Program Analysis

Symbolic and Concolic Execution (Part 3)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Winter 2020/2021

Overview

1. **Classical Symbolic Execution**
2. **Challenges of Symbolic Execution**
3. **Concolic Testing** ←
4. **Large-Scale Application in Practice**

Mostly based on these papers:

- *DART: directed automated random testing*, Godefroid et al., PLDI'05
- *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*, Cadar et al., OSDI'08
- *Automated Whitebox Fuzz Testing*, Godefroid et al., NDSS'08

Concolic Testing

Mix concrete and symbolic execution = "concolic"

- Perform concrete and symbolic execution side-by-side
- Gather path constraints while program executes
- After one execution, negate one decision, and re-execute with new input that triggers another path

Example

```
function double(n) {
  return 2 * n;
}

function testMe(x, y) {
  var z = double(y);
  if (z === x) {
    if (x > y + 10) {
      throw "Error";
    }
  }
}
```

Concrete execution

$a = b = c = 1$

$x = y = z = 0$

true

$x = -2$

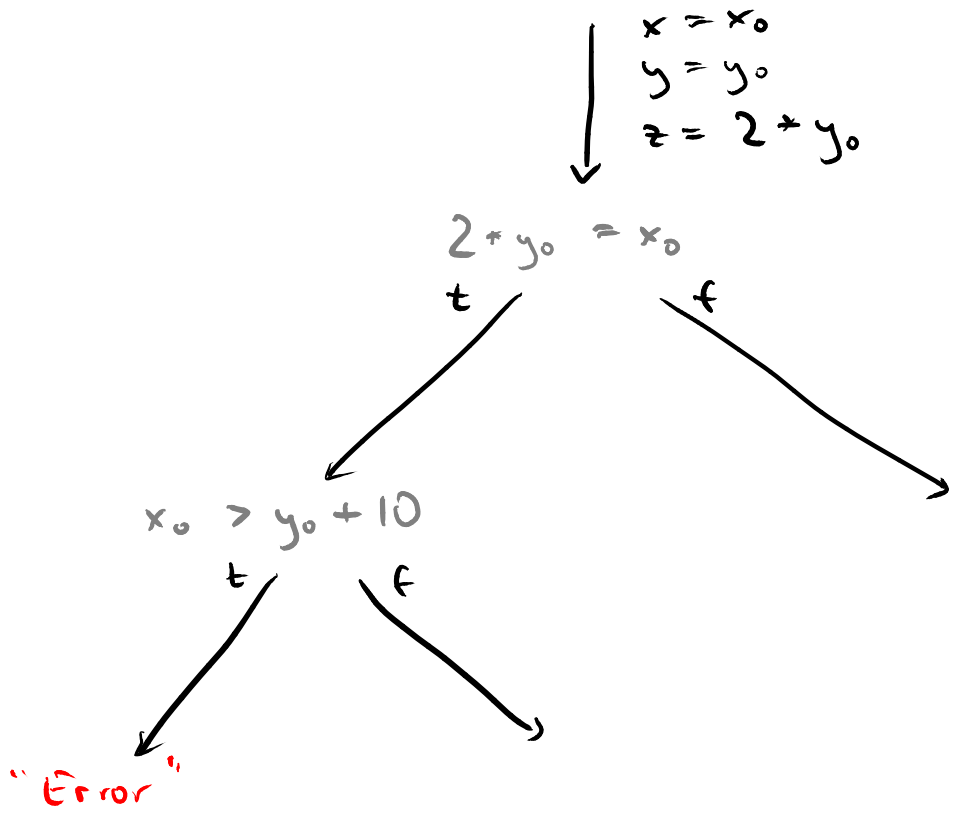
true

false

$z = 2$

$-2 + 0 + 2 \neq 3$ ✓

Symbolic execution



Execution 1

	Concrete exec.	Symbolic exec.	Path conditions
After entering fct.	$x = 22$, $y = 7$	$x = x_0$, $y = y_0$	
After call to "double"	$x = 22$, $y = 7$, $z = 14$	$x = x_0$, $y = y_0$, $z = 2 \cdot y_0$	
After outer "if"	$x = 22$, $y = 7$, $z = 14$	$x = x_0$, $y = y_0$, $z = 2 \cdot y_0$	$2 \cdot y_0 \neq x_0$

Solve: $2 \cdot y_0 = x_0$

Solution: $x_0 = 2$, $y_0 = 1$

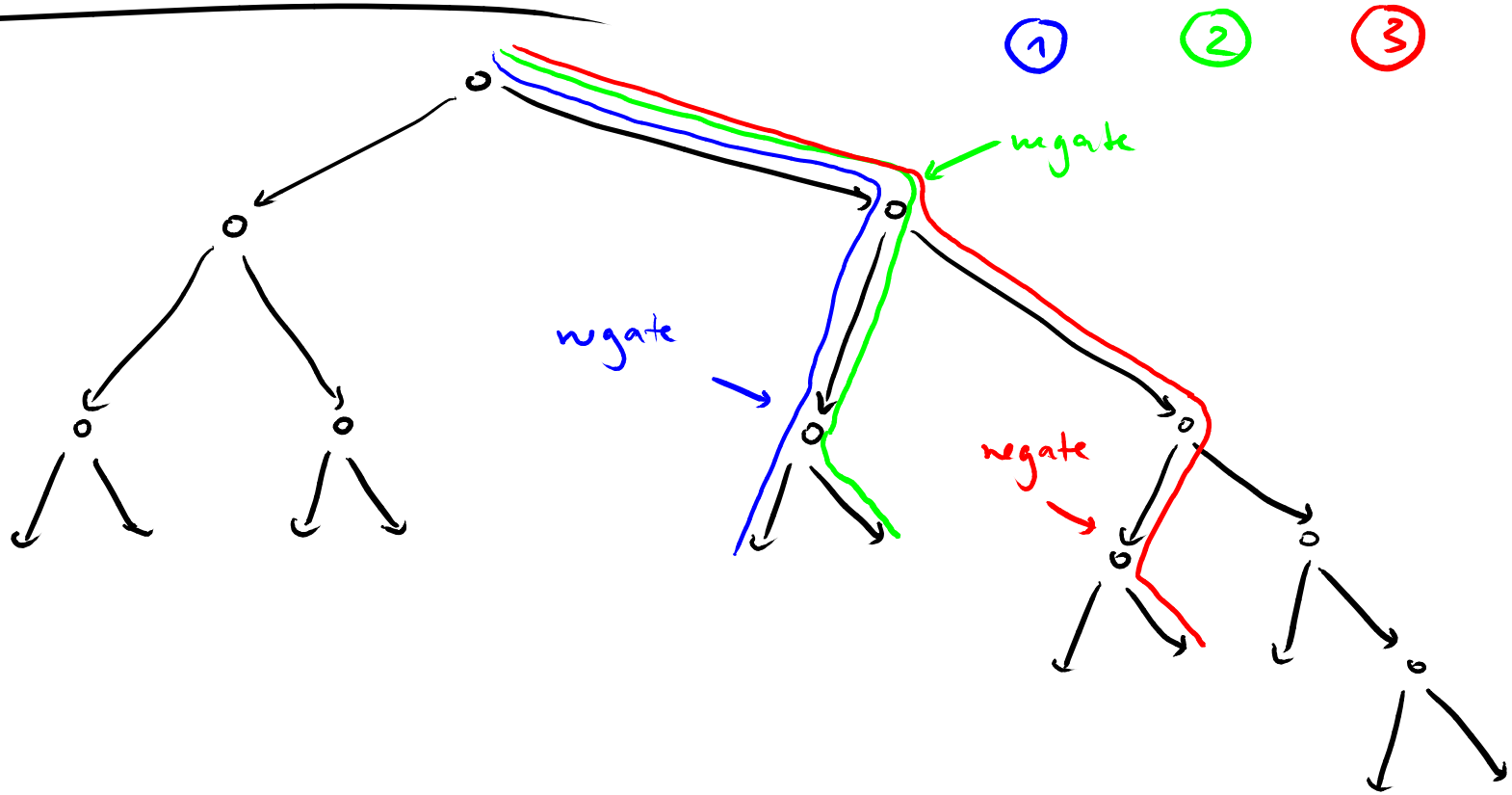
Execution 2:

	Concrete exec.	Symbolic exec.	Path conditions
After entering fct.	$x = 2,\ y = 1$	$x = x_0,\ y = y_0$	
After call to "double"	$x = 2,\ y = 1,\ z = 2$	$x = x_0,\ y = y_0,\ z = 2 \cdot y_0$	
After outer "if"	"	"	$2 \cdot y_0 = x_0$
After inner "if"	"	"	$2 \cdot y_0 = x_0 \ \wedge\ x_0 \leq y_0 + 10$

Solve: $2 \cdot y_0 = x_0 \ \wedge\ x_0 > y_0 + 10$

Solution: $\underbrace{x_0 = 30,\ y_0 = 15}_{\text{Hits "Error"}}$

Exploring the Execution Tree



Algorithm

Repeat until all paths are covered

- Execute program with concrete input i and collect symbolic constraints at branch points: C
- Negate one constraint to force taking an alternative branch b': Constraints C'
- Call constraint solver to find solution for C': New concrete input i'
- Execute with i' to take branch b'
- Check at runtime that b' is indeed taken
  Otherwise: ”divergent execution”

Divergent Execution: Example

```
function f(a) {
    if (Math.random() < 0.5) {
        if (a > 1) {
            console.log("yes");
        }
    }
}
```

Exec. 1

$a = 0$

true

false

path constraint:

$a_0 \leq 1$

negate & solve:

$a_0 = 2$

Exec. 2

$a = 2$

false

→ Divergent execution

Benefits of Concolic Approach

When symbolic reasoning is impossible or impractical, fall back to concrete values

- Native/system/API functions
- Operations not handled by solver (e.g., floating point operations)