Probabilistic Program Analysis

Symbolic Execution + Model Counting + Reinforcement Learning

Matthew B. Dwyer

Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, Nebraska USA

Summer 2014

Program Correctness

How do you tell if your program works correctly?

(hint: this is a two part answer)

Program Correctness

How do you tell if your program works correctly?

(hint: this is a two part answer)

- Specify what it means to be correct

Program Correctness

How do you tell if your program works correctly?

(hint: this is a two part answer)

- Specify what it means to be correct
- Provide evidence that the program's behavior matches the specification

Program Correctness

How can you specify what it means to be correct?

Program Correctness

How can you specify what it means to be correct?

- oracle: $completeTriple(a, b) = c \iff \sqrt{a^2 + b^2} = c \vee \sqrt{a^2 + c^2} = b \vee \ldots$

Program Correctness

How can you specify what it means to be correct?

- oracle: $completeTriple(a, b) = c \iff \sqrt{a^2 + b^2} = c \lor \sqrt{a^2 + c^2} = b \lor \ldots$
- assert: `assert a > 0;`

Program Correctness

How can you specify what it means to be correct?

- oracle: $\mathit{completeTriple}(a, b) = c \iff \sqrt{a^2 + b^2} = c \vee \sqrt{a^2 + c^2} = b \vee \ldots$
- assert: `assert a > 0;`

What kind of evidence can you provide?

- read the code

Program Correctness

How can you specify what it means to be correct?

- oracle: $completeTriple(a, b) = c \iff \sqrt{a^2 + b^2} = c \lor \sqrt{a^2 + c^2} = b \lor \ldots$
- assert: `assert a > 0;`

What kind of evidence can you provide?

- read the code
- testing: $completeTriple(3, 4) = c \iff \sqrt{3^2 + 4^2} = \sqrt{25} = 5 = c \lor \ldots$

Program Correctness

How can you specify what it means to be correct?

- oracle: $\mathit{completeTriple}(a, b) = c \iff \sqrt{a^2 + b^2} = c \vee \sqrt{a^2 + c^2} = b \vee \ldots$
- assert: `assert a > 0;`

What kind of evidence can you provide?

- read the code
- testing: $\mathit{completeTriple}(3, 4) = c \iff \sqrt{3^2 + 4^2} = \sqrt{25} = 5 = c \vee \ldots$

Wouldn't it be nice if you could systematically check all/most of the program's behavior against the specification?

Program analysis in a nutshell

Consider the following program fragment:

```
int classify(int a, int b, int c) {
  if (a==b)
    type+=1;
  if (a==c)
    type+=2;
  if (b==c)
    type+=3;
  assert type != 6; ...
```

Is there a trace that leads to the `assert` being violated?

Program analysis in a nutshell

Consider the following program fragment:

```
1 int classify(int a, int b, int c) {
2   if (a==b)
3     type+=1;
4   if (a==c)
5     type+=2;
6   if (b==c)
7     type+=3;
8   assert type != 6; ...
```

A **set** of program traces

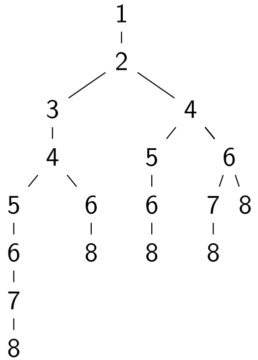
$$\{[1,2,4,6,8],\ [1,2,3,4,6,8],\ [1,2,4,5,6,8],\ [1,2,4,6,7,8],\ \ldots\}$$

Is there a trace that leads to the `assert` being violated?

Consider the following program fragment:

```
1 int classify(int a, int b, int c) {
2   if (a==b)
3     type+=1;
4   if (a==c)
5     type+=2;
6   if (b==c)
7     type+=3;
8   assert type != 6; ...
```

A **tree** of program traces



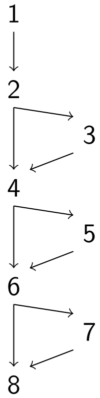
Is there a trace that leads to the assert being violated?

Program analysis in a nutshell

Consider the following program fragment:

```
1 int classify(int a, int b, int c) {
2   if (a==b)
3     type+=1;
4   if (a==c)
5     type+=2;
6   if (b==c)
7     type+=3;
8   assert type != 6; ...
```

A **graph** of program traces



Is there a trace that leads to the `assert` being violated?

Program Analysis in a nutshell

Graphs can encode large sets of program traces

Program Analysis in a nutshell

Graphs can encode large sets of program traces

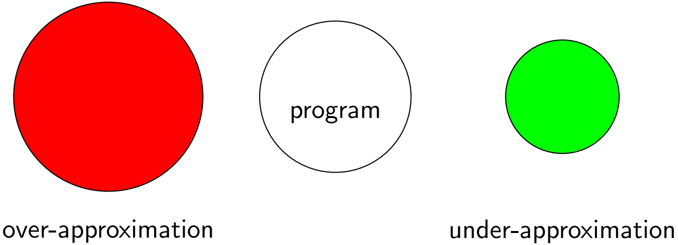
- exploit well-understood graph algorithms for analysis
- e.g., DFS, intersection/union over prefixes/suffixes of paths, etc.

Program Analysis in a nutshell

Graphs can encode large sets of program traces

- exploit well-understood graph algorithms for analysis
- e.g., DFS, intersection/union over prefixes/suffixes of paths, etc.

... but they approximate the set of traces

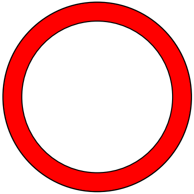


Program Analysis in a nutshell

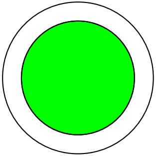
Graphs can encode large sets of program traces

- exploit well-understood graph algorithms for analysis
- e.g., DFS, intersection/union over prefixes/suffixes of paths, etc.

... but they approximate the set of traces



extra behavior

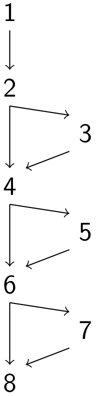


missing behavior

Where is the overapproximation?

```
1 int classify(int a, int b, int c) {
2   if (a==b)
3     type+=1;
4   if (a==c)
5     type+=2;
6   if (b==c)
7     type+=3;
8   assert type != 6; ...
```

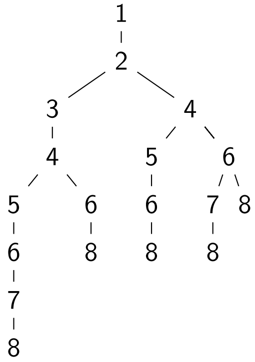
A **graph** of program traces



Underapproximating Symbolic Execution

```
1 int classify(int a, int b, int c) {
2   if (a==b)
3     type+=1;
4   if (a==c)
5     type+=2;
6   if (b==c)
7     type+=3;
8   assert type != 6; ...
```

A **tree** of program traces



Basic symbolic execution algorithm

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

while $\neg branch(l)$ **do**
    $m \leftarrow m\langle v, e\rangle$
    $l \leftarrow next(l)$
end while
$c \leftarrow m[cond(l)]$
if $\mathrm{SAT}(\phi \wedge c)$ **then**
    `symbolicExecute`$(target(l), \phi \wedge c, m)$
end if
if $\mathrm{SAT}(\phi \wedge \neg c)$ **then**
    `symbolicExecute`$(next(l), \phi \wedge \neg c, m)$
end if

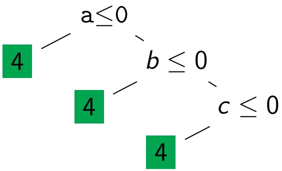
A simple example ...

```
int classify(int a, int b, int c) {
  if (a<=0 || b<=0 || c<=0) return 4;
  int type=0;
  if (a==b) type+=1;
  if (a==c) type+=2;
  if (b==c) type+=3;
  if (type==0) {
    if (a+b<=c || b+c<=a || a+c>=b) type=4;
    else type=1;
    return type;
  }
  if (type>3) type=3;
  else if (type==1 && a+b>c) type=2;
  else if (type==2 && a+c>b) type=2;
  else if (type==3 && b+c>a) type=2;
  else type=4;
  return type;
}
```

A simple example ...

```
int classify(int a, int b, int c) {
  if (a<=0 || b<=0 || c<=0) return 4;
  int type=0;
  if (a==b) type+=1;
  if (a==c) type+=2;
  if (b==c) type+=3;
  if (type==0) {
    if (a+b<=c || b+c<=a || a+c>=b) type=4;
    else type=1;
    return type;
  }
  if (type>3) type=3;
  else if (type==1 && a+b>c) type=2;
  else if (type==2 && a+c>b) type=2;
  else if (type==3 && b+c>a) type=2;
  else type=4;
  return type;
}
```

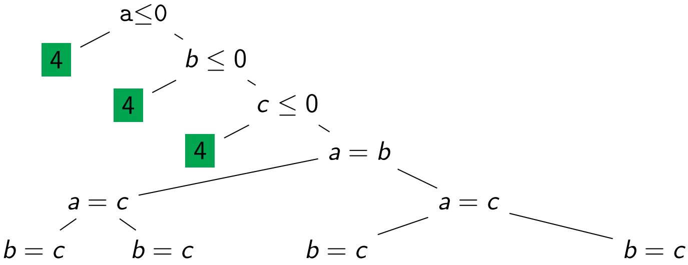
Symbolic execution tree



A simple example ...

```
int classify(int a, int b, int c) {
  if (a<=0 || b<=0 || c<=0) return 4;
  int type=0;
  if (a==b) type+=1;
  if (a==c) type+=2;
  if (b==c) type+=3;
  if (type==0) {
    if (a+b<=c || b+c<=a || a+c>=b) type=4;
    else type=1;
    return type;
  }
  if (type>3) type=3;
  else if (type==1 && a+b>c) type=2;
  else if (type==2 && a+c>b) type=2;
  else if (type==3 && b+c>a) type=2;
  else type=4;
  return type;
}
```

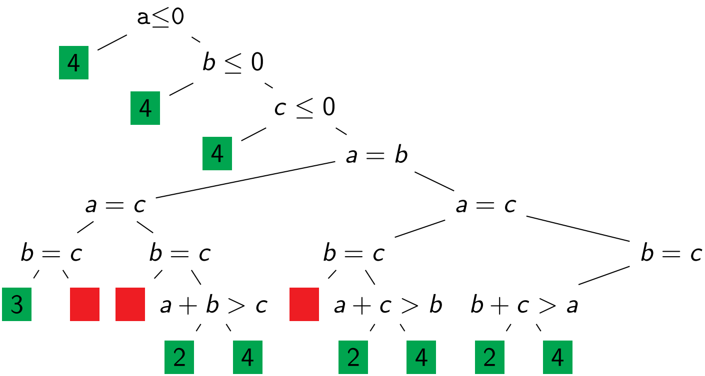
Symbolic execution tree



A simple example ...

```
int classify(int a, int b, int c) {
  if (a<=0 || b<=0 || c<=0) return 4;
  int type=0;
  if (a==b) type+=1;
  if (a==c) type+=2;
  if (b==c) type+=3;
  if (type==0) {
    if (a+b<=c || b+c<=a || a+c>=b) type=4;
    else type=1;
    return type;
  }
  if (type>3) type=3;
  else if (type==1 && a+b>c) type=2;
  else if (type==2 && a+c>b) type=2;
  else if (type==3 && b+c>a) type=2;
  else type=4;
  return type;
}
```

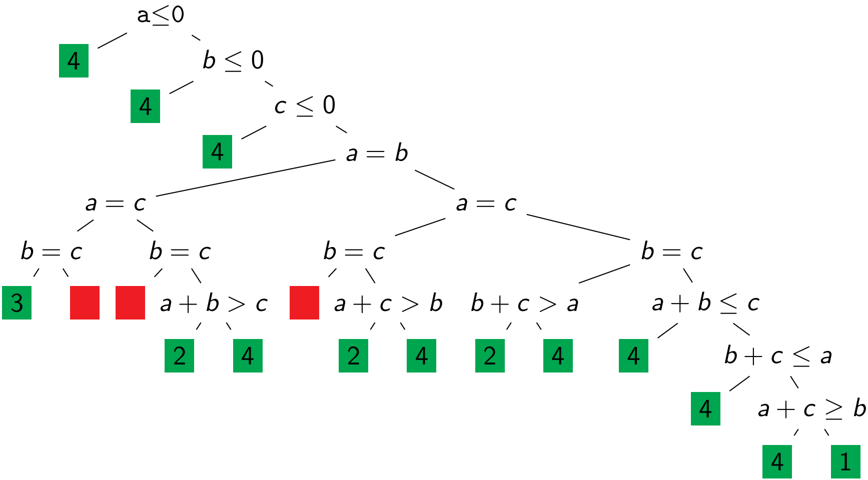
Symbolic execution tree



A simple example ...

```
int classify(int a, int b, int c) {
  if (a<=0 || b<=0 || c<=0) return 4;
  int type=0;
  if (a==b) type+=1;
  if (a==c) type+=2;
  if (b==c) type+=3;
  if (type==0) {
    if (a+b<=c || b+c<=a || a+c>=b) type=4;
    else type=1;
    return type;
  }
  if (type>3) type=3;
  else if (type==1 && a+b>c) type=2;
  else if (type==2 && a+c>b) type=2;
  else if (type==3 && b+c>a) type=2;
  else type=4;
  return type;
}
```

Symbolic execution tree



Some observations

- There are 14 distinct paths (Green): from 1 to 9 branches
- 1 path returns "scalene" (1); 3 return "isoscelese" (2); and 1 returns "equilateral" (3)
- 3 paths are pruned because constraints are unsat (Red)
- Interesting symmetries are involved in the "isoscelese" and unsat cases

Analyzing program behavior

The symbolic execution could be used to ...

- check contracts, e.g., if $a = b = c$ then $return = 3$
- generate a suite of 14 thorough tests
- demonstrate that it is possible to return values $[1, 4]$

... and support answering other yes/no questions about program behavior.

Moving beyond yes/no questions

We are interested in exploring how to ...

- determine the chance that a contract holds (1 is a special case)
- focus testing on *rare* paths (likely ones are easy to hit)
- determine how frequently a given value is returned

Moving beyond yes/no questions

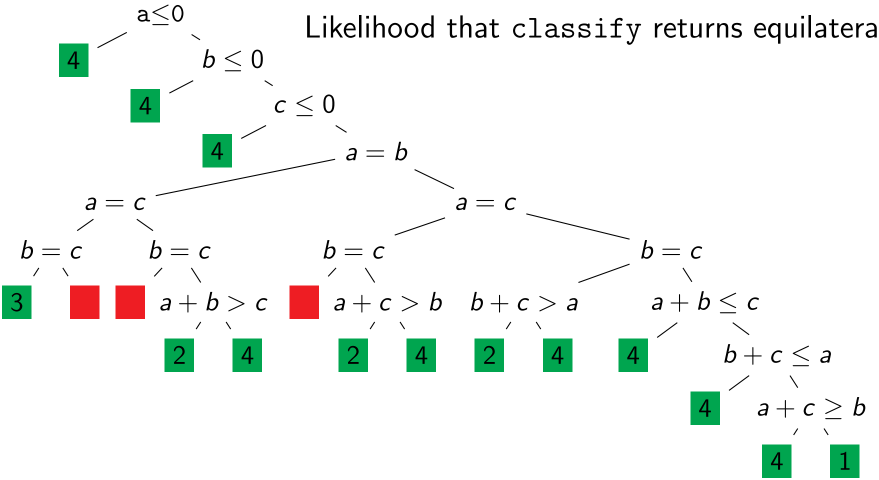
We are interested in exploring how to ...

- determine the chance that a contract holds (1 is a special case)
- focus testing on *rare* paths (likely ones are easy to hit)
- determine how frequently a given value is returned

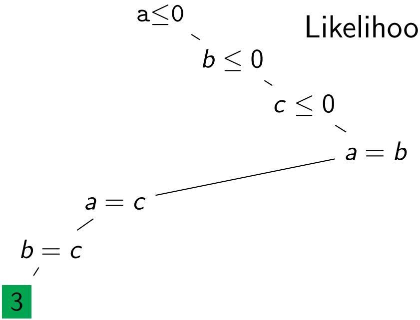
When yes/no questions cannot be answered can we quantify what was discovered during program analysis?

Assume `ints` are drawn uniformly from $[-1000, 1000]$

Likelihood that `classify` returns equilateral?



Assume `ints` are drawn uniformly from $[-1000, 1000]$



Likelihood that `classify` returns equilateral?

Assume `ints` are drawn uniformly from $[-1000, 1000]$

Likelihood that `classify` returns equilateral?

$a \leq 0$

$b \leq 0$

$c \leq 0$

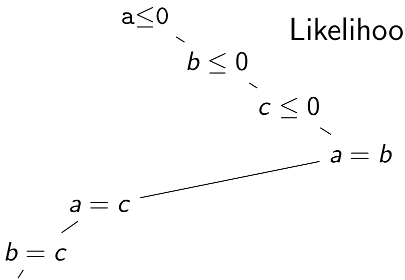
$a = b$

$a = c$

$b = c$

$\boxed{3}$ $\neg(a \leq 0) \wedge \neg(b \leq 0) \wedge \neg(c \leq 0) \wedge (a = b) \wedge (a = c) \wedge (b = c)$

Assume `ints` are drawn uniformly from $[-1000, 1000]$



Likelihood that `classify` returns equilateral?

3 $\neg(a \leq 0) \wedge \neg(b \leq 0) \wedge \neg(c \leq 0) \wedge (a = b) \wedge (a = c) \wedge (b = c)$

How many inputs satisfy this path condition?

Assume `ints` are drawn uniformly from $[-1000, 1000]$

Likelihood that `classify` returns equilateral?

a≤0

$b \leq 0$

$c \leq 0$

$a = b$

$a = c$

$b = c$

$\boxed{3}$ $\neg(a \leq 0) \wedge \neg(b \leq 0) \wedge \neg(c \leq 0) \wedge (a = b) \wedge (a = c) \wedge (b = c)$

How many inputs satisfy this path condition?

1000

Assume `ints` are drawn uniformly from $[-1000, 1000]$

a≤0

$b \leq 0$

$c \leq 0$

$a = b$

$a = c$

$b = c$

$\boxed{3}$ $\neg(a \leq 0) \wedge \neg(b \leq 0) \wedge \neg(c \leq 0) \wedge (a = b) \wedge (a = c) \wedge (b = c)$

Likelihood that `classify` returns equilateral?

$1.25e^{-7}$

How many inputs satisfy this path condition?

1000

Some observations

- You may be able to calculate these probabilities because you know about triangles.

Some observations

- You may be able to calculate these probabilities because you know about triangles.
- We want to calculate the probability that a program's execution ...
  - returns a value, reaches a statement, ...

Some observations

- You may be able to calculate these probabilities because you know about triangles.
- We want to calculate the probability that a program's execution ...
  - returns a value, reaches a statement, ...
- Adapting symbolic execution to perform these computations involves ...
  - calculating the paths of interest
  - calculating the probability of taking those paths
  - combining those probabilities appropriately

UNIVERSITY OF Nebraska Lincoln

Probabilistic symbolic execution algorithm

Algorithm 2 `probSymbolicExecute`$(l, \phi, m, \boxed{\texttt{p}})$

while $\neg branch(l)$ **do**
  $m \leftarrow m\langle v, e\rangle$
  $l \leftarrow next(l)$
end while
$c \leftarrow m[cond(l)]$
$\boxed{\phi' \leftarrow slice(\phi, c)}$
$\boxed{p_c \leftarrow prob(\phi' \wedge c)/prob(\phi')}$
if $\boxed{p_c > 0}$ **then**
  `probSymbolicExecute`$(target(l), \phi \wedge c, m, \boxed{p * p_c})$
end if
if $\boxed{p_c < 1}$ **then**
  `probSymbolicExecute`$(next(l), \phi \wedge \neg c, m, \boxed{p * (1 - p_c)})$
end if

Key algorithmic features

Slicing the path condition, i.e., $\mathit{slice}(\phi, c)$

- reduces formula size which reduces cost of $\mathit{prob}(\cdot)$
- exposes opportunities for reusing computation in $\mathit{prob}(\cdot)$

Key algorithmic features

Slicing the path condition, i.e., $slice(\phi, c)$

- reduces formula size which reduces cost of $prob(\cdot)$
- exposes opportunities for reusing computation in $prob(\cdot)$

Calculating the conditional probability p_c of c

- requires model counting of path condition
- determines satisfiability of branches, i.e., $p_c > 0 \implies SAT(\phi \wedge c)$
- allows inference of off-branch probability, i.e., $p * (1 - p_c)$
- depth-first nature of symbolic execution ensures that $prob(\phi')$ will be reused
- slicing assures independence in computing p_c, i.e., $\phi - \phi'$ factored out

Key algorithmic features

Slicing the path condition, i.e., *slice*(ϕ, c)

- reduces formula size which reduces cost of *prob*$(\cdot)$
- exposes opportunities for reusing computation in *prob*$(\cdot)$

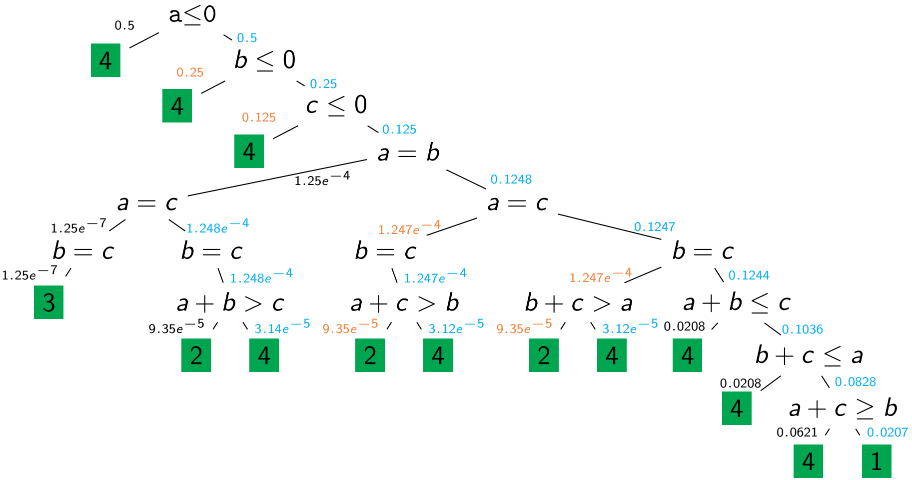
Calculating the conditional probability p_c of c

- requires model counting of path condition
- determines satisfiability of branches, i.e., $p_c > 0 \implies SAT(\phi \wedge c)$
- allows inference of off-branch probability, i.e., $p * (1 - p_c)$
- depth-first nature of symbolic execution ensures that *prob*(ϕ') will be reused
- slicing assures independence in computing p_c, i.e., $\phi - \phi'$ factored out

This allows the algorithm to compute path probabilities cost-effectively.

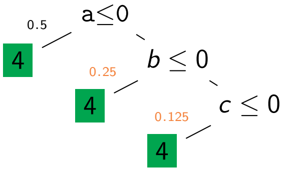
University of Nebraska Lincoln

Probabilistic symbolic execution tree

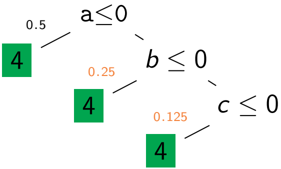


29 branches: 8 counting queries, 6 reused, 15 inferred

Probabilistic symbolic execution tree

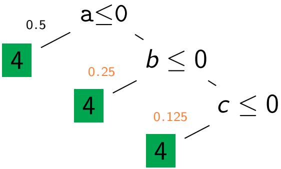


Probabilistic symbolic execution tree



$$slice(true, a \leq 0) = true$$

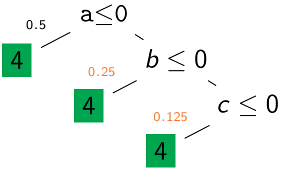
Probabilistic symbolic execution tree



$$slice(true, a \leq 0) = true$$

$$p_c = prob(a \leq 0)/prob(true)$$

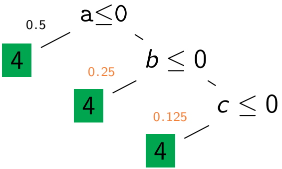
Probabilistic symbolic execution tree



$$slice(true, a \leq 0) = true$$
$$p_c = prob(a \leq 0)/prob(true)$$

$$slice(a \leq 0 \wedge b \leq 0, c \leq 0) = true$$

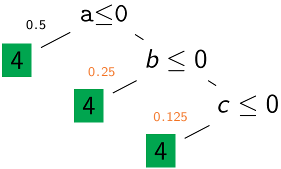
Probabilistic symbolic execution tree



$$slice(true, a \leq 0) = true$$
$$p_c = prob(a \leq 0)/prob(true)$$

$$slice(a \leq 0 \wedge b \leq 0, c \leq 0) = true$$
$$p_c = prob(c \leq 0)/prob(true)$$

Probabilistic symbolic execution tree



$$slice(true, a \leq 0) = true$$
$$p_c = prob(a \leq 0)/prob(true)$$

$$slice(a \leq 0 \wedge b \leq 0, c \leq 0) = true$$
$$p_c = prob(c \leq 0)/prob(true)$$

Normalization of constraints, e.g., $a \mapsto v_1$, $c \mapsto v_1$, enables reuse in calculating p_c

Calculating $prob(\cdot)$

We report on support for linear integer arithmetic (LIA) constraints using LattE

- computes the number of *lattice* points in a convex polytope;
- constraints encoded as system of inequalities, $Ax \leq B$;
- does not support disjunction or disequality constraints, i.e., $x \neq c$

Calculating $prob(\cdot)$

We report on support for linear integer arithmetic (LIA) constraints using LattE

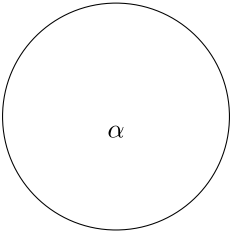
- computes the number of *lattice* points in a convex polytope;
- constraints encoded as system of inequalities, $Ax \leq B$;
- does not support disjunction or disequality constraints, i.e., $x \neq c$

Our calculation relies on "counting" the number of solutions of a set of related constraints using LattE and combining the results.

- $count = count_{\wedge}(\bigwedge_{ineqSet}) - count_{\vee}(\bigvee_{exSet})$
- `return` $count / \prod_{v \in vars} dom(v)$

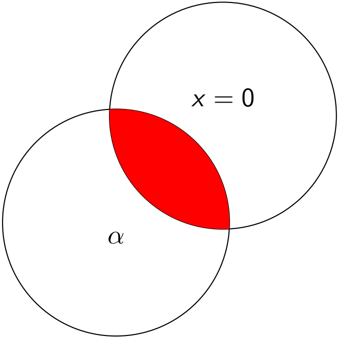
$\alpha \wedge (x \neq 0) \wedge (y \neq 0)$ cannot be expressed directly

$\alpha \wedge (x \neq 0) \wedge (y \neq 0)$ cannot be expressed directly



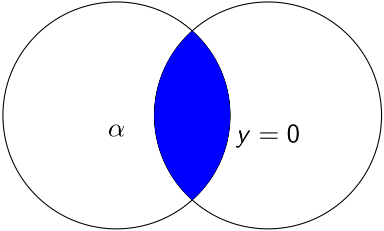
Count the solutions to α

$\alpha \wedge (x \neq 0) \wedge (y \neq 0)$ cannot be expressed directly



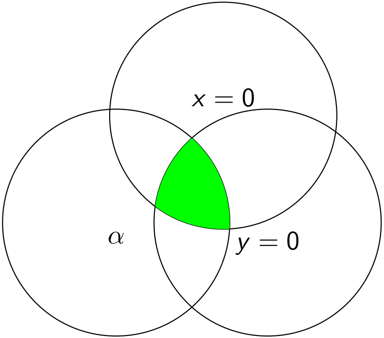
Remove the count of solutions to $\alpha \wedge (x = 0)$

$\alpha \wedge (x \neq 0) \wedge (y \neq 0)$ cannot be expressed directly



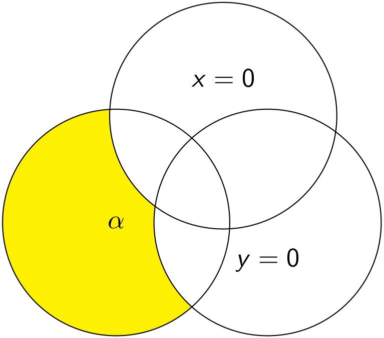
Remove the count of solutions to $\alpha \wedge (y = 0)$

$\alpha \wedge (x \neq 0) \wedge (y \neq 0)$ cannot be expressed directly



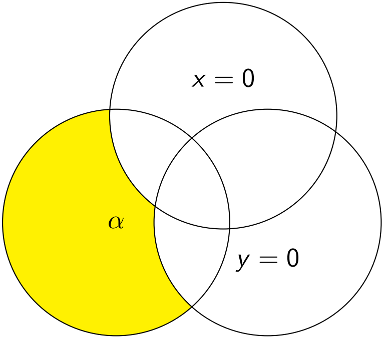
Add back the count of solutions to $\alpha \wedge (x = 0) \wedge (y = 0)$

$\alpha \wedge (x \neq 0) \wedge (y \neq 0)$ cannot be expressed directly



This results in the count of $\alpha \wedge (x \neq 0) \wedge (y \neq 0)$

$\alpha \wedge (x \neq 0) \wedge (y \neq 0)$ cannot be expressed directly



This results in the count of $\alpha \wedge (x \neq 0) \wedge (y \neq 0)$

Complexity is exponential in number of disequality constraints

Optimizing $count_{\wedge}(\cdot)$

Our experience with LattE revealed that its execution time

- is not dependent on the size of variable domains;
- is highly dependent on the number of variables (dimension of the polytope);
- is highly dependent on the number of constraints (faces of the polytope);

UNIVERSITY OF Nebraska Lincoln

Optimizing $count_{\wedge}(\cdot)$

Our experience with LattE revealed that its execution time

- is not dependent on the size of variable domains;
- is highly dependent on the number of variables (dimension of the polytope);
- is highly dependent on the number of constraints (faces of the polytope);

Our experience with sliced PCs revealed that

- a significant portion of the PC, and many variables, can be eliminated;
- sliced PCs, if normalized, recur throughout the symbolic execution tree

Optimizing $count_{\wedge}(\cdot)$

Our experience with LattE revealed that its execution time

- is not dependent on the size of variable domains;
- is highly dependent on the number of variables (dimension of the polytope);
- is highly dependent on the number of constraints (faces of the polytope);

Our experience with sliced PCs revealed that

- a significant portion of the PC, and many variables, can be eliminated;
- sliced PCs, if normalized, recur throughout the symbolic execution tree

Our implementation of $count_{\wedge}(\cdot)$

- normalizes the inequality system
- caches the counts computed for each system; and
- checks the cache before invoking LattE.

How well does this work?

Our ISSTA 2012 paper describes several usage scenarios ...

- finding a bug by looking at anomalies in path probabilities;
- assessing the probability of covering lines of code; and
- characterizing the likelihood of detected bugs

How well does this work?

Our ISSTA 2012 paper describes several usage scenarios ...

- finding a bug by looking at anomalies in path probabilities;
- assessing the probability of covering lines of code; and
- characterizing the likelihood of detected bugs

I'll focus on the necessity of the optimizations we've developed.

Memoization and Slicing are key

- Ran on Binomial Heap and TreeMap collections with all possible input sequences (adds, removes, etc.) of length 4;
- times reported in seconds

Memoization and Slicing are key

- Ran on Binomial Heap and TreeMap collections with all possible input sequences (adds, removes, etc.) of length 4;
- times reported in seconds

Subject	Memoize	Slice	PC Red.	Var Red.	*prob*(·)	LattE	Reused	LattE time	Total time
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

Probabilistic Symbolic Execution for Concurrency

Non-deterministic choice is used to model a lack of information

- e.g., details of OS scheduling algorithm

Probabilistic Symbolic Execution for Concurrency

Non-deterministic choice is used to model a lack of information

- e.g., details of OS scheduling algorithm

Two additional challenges

- What conditional probability should we use for a non-deterministic choice?
- State space explosion

We use a sampling-based analysis with reinforcement learning

Probabilistic Symbolic Execution for Concurrency

View program's behavior as a tree-structured Markov Decision Process

- symbolic execution tree with non-deterministic choice nodes

Probabilistic Symbolic Execution for Concurrency

View program's behavior as a tree-structured Markov Decision Process

- symbolic execution tree with non-deterministic choice nodes

Randomly sample paths, but unlike Monte Carlo methods

- each sampled path, p, contributes mass proportional to $count(PC_p)$
- a sampled path is biased against being revisited

Probabilistic Symbolic Execution for Concurrency

View program's behavior as a tree-structured Markov Decision Process

- symbolic execution tree with non-deterministic choice nodes

Randomly sample paths, but unlike Monte Carlo methods

- each sampled path, p, contributes mass proportional to $count(PC_p)$
- a sampled path is biased against being revisited

We compute the maximum probability of an event (e.g., `assert`)

- resolve non-deterministic choice to use max probability of child
- calculation is bottom up like value-iteration

Probabilistic Symbolic Execution for Concurrency

View program's behavior as a tree-structured Markov Decision Process

- symbolic execution tree with non-deterministic choice nodes

Randomly sample paths, but unlike Monte Carlo methods

- each sampled path, p, contributes mass proportional to $count(PC_p)$
- a sampled path is biased against being revisited

We compute the maximum probability of an event (e.g., `assert`)

- resolve non-deterministic choice to use max probability of child
- calculation is bottom up like value-iteration

MDP support works for any source of non-determinism, e.g., abstract post, unknown input values

Probabilistic Symbolic Execution for Concurrency

View program's behavior as a tree-structured Markov Decision Process

- symbolic execution tree with non-deterministic choice nodes

Randomly sample paths, but unlike Monte Carlo methods

- each sampled path, p, contributes mass proportional to $count(PC_p)$
- a sampled path is biased against being revisited

We compute the maximum probability of an event (e.g., `assert`)

- resolve non-deterministic choice to use max probability of child
- calculation is bottom up like value-iteration

MDP support works for any source of non-determinism, e.g., abstract post, unknown input values

Fastest and most precise method to date (submitted to ASE 2014)

Availability and Ongoing work

An SPF extension that implements our approach is available

- http://probsym.googlecode.com
- improved over results in paper; uses the Green solver interface
- http://green-solver.googlecode.com
- multiple MDP-based algorithms (exact and sampling based)

Availability and Ongoing work

An SPF extension that implements our approach is available

- `http://probsym.googlecode.com`
- improved over results in paper; uses the Green solver interface
- `http://green-solver.googlecode.com`
- multiple MDP-based algorithms (exact and sampling based)

We have added ...

- confidence-based model counting when exact counters are inefficient (PLDI'14);
- support for user defined input probability distributions; and
- parallelism to exploit sample independence

Probabilistic Program Analysis

Symbolic Execution + Model Counting + Reinforcement Learning

Matthew B. Dwyer

Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, Nebraska USA

Summer 2014