# CS1632, Lecture 17: Static Analysis, Part 3

Wonsun Ahn

# Symbolic Model Checking

- Model checking can be categorized into:
  1. Enumerative model checking
     - What we learned in the last chapter
     - Hard to escape state explosion
  2. Symbolic model checking
     - What we will learn in this chapter
     - Model checking using *symbolic execution*
     - Can fundamentally solve the state explosion problem

# Symbolic Execution

- **Symbolic execution**: Assigning *symbolic expressions* instead of actual values to variables during execution
  - Instead of `x = 1, y = true`, …
  - `x = A + 1`, `y = A * B`, …
- **Symbolic expression**: An expression using *symbolic values*
  - `A + 1`, `A * B`, …
- **Symbolic value**: Math symbol that stands for an *input value*
  - `A, B, …, X, Y, Z`
- Idea:
  - If `x == A+1`, `y == A+2` at source line `assert(x < y)`

  → Model checker can prove though math that it always passes, for every input value without having to try them one by one!

# Notation We Will Use

- Program variables: lower case
  - `int x, y, z;`
- Symbolic values: UPPER CASE
  - `A, B, …, X, Y, Z`
  - Represent input values that are given to the code
  - Can be values from user input
  - Can be values from command line arguments
  - If testing a method, can be values passed into parameters

# Example: Enumerative Model Checking

### Code that swaps 2 integers

```
int x, y;  // parameters
if (x > y) {
  x = x + y;
  y = x – y;
  x = x – y;
  if (x > y)
    assert false;
}
```

### Execution Path for x=1, y=0

x: 1, y: 0
↓
1 > 0 ? true
↓
x = 1 + 0 = 1
↓
y = 1 – 0 = 1
↓
x = 1 – 1 = 0
↓
0 > 1 ? false

- Must do this for all values of x and y.
- But is that how a human would do it?

# Symbolic Model Checking

- Trace through a program like a human being would
- In a symbolic execution:
  - Inputs are *symbolic values* instead of concrete data values
  - Variables are *symbolic expressions* on the *symbolic values*
- Example:

```
int x, y; // parameters

x = x + y;

y = x – y;

x = x – y;
```

[Code]

x: 1, y: 0
↓
x = 1 + 0 = 1
↓
y = 1 – 0 = 1
↓
x = 1 – 1 = 0

[Concrete]

x: A, y: B
↓
x = A + B
↓
y = A + B – B = A
↓
x = A + B – A = B

[Symbolic]

- Symbolic execution proves that the swap works for all A and B!

# Symbolic Model Checking

- What if there is path divergence?
  - if statement
  - for loop
  - while loop

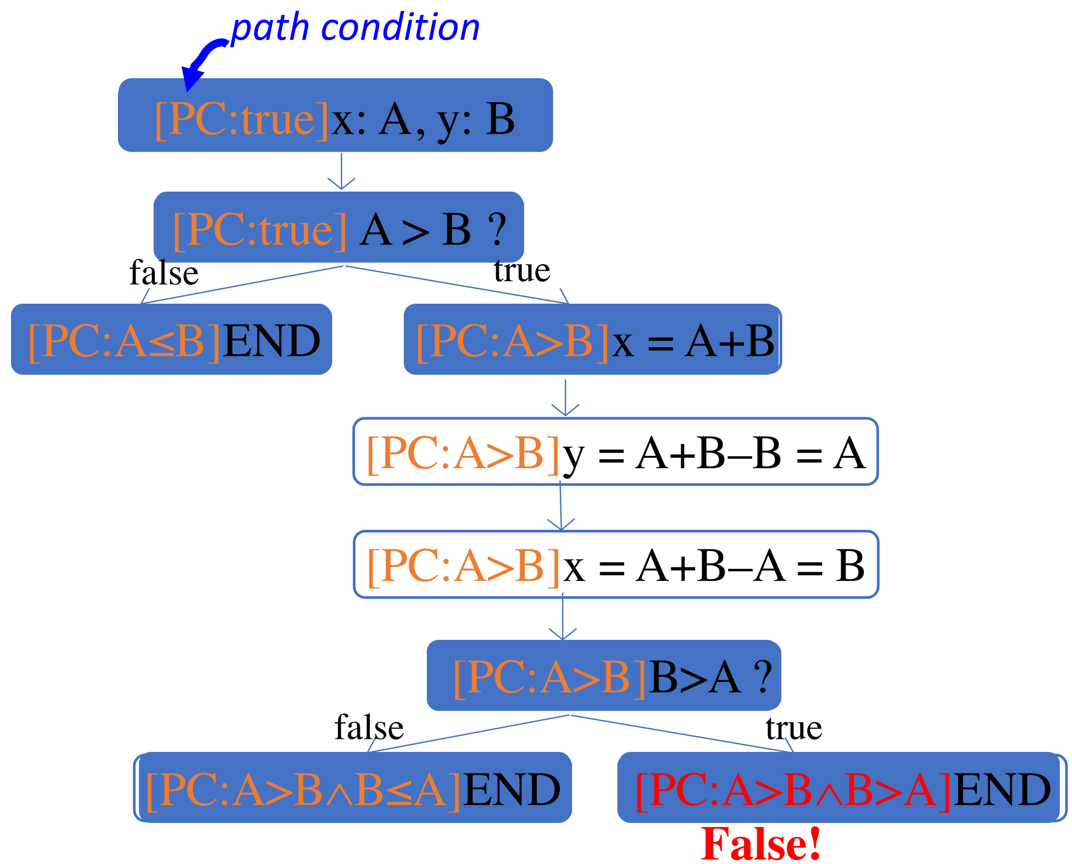
- For each path, build a Path Condition (PC)
  - Condition on symbolic values (the As and the Bs)

# Example: Symbolic Execution

**Code that swaps 2 integers:**

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

**Symbolic Execution Tree:**



# Is the Path Condition Feasible?

- Each path condition is checked using a constraint solver

$A > B \land B > A \Rightarrow$ $\boxed{\text{Constraint Solver}}$ $\Rightarrow$ {} Empty Set!

- If path is infeasible, does not continue down that path
  - Hence, assert false is never reached

# Symbolic Model Checking Uses

- Prove a program correct
  - Much less state explosion than enumerative checking
  - Now proving correctness suddenly becomes feasible
- Generate test cases
  - Generate input values that trigger a defect
  - Input values can be generated out of path conditions
- Generate program invariants
  - Invariants enhance programmer's understanding of code
  - Invariants can also be generated out of path conditions

# Generating Test Cases out of Path Conditions

Method m (x, y, d):

```
1:  d=d+1;
2:  if (x > y)
3:    return d / (x-y);
     else
4:    return d / (y-x);
```

Symbolic execution tree:

[PC: *true*] x: X, y: Y, d: D

[1:] [PC: *true*] d = D+1

[2:] [PC: *true*] X > Y ?

[3:] [PC: *X>Y*] return (D+1)/(X-Y)

[4:] [PC: *X<=Y & Y-X!=0*] return (D+1)/(Y-X)

[4:] [PC: *X<=Y & Y-X=0*] Division by zero!

*Solve path conditions → test inputs*

# Auto-generated JUnit Tests

- Constraint solver returns set of values satisfying each PC
  → Choose one of them as a test case

```
@Test public void t1() {      Pass      PC: X > Y
  m(1, 0, 1);
}
@Test public void t2() {      Pass      PC: X<=Y & Y-X!=0  ⇔ X < Y
  m(0, 1, 1);
}
@Test public void t3() {      Fail ✗    PC: X<=Y & Y-X=0  ⇔ X=Y
  m(1, 1, 1);
}
```

☛ Can achieve full path coverage by solving each PC

# Generating Invariants out of Path Conditions

- Pre-condition (insert at beginning of method):
  - assert x!=y;
- Post-condition (insert at end of method):
  - assert result == ((x>y) ? (d+1)/(x-y) : (d+1)/(y-x));
- Each method can be annotated with invariants
  - Can be checked against specifications for defects
  - Can enhance programmer's understanding of method

# Symbolic Model Checking Challenges

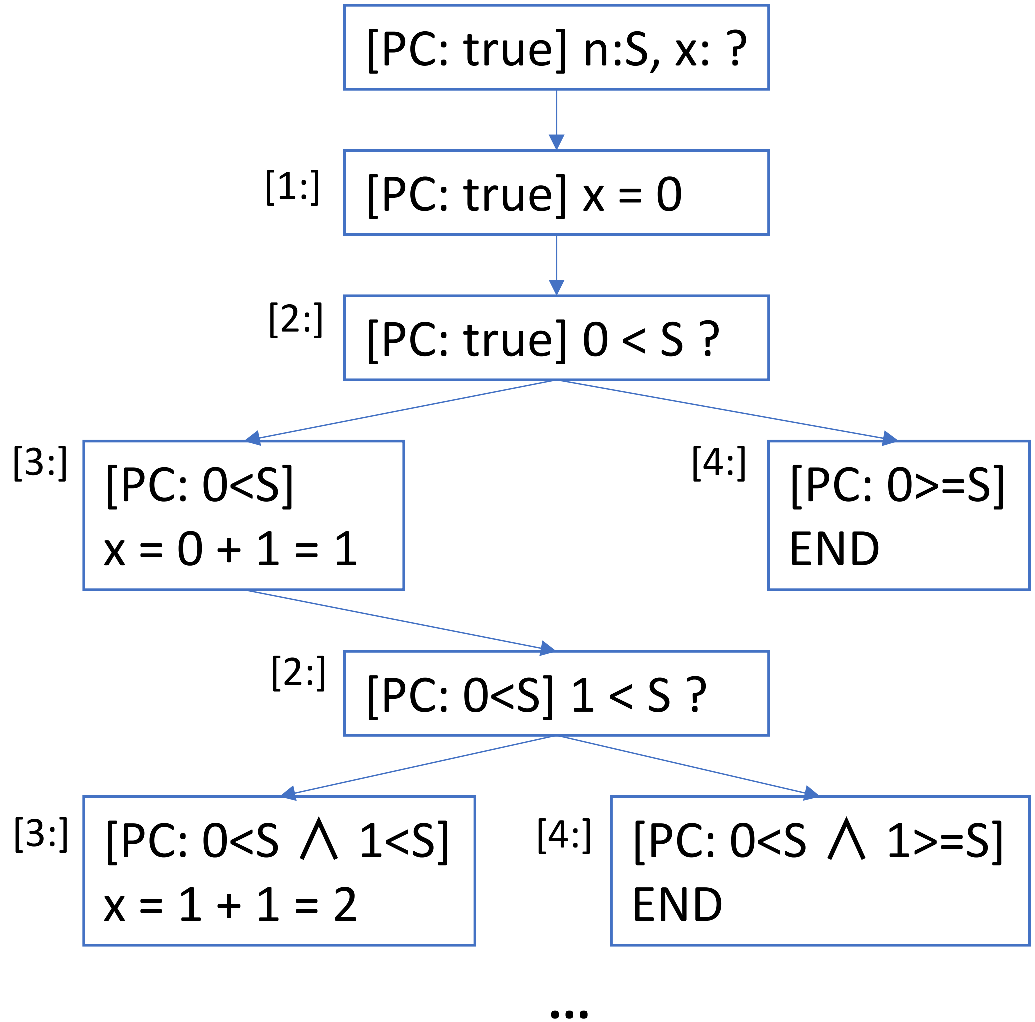
- Symbolic model checking does have challenges
- … Or every one would be using symbolic model checking
- Some examples are:
  - Loops
  - Complex math constraints
  - Complex data structures

# Challenges: Loops

## Example Code

```
void test(int n) {
1:  int x = 0;
2:  while(x < n) {
3:    x = x + 1;
4:  }
}
```

## Infinite symbolic execution tree



# Challenges: Complex Math Constraints

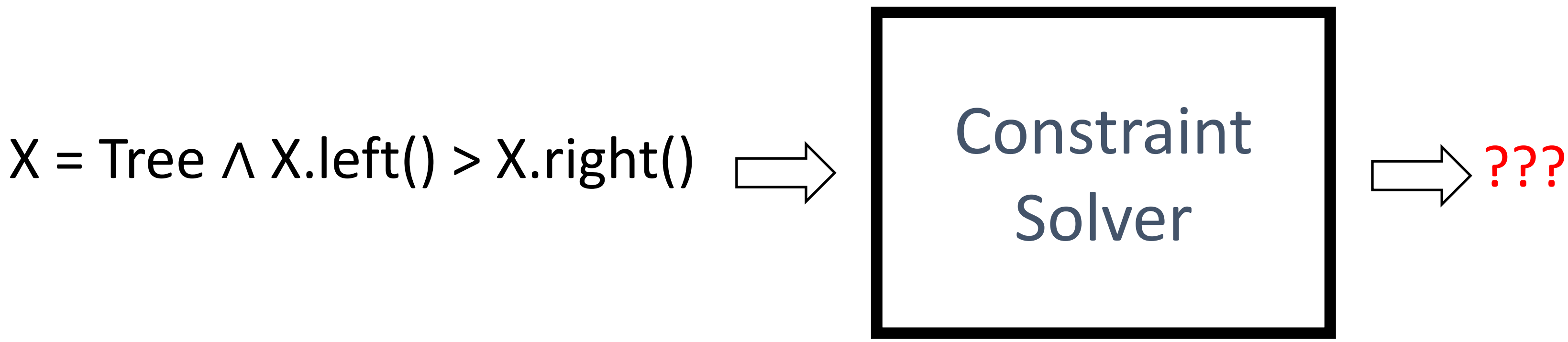
- Constraint solvers are not particularly good at math

$X > Y \land \text{hash}(X) > \text{hash}(Y)$ ⇨ **Constraint Solver** ⇨ ???

- If above constraint was an if condition:
  if ($X > Y \land \text{hash}(X) < \text{hash}(Y)$) { assert false; }
  - ☛ Will have a hard time checking whether assert fires

# Challenges: Complex Data Structures

- Complex data structures are confusing to solvers



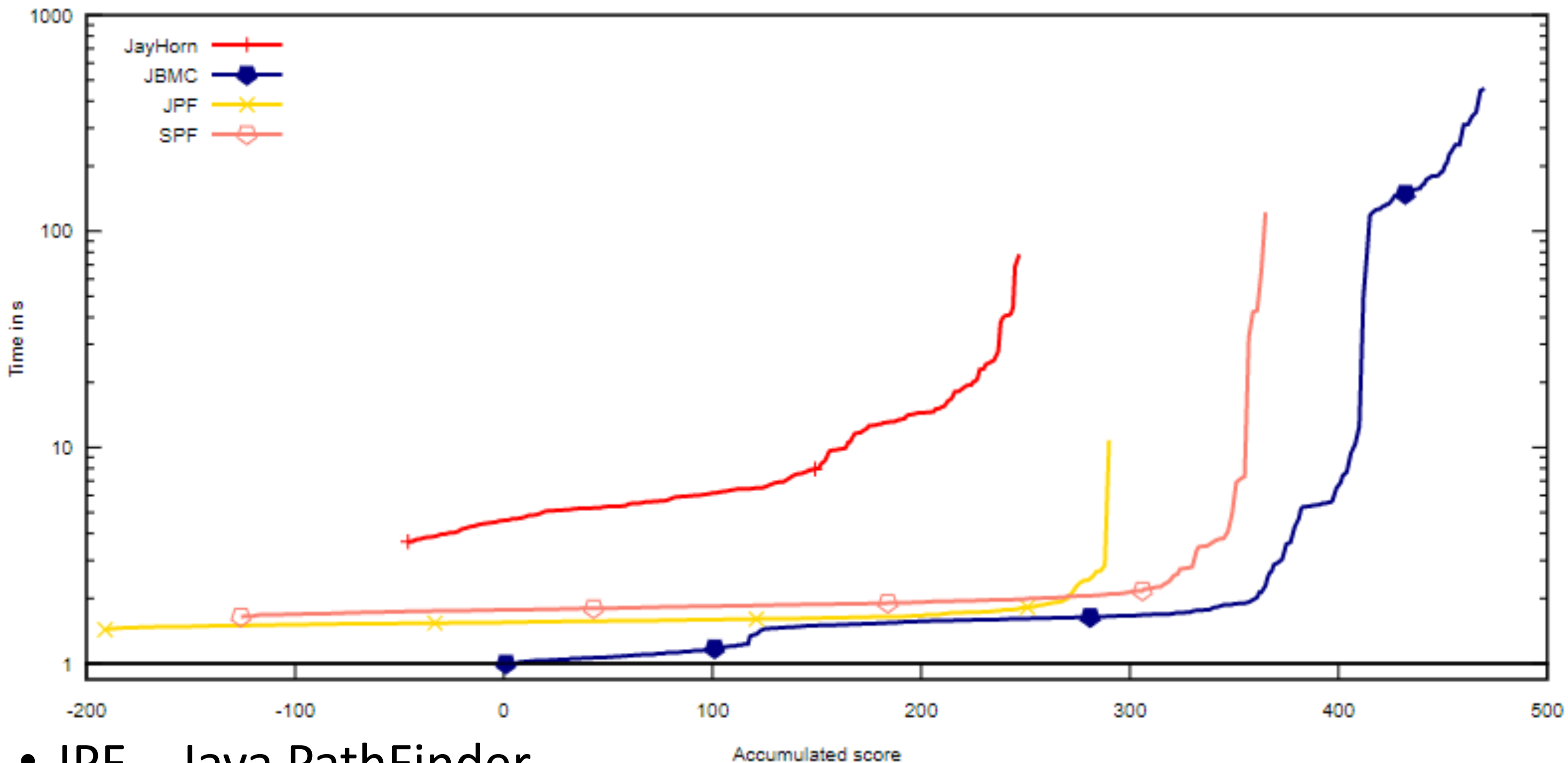

- In order to solve above constraint, solver must know:
  - What a tree data structures looks like
  - What left() means and what right() means
- Solvers know some data structures, but not many

# The Best of Both Worlds

- Symbolic Model Checking (Symbolic Execution)
  + Much less state explosion
  - Hard time dealing with loops, math, data structures
- Enumerative Model Checking (Concrete Execution)
  - Serious state explosion
  + No problems with loops, math, data structures
  (just execute the loop, math, or data structure code)
- The best of both worlds: Concolic Execution
  - Concolic = Concrete + Symbolic
  - Symbolic Java Path Finder does exactly this!
  - Chooses between the two depending on the method

# Model Checking is Getting Better Every Year

https://sv-comp.sosy-lab.org/2019/results/results-verified/




- JPF – Java PathFinder
- SPF – Symbolic Java PathFinder (JPF with symbolic execution)
- JBMC – Java Bounded Model Checker (2018 newcomer)

# References


- Ranjit Jhala and Rupak Majumdar. 2009. "Software model checking". ACM Computing Surveys: https://people.mpi-sws.org/~rupak/Papers/SoftwareModelChecking.pdf

- Cristian Cadar and Koushik Sen. 2013. "Symbolic execution for software testing: three decades later". Communications of the ACM: https://people.eecs.berkeley.edu/~ksen/papers/cacm13.pdf

- 8th Competition on Software Verification (SV-COMP), 2019: https://sv-comp.sosy-lab.org/2019/results/results-verified/