SEENG3320/6320: Software Verification and Validation

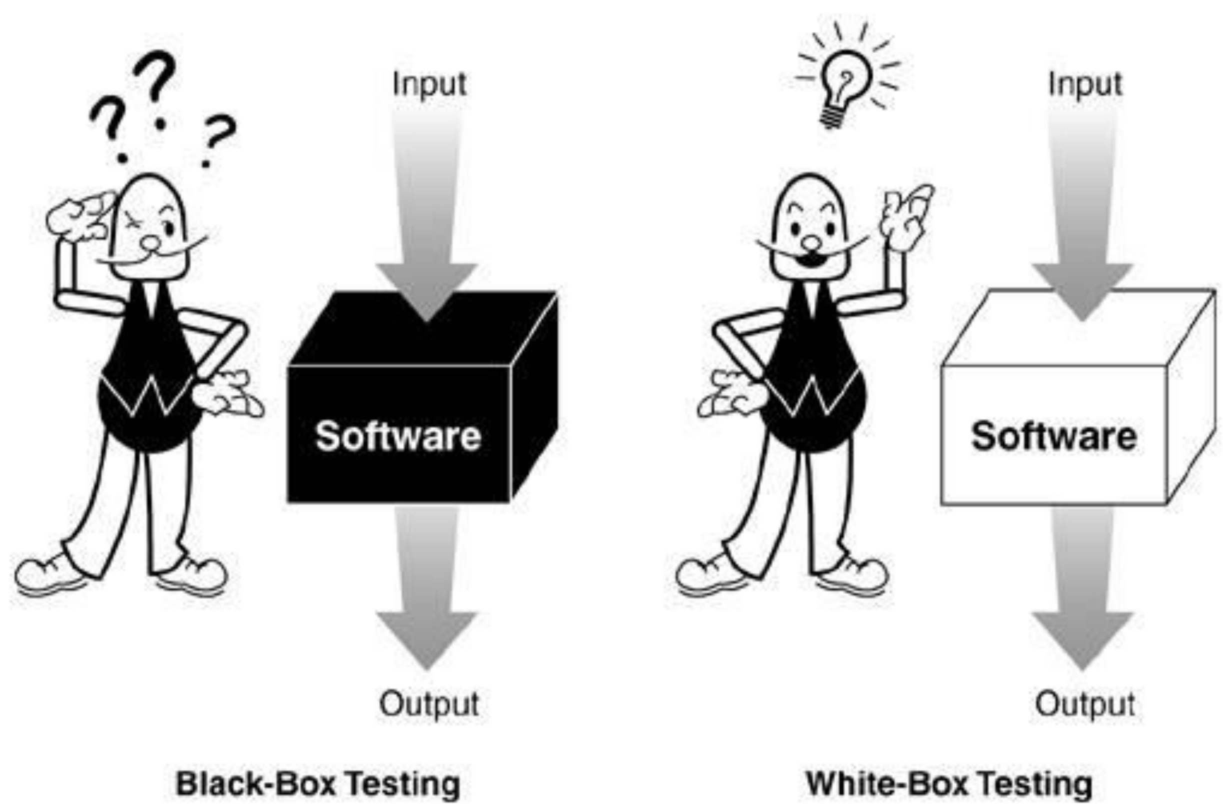
School of Electrical Engineering and Computing

Semester 1, 2020

Symbolic Execution

Types of test generation

- Black-box (functional) vs. white-box (structural) test generation
- Black-box test generation: Generating test cases based on the functionality of the software
- White-box test generation: Generating test cases based on the source-code structure of the program



Symbolic Execution

- **Symbolic execution** (also called **symbolic evaluation**) is a means of analyzing a program to determine what inputs cause each part of a program to execute.
- ’Execute’ programs with symbols: we track symbolic state rather than concrete input
- When ’execute’ one path, we actually simulate many test runs, since we are considering all the inputs that can exercise the same path.
- Firs proposed by James C King in 1975.

Concrete Execution Verse Symbolic Execution

During a concrete execution:

- The program would read a concrete input value (e.g., 5) and assign it to y.
- Execution would then proceed with the multiplication and the conditional branch.

During a symbolic execution:

- The program reads a symbolic value (e.g., `Y`) and assigns it to y.
- The program would then proceed with the multiplication and assign `Y * 2` to `z`. When reaching the `if` statement, it would evaluate `Y * 2 == 12`.

```
int f() {
  ...
  y = read();
  z = y * 2;
  if (z == 12) {
    fail();
  } else {
    printf("OK");
  }
}
```

Path Conditions:

```
Path 1: Y * 2 == 12
Path 2: Y * 2 != 12
```

Basic Concepts

- Consider the following code

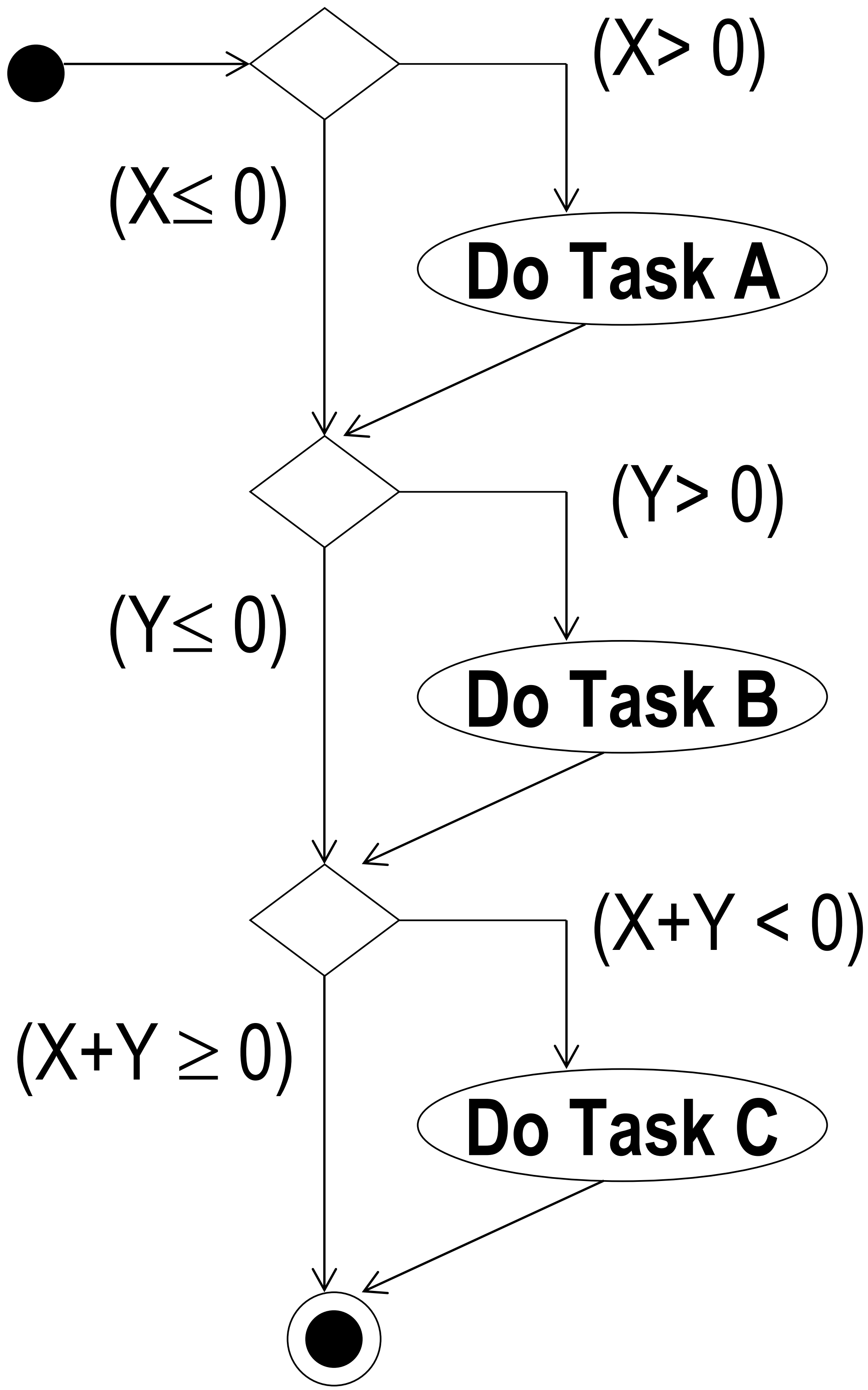
```
public void methodZ()
{
    INPUT 2 values of double type: X and Y
    Define Z = X+Y
    If (X> 0)      { /*do Task A*/ }
    If (Y>0)       { /*do Task B*/ }
    If (Z < 0)     { /*do Task C*/ }
}
```

Convert (**Z** < 0) to (**X**+**Y** <0).
This process is called "***predicate interpretation***" , which ensures that the **predicate** is made up by the **input variables**.

Basic Concepts (cont.)

- Path condition: conjunction of predicate interpretations along a path
- Executable (feasible) path
  - If a **path** can be executed by at least one **input**, the path is called executable path
- Infeasible path
  - The *path condition* can never be **satisfied**.
    - NO test cases can be generated for an infeasible path

Basic Concepts (cont.)



Path 1: $(X > 0)$ and $(Y>0)$ and $(X+Y < 0)$ **Infeasible**
Path 2: $(X > 0)$ and $(Y>0)$ and $(X+Y \geq 0)$ Feasible
Path 3: $(X > 0)$ and $(Y\leq 0)$ and $(X+Y < 0)$ Feasible
Path 4: $(X > 0)$ and $(Y\leq 0)$ and $(X+Y \geq 0)$ Feasible
Path 5: $(X\leq 0)$ and $(Y>0)$ and $(X+Y < 0)$ Feasible
Path 6: $(X\leq 0)$ and $(Y>0)$ and $(X+Y \geq 0)$ Feasible
Path 7: $(X\leq 0)$ and $(Y\leq 0)$ and $(X+Y < 0)$ Feasible
Path 8: $(X\leq 0)$ and $(Y\leq 0)$ and $(X+Y \geq 0)$ Feasible

7 paths are feasible because they can be executed by at least **ONE input**

Basic Concepts (cont.)

- We can refine the path conditions for Paths 2-8 as follows:

Path 2: $(X > 0)$ and $(Y > 0)$ and $(X+Y > 0)$	Feasible
Path 3: $(X > 0)$ and $(Y < 0)$ and $(X+Y < 0)$	Feasible
Path 4: $(X > 0)$ and $(Y \leq 0)$ and $(X+Y \geq 0)$	Feasible
Path 5: $(X < 0)$ and $(Y > 0)$ and $(X+Y < 0)$	Feasible
Path 6: $(X \leq 0)$ and $(Y > 0)$ and $(X+Y \geq 0)$	Feasible
Path 7: $(X \leq 0)$ and $(Y \leq 0)$ and $(X+Y < 0)$ and $(x \neq y = 0)$	Feasible
Path 8: $X = Y = 0$	Feasible

Basic Concepts (cont.)

- Input sub-domain
    - A set of inputs satisfying a path condition
- "**Searching** an input to execute a path" **is equivalent** to "**solving** the associated path condition"

Searching = Solving

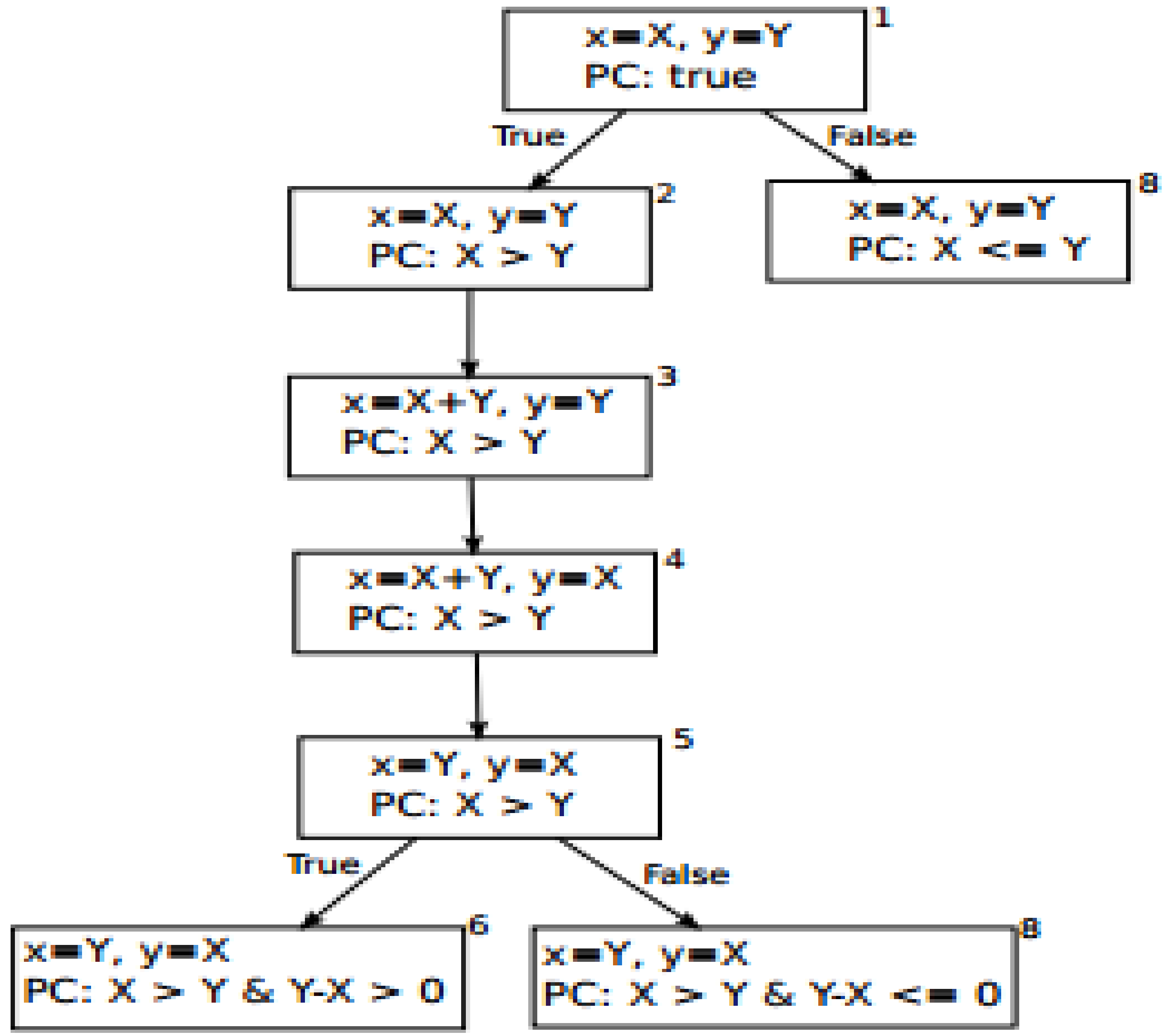
Test Case Generation based on Symbolic Execution

```
    int x, y;
1   if(x > y){
2     x = x+y;
3     y = x-y;
4     x = x-y;
5     if(x - y > 0)
6       assert false;
7   }
8   print(x, y)
```

(a)

Path	PC	Program Input
1,8	X <= Y	X=1 Y=1
1,2,3,4,5,8	X>Y & Y-X<=0	X=2 Y=1
1,2,3,4,5,6	X>Y & Y-X>0	none

(c)

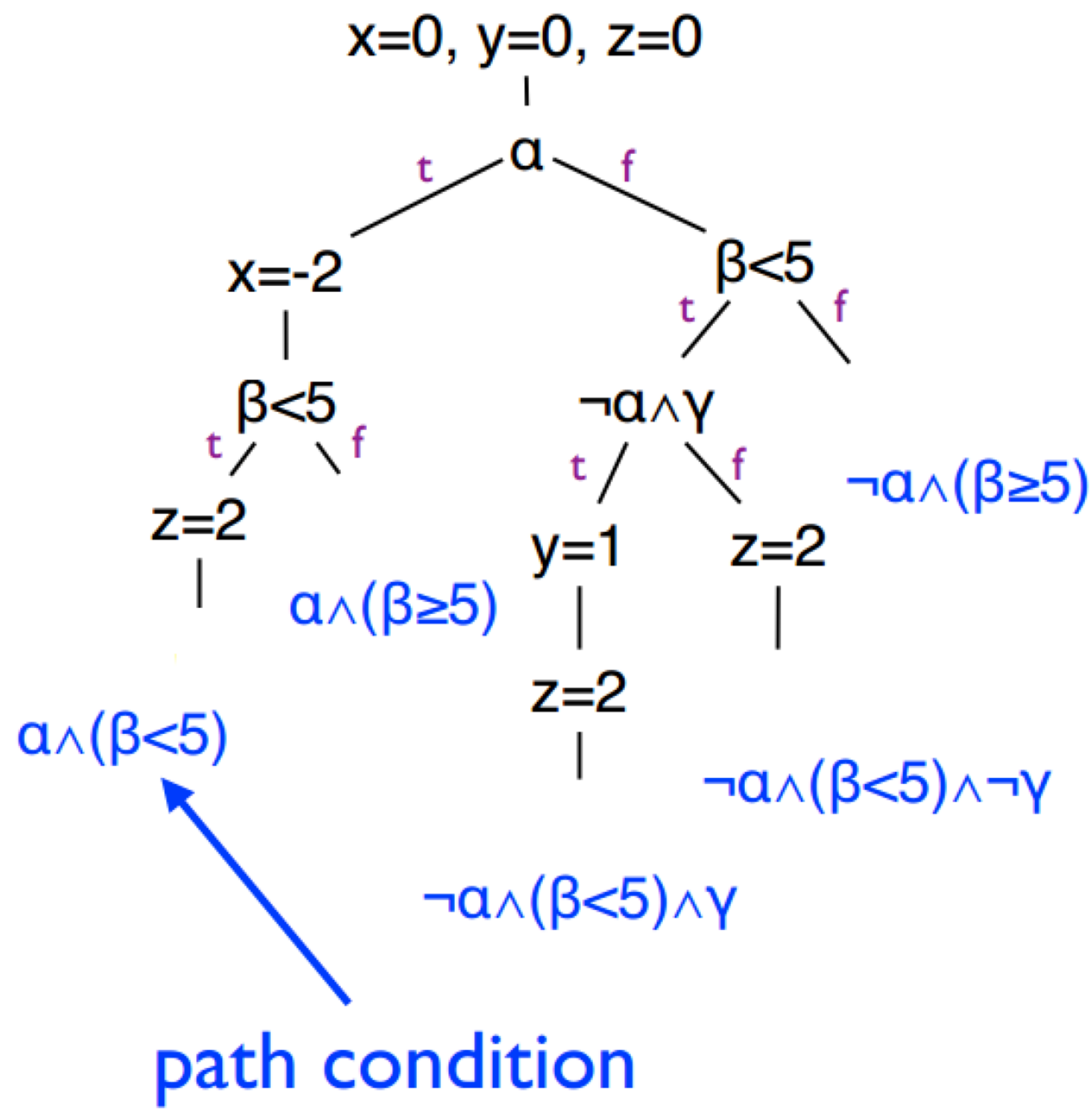


(b)

Test Case Generation based on Symbolic Execution

```
bool a= α;
int b = β; bool c = γ;
                    // symbolic
```

```
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



Test case 1: α = true; $\beta = 1$

Test case 2: α = true; $\beta = 6$

…

Example

`y = read();`	y=s, s is a symbolic variable for input
`p = 1;`	$p = 1, y = s$
`while(y < 10){`	$p = 1, y = s$
`    y = y + 1;`	$s<10, y = s + 1, p = 1$
`    if y >2`	
`      p = p + 1;`	$2 < s + 1< 10, y = s + 1, p = 2$
`    else`	
`      p = p + 2;`	$s + 1<=2, y = s + 1, p = 3$
`}`	
`print (p);`	

Quiz

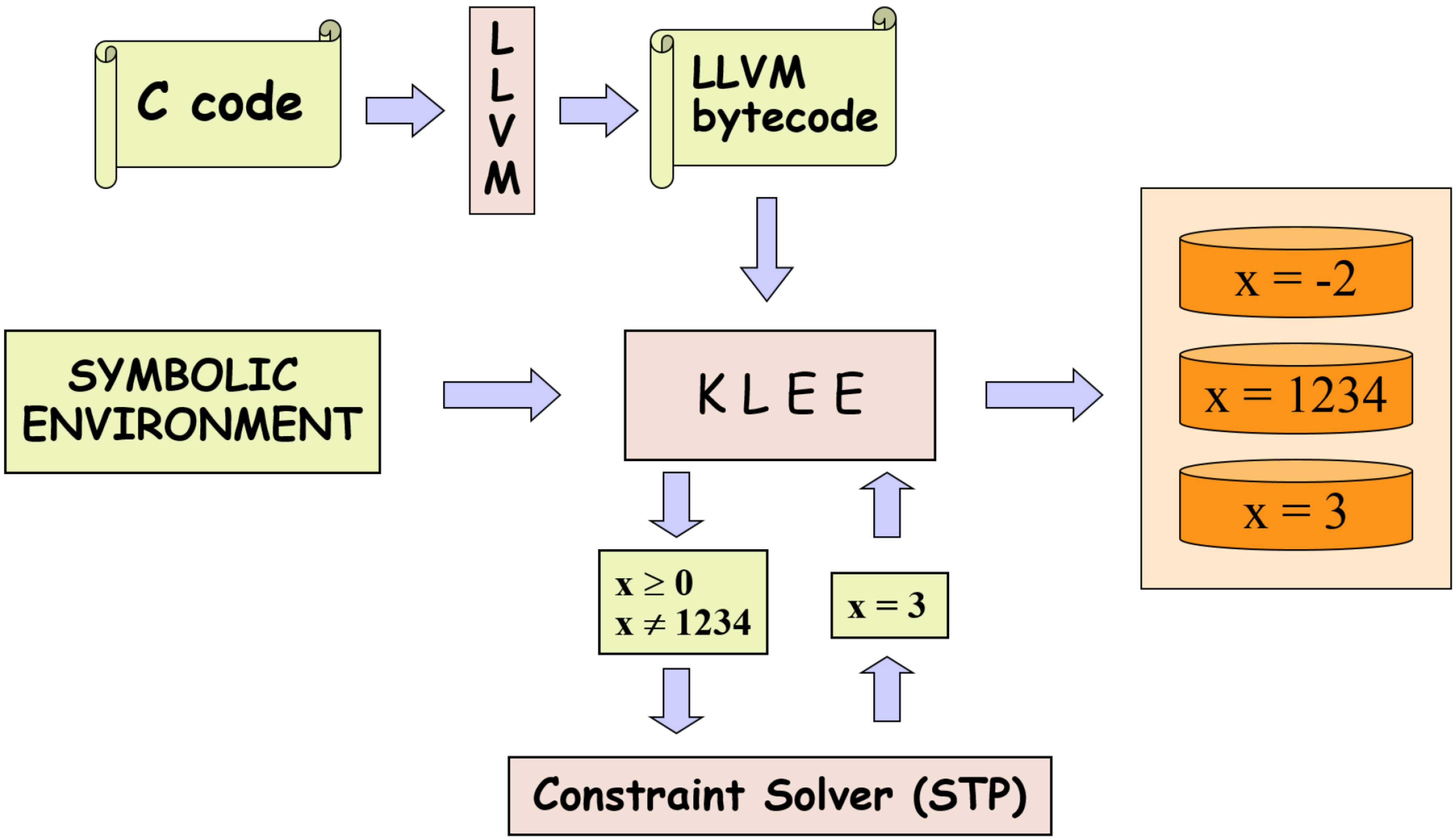
```
int foo(int i){
        int j = 2*i;
        i = i++;
        i = i * j;
        if ( i < 1 )
                i =   -i;
        return i;
}
```

- Perform symbolic execution of this code:
  - What are the path conditions ?
  - Generate test cases for each path

Tool Support

- Java：
  - Java PathFinder: http://babelfish.arc.nasa.gov/trac/jpf
  - JCUTE: https://github.com/osl/jcute
- C：
  - CUTE/S2E/Crest
  - KLEE: http://klee.github.io/
- .Net:
  - Pex: http://research.microsoft.com/en-us/projects/pex/
- ……

KLEE



- Using the LLVM compiler infrastructure to compile C code to bytecode.
- Modelling program environment.
- Using STP, an efficient constraint solver, to solve path conditions.

Constraint Solver

SAT: find an assignment to a set of Boolean variables that makes the Boolean formula true

Complexity: NP-Complete

https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

SMT (Satisfiability Modulo Theories) = SAT++
An SMT formula is a Boolean combination of formulas over first-order theories

https://en.wikipedia.org/wiki/Satisfiability_modulo_theories

Problems of static symbolic execution

- Path explosion
  - Remember n branches will cause 2^n paths
  - Infinite paths for unbounded loops
  - Calculate constraints on all paths is infeasible for real software
- Too complex constraint
  - The constraint gets very complex for large programs
  - Not to mention the resolving part is NPC

Further Reading

- R. G. Hamlet, “Testing programs with the aid of a compiler,” TSE, vol. 3, 1977.
- R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Hints on test data selection: Help for the practicing programmer,” Computer, vol. 11, 1978.
- A. J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf, “An experimental determination of sufficient mutation operators,” ACM TOSEM, vol. 5, no. 2, pp. 99–118, 1996.
- James C. King, Symbolic execution and program testing, Communications of the ACM, volume 19, number 7, 1976, 385—394.
- Cadar, Cristian; Dunbar, Daniel; Engler, Dawson, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs". Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. OSDI'08. Berkeley, CA, USA, USENIX Association, pp. 209–224.

Thanks!


Testing
Software
development
design
articles
management
test
quality
code
web
use
good
risk
usability
Agile
People
Project
information
article
system
check