UT ECE | THE UNIVERSITY OF TEXAS AT AUSTIN
DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

EE360T/382V Software Testing

khurshid@ece.utexas.edu

April 18, 2018

Overview

Last time

- Systematic test input generation

Today

- Symbolic execution

Next time

- Testing multi-threaded code
- Testing websites

Reminder

- **Exam 3 on Monday, April 30** – in-class, closed book
  - Focus: Practical considerations (Text), systematic testing, non-det. choice, symbolic execution

Quick re-cap from last time

Non-deterministic choice in Java PathFinder

- `getInt(min, max)`
  - Systematically explore each value between min and max (inclusive), where *min* ≤ *max*
- `getBoolean()`
  - Systematically explore true and false

```
static void m() {
  int x = Verify.getInt(0, 2);
  boolean b = Verify.getBoolean();
  System.out.println(x + ", " + b);
}
```

```
0, false
0, true
1, false
1, true
2, false
2, true
```

Quick exercise

What is the console output of the following program?

```
static void p() {
  int x = 0;
  boolean b = false;
  x = Verify.getInt(0, 2);
  for (int i = 0; i < x; i++) {
    b = Verify.getBoolean();
  }
  System.out.println(x + ", " + b);
}
```

0, false
1, false
1, true
2, false
2, true
2, false
2, true

Symbolic execution

Technique for executing the program on symbolic input values to analyze its behaviors

- Invented in 70's; heavily studied in last 15 years
- Explores symbolic execution tree of (bounded) paths
  - For each path, builds a path condition that represents inputs that execute that path
  - Checks satisfiability of path conditions to try to avoid infeasible paths

Program state: symbolic values of variables, path condition, and counter

Various applications, e.g., for test input generation

Traditional focus: programs with primitives and arrays

Simple example – straight-line code
[Based on Fig. 1, KingCACM'76]

```
static int sum(int a, int b, int c) {
  int x = a + b;
  int y = b + c;
  int z = x + y - b;
  return z;
}
```

Concrete execution – Input: a = 1, b = 2, c = 3

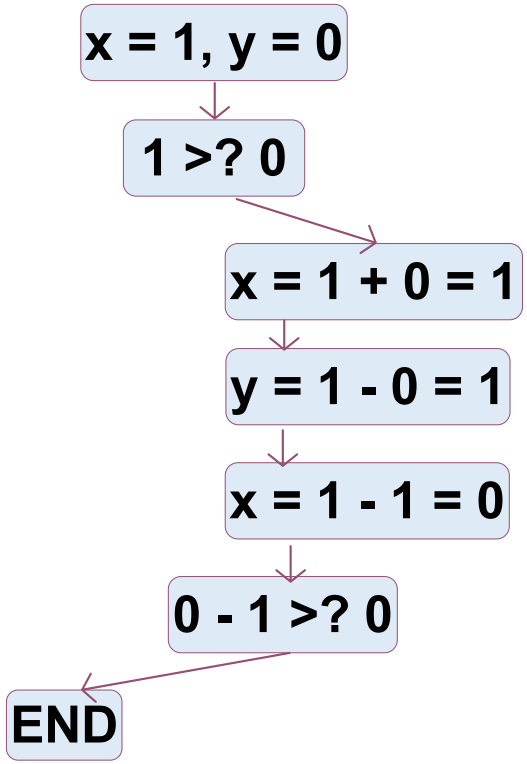
- Output: 6

Symbolic excution

- Input: a = A, b = B, c = C
- Output: (((A + B) + (B + C)) - B), i.e., A + B + C
- Path condition: true

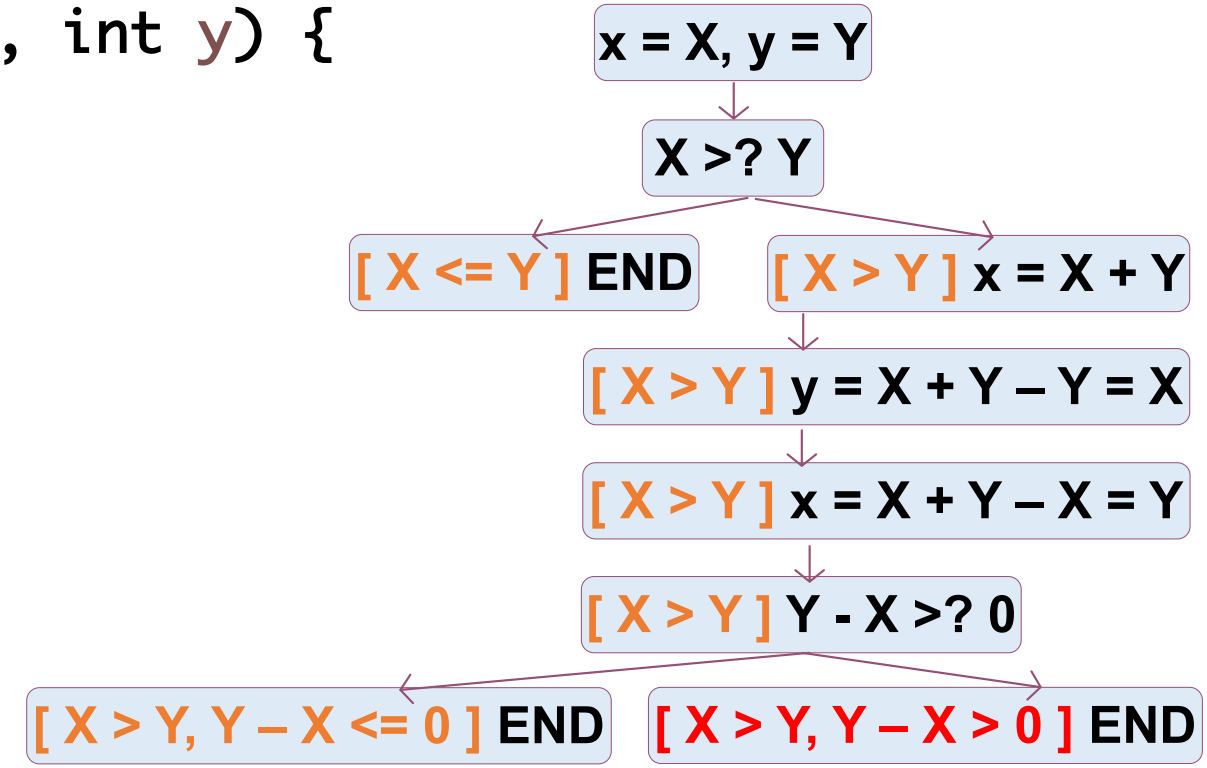
Example: Concrete execution path

```
static void m(int x, int y) {
  if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0)
      assert false;
  }
}
```



Example: Symbolic execution tree

```
static void m(int x, int y) {
  if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0)
      assert false;
  }
}
```



Symbolic execution: Method with fixed number of inputs with primitive types

Given a method *m* to execute symbolically:

1. Identify *m*'s inputs and assign a symbol to each
2. Initialize a path condition *pc* to true
3. Execute *m* following standard Java semantics except for operations on symbols
   - Evaluate symbolic expressions algebraically
   - For Boolean condition *c*, assume non-deterministically: (1) *c* is true; and (2) *c* is false
     - For each conditional branch taken, update *pc* with branch condition – over input symbols
       - If possible, check satisfiability of *pc*

Practical issues

What about loops and recursion?

- Conceptually, follow Java semantics to (dynamically) unroll loops and inline methods invoked as needed
  - Cannot continue indefinitely!

When to terminate path exploration?

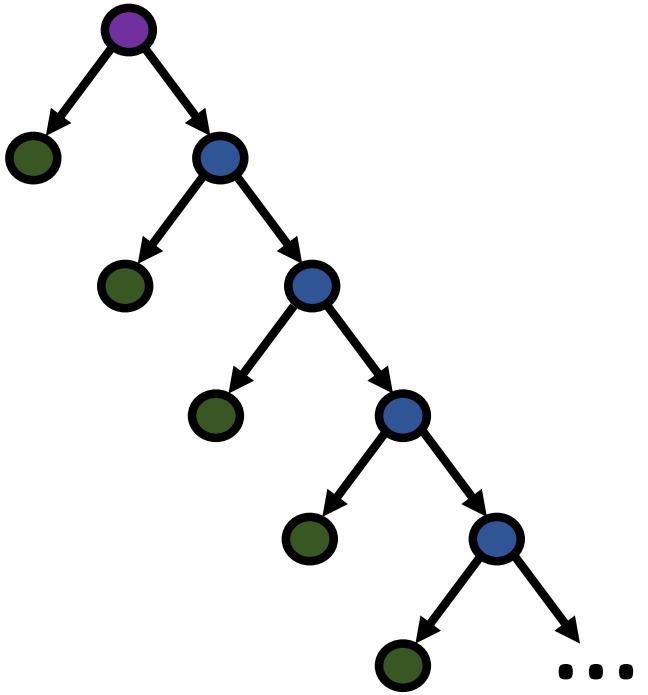
- One way: use a bounded depth, e.g., ≤ *k* branches

How to check path condition feasibility?

- In general, undecidable
- In practice, automated constraint solvers (SAT/SMT) can check some useful classes of constraints

Example: While-loop
[Based on Fig. 4, KingCACM'76]

```
static int power(int x, int y) {
  int z = 1;
  int j = 1;
  while (y >= j) {
    z = z * x;
    j = j + 1;
  }
  return z;
}
```



Depth bound: ≤ 5 conditional branches

Symbolic execution of power

PC: (Y < 1)

- Solution: Y = 0; Output: 1

PC: (Y >= 1), (Y < 2)

- Solution: Y = 1; Output: (1 * X)

PC: (Y >= 1), (Y >= 2), (Y < 3)

- Solution: Y = 2; Output: ((1 * X) * X)

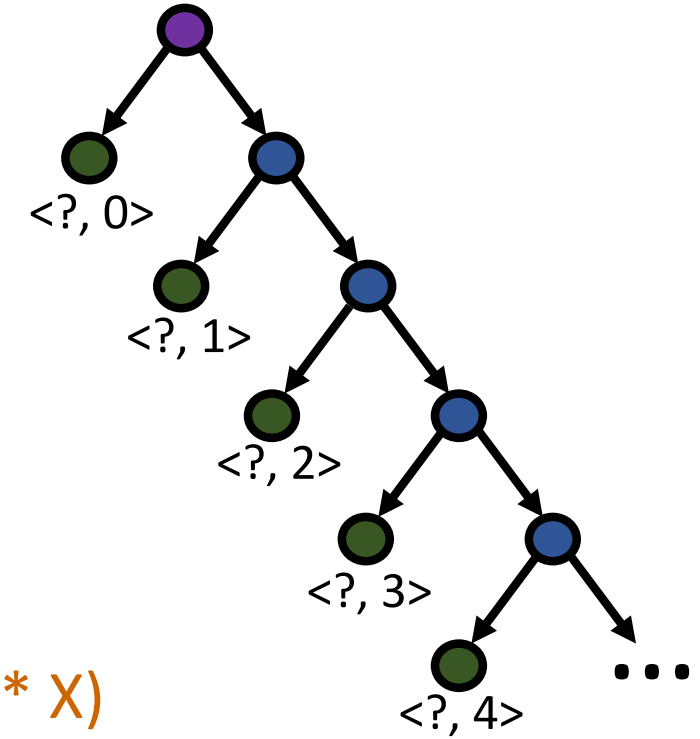
PC: (Y >= 1), (Y >= 2), (Y >= 3), (Y < 4)

- Solution: Y = 3; Output: (((1 * X) * X) * X)

PC: (Y >= 1), (Y >= 2), (Y >= 3), (Y >= 4), (Y < 5)

- Solution: Y = 4; Output: ((((1 * X) * X) * X) * X)

…



Inputs with non-primitive types

How to handle arrays of primitives?

- Array could be null, or have length 0, 1, 2, ...
  - Each array element is a symbolic value
- In general, check for array access out of bounds
- To simplify, assume and state explicitly:
  - Array has a given constant length, e.g., 3

Not in scope

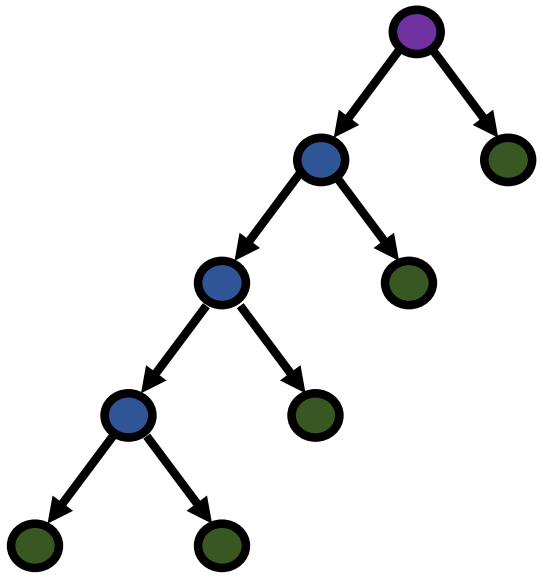
How to handle references (non-arrays)?

- Add constraints on references to the PC, e.g.,
  `N != null && N.elem > 0 && N.next == null;`
- Lazy initialization: Non-det. try all concrete values, e.g., null, existing objects, a new object

How to handle arrays of references?

Example: Integer array

```
static int search(int[] a, int x) {
  for (int i = 0; i < a.length; i++) {
    if (a[i] == x) {
      return i;
    }
  }
  return -1;
}
```



Assume:

```
a != null && a.length == 4
```

Symbolic execution of search

PC: (A[0] != X), (A[1] != X), (A[2] != X), (A[3] != X)

- Solution: <[0, 0, 0, 0], 1>; Output: -1

PC: (A[0] != X), (A[1] != X), (A[2] != X), (A[3] == X)

- Solution: <[1, 1, 1, 0], 0>; Output: 3

PC: (A[0] != X), (A[1] != X), (A[2] == X)

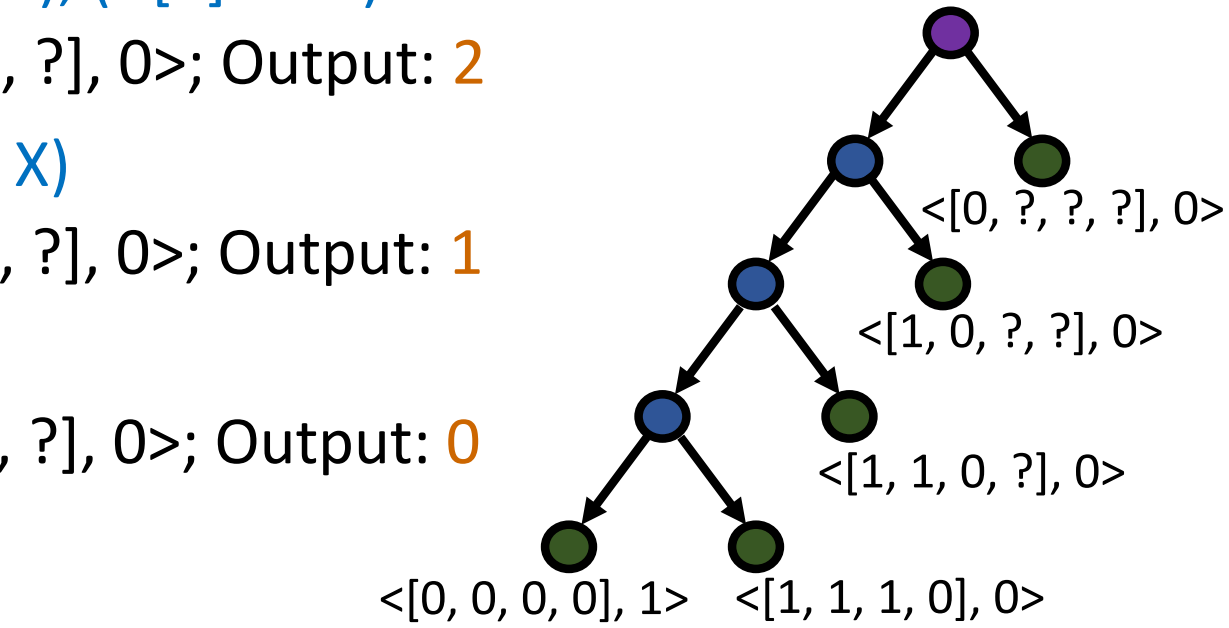
- Solution: <[1, 1, 0, ?], 0>; Output: 2

PC: (A[0] != X), (A[1] == X)

- Solution: <[1, 0, ?, ?], 0>; Output: 1

PC: (A[0] == X)

- Solution: <[0, ?, ?, ?], 0>; Output: 0



Further reading

James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7): 385-394, 1976

Programming Languages | B. Wegbreit Editor

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

This paper describes the symbolic execution of programs. Instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values. The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols. The difficult, yet interesting issues arise during the symbolic execution of conditional branch type statements. A particular system called EFFIGY which provides symbolic execution for program testing and debugging is also described. It interpretively executes programs written in a simple PL/I style programming language. It includes many standard debugging features, the ability to manage and to prove things about symbolic expressions, a simple program testing manager, and a program verifier. A brief discussion of the relationship between symbolic execution and program proving is also included.

1. Introduction

The large-scale production of reliable programs is one of the fundamental requirements for applying computers to today's challenging problems. Several techniques are used in practice; others are the focus of current research. The work reported in this paper is directed at assuring that a program meets its requirements even when formal specifications are not given. The current technology in this area is basically a testing technology. That is, some small sample of the data that a program is expected to handle is presented to the program. If the program is judged to produce correct results for the sample, it is assumed to be correct. Much current work [11] focuses on the question of how to choose this sample.

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.

Program testing and program proving can be considered as extreme alternatives. While testing, a programmer can be assured that sample test runs work correctly by carefully checking the results. The correct execution for inputs not in the sample is still in doubt. Alternatively, in program proving the programmer formally proves that the program meets its specification for all executions without being required to execute the program at all. To do this he gives a precise specification of the correct program behavior and then follows a formal proof procedure to show that the program and the specification are consistent. The confidence in this method hinges on the care and accuracy employed in both the creation of the specification and in the construction of the proof steps, as well as on the attention to machine dependent issues such as overflow, rounding

... symbolic execution for *testing programs* is a more exploitable technique in the short term than the more general one of program verification.

James King
CACM 19:7, 1976

Two current tools for symbolic execution
from academia

Java

- Symbolic PathFinder
  - https://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc
  - Based on Java PathFinder
    - https://github.com/javapathfinder

C

- KLEE
  - https://klee.github.io/
  - Based on LLVM compiler infrastructure

Exercise

List the path conditions for the following program, and provide a solution for each path condition

```
static boolean isSum2(int[] a, int sum) {
  for (int i = 0; i < a.length; i++) {
    for (int j = i + 1; j < a.length; j++) {
      if (a[i] + a[j] == sum) return true;
    }
  }
  return false;
}
```

Assume: `a != null && a.length == 3`

?/!