

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 $x = 1, y = \text{true}, \dots$
 - $x = A + 1, y = A * B, \dots$
- **Symbolic expression:** An expression using *symbolic values*
 - $A + 1, A * B, \dots$
- **Symbolic value:** Math symbol that stands for an *input value*
 - A, B, \dots, X, Y, Z
- Idea:
 - If $x == X + 1$ and $y == X + 2$,
we can prove that invariant `assert (x < y)` always holds.
 - Holds for every input value without having to try 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 external to the code

- Can be values from user input

- Can be values from command line arguments

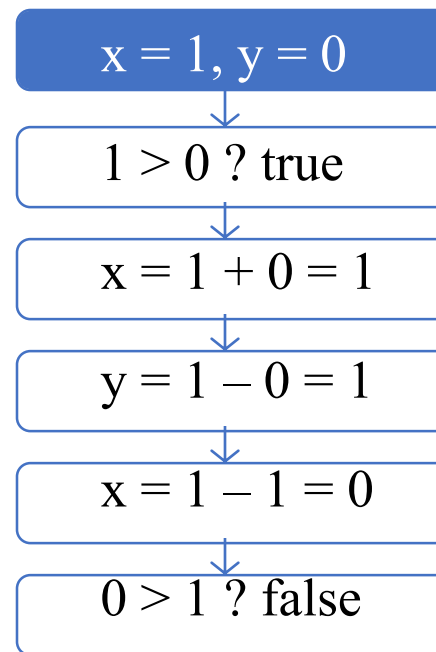
- If testing method, can be values passed into parameters

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



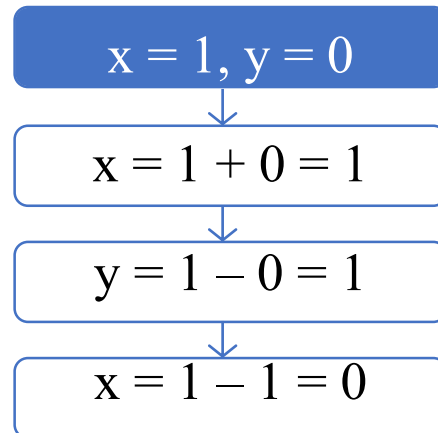
- Must do this for all values of x and y.
- But is that how a human would do it?

Symbolic Model Checking

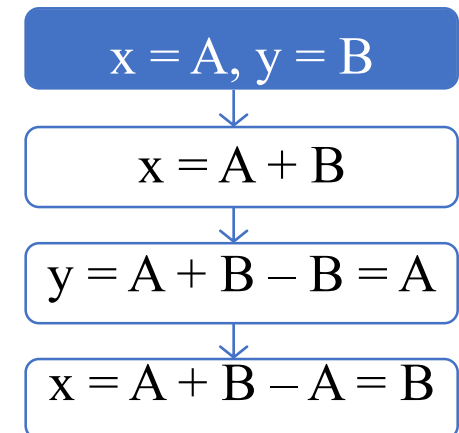
- Trace through a program like a human being would
- In a symbolic execution:
 - Inputs are *symbolic values* instead of concrete data values
 - Variables are *symbolic expressions* on the *symbolic values*
- Example:

```
int x, y; // parameters  
x = x + y;  
y = x - y;  
x = x - y;
```

[Code]



[Concrete]



[Symbolic]

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

Symbolic Model Checking

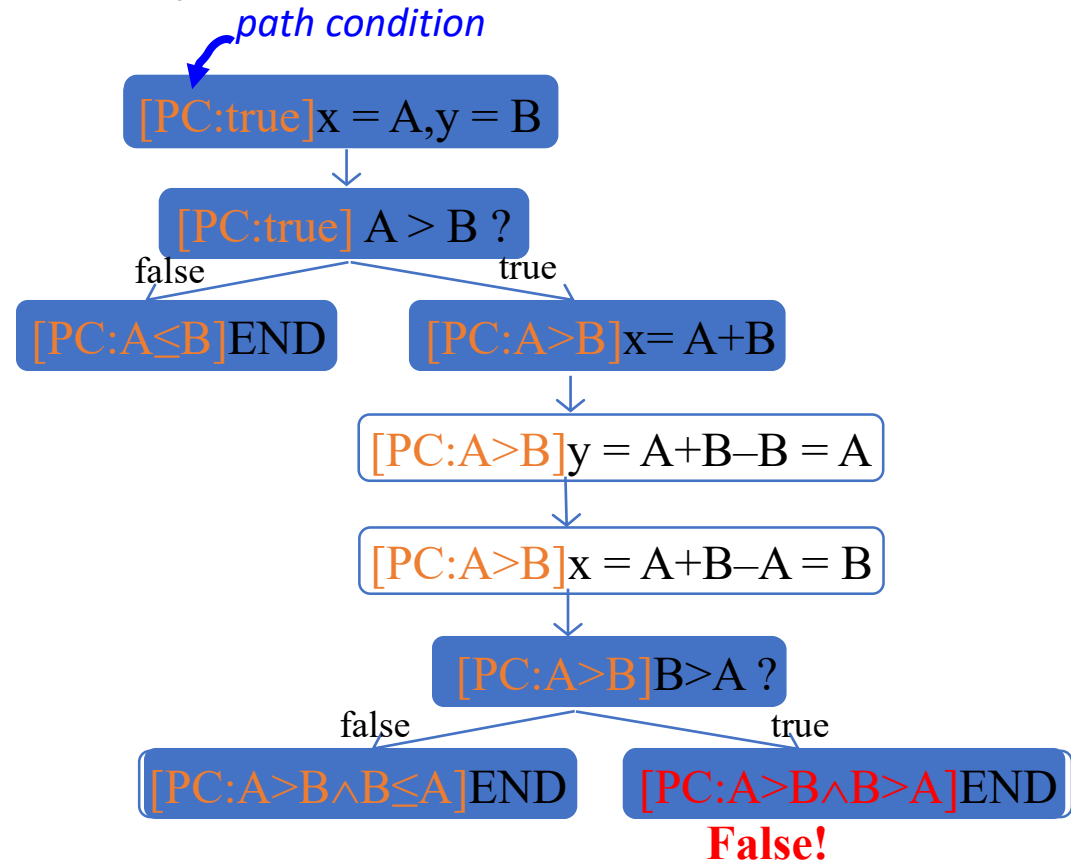
- What if there is path divergence?
 - if statement
 - for loop
 - while loop
- For each path, build a **Path Condition (PC)**
 - Condition on symbolic values (the As and the Bs)

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



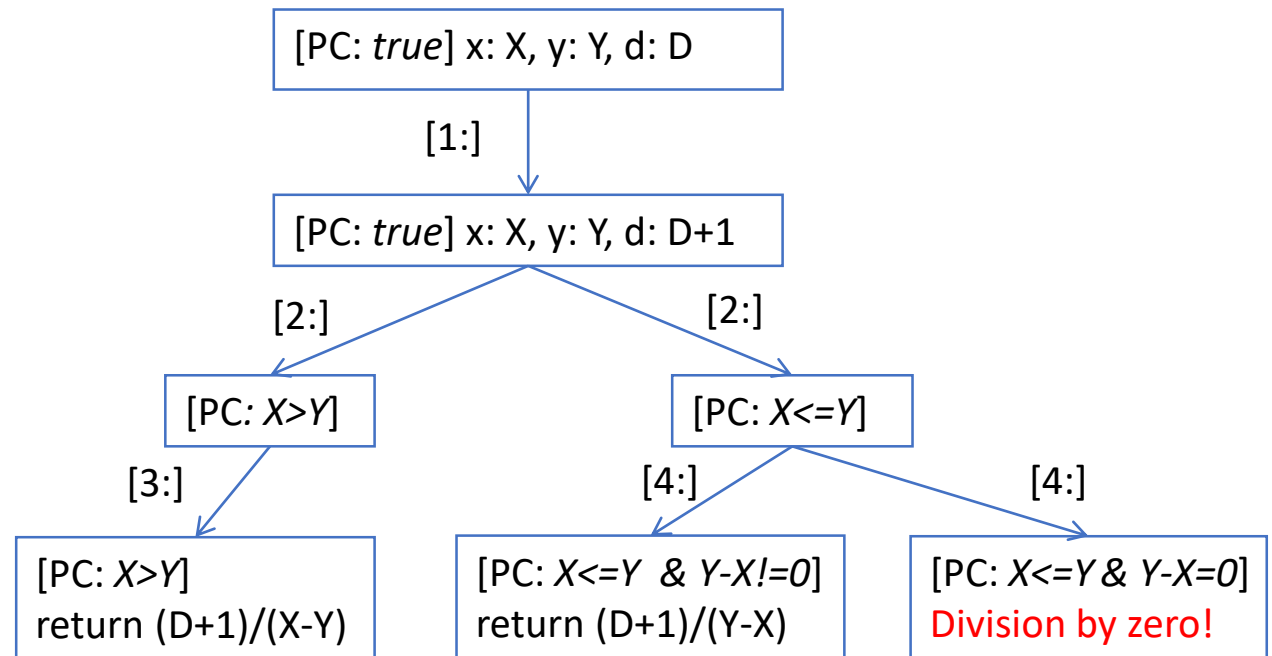
- If path is infeasible, does not continue down that path
 - Hence, **assert false** is never reached

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

Symbolic execution tree:



Solve path conditions → test inputs

Auto-generated JUnit Tests

```
@Test public void t1() {  
    m(1, 0, 1);  
}
```

Pass ✓ PC: $X > Y$

```
@Test public void t2() {  
    m(0, 1, 1);  
}
```

Pass ✓ PC: $X \leq Y \ \& \ Y - X \neq 0 \Leftrightarrow X < Y$

```
@Test public void t3() {  
    m(1, 1, 1);  
}
```

Fail ✗ PC: $X \leq Y \ \& \ Y - X = 0 \Leftrightarrow X = Y$

☛ Achieves full path coverage

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

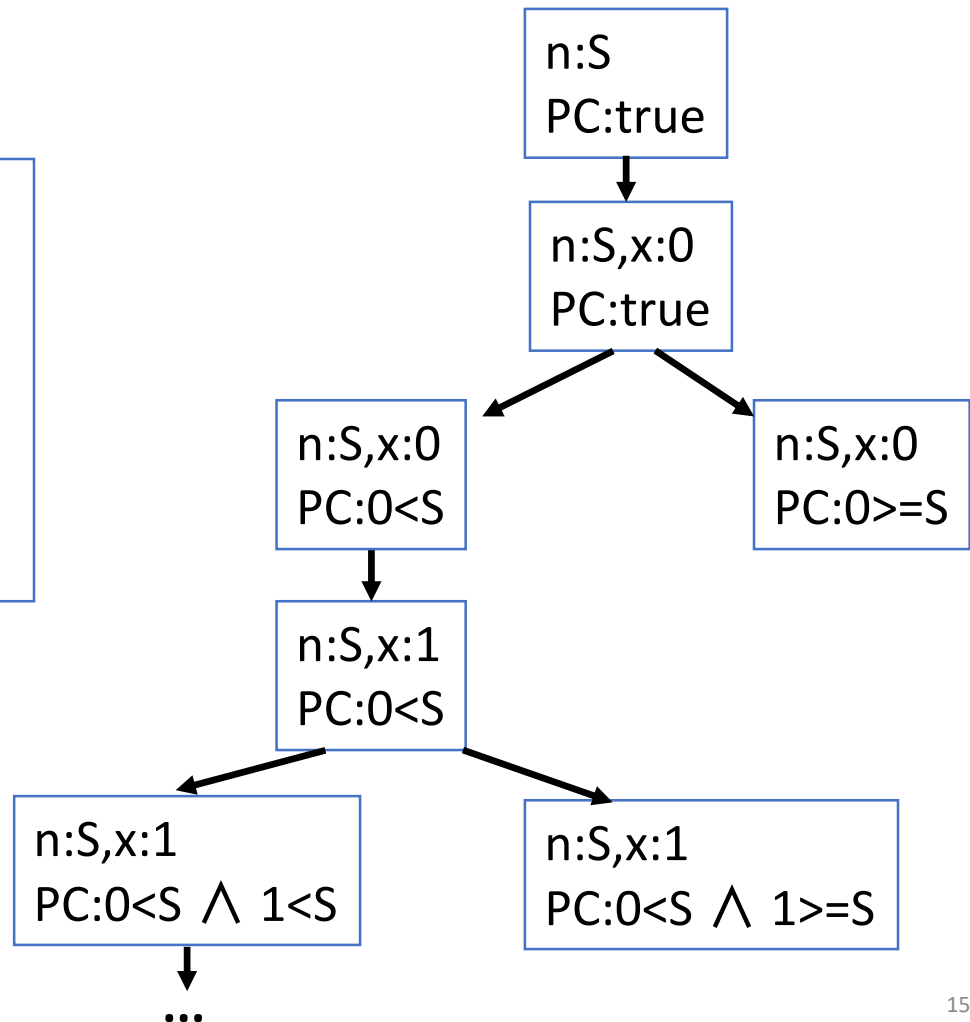
- Symbolic model checking does have challenges
- ... Or every one would be using symbolic model checking
- Some examples are:
 - Loops
 - Complex math constraints
 - Complex data structures

Challenges: Loops

Example Code

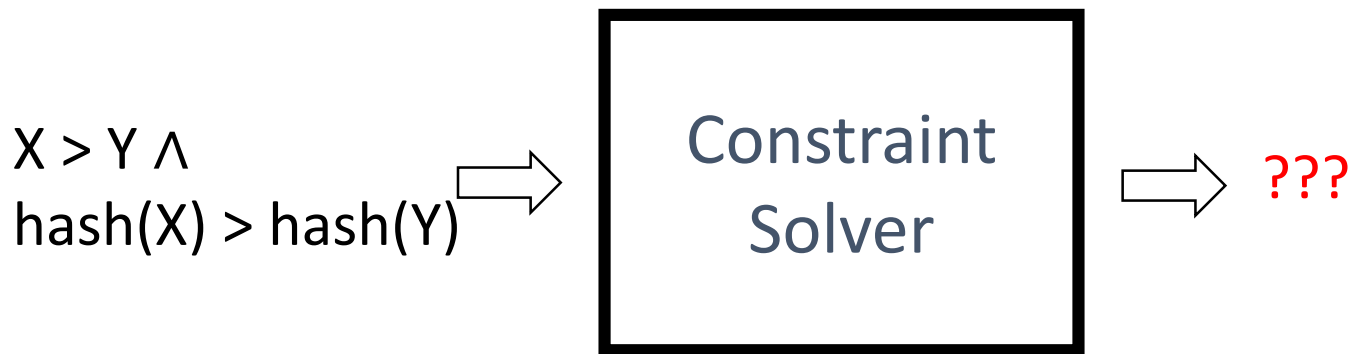
```
void test(int n) {  
  int x = 0;  
  while(x < n) {  
    x = x + 1;  
  }  
}
```

Infinite symbolic execution tree



Challenges: Complex Math Constraints

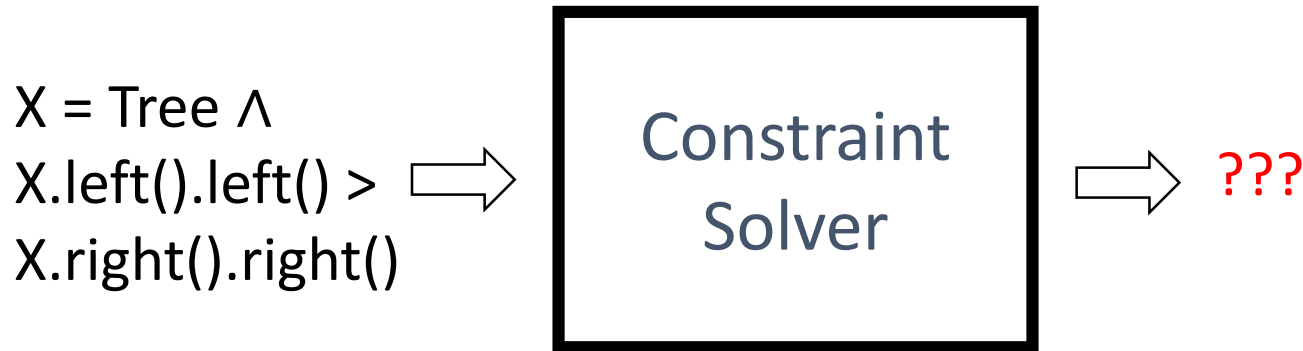
- Constraint solvers are not particularly good at math



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

Challenges: Complex Data Structures

- Complex data structures are confusing to solvers



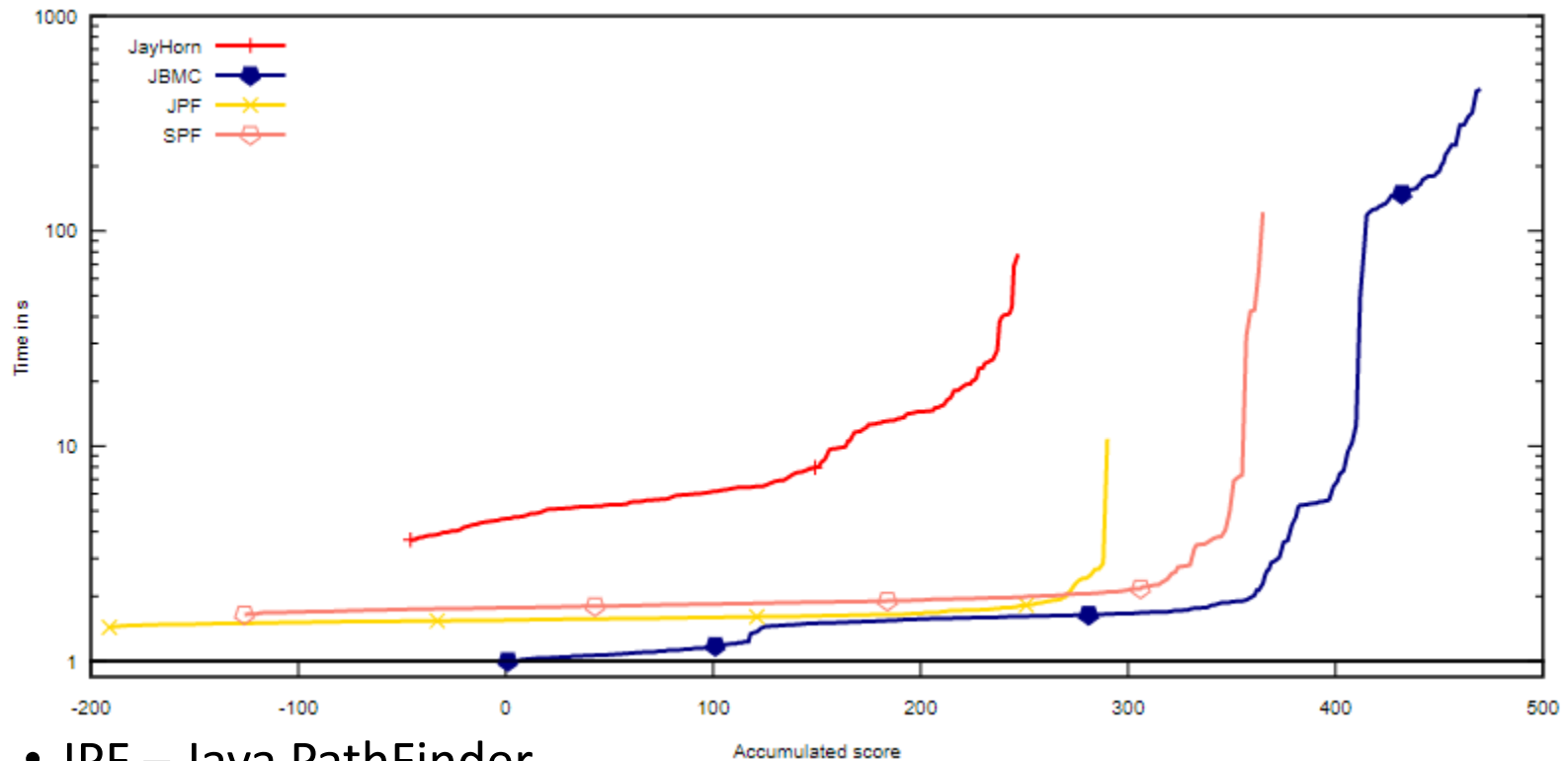
- In order to solve above constraint, solver must know:
 - What a tree data structures looks like
 - What `left()` means and what `right()` means
- Solvers know some data structures, but not many

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/>