slicing (i.e., memoization only) does not reduce the size of the PC. LattE's runtime is known to be strongly dependent on both the number of inequalities and the dimension of the polytope [4], so this helps performance. Second, when PCs are sliced there are more opportunities for memoization to reuse computations from different parts of the symbolic execution tree. Without slicing this type of reuse would not be possible, since paths that differ in some branch would never be amenable to memoization.

Memoization can exploit the opportunities presented by slicing, but memoization also helps mitigate the combinatorial blowup in calls to LattE due to the presence of disequalities even when slicing is disable. The data bear this out. When slicing is disabled for TreeMap 4695/17073 (29%) of the LattE calls are memoized – this memoization is due exclusively to the presence of disequality constraints. With slicing enabled only 562/2826 (19.8%) of the LattE calls are memoized. This drop in the percentage of memoized calls is because now memoization happens on the sliced PCs, short-circuiting the need to memoize the combinatorially many calls due to disequalities that arise in evaluating Equation 1.

Currently Symbolic PathFinder does not support slicing for regular feasibility checks, but on the strength of these reductions we intend to introduce this optimization also for classic symbolic execution.

5. RELATED WORK

Our work can be seen as a form of profiling. The idea of path profiling is not new [7, 28]. So-called "hot paths" are useful for optimization (especially branch prediction) and test generation, and researchers have used profiling [1], static analysis [3], and — most recently — symbolic execution [18, 20] to determine which paths are hot. The latter work is clearly the closest to our own. The authors' starting point is the calculation of path frequencies. They use symbolic execution to generate path conditions and volume computation to calculate the frequencies and, later, path probabilities. Unfortunately, the description is quite brief (only four pages) and while they mention the possibility of PC slicing, they do not develop it nor memoization which make such a significant difference in our work.

One view of our work is as an enhanced form of static analysis. Garbervetsky et al. [10, 11] use a form of model counting similar to ours to calculate the number of visits to an allocation site which is then used to predict memory regions associated with Java methods. A conjunction of local invariants along a control path is taken and model counting is used to derive from this a parameterized formula that counts the number of solutions to the conjunction.

Another view of our work is that we enhance the semantics of programs with probabilities. Others have taken a more formal approach to this idea: Morgan and McIver extend weakest precondition analysis [23] while Monniaux bases his work on denotational semantics [22]. Unfortunately this work remains limited to manual analyses. It is worth pointing out that this work (and ours) is different from probabilistic model checking [17], where the probabili-

ties of transitions are known a priori.

Model counting is frequently used for Artificial Intelligence problems (such as bounded-length adversarial and contingency planning, and probabilistic reasoning, including Bayesian net reasoning [29]) and for hard combinatorial problems, such as combinatorial designs. Gomes et al. [12] is a good survey of model counting and its applications.

We showed two applications of our work related to testing: probability of obtaining coverage and bug finding/localization. Random testing is a well studied field and it works on the assumption that the probability of obtaining good coverage is fairly high (see [30] for an extensive survey on how well random testing works for containers, including the ones used in this paper). Our work can be seen as an approach to quantify just how well random testing will work for a specific program. Bug localization is an equally well studied field and the spectrum based techniques (for example, see Tarantula [14]) focussing on the *suspiciousness* of a statement is most closely related to our approach. The basic approach is to consider how often a line of code appears in failing and passing runs. We believe probabilities can greatly enhance this form of fault localization.

6. CONCLUSION AND FUTURE WORK

Recent years have witnessed an increasing trend in program analyses targetted at non-functional properties. For example, several researchers [2, 13, 36] have used symbolic execution to attempt to characterize the performance of programs. This requires a shift from asking questions about whether a program execution is possible or not – a decision question – to quantifying characteristics of a program executions.

In this paper, we have introduced an approach that extends symbolic execution to perform a specific type of quantitative analysis – the calculation of path probabilities. We believe that there are many possible applications for such an analysis and we have illustrated three such applications in this paper. One might also imagine adapting existing approaches to differential program analysis [25, 26] to prioritize the analysis of program differences that are most likely to be executed. There have been several approaches suggested in the literature for directing symbolic execution to more profitable portions of the execution tree using heuristics [35, 21]. It may be fruitful to explore how probabilities computed during symbolic execution might direct the progress symbolic execution. For example, to bias test generation to unlikely execution paths.

Our probabilistic symbolic execution extension now forms part of the standard release of SPF. Scalability is an issue: our technique is more expensive than "pure" symbolic execution, but any program that can be analysed with the latter, is also amenable to the former. Moreover, the cost of invoking LattE is, in our experience, independent of the variable domain sizes. We plan to explore several further optimizations to its performance. We have, for instance, prototyped schemes for computing different canonical column and row orders to increase the effectiveness of memoization. We also have prototyped a new form of symmetry reduction that has the potential to significantly reduce execution cost. In addition to performance optimizations, we plan to extend the model counter support with more theories, e.g., Linear Real Arithmetic and Strings, and to support probability mass and distribution functions using LattE's integration support. Currently we also don't support references, but we believe an extension to support lazy initialization of reference types as in [15] is not hard.

Finally, the experience of classic symbolic execution has taught us that there will always be programs for which decision procedure support is lacking. We expect model counting to be limited in much the same way, thus we will explore the use of statistical sampling techniques to provide probability estimates in cases where model counting is ineffective. This will allow for a type of symbolic-concrete probabilistic symbolic execution.

7. ACKNOWLEDGMENTS

We would like to thank Steve Kroon, Aline Uwimbabazi, and Brink van der Merwe for fruitful discussions about the development of the ideas in this paper. Matthew Dwyer's work was supported, in part, by a Research Award from the United State's Fulbright Scholar Program to visit Stellenbosch University, South Africa and by the AFOSR under Award #FA9550-10-1-0406.

8. REFERENCES

[1] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 46–57. IEEE, Dec. 1996.

[2] J. Burnim, S. Juvekar, and K. Sen. Wise: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering*, pages 463–473, May 2009.

[3] R. P. L. Buse and W. Weimer. The road not taken: Estimating path execution frequency statically. In *Proceedings of the 31st International Conference on Software Engineering*, pages 144–154. ACM, May 2009.

[4] J. A. De Loera, B. Dutra, M. Köppe, S. Moreinis, G. Pinto, and J. Wu. Software for exact integration of polynomials over polyhedra. arXiv:1108.0117v2 [math.MG], 2011.

[5] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS #4963, pages 337–340. Springer, 2008.

[6] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Software Eng.*, 17(9):900–910, Sept. 1991.

[7] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Computers*, C–30(7):478–490, July 1981.

[8] J. Galambos and I. Simonelli. *Bonferroni-Type Inequalities with Applications*. Springer-Verlag, 1996.

[9] J. P. Galeotti, N. Rosner, C. L. Pombo, and M. F. Frias. Analysis of invariants for efficient bounded verification. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 25–36, July 2010.

[10] D. Garbervetsky, S. Yovine, V. A. Braberman, M. Rouaux, and A. Taboada. On transforming Java-like programs into memory-predictable code. In *Proceedings of the 7th International Workshop on*

Java Technologies for Real-Time and Embedded Systems, pages 140–149. ACM, Sept. 2009.

[11] D. Garbervetsky, S. Yovine, V. A. Braberman, M. Rouaux, and A. Taboada. Quantitative dynamic-memory analysis for Java. *Concurrency and Computation: Practice and Experience*, 23(14):1665–1678, Sept. 2011.

[12] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 633–654. IOS Press, 2009.

[13] S. Gulwani. SPEED: Symbolic complexity bound analysis. In *Proceedings of the 21st International Conference on Computer Aided Verification*, LNCS #5643, pages 51–62. Springer, June–July 2009.

[14] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, pages 273–282. ACM, Nov. 2005.

[15] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS #2619, pages 553–568. Springer, Apr. 2003.

[16] M. Köppe. A primal Barvinok algorithm based on irrational decompositions. *SIAM J. Discrete Math.*, 21(1):220–236, 2007.

[17] M. Z. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS #2280, pages 52–66. Springer, Apr. 2002.

[18] S. Liu and J. Zhang. Program analysis: from qualitative analysis to quantitative analysis. In *Proceedings of the 33rd International Conference on Software Engineering – NIER Track*, pages 956–959. ACM, May 2011.

[19] J. A. D. Loera, R. Hemmecke, J. Tauzer, and R. Yoshida. Effective lattice point counting in rational convex polytopes. *J. Symb. Comput.*, 38(4):1273–1302, Oct. 2004.

[20] F. Ma, S. Liu, and J. Zhang. Volume computation for Boolean combination of linear arithmetic constraints. In *Proceedings of the 22nd International Conference on Automated Deduction*, LNCS #5663, pages 453–468. Springer, Aug. 2009.

[21] K.-K. Ma, K. Yit Phang, J. Foster, and M. Hicks. Directed symbolic execution. In *Proceedings of the 18th International Static Analysis Symposium*, LNCS #6887, pages 95–111. Springer, Sept. 2011.

[22] D. Monniaux. Abstract interpretation of probabilistic semantics. In *Proceedings of the 7th International Static Analysis Symposium*, LNCS #1824, pages 322–339. Springer, June 2000.

[23] C. C. Morgan and A. K. McIver. *pGCL*: Formal reasoning for random algorithms. *South African Comp Jnl*, 22:14–27, Mar. 1999.

[24] G. J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.

[25] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 226–237, Nov. 2008.

[26] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 504–515, June 2011.

[27] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 179–180. ACM, Sept. 2010.

[28] G. Ramalingam. Data flow frequency analysis. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 267–277, May 1996.

[29] T. Sang, P. Beame, and H. A. Kautz. Performing Bayesian inference by weighted model counting. In *Proceedings of the 20th National Conference on Artificial Intelligence and the 17th Innovative Applications of Artificial Intelligence Conference*, pages 475–482. AAAI Press / The MIT Press, July 2005.

[30] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov. Testing container classes: Random or systematic? In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering*, LNCS #6603, pages 262–277. Springer, Mar.–Apr. 2011.

[31] UC Davis, Mathematics. Latte integrale. `http://www.math.ucdavis.edu/~latte`.

[32] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, Apr. 2003.

[33] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for Java containers using state matching. In L. L. Pollock and M. Pezzè, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 37–48. ACM, July 2006.

[34] R. Williams. Triangle classification problem. `http://russcon.org/triangle_classification.html`.

[35] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 359–368, June–July 2009.

[36] P. Zhang, S. G. Elbaum, and M. B. Dwyer. Automatic generation of load tests. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 43–52, Nov. 2011.