CS1632, Lecture 19: Static Analysis, Part 3

Wonsun Ahn

State Space Reduction Techniques

- State collapsing
- Heuristic state approximation
- Hash compaction
- Heap canonicalization
- Symbolic execution

- So far we have been looking at *enumerative* model checking
  - Enumerating all the states a program can be in
  - No matter how hard you try, no escaping state explosion
- Symbolic execution can fundamentally change the equation

Let’s take a step back

- Model checking
  - Abstract model checking: models abstraction of states
  - Concrete model checking: models actual program states
    - ☛We are here
- Concrete model checking can be subdivided into:
  - Concrete *enumerative* model checking
  - Concrete *symbolic* model checking
    - Concrete model checking using symbolic execution
    - ☛And we are here

Example: Enumerative Model Checking

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;
}
```

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;

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]

$x = X, y = Y$

↓

$x = X + Y$

↓

$y = X + Y - Y = X$

↓

$x = X + Y - X = Y$

[Symbolic]

- Symbolic execution proves that the swap works for all X and Y!

Symbolic Model Checking

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

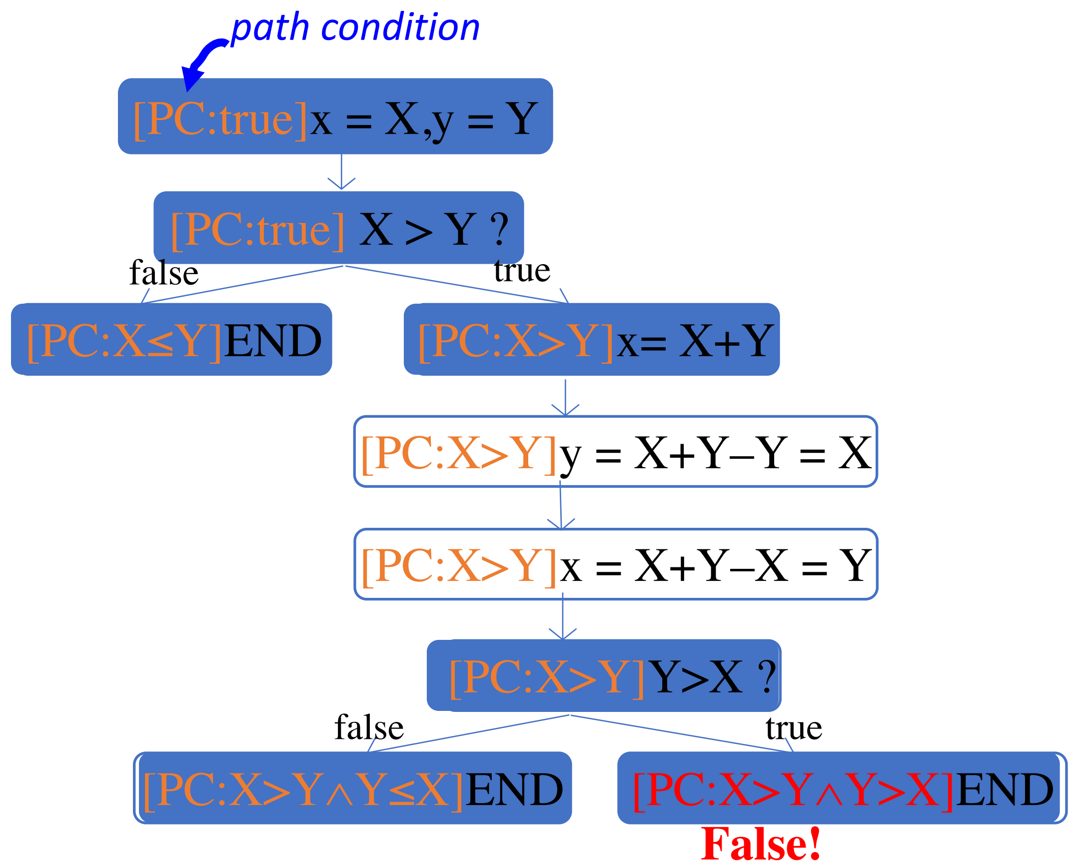
- For each path, build a Path Condition (PC)
  - Condition on symbolic values (the Xs and the Ys)

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

$X > Y \wedge Y > X \Rightarrow$ $\boxed{\text{Constraint Solver}}$ $\Rightarrow \{\}$ Empty Set!

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

Symbolic Model Checking Uses

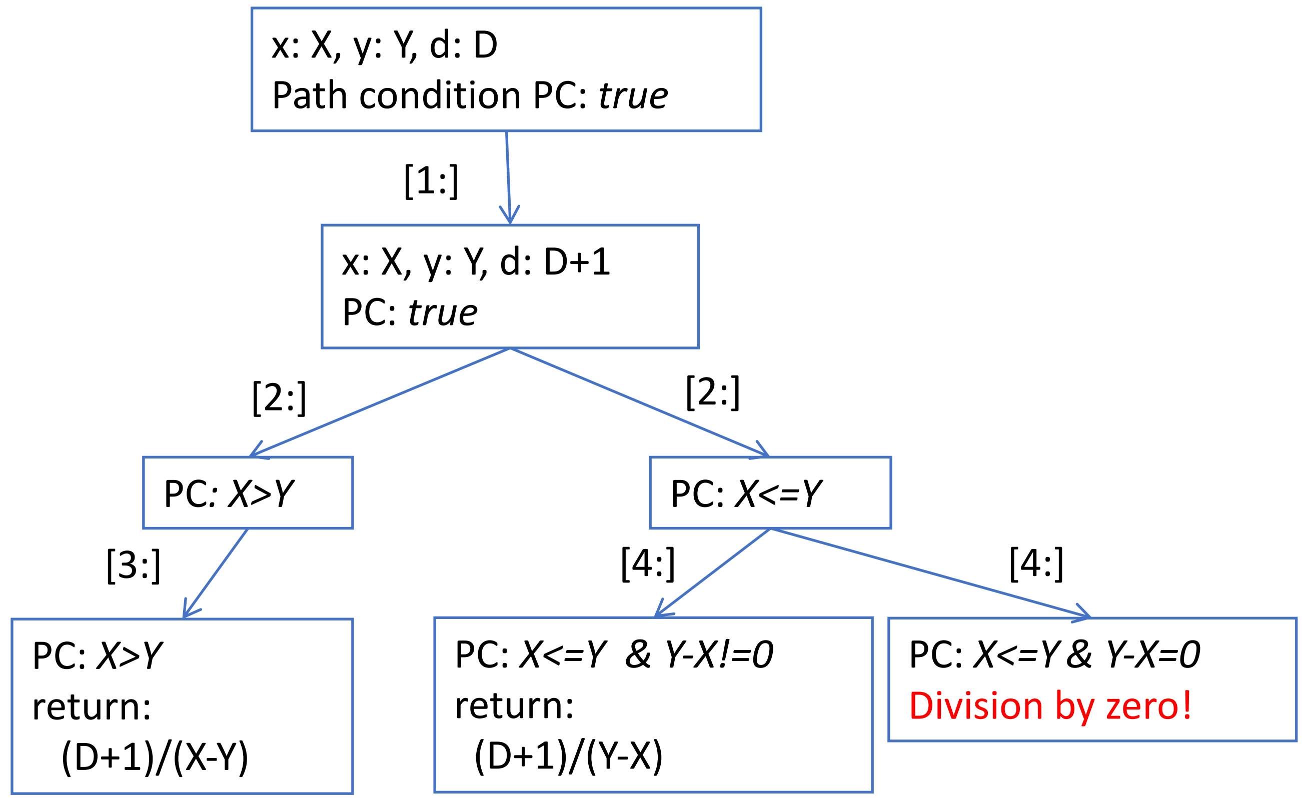
- Prove a program correct
  - Much less state explosion than enumerative checking
  - Now proving correctness suddenly becomes feasible
- Generate test cases
  - Generate input values that trigger a defect
  - Input values can be generated out of path conditions
- Generate program invariants
  - Invariants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



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: $X<=Y$ & $Y-X!=0$ ⬄ $X < Y$

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

Fail ✗   PC: $X<=Y$ & $Y-X=0$ ⬄ $X=Y$

☛ Achieves full path coverage

Generating Invariants out of Path Conditions

- Pre-condition:
  - “x!=y”

- Post-condition:
  - “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

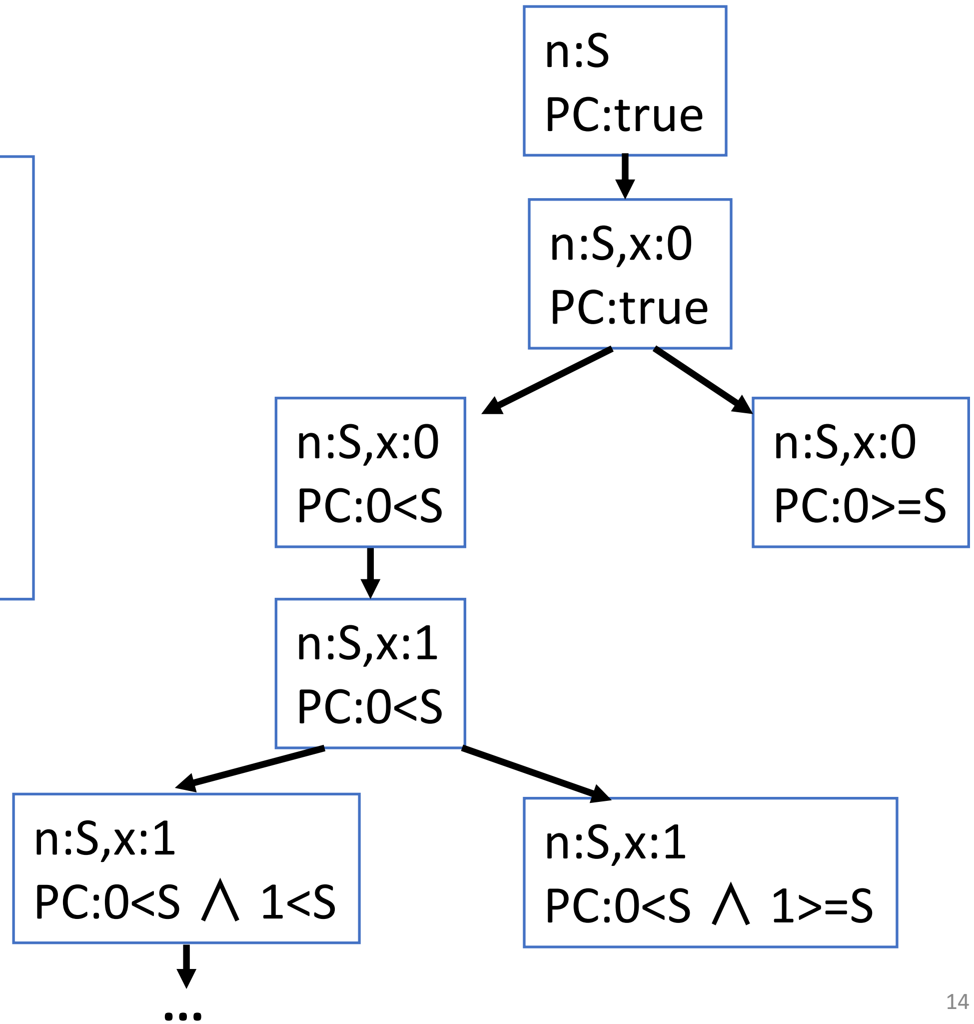
- Symbolic model checking does have challenges
- … Or every one would be using symbolic model checking
- Some examples are:
  - Loops
  - Complex math constraints
  - Complex data structures

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

$X > Y \land$
$hash(X) > hash(Y)$ $\Rightarrow$ Constraint Solver $\Rightarrow$???

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

Challenges: Complex Data Structures

- Complex data structures are confusing to solvers

X = Tree ∧
X.left().left() >
X.right().right() ⇨ Constraint Solver ⇨ ???

- 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
  - a.k.a. DART(Directed Automated Random Testing)

Automated Random Testing

- Where have I heard that before? Hmm…
- Stochastic Testing is an automated random test
  - Randomly selects values to check given property
- Fuzz Testing is also an automated random test
  - Randomly fuzzes inputs in corpus to expand coverage

- Directed Automated Random Testing
  - Also uses random input for the initial run
  - But subsequently uses symbolic execution to direct search

DART (Directed Automated Random Testing)

1. Run the program starting with some random inputs
2. Gather symbolic constraints at conditional statements
3. Use a constraint solver to generate new test inputs
   (New test inputs should exercise new path)
4. Go back to 1.

* Repeat until all paths are covered

- So what’s different from pure symbolic execution?
  - Now we have concrete values as well as symbolic values
  - Now constraint solver can do a much better job

Directed Search

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

Concrete Execution	Symbolic Execution	Path Constraint
x = 0, y = 0	create symbolic variables X, Y	

Directed Search

Concrete Execution	Symbolic Execution	Path Constraint
	create symbolic variables X, Y	
		$X \leq Y$
	Solve: $!(X \leq Y)$ Solution: X=1, Y=0	
x = 0, y = 0		

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

Directed Search

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

Concrete Execution	Symbolic Execution	Path Constraint
x = 1, y = 0	create symbolic variables X, Y	

Directed Search

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

Concrete Execution	Symbolic Execution	Path Constraint
	create symbolic variables X, Y	
x = 1, y = 0		X > Y

Directed Search

Concrete Execution	Symbolic Execution	Path Constraint
	create symbolic variables X, Y	
		X > Y
x = 1, y = 0	x = X+Y	

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

Directed Search

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

Concrete Execution	Symbolic Execution	Path Constraint
	create symbolic variables X, Y	
		X > Y
	x = X+Y	
x = 1, y = 1	y = X	

Directed Search

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

Concrete Execution	Symbolic Execution	Path Constraint
	create symbolic variables X, Y	
		X > Y
	y = X	
x = 0, y = 1	x = Y	

Directed Search

Concrete Execution	Symbolic Execution	Path Constraint
	create symbolic variables X, Y	
		$X > Y$
	$y = X$	
	$x = Y$	
$x = 0, y = 1$		$Y \leq X$

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

Solve: $X > Y$ AND $!(Y \leq X)$

Impossible: DONE!

DART (Directed Automated Random Testing)

- Gaining popularity in industry
  + Unlike symbolic execution, can work on complex apps
  + Unlike stochastic testing, can achieve very high coverage

- Many tools
  - PEX, SAGE, YOGI (Microsoft)
  - KLEE: LLVM open source project

- Many applications
  - Bug finding, security, web and database applications, etc.

State Space Reduction Techniques

- State collapsing
- Heuristic state approximation
- Hash compaction
- Heap canonicalization
- Symbolic execution

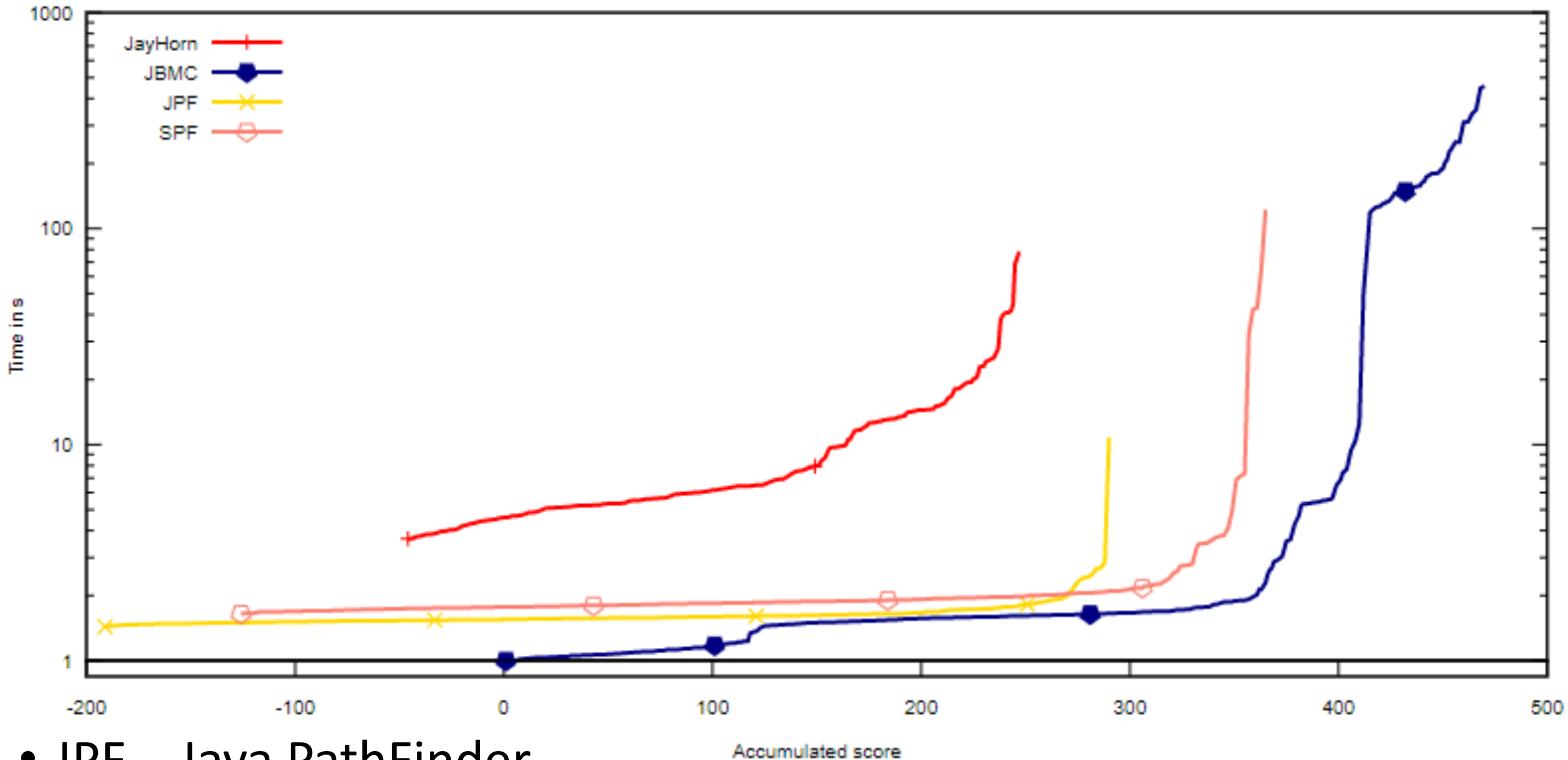
- What if the state space is *still* too large?
- One recourse – reduce the problem size

Reducing the problem size

- When state space explosion prevents exhaustive exploration, What are the alternatives?
    1. Put a cap on problem size and exhaustively explore
    2. Put a cap on time / space and do a partial exploration
- In most cases, putting a cap on problem size is better
    - Most corner cases exhibit with a relatively small problem size
    - Correctness with smaller sizes usually translate to larger sizes
- Examples of capping problem size
    - Instead of checking an infinite tree structure, check a limited tree
    - Instead of checking infinite number of players, check just 10
    - Etc.

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/