Software Testing, Quality Assurance and Maintenance Winter 2025

Lecture 7 — January 27, 2025

Patrick Lam *version 1*

Here's a program.

```
def Foo(x,y):
""" requires: x and y are int
    ensures: returns floor(max(xy)/min(x,y))"""
  if x > y:
    return x / y
  else:
    return y / x
```

We are going to use *symbolic execution* to automatically generate a test suite that:

- achieves full branch coverage;
- identifies dead code; and,
- discovers whether division by 0 is possible.

Symbolic Execution in a Nutshell. In four steps:

1. *transform* the program to add explicit tests for division by 0 (oracles);
2. *traverse* the program to compute each program path;
   path1: `x > y, y == 0`; path2: `x > y, y != 0, return x / y`; etc.
3. *solve* constraints for each path using a constraint (or logic) solver;
   path1: `x=10,y=0`; path2: `x=10,y=1`; etc.
4. *run* the program on tests generated by the previous step.

All testing is now automatic.

Here's the transformed program.

```
def Foo(x,y):
""" requires: x and y are int
    ensures: returns floor(max(xy)/min(x,y))"""
  if x > y:
    assert y != 0
    return x / y
  else:
    assert x != 0
    return y / x
```

Main Components of Symbolic Execution. As above, we have:

Traversing: automatically exploring program paths, executing the program on symbolic input values (vs the concrete values that the semantics operates on); forking execution at each branch; and recording branching conditions.

Solving constraints: deciding path feasibility, and generating test cases to get paths and to find bugs.

Traversing Paths. Let's enumerate all of the paths.

1. `x > y, y == 0`: assertion fails
2. `x > y, y != 0`: reaches `return x / y`
3. `x <= y, x == 0`: assertion fails
4. `x <= y, x != 0`: reaches `return y / x`

Solving Constraints. We generate a set of constraints and ask z3 to solve them for us:

```
(declare-fun x () Int)
(declare-fun y () Int)
(assert (> x y))
(assert (not (= y 0)))
(check-sat)
(get-model)
```

History of symbolic execution

Symbolic execution is actually quite old; here are references from 1975 by James C. King; and Robert S. Boyer, Bernard Elspas, and Karl N. Levitt: [BEL75, Kin75].

> Recent work on proving the correctness of programs by formal analysis [5] shows great promise and appears to be the ultimate technique for producing reliable programs. However, the practical accomplishments in this area fall short of a tool for routine use. Fundamental problems in reducing the theory to practice are not likely to be solved in the immediate future.

I will also point out that even if the software works perfectly (never crashes, for instance), you still don't know that the software is doing the right thing. You need SE 463 (requirements) for that.

In any case, in the 2000s, SAT solvers and their big brothers SMT solvers made SAT feasible in practice. In the context of formal verification, constraint solving is easy. Classical verification algorithms—with worst-case exponential running time—became viable in practice even if not in theory.

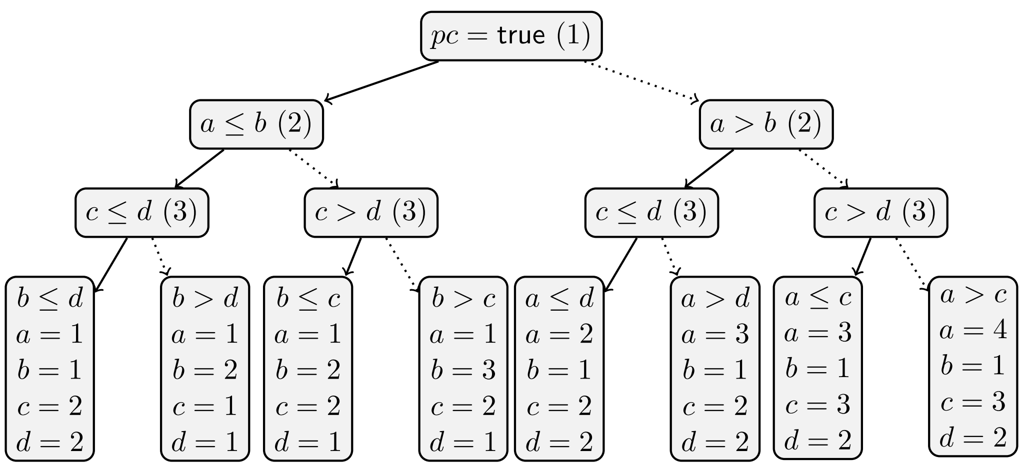
There were also conceptual breakthroughs in the mid-2000s with dynamic symbolic execution, which we'll see in Lecture 9.

Symbolic Execution Illustrated

Let's continue with an example.

```
int max4(int a, int b, int c, int d) {
  return max2(max2(a, b /* (1) */), max2(c, d /* (2) */) /* (3) */);
}

int max2(int x, int y) {
  if (x <= y) return y;
  else return x;
}
```

We can exhaustively explore all of the paths. Now, *pc* means path condition, and it's the thing inside the boxes. Solid lines are the true branch, while dashed lines are the false branch. This tree is different from the one in the slides, but I think it's right. Each node shows the result of the previous test. In this picture (and only in this picture—for space reasons), you get the path condition by conjoining the conditions that lead to your path, except for true; for instance, the full path condition at the left-most leaf is $a \le b \wedge c \le d \wedge b \le d$.



I manually worked out these test cases to meet the constraints, but of course we can also get z3 to compute it too. That's the whole reason we're talking about this.

```
(declare-fun a () Int)
(declare-fun b () Int)
(declare-fun c () Int)
(declare-fun d () Int)
(assert (< 0 a))
(assert (< 0 b))
(assert (< 0 c))
(assert (< 0 d))
(assert (<= a b))
(assert (> c d))
(assert (<= b c))
```

```
(check-sat)
(get-model)
```

If you run this, then you get:

```
sat
(
  (define-fun d () Int
    1)
  (define-fun a () Int
    1)
  (define-fun c () Int
    2)
  (define-fun b () Int
    1)
)
```

A Symbolic Execution Example

We are now going to illustrate how symbolic execution works on this code:

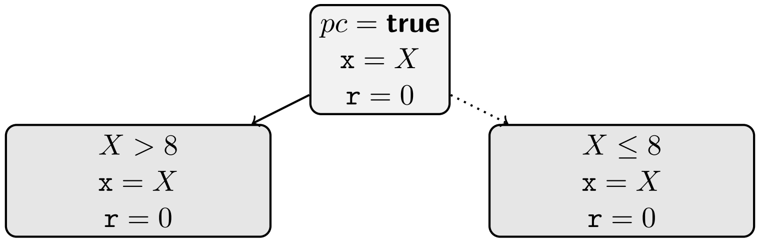
```
  int proc(int x) {
    int r = 0;

    if (x > 8) { // (1)
      r = x - 7
    }

    if (x < 5) { // (2)
      r = x - 2;
    }
  }
```

The initial symbolic state, after executing `r=0`, is this:

$$\boxed{\begin{array}{c} pc = \textbf{true} \\ \mathtt{x} = X \\ \mathtt{r} = 0 \end{array}}$$

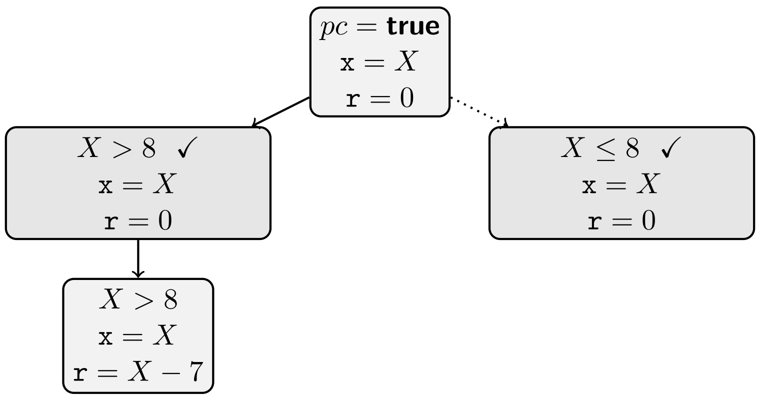
The start of the method is always reachable, so $pc = \textbf{true}$. There is one input X stored in program variable `x`. We also know that `r` is 0 after its initialization.

Program point (1) is a branch, so there are two possible symbolic states coming out of the branch statement. The path condition is what has to be true to reach a particular point. In this case, on the true branch, it must be the case that $X > 8$ (that's how you get there); and conversely for the false branch, $X \leq 8$. We encode that in the path condition. In these pictures we use the convention that the solid line denotes the then-branch, while the dotted line denotes the else-branch.

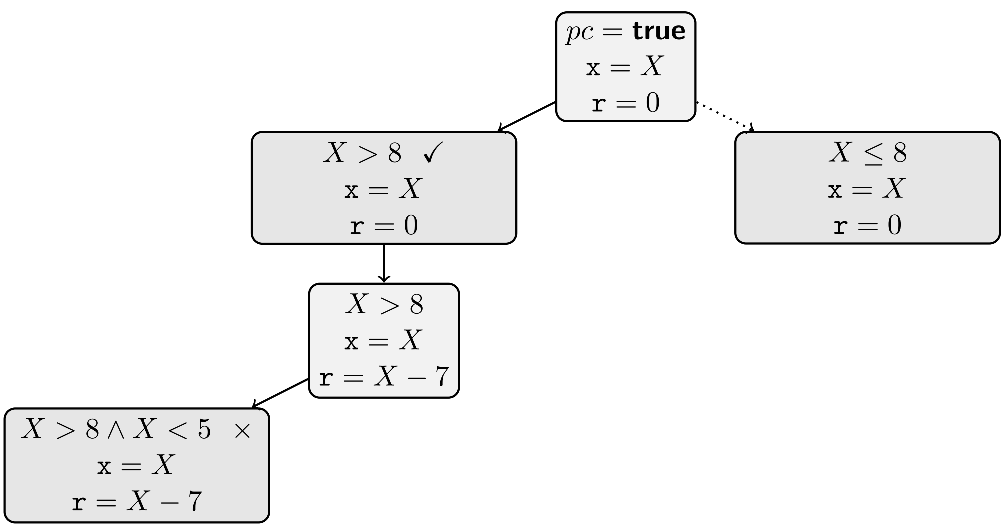


We ask the SAT solver if both path conditions $X > 8$ and $X \leq 8$ are satisfiable. They are (✓).

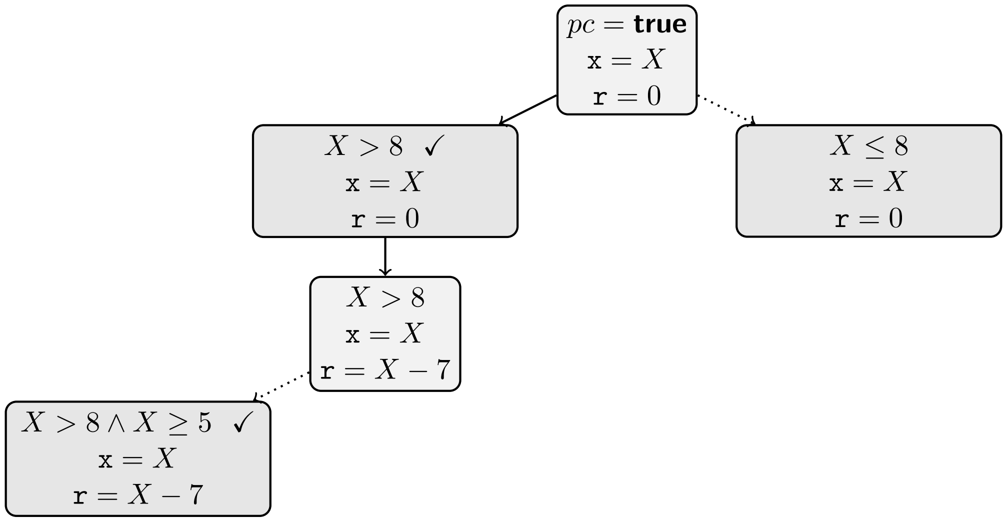
We continue executing the code in the then-branch, updating `r` with its new symbolic value, $X - 7$.



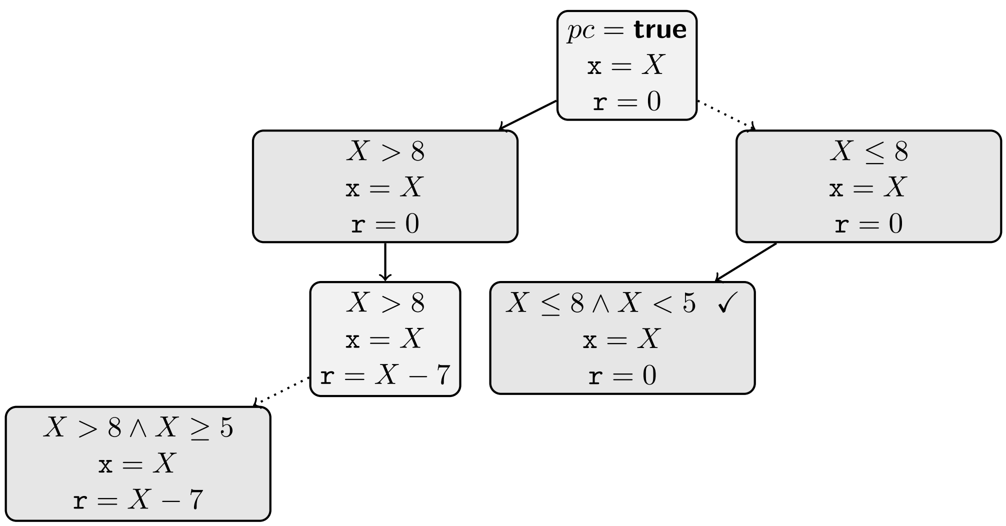
Execution continues to the second conditional (2). This induces another node in the tree. (Actually, 2 nodes, but let's start with the true branch only).



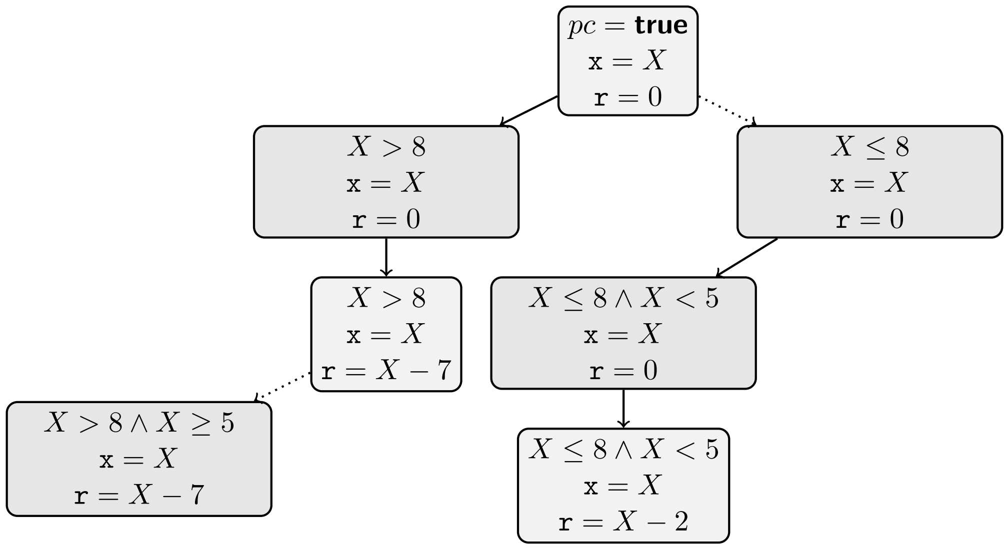
The SMT solver says that path condition is unsatisfiable ($\times$): there is no X such that $X > 8$ and $X < 5$. We throw away that state and instead explore the else branch. The else branch's path condition is indeed satisfiable (✓). The else branch is empty, so we proceed to the return and end that path.



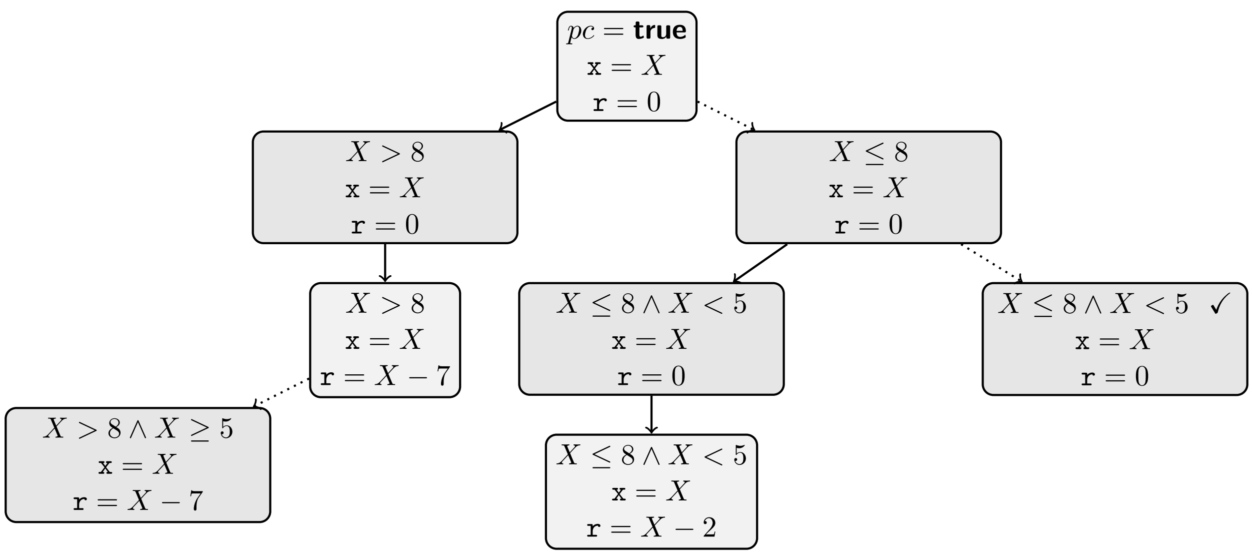
We continue with the else branch of (1), which proceeds directly to conditional (2).



The resulting path condition after (1) is false and (2) is true, $X \leq 8 \wedge X < 5$, is satisfiable (✓). We continue executing the code in the then-branch and assign to `r` symbolic value $X - 2$.



The final path to explore is the else-branch of conditional (2). That path condition, $X \le 8 \wedge X \ge 5$, is also satisfiable (✓).



Whenever we had asked the SMT solver whether a path condition was satisfiable, we also requested a satisfying assignment, which I didn't show you at the time. But some satisfying assignments are, from left to right, $X = 9$; $X = 4$; $X = 7$, and they induce the test cases `proc(9)`, `proc(4)`, and `proc(7)`. This test suite achieves full path coverage on this function: it explores all feasible paths from entry to return (not just branches!).

Defining symbolic execution

We've seen a couple of examples of symbolic execution. This technique analyses programs by tracking symbolic values (like X above) rather than actual concrete values; it is thus a form of static analysis. These symbolic values enable (symbolic) reasoning about *all* inputs that take the same path through the program, not just certain concrete values.

Symbolic values stand in for input variables. Programs can operate on a range of input values, and we don't want to commit to any specific values at analysis time, so we just leave it symbolic.

Path conditions. The (symbolic) path conditions characterize what must hold on a given path, and the symbolic state summarizes the effects of the execution on all possible program states.

A path condition for a path P is a formula pc such that pc is satisfiable if and only if P is executable.

In symbolic execution, we use a theorem prover, or constraint solver (e.g. z3), to check if a path condition is satisfiable and the path can be taken.

Symbolic state. A *symbolic state* is a pair $S = (Env, pc)$, where:

- Environment $Env : L \to E$ is a mapping from program variables to symbolic expressions (i.e. first-order logic terms);
- Path condition pc is a first-order logic formula.

We had concrete states in the operational semantics. Now we can define when a concrete state $M : L \to \mathbb{Z}$ satisfies ($\models$) a symbolic state $S = (Env, pc)$:

$$M \models (Env, pc) \text{ iff } \left((\bigwedge_{v \in L} M(v) = Env(v)) \land pc \text{ is SAT} \right)$$

We had the operational semantics on concrete states before. It is possible to extend them to symbolic states. Each program statement updates symbolic variables and the path condition, in a way that is consistent with the operational semantics.

Symbolic state satisfiability. Let's look at a symbolic state and two concrete states that may or may not satisfy it. Here's the symbolic state.

$$Env = \begin{cases} x \mapsto X \\ y \mapsto Y \end{cases} \qquad pc = X > 5 \land Y < 3$$

Now, what about concrete state

$$[x \mapsto 10, y \mapsto 1] \models ?S$$

Here, Env is saying that $x = X$ and $y = Y$, so we just have concrete path condition $10 > 5 \land 1 < 3$, which is true, so the concrete state above does satisfy this symbolic state.

For concrete state

$$[x \mapsto 1, y \mapsto 10] \models ?S$$

we have $1 > 5 \land 10 < 3$, which is false, so that concrete state does not satisfy the symbolic state.

A new example of symbolic state (slightly modified from the slides):

$$Env = \begin{cases} x \mapsto X + Y \\ y \mapsto Y - X \end{cases} \qquad pc = 2 * X - Y > 0$$

Now, what if concrete state was

$$[x \mapsto 10, y \mapsto 2] \models ?S$$

We can use linear algebra on the equations $10 = X + Y$ and $2 = Y - X$ to get $X = 4, Y = 6$. That gives concrete path condition $2 * 4 - 6 > 0$, which evaluates to true, making this concrete state satisfy the concrete state. (I think the example in the slides is broken, because we're supposed to be working over integers here.)

With concrete state

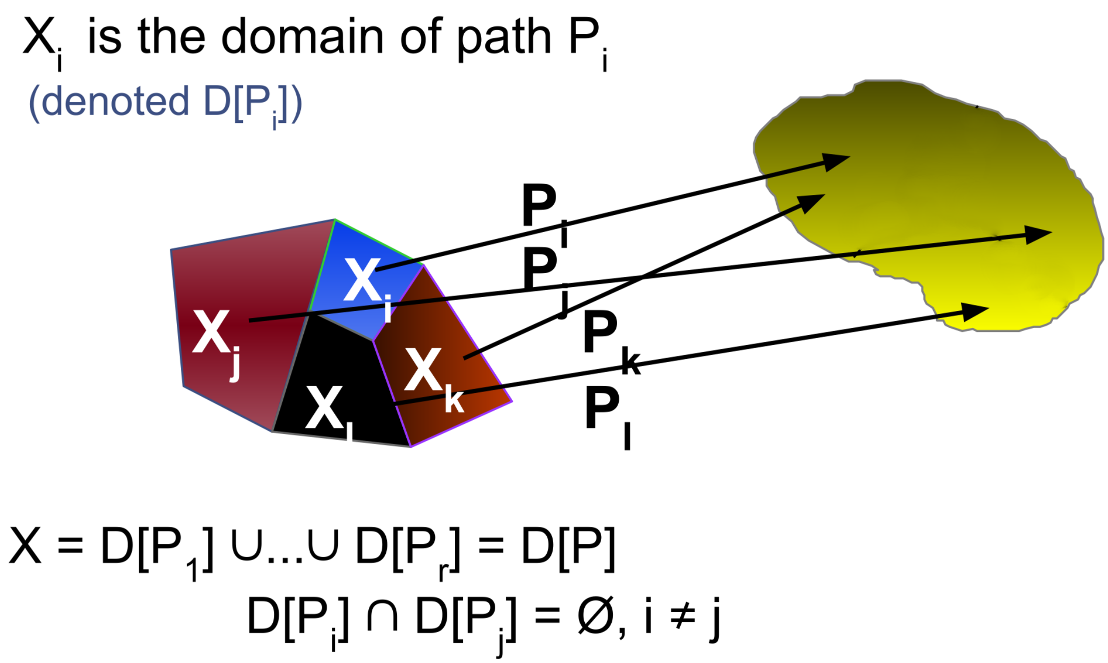
$$[x \mapsto 2, y \mapsto 10] \models ?S$$

linear algebra gives us $X = -8, Y = 6$, so we have path condition $2 * (-8) - 6 > 0$, which is false, and so this concrete state does not satisfy the given symbolic state.

Another view of symbolic execution. Symbolic execution is quite path-based. We can say that symbolic execution creates a functional representation of each path in the control-flow graph of a program, i.e. under symbolic execution, we view each program path as a function from inputs to outputs.

We can define the domain for path P_i, or $D[P_i]$, to be the set of inputs that force the program to take path P_i.

The program P as a whole can be thought of as a collection of partial functions $P_1, \ldots, P_r$, where each P_i maps some (disjoint) part of the program input to output. Here's a picture from the slides, though I disagree with partitioning the output space; outputs can be anything.



A Third Symbolic Execution Example

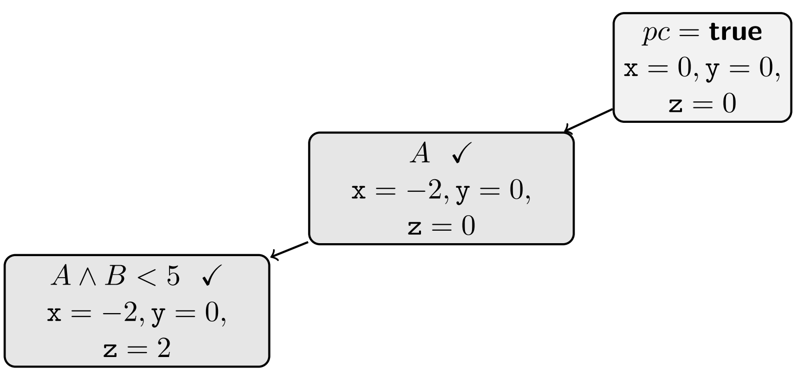
Once again, we can use symbolic execution to find an assertion violation.

```
proc(int a, int b, int c) {
  int x = 0, y = 0, z = 0;
  if (a) { // (1)
    x = -2;
  }
  if (b < 5) { // (2)
    if (!a && c) { // (3)
      y = 1;
    }
    z = 2;
  }
  assert (x + y + z != 3);
}
```

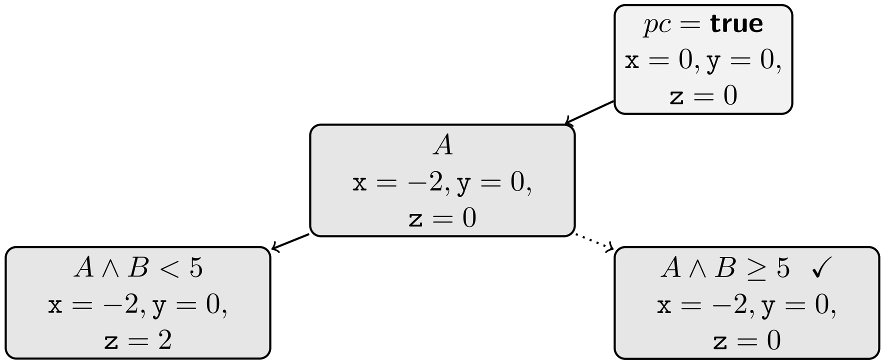
We are going to say that $a = A, b = B, c = C$ always, and leave them out of the symbolic state for space reasons. The initial state is:

$$\begin{array}{c} pc = \textbf{true} \\ \mathtt{x} = 0, \mathtt{y} = 0, \\ \mathtt{z} = 0 \end{array}$$

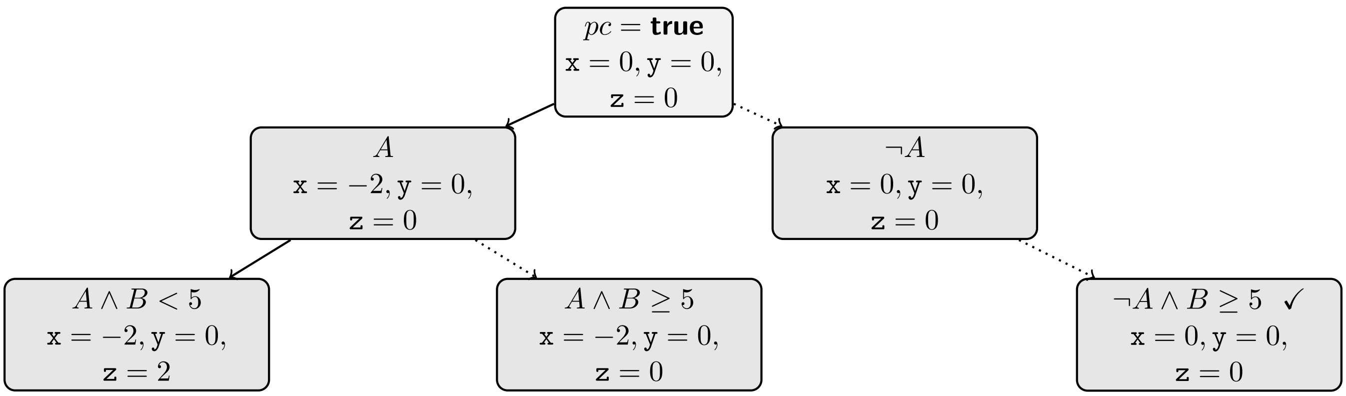
I'm going to combine visiting the true-branch of (1) as well as the true-branch of (2) in the following picture. Both path conditions A and $A \wedge B < 5$ are satisfiable (✓).



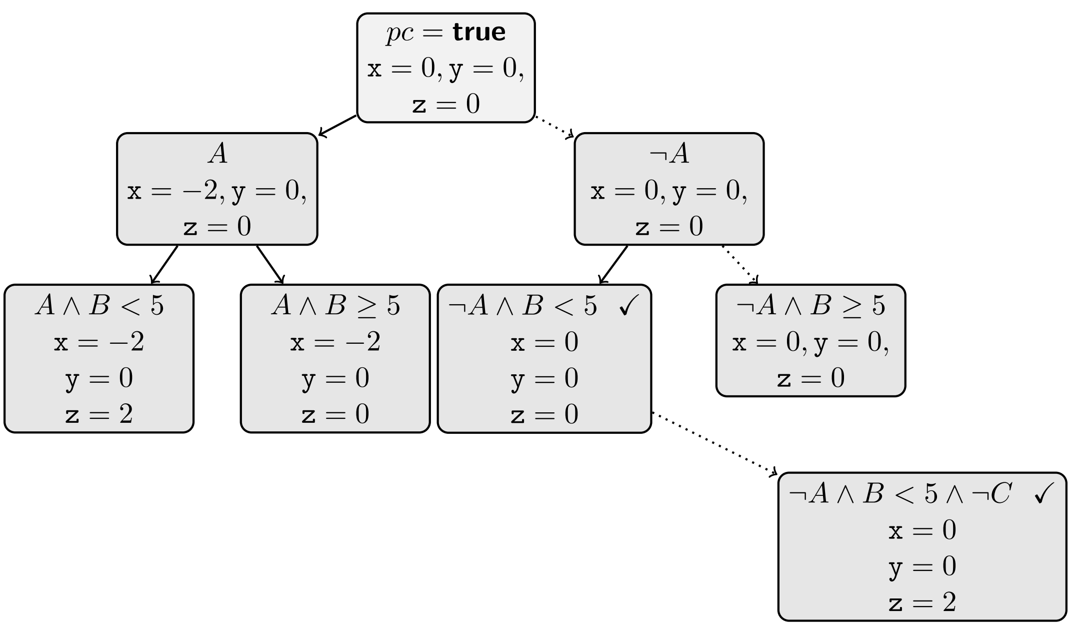
We can also add the else-branch of (2), which has a satisfiable path condition (✓) and has no body.



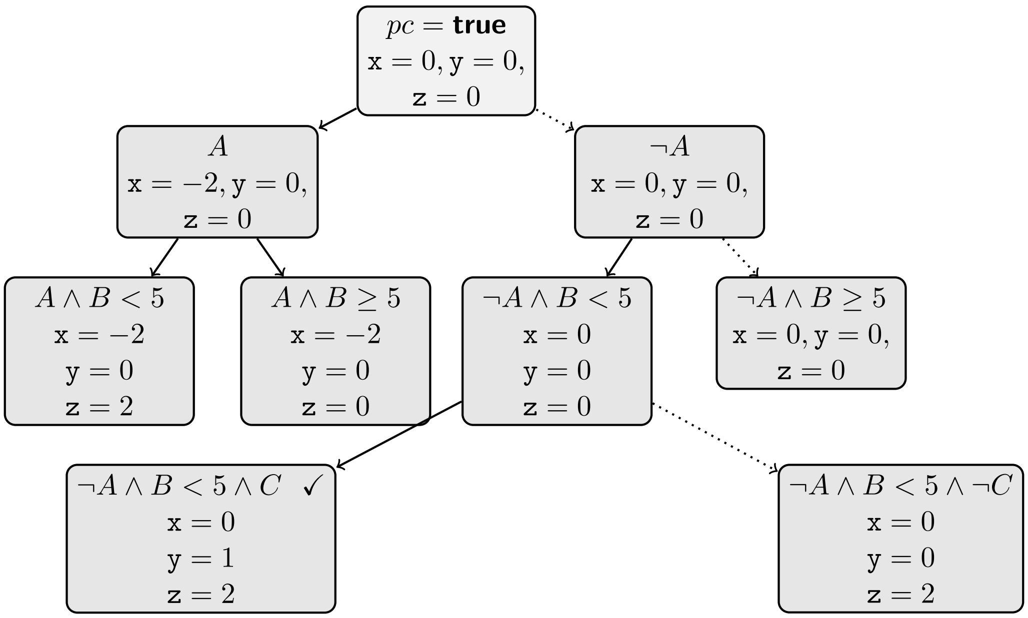
Now we visit the else-branch of (1) and also the else-branch of (2), which yields satisfiable (✓) path condition $\neg A \wedge B \geq 5$.



And there are still unexplored paths through the true-branch of (2) leading to (3), for which we'll explore the else-branch first. This yields satisfiable (✓) path condition $\neg A \wedge B < 5 \wedge \neg C$. (I simplified $\neg(\neg a \wedge c)$ to just $a \vee \neg c$ to get the path condition here).



Finally, we explore the then-branch of (3), yielding an satisfiable path condition $\neg A \wedge B < 5 \wedge \neg A \wedge C$ (which I've simplified).



This path fails the assert. We can ask the SMT solver to find a test case that violates the assertion, which is A false, $B = 4$, and C true, yielding $x = 0, y = 1, z = 2$, so that the assert is checking $0 + 1 + 2 \neq 3$, which fails as desired.

Finding Bugs using Symbolic Execution

We've seen that symbolic execution enumerates paths. It will therefore find bugs that trigger when a specific path executes. That's not quite enough on its own, though. We saw how we rewrote programs to use specific asserts, as in the division by zero example. In general, finding a bug requires finding the conditions that trigger it. Bugs include assertion failures, buffer overflows, division by zero, etc.

We go one more step beyond just having asserts, and compile these assertions into conditionals. This treats assertions as conditions and creates explicit error paths. Since we are exploring all paths, we will explore the error path (containing an `error()` call) if it is reachable.

So, we compile from

```
assert x != NULL
```

into

```
if (x == NULL)
     error();
```

and show that the `error()` call is reachable or unreachable.

The rewriting process, or instrumenting the programs with properties, can translate any safety property ("bad things don't happen") into reachability (of an `error()` call). There is some similarity to fuzzing.

Rewriting can be explicit or implicit. For explicit rewriting, this is like sanitizers, and we instrument the code with checks. But the symbolic engine can also implicitly inject extra checks at runtime.

Checks might look like this:

$$\begin{aligned} \texttt{y = 100 / x} &\Rightarrow \texttt{assert x != 0; y = 100/x} \text{ (division by zero)} \\ \texttt{a[x] = 10} &\Rightarrow \texttt{assert x >= 0 \&\& x < len(a)} \text{ (array bounds)} \end{aligned}$$

Problems of (Classical) Symbolic Execution

Of course, I've shown you cases where symbolic execution works. This isn't real code. What happens for real?

Some code is hard to analyze. Even if it generates innocuous-looking constraints, the resulting constraints might be beyond the abilities of our SMT solvers. And cryptographic hashes are definitely hard to invert.

There's also the path explosion problem. The number of paths in the program is exponential in the size of the program. The paths are caused by control flow, loops, procedures, concurrency, etc.

And, to analyze real code, you need to work with more than just integers. There are pointers and data structures; files and databases; networks and sockets; and threads and thread schedules, among other things. There has to be some way of handling these. Which is why we'll talk about dynamic symbolic execution soon.

References

[BEL75] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT—a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, page 234–245, New York, NY, USA, 1975. Association for Computing Machinery.

[Kin75] James C. King. A new approach to program testing. In *Proceedings of the International Conference on Reliable Software*, page 228–233, New York, NY, USA, 1975. Association for Computing Machinery.