Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 14 — February 27, 2026

Patrick Lam *version 1*

Here's a function.

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

Previously, we've talked about various methods for testing functions: writing a test suite manually; fuzzing; and property-based testing.

We're going to add to our arsenal with *symbolic execution*, which will automatically generate[1] a test suite that:

- achieves full branch coverage (actually even better: full path coverage);
- identifies dead code; and,
- discovers whether division by 0 is possible.

More generally, symbolic execution is a deterministic technique which automatically analyzes some code and generates tests to determine reachability of each line of that code.

We need symbolic execution to understand how the bounded model checker Kani and the auto-active program verifier Dafny work. You can use both of these tools without understanding symbolic execution, at least until things to wrong. But we are studying how these tools work, so we'll need at least some notions of symbolic execution.

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.

Under symbolic execution, all testing is now automatic. And, for some definitions of exhaustive (path coverage), the testing is also exhaustive.

Here's the transformed program.

[1]if conditions are right...

```
def Foo(x,y):
""" requires: x and y are int
    ensures: returns floor(max(x,y)/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 we have when we run the program); creating 2 new paths for each branch; and recording branch 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. For each path, we generate a set of constraints and ask the z3 SMT solver to solve them for us. Here's path #2.

```
(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 Robert S. Boyer, Bernard Elspas, and Karl N. Levitt; and James C. King: [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.

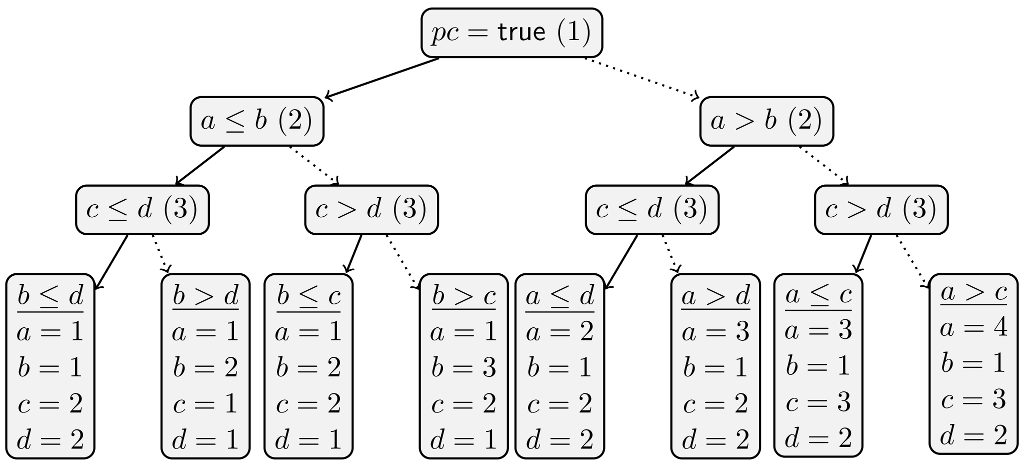
In any case, in the 2000s, SAT solvers and their big brothers (SMT solvers) made SAT feasible in practice. In the context of formal verification, constraint solving is easy. Classical verification algorithms—with worst-case exponential running time—became viable in practice even if not in theory.

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. Hereafter, *pc* means path condition (rather than program counter as before), and it's the thing inside the boxes. Solid lines are the true branch, while dashed lines are the false branch. 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; for instance, the full path condition at the left-most leaf is $a \leq b \wedge c \leq d \wedge b \leq d$.


$pc = \text{true}$ (1)
$a \leq b$ (2)
$a > b$ (2)
$c \leq d$ (3)
$c > d$ (3)
$c \leq d$ (3)
$c > d$ (3)
$b \leq d$, $a = 1$, $b = 1$, $c = 2$, $d = 2$
$b > d$, $a = 1$, $b = 2$, $c = 1$, $d = 1$
$b \leq c$, $a = 1$, $b = 2$, $c = 2$, $d = 1$
$b > c$, $a = 1$, $b = 3$, $c = 2$, $d = 1$
$a \leq d$, $a = 2$, $b = 1$, $c = 2$, $d = 2$
$a > d$, $a = 3$, $b = 1$, $c = 2$, $d = 2$
$a \leq c$, $a = 3$, $b = 1$, $c = 3$, $d = 2$
$a > c$, $a = 4$, $b = 1$, $c = 3$, $d = 2$

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:

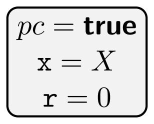
```
sat
(
  (define-fun d () Int
    1)
  (define-fun a () Int
    1)
  (define-fun c () Int
    2)
  (define-fun b () Int
    1)
)
```

A Symbolic Execution Example

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

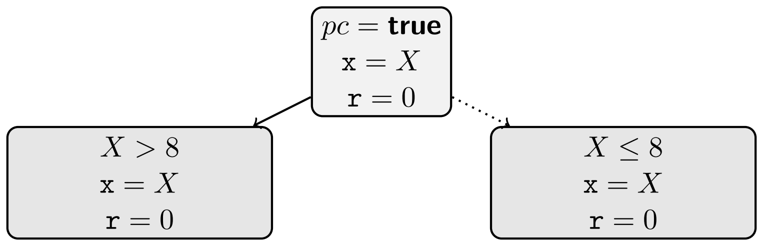
```
  int proc(int x) {
    int r = 0;

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

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

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



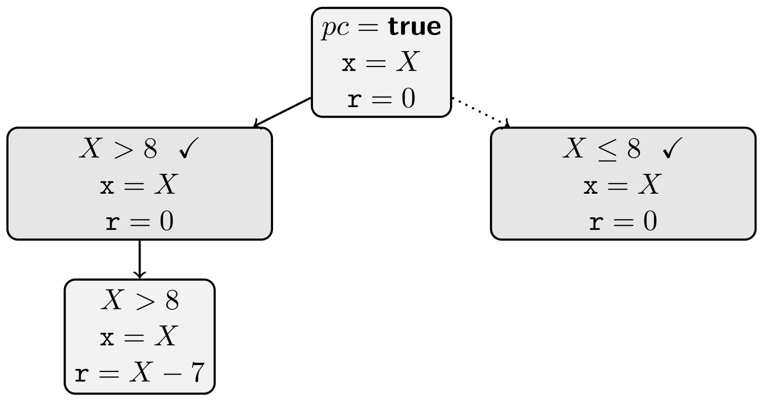
The start of the method is always reachable, so $pc = \textbf{true}$. There is one input X stored in program variable `x`. We also know that `r` is 0 after its initialization.

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. We continue the convention that the solid line denotes the then-branch, while the dotted line denotes the else-branch.

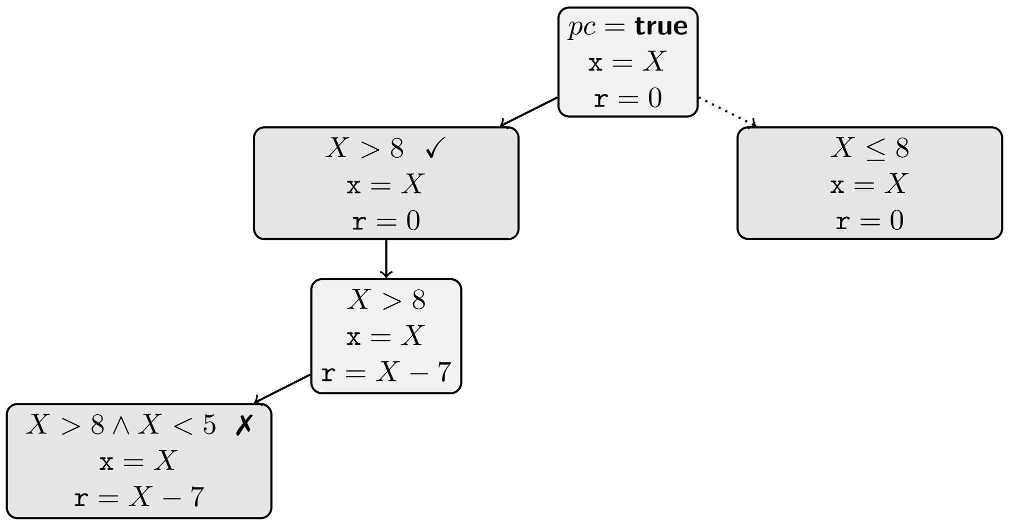


We ask the SMT solver if both path conditions $X > 8$ and $X \leq 8$ are satisfiable. They are (✓).

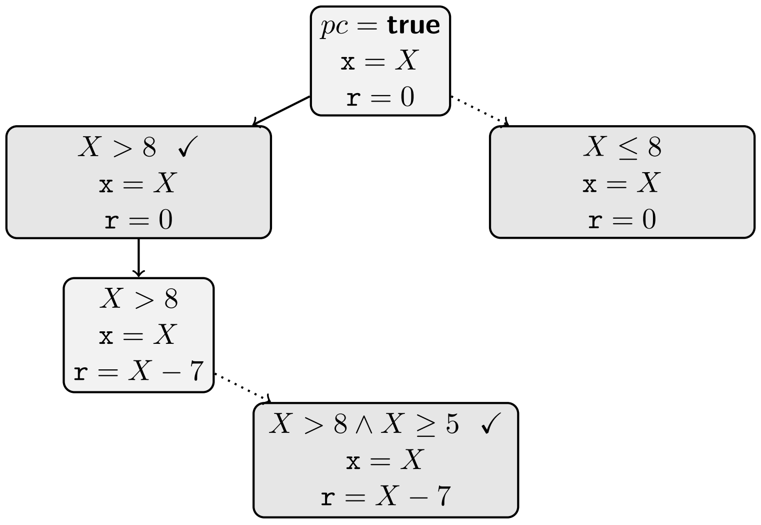
We continue executing the code in the then-branch, updating `r` with its new symbolic value, $X - 7$.



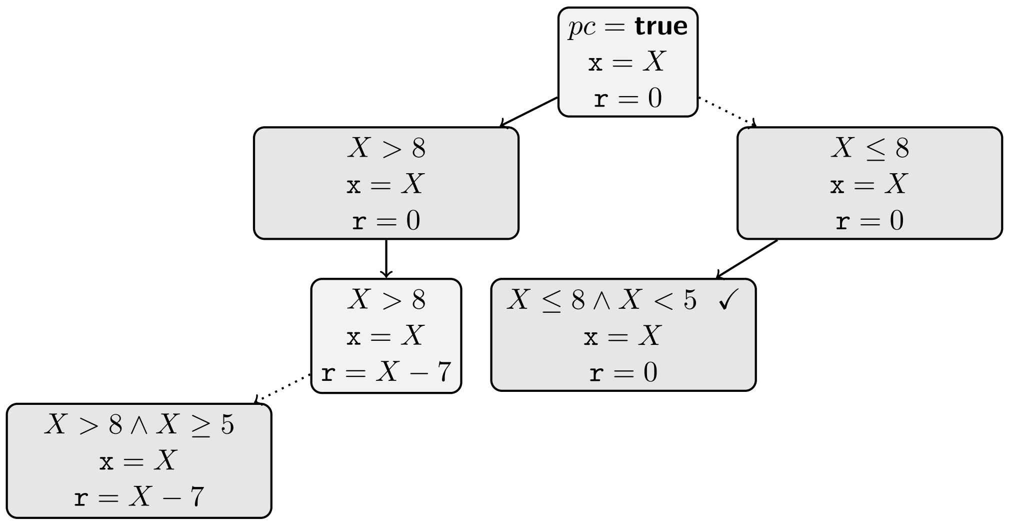
Execution continues to the second conditional (2). This induces another node in the tree. (Actually, 2 nodes, but let's start with the true branch only).



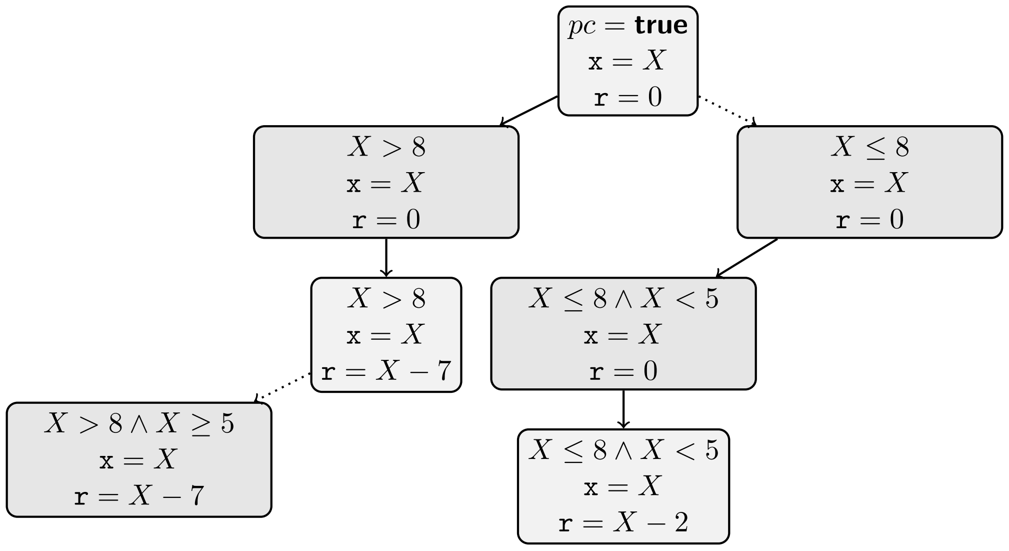
The SMT solver says that the true branch's path condition is unsatisfiable (✗): there is no X such that $X > 8$ and $X < 5$. We throw away that state and instead explore the else branch. The else branch's path condition is indeed satisfiable (✓). The else branch is empty, so we proceed to the return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



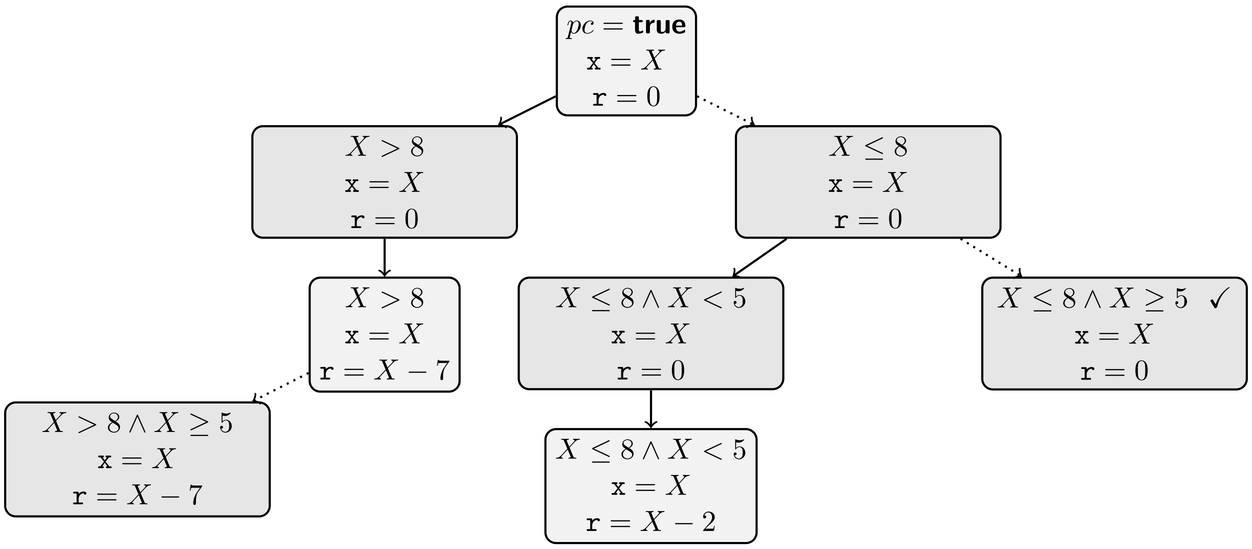
We continue with the else branch of (1), which proceeds directly to conditional (2).



The resulting path condition after (1) is false and (2) is true, $X \leq 8 \land X < 5$, 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, $X \leq 8 \land X \geq 5$, 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.

The notion of symbolic values is going to be key to understanding how both Kani and Dafny work.

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 a constraint solver (like z3), to check if a path condition is satisfiable and the path can be taken. A satisfying assignment can be used as an input for the program to execute the path of interest.

A Third Symbolic Execution Example

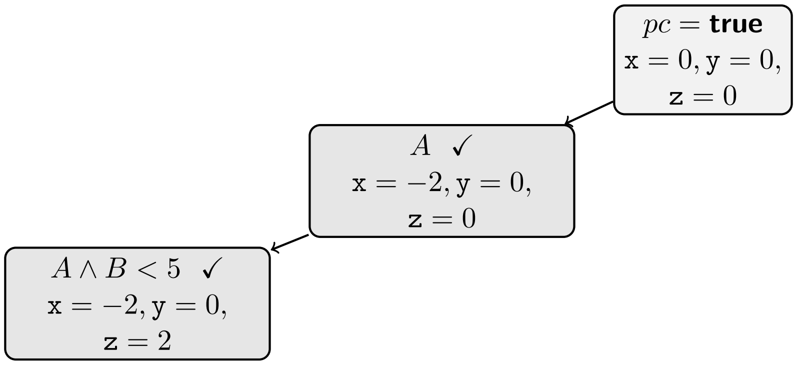
Once again, we can use symbolic execution to find an assertion violation.

```
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);
}
```

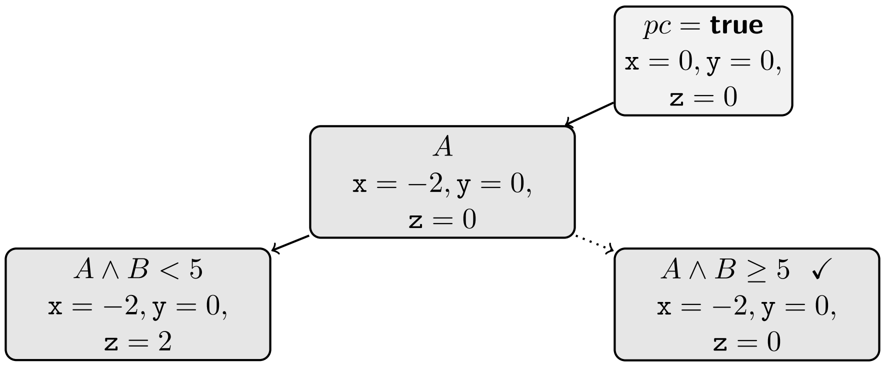
We are going to say that $a = A, b = B, c = C$ always, and leave them out of the symbolic state for space reasons. The initial state is:

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

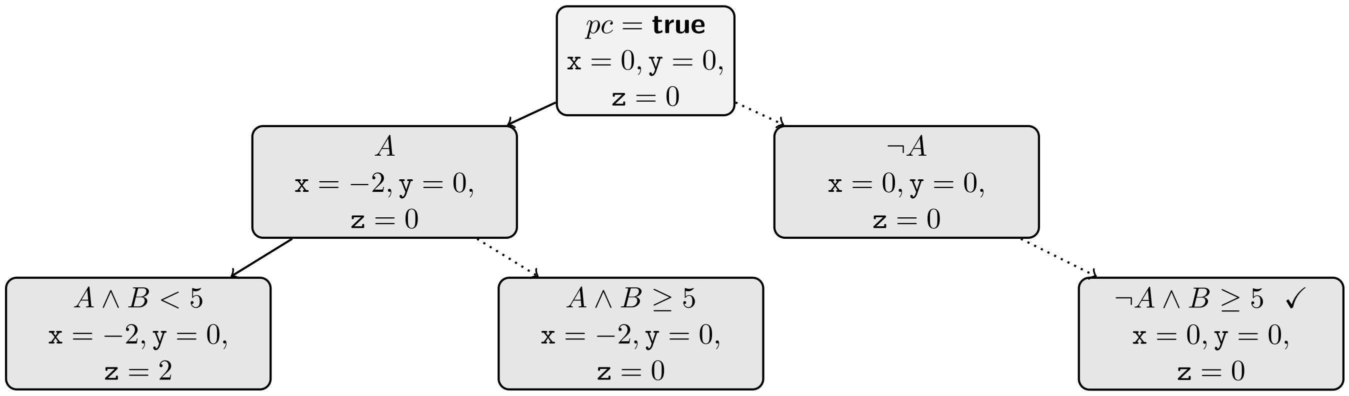
I'm going to combine visiting the true-branch of (1) as well as the true-branch of (2) in the following picture. Both path conditions A and $A \wedge B < 5$ are satisfiable (✓). We can't visit (3)'s true-branch because the path condition inside that branch, $A \wedge B < 5 \wedge (\neg A \wedge C)$, is unsatisfiable (✗).


$pc = \textbf{true}$
$\mathtt{x} = 0, \mathtt{y} = 0, \mathtt{z} = 0$
A ✓
$\mathtt{x} = -2, \mathtt{y} = 0, \mathtt{z} = 0$
$A \wedge B < 5$ ✓
$\mathtt{x} = -2, \mathtt{y} = 0, \mathtt{z} = 2$

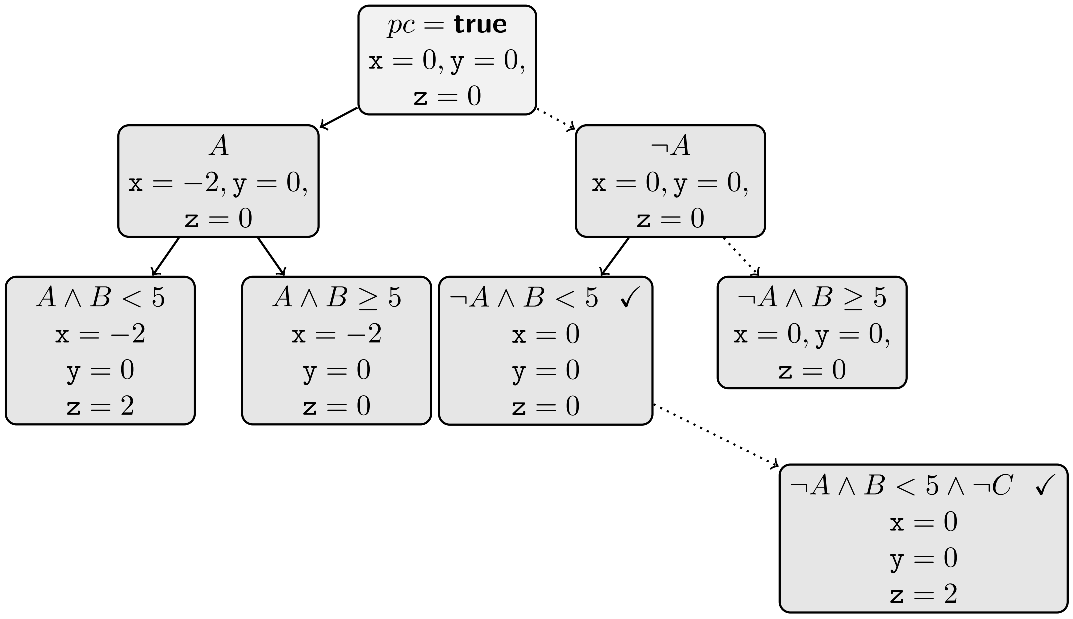
We can also add the else-branch of (2), which has a satisfiable path condition (✓) and has no body.


$pc = \textbf{true}$ x = 0, y = 0, z = 0
A x = −2, y = 0, z = 0
$A \wedge B < 5$ x = −2, y = 0, z = 2
$A \wedge B \geq 5$ ✓ x = −2, y = 0, z = 0

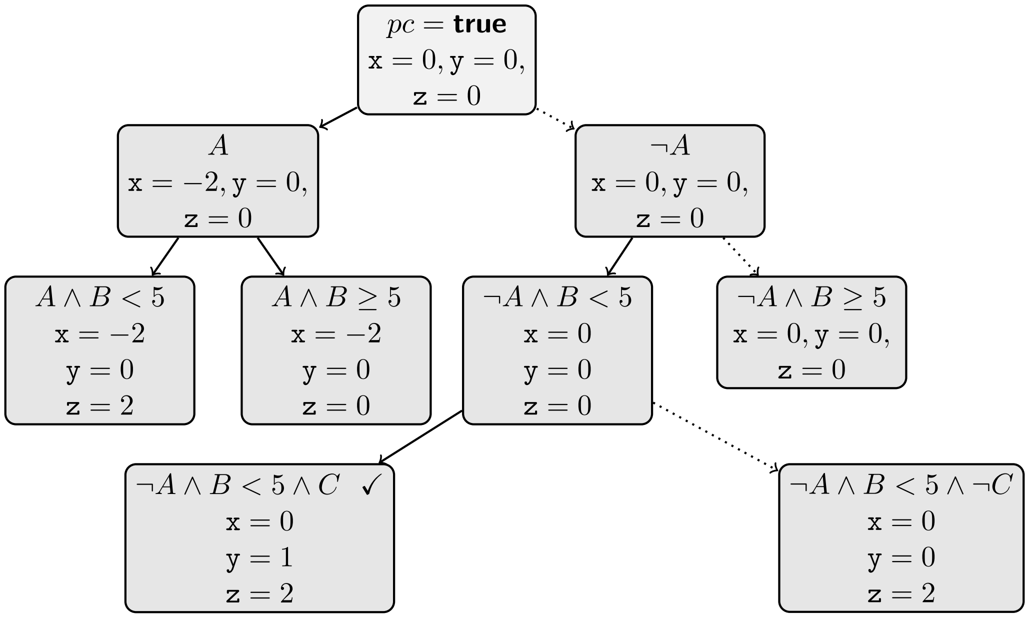
Now we visit the else-branch of (1) and also the else-branch of (2), which yields satisfiable (✓) path condition $\neg A \wedge B \geq 5$.


$pc = \textbf{true}$ x = 0, y = 0, z = 0
A x = −2, y = 0, z = 0
$\neg A$ x = 0, y = 0, z = 0
$A \wedge B < 5$ x = −2, y = 0, z = 2
$A \wedge B \geq 5$ x = −2, y = 0, z = 0
$\neg A \wedge B \geq 5$ ✓ x = 0, y = 0, z = 0

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).


$pc = \textbf{true}$ x = 0, y = 0, z = 0
A x = −2, y = 0, z = 0
$\neg A$ x = 0, y = 0, z = 0
$A \wedge B < 5$ x = −2 y = 0 z = 2
$A \wedge B \geq 5$ x = −2 y = 0 z = 0
$\neg A \wedge B < 5$ ✓ x = 0 y = 0 z = 0
$\neg A \wedge B \geq 5$ x = 0, y = 0, z = 0
$\neg A \wedge B < 5 \wedge \neg C$ ✓ x = 0 y = 0 z = 2

Finally, we explore the then-branch of (3), yielding a 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. Like fuzzing: 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 using fuzzing plus explicit assertions to find problems.

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:

```
y = 100 / x  ⇒  assert x != 0; y = 100/x (division by zero)
  a[x] = 10  ⇒  assert x >= 0 && x < len(a) (array bounds)
```

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. For instance, cryptographic hashes are definitely hard to invert.

There's also the path explosion problem. The number of paths in the program is at least exponential in the size of the program. Control flow, loops, procedures, concurrency, etc., can cause lots of paths—a potentially infinite number, in the case of loops.

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.

But, for the purpose of this course, you now know enough about symbolic execution to make sense of bounded model checking.

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.