Software Testing, Quality Assurance & Maintenance—Lecture 14

Patrick Lam
University of Waterloo

February 27, 2026

Part I

Motivating Symbolic Execution

Consider this 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
```

How to test? So far:

- manually-written test suite;
- fuzzing;
- property-based testing.

Symbolic execution: magic?

We introduce symbolic execution.

- Achieves full branch (actually, path) coverage;
- Identifies dead code;
- Discovers whether division by 0 is possible.

(How well does fuzzing work on the example?)

About symbolic execution

Symbolic execution is a deterministic technique which

- automatically analyzes some code, and
- generates tests to determine reachability of each line of that code.

Why?

Must understand symbolic execution to understand bounded model checker Kani and auto-active program verifier Dafny.

Can use these tools without understanding.

We are here to understand.

Part II

How Symbolic Execution Works: a Worked Example

Four Steps to Symbolic Execution

if we are looking for division by 0 errors:

- *transform* program to add oracles—
  tests for division by 0;
- *traverse* & compute each program path;
  path1: `x > y, y == 0`;
  path2: `x > y, y != 0, return x / y`; etc.
- *solve* constraints for each path;
  path1: `x=10,y=0`;
  path2: `x=10,y=1`; etc.
- *run* the program on generated tests.

Implications

All testing is now automatic.

This testing is also exhaustive,
with respect to path coverage.

Transformed function

With the asserts:

```
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
```

The Next Two Steps

Traversing:
for each program path, execute program on symbolic input values;
record branch conditions.

Solving constraints:
decide path feasibility;
generate test cases to reach paths and to find bugs.

Transformed function

With the asserts:

```
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
```

Traversing Paths

Enumerating all 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

The z3 SMT solver can solve this example, corresponding to path #2:

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

Solution

```
sat
(
  (define-fun y () Int
    1)
  (define-fun x () Int
    2)
)
```

History of Symbolic Execution

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.

(from a 1975 paper)

What about today?

1. Even if the software never crashes, still need it to do the right thing (validation).

2. Verification, though, is more feasible in practice with industrial-strength SAT/SMT solvers: constraint solving is easy.

Part III

Symbolic Execution: Path Conditions

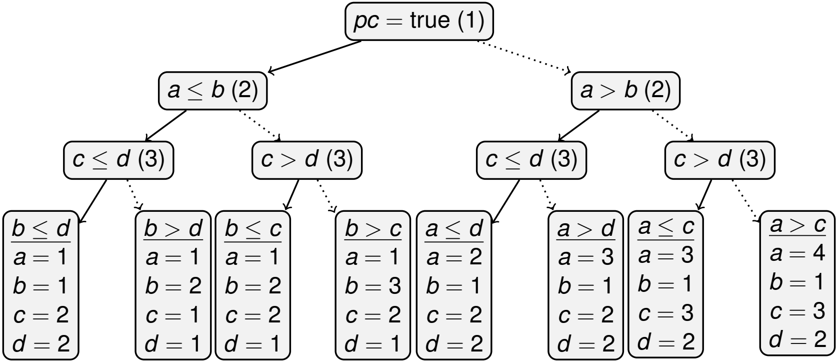
Our next example

```
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 will explore all the paths.

All the paths



pc = path condition; solid = true branch; dashed = false branch.

Here (and only here), get *pc* by conjoining conditions on your path; e.g. leftmost leaf has *pc*: $a \leq b \wedge c \leq d \wedge b \leq d$.

A test case

We ask z3 to compute values for a, b, c, d, based on pc
$a \leq b \wedge c \leq d \wedge b \leq d$.

$$\boxed{\begin{array}{c} \underline{b \leq d} \\ a = 1 \\ b = 1 \\ c = 2 \\ d = 2 \end{array}}$$

Running z3

$$a \leq b \wedge c > d \wedge b \leq c$$

Input:

```
(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)
```

Output:

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

Part IV

Symbolic Execution: Example 1

Another proc

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

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

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

Initial symbolic state

After executing `r=0`:

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

1. method start is always reachable, so $pc = \textbf{true}$
2. the sole input, symbolic X, is stored in $\mathtt{x}$
3. $\mathtt{r}$ is 0

same proc again

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

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

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

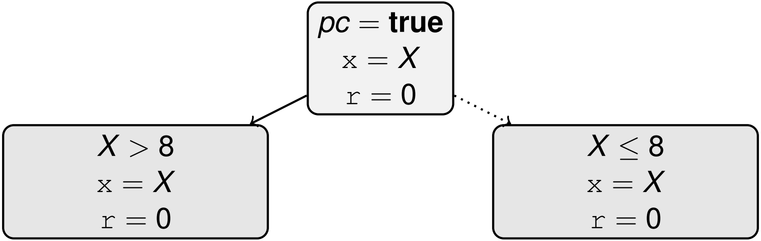
Symbolically executing the if

point (1) is `if (x > 8)`: 2 possible symbolic states after.

pc is what has to be true to reach a point.

on true branch, must have $X > 8$;
on false branch, $X \leq 8$.

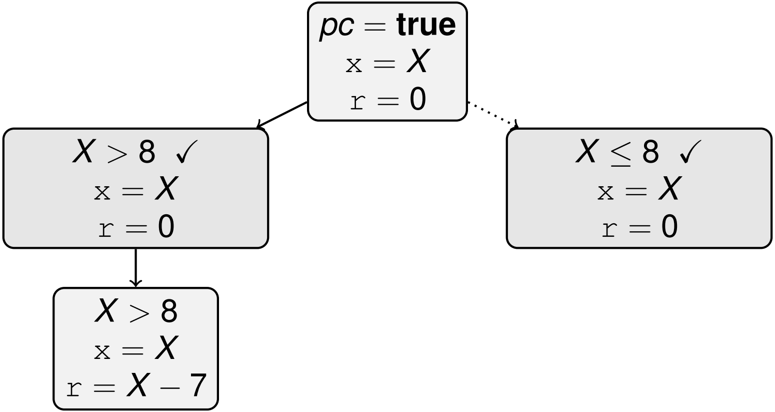
Encode this in *pc*.



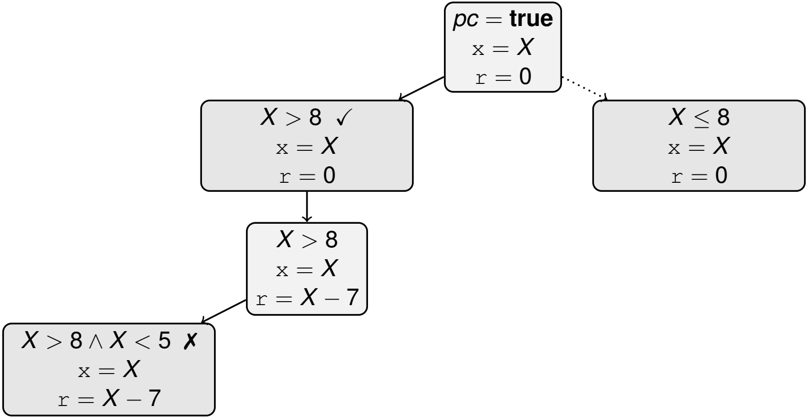
Ask SMT solver if path conditions $X > 8$ and $X \leq 8$ are satisfiable: yes ($\checkmark$).

then-branch code: `r = x - 7`

Update `r` with its new symbolic value, $X - 7$.

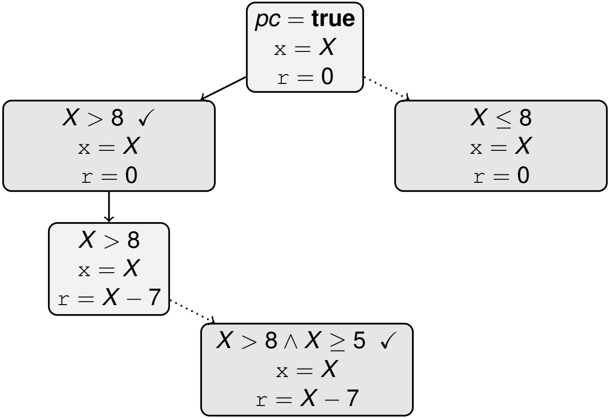


`if x < 5` (2) then-branch



Now unsatisfiable (can't have $X > 8 \wedge X < 5$); throw it away.

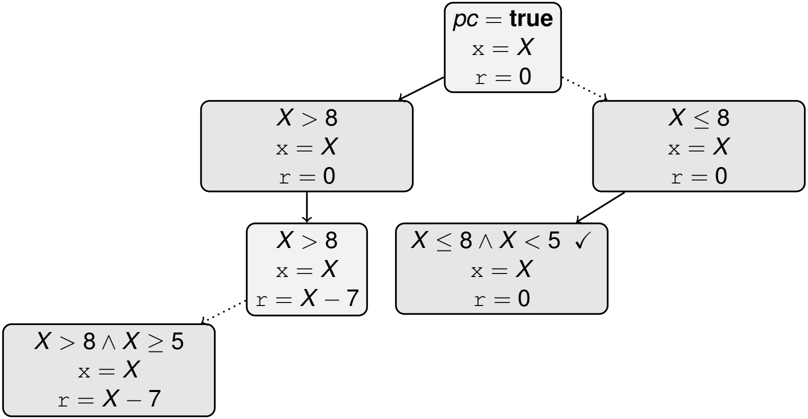
second conditional (2) `if x < 5` and else-branch



else branch path condition is satisfiable (✓);
proceed to the return and end that path.

back up to (1) else-branch and (2) then-branch

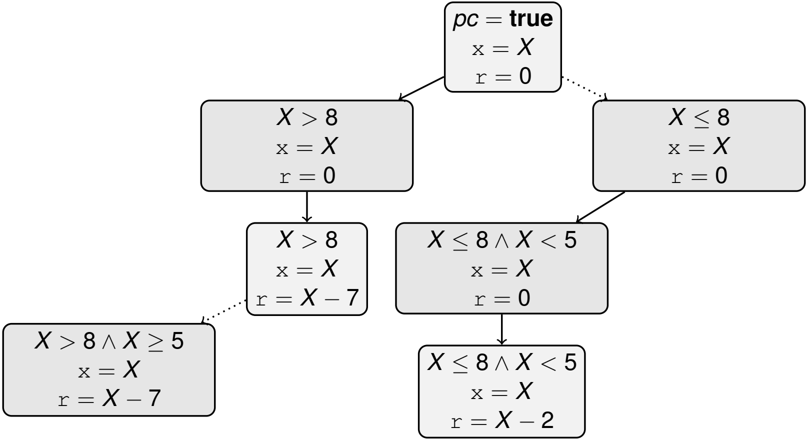
(1) else-branch proceeds directly to conditional (2):



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

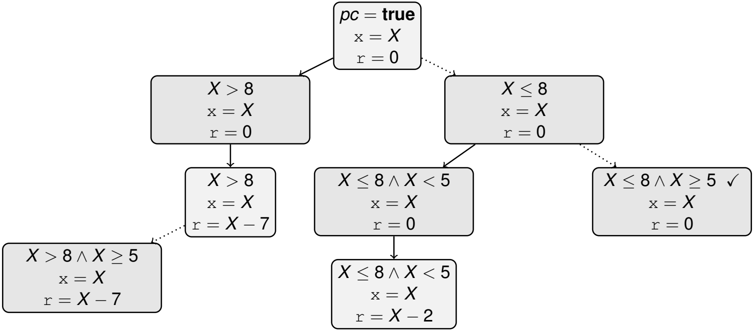
finishing (2) then-branch

We continue executing the code in the then-branch and assign to `r` the symbolic value $X - 2$.



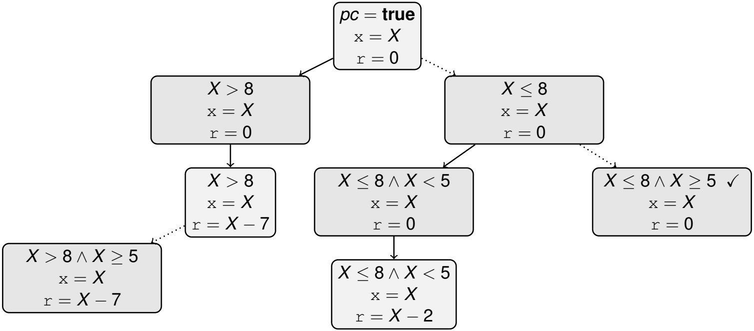
finally (2) else-branch

That path condition, $X \leq 8 \wedge X \geq 5$, is also satisfiable (✓).



Satisfying assignments ✓ ✗

We asked SMT solver about ✓ versus ✗.
At the same time, we also requested satisfying assignments.



Some satisfying assignments, from left to right:

$$X = 9;\ X = 4;\ X = 7;$$

test cases:

`proc(9),proc(4),proc(7)`

Explores all feasible paths.

Defining symbolic execution

We've seen some examples.
Summing up:

- track symbolic values (e.g. X) rather than actual concrete values;
- enable symbolic reasoning about all inputs taking a given path.

Symbolic value: stands in for input variable.
Don't need to commit to any specific values.

The concept of symbolic value is key to Kani and Dafny.

Path conditions

(Symbolic) path condition: characterize what must hold on a given path.

Symbolic state: summarizes the effects of the execution on all possible program states.

A path condition for a path P is a formula pc s.t. pc is satisfiable iff P is executable.

In symbolic execution: 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.

Part V

Symbolic Execution: Example 2

One more symbolic execution example

Symbolic execution can find 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 will say $a = A, b = B, c = C$ always, leaving them out of symbolic state.

Initial state

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

Skipping ahead

true-branch (1) plus true-branch (2):

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

$$\begin{array}{c} A \;\checkmark \\ \mathtt{x} = -2, \mathtt{y} = 0, \\ \mathtt{z} = 0 \end{array}$$

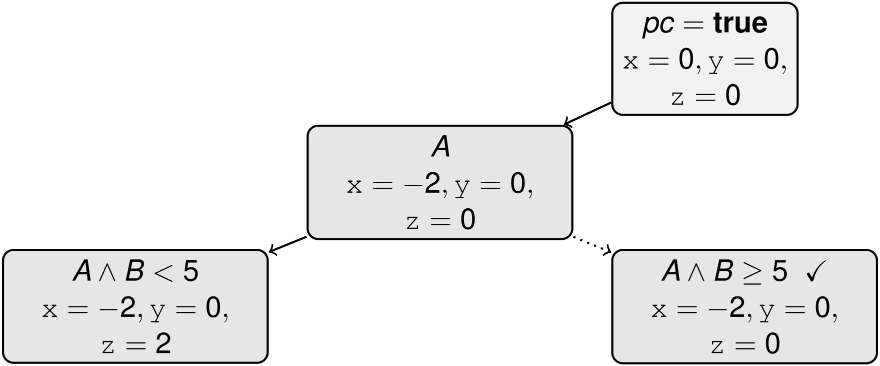
$$\begin{array}{c} A \wedge B < 5 \;\checkmark \\ \mathtt{x} = -2, \mathtt{y} = 0, \\ \mathtt{z} = 2 \end{array}$$

A and $A \wedge B < 5$ both satisfiable ($\checkmark$).

can't visit (3)'s true-branch because pc inside that branch, $A \wedge B < 5 \wedge (\neg A \wedge C)$, is unsatisfiable (✗).

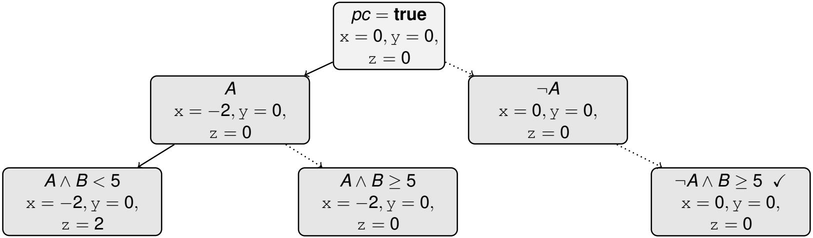
adding (2) else-branch

satisfiable (✓) path condition and no body:



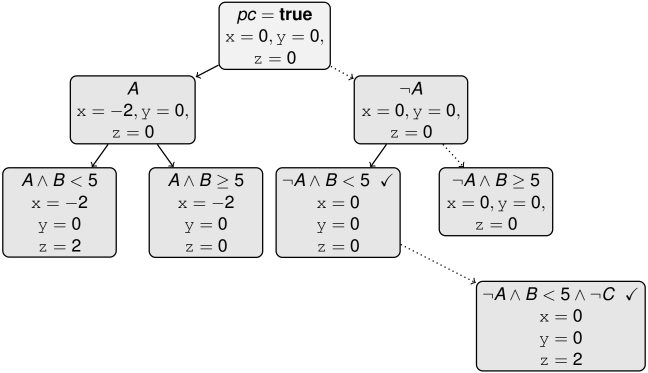
adding (1) else-branch and (2) else-branch

yields satisfiable (✓) path condition $\neg A \wedge B \geq 5$.



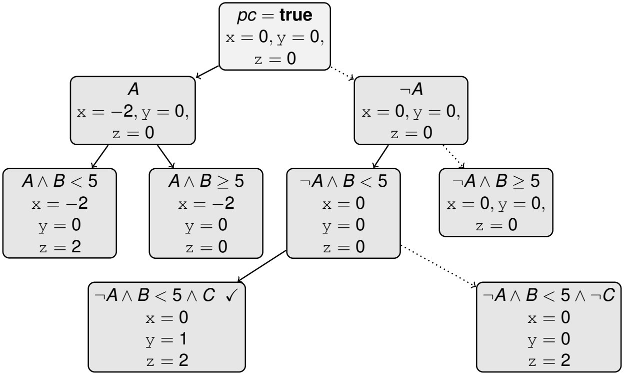
(2) true-branch leading to (3)'s else-branch

yields satisfiable (✓) path condition $\neg A \wedge B < 5 \wedge \neg A \wedge C$



finally: (3) then-branch

path condition $\neg A \wedge B < 5 \wedge \neg A \wedge C$



This path fails the assert `(x + y + z != 3)`.
SMT solver tells us A false, $B = 4$, and C true.
have $x = 0, y = 1, z = 2$, i.e. $0 + 1 + 2 \neq 3$, fails as desired.

Part VI

Symbolic Execution Commentary

Finding Bugs using Symbolic Execution

Symbolic execution enumerates paths;
thus, finds bugs triggered on a specific path.

Like fuzzing: use specific asserts.

To find a bug: find conditions that trigger it.

Bugs: assertion failures, buffer overflows, division by zero, etc.

Asserts versus conditionals

Explicit error paths: compile from

```
assert x != NULL
```

into

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

Since we explore all paths, we will explore the error path (containing an `error()` call) if it is reachable.

Implications of rewriting

Rewriting/instrumenting programs with properties:

translates any safety property (“bad things don’t happen”) into reachability (of an `error()` call).

Rewriting: explicit or implicit

Explicit: like sanitizers, instrument the code with checks.

Symbolic engine can also implicitly inject extra checks at runtime.

Checks might look like this:

`y = 100 / x` $\Rightarrow$ `assert x != 0; y = 100/x` (division by zero)

`a[x] = 10` $\Rightarrow$ `assert x >= 0 && x < len(a)` (array bounds)

Problems of (Classical) Symbolic Execution

We've seen selected examples.
Real-world?

Some code is hard to analyze.
Resulting constraints might be beyond the abilities of our SMT solvers.
e.g. cryptographic hashes are definitely hard to invert.

Problems of (Classical) Symbolic Execution II

Also: the path explosion problem.
# 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—potentially infinite.

Problems of (Classical) Symbolic Execution III

To analyze real code, you need to work with more than just integers.

- pointers and data structures;
- files and databases;
- networks and sockets;
- threads and thread schedules; etc.

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.