软件分析与架构设计

符号执行

何冬杰

重庆大学

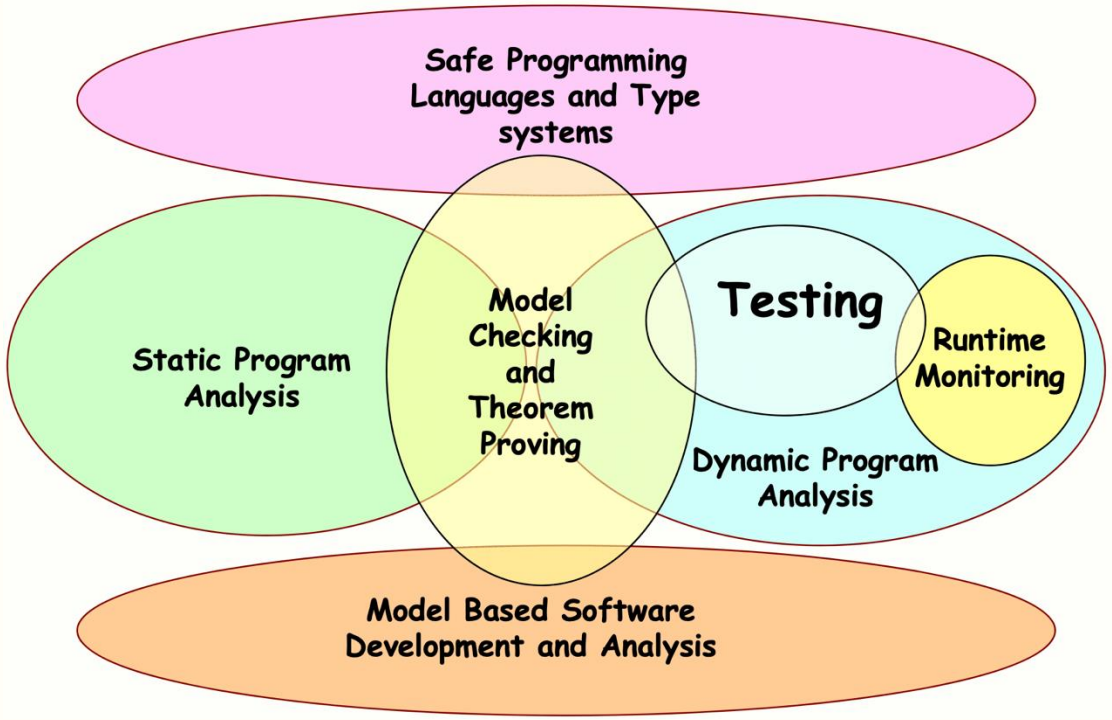
软件质量保障（QA）

❑Today, QA is mostly testing

> “50% of my company employees are testers, and the rest spends 50% of their time testing!” —— Bill Gates 1995

- ➢Still true today (2025)

❑Big Picture of QA



Program testing

- **Program testing vs verification**
  - Verification of assert(Q): $\forall x: P \Rightarrow Q$
  - Bug-finding of assert(Q): $\exists x: P \wedge \neg Q$
- **Bugs and Reachability**
  - Convert error case into reachability problem
    - assert(p) -> if(!p) ERROR;
    - *x -> if(x == NULL) { ERROR; } return *x;
    - a[i] ->if(i < 0 || i >a.length) {ERROR; } return a[i];
  - "Bug finding" is now just about finding inputs that execute every program path
- **Random testing**
- **符号执行：可以显著提高测试覆盖率**

符号执行（Symbolic Execution)

- Reason about behavior of program by “executing” it with symbolic values
  - 每次分析一条路径，按某种顺序遍历路径
- Originally proposed by James King (1976, CACM) and Lori Clarke (1976, IEEE TSE)
- Became practical around 2005
  - because of advances in constraint solving (SMT solvers)

符号执行举例

```
01 int x = y = z = 0;
02 if (a) {
03   x = -2;
04 }
05 if (b < 5) {
06   if (!a && c) {
07     y = 1;
08   }
09   z = 2;
10 }
11 assert(x + y + z != 3);
```

Concrete execution

a = b = c = 1

x = y = z = 0

true

x = -2

true

false

z = 2

-2 + 0 + 2 ≠ 3 ✓

Symbolic execution on a path

a = a0, b=b0,c=c0
x = y = z = 0
true
x = -2
false
-2+0+0 != 3

Symbolic Values and Path Conditions

- **Symbolic values:**
  - Unknown values, e.g., user inputs, are kept symbolically
  - Symbolic state maps variables to symbolic values
- **Path Conditions:**
  - Quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far

Symbolic input values: x_0 , y_0

Symbolic state:

$z = x_0 + y_0$

```
function f(x, y) {
  z = x + y;
  if (z > 0) {
    ...
  }
}
```

Path condition:

$x_0 + y_0 > 0$

Satisfiability of Formulas

- **Determine whether a path is feasible:**
  - Check if its path condition is satisfiable
- **Done by powerful SMT/SAT solvers**
  - SAT = satisfiability, SMT = satisfiability modulo theory
  - E.g., Z3, Yices, STP
  - For a satisfiable formula, solvers also provide a concrete solution
  - Examples:
    - a0 + b0 > 1: Satisfiable, one solution: a0 = 1, b0 = 1
    - (a0 + b0 < 0) Λ (a0 - 1> 5) Λ (b0 > 0): Unsatisfiable

Formalizing Symbolic Execution

- **Guards:**
  - $\mathrm{g} ::= \text{true} \mid \text{false} \mid \text{not g} \mid g_1\ op_b\ g_2 \mid a_{s1}\ op_r\ a_{s2}$
- **Symbolic Formulas** $\sum \in \mathrm{Var} \rightarrow \boldsymbol{a_s}$
  - $a_s ::= \alpha \mid \mathrm{n} \mid a_{s1}\ op_r\ a_{s2}$
- **Symbolic Evaluation of Expressions and Statements**

Expr

$$\frac{}{\langle \Sigma, n \rangle \Downarrow n}\ \textit{big-int}$$

$$\frac{}{\langle \Sigma, x \rangle \Downarrow \Sigma(x)}\ \textit{big-var}$$

$$\frac{\langle \Sigma, a_1 \rangle \Downarrow a_{s1} \quad \langle \Sigma, a_2 \rangle \Downarrow a_{s2}}{\langle \Sigma, a_1 + a_2 \rangle \Downarrow a_{s1} + a_{s2}}\ \textit{big-add}$$

Stmt

$$\frac{}{\langle g, \Sigma, \texttt{skip} \rangle \Downarrow \langle g, \Sigma \rangle}\ \textit{big-skip}$$

$$\frac{\langle g, \Sigma, s_1 \rangle \Downarrow \langle g', \Sigma' \rangle \quad \langle g', \Sigma', s_2 \rangle \Downarrow \langle g'', \Sigma'' \rangle}{\langle g, \Sigma, s_1; s_2 \rangle \Downarrow \langle g'', \Sigma'' \rangle}\ \textit{big-seq}$$

$$\frac{\langle \Sigma, a \rangle \Downarrow a_s}{\langle g, \Sigma, x := a \rangle \Downarrow \langle g, \Sigma[x \mapsto a_s] \rangle}\ \textit{big-assign}$$

Formalizing Symbolic Execution

❑ Branching

$$\frac{\langle \Sigma, b\rangle \Downarrow g' \quad g \wedge g' \mathtt{SAT} \quad \langle g \wedge g', \Sigma, s_1\rangle \Downarrow \langle g'', \Sigma'\rangle}{\langle g, \Sigma, \mathtt{if}\ b\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2, \rangle \Downarrow \langle g'', \Sigma'\rangle}\ \textit{big-iftrue}$$

$$\frac{\langle \Sigma, b\rangle \Downarrow g' \quad g \wedge \neg g' \mathtt{SAT} \quad \langle g \wedge \neg g', \Sigma, s_2\rangle \Downarrow \langle g'', \Sigma'\rangle}{\langle g, \Sigma, \mathtt{if}\ b\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2, \rangle \Downarrow \langle g'', \Sigma'\rangle}\ \textit{big-iffalse}$$

❑Loops

$$\frac{\langle \Sigma, b\rangle \Downarrow g' \quad g \wedge g' \mathtt{SAT} \quad \langle g \wedge g', \Sigma, s; \mathtt{while}\ b\ \mathtt{do}\ s\rangle \Downarrow \langle g'', \Sigma'\rangle}{\langle g, \Sigma, \mathtt{while}\ b\ \mathtt{do}\ s, \rangle \Downarrow \langle g'', \Sigma'\rangle}\ \textit{big-whiletrue}$$

Q. What's wrong here?

$$\frac{\langle \Sigma, b\rangle \Downarrow g' \quad g \wedge \neg g' \mathtt{SAT}}{\langle g, \Sigma, \mathtt{while}\ b\ \mathtt{do}\ s, \rangle \Downarrow \langle g \wedge \neg g', \Sigma\rangle}\ \textit{big-whilefalse}$$

Symbolic Execution of Loops

- **Loop invariants can be used if given**
- **But we can choose to explore only partial set of paths**
  - k-bounded loops (often: k < 3)
  - "Unsound" for verification
  - Sound but "Incomplete" for bug finding
    - formulas for a given path can be solved to find a witness = test input

$$\frac{k > 0 \quad \langle \Sigma, b \rangle \Downarrow g' \quad g \wedge g' \mathtt{SAT} \quad \langle g \wedge g', \Sigma, s; \mathtt{while_{k-1}}\ b\ \mathtt{do}\ s \rangle \Downarrow \langle g'', \Sigma' \rangle}{\langle g, \Sigma, \mathtt{while_k}\ b\ \mathtt{do}\ s, \rangle \Downarrow \langle g'', \Sigma' \rangle}\ \textit{big-whiletrue}$$

$$\frac{\langle \Sigma, b \rangle \Downarrow g' \quad g \wedge \neg g' \mathtt{SAT}}{\langle g, \Sigma, \mathtt{while_k}\ b\ \mathtt{do}\ s, \rangle \Downarrow \langle g \wedge \neg g', \Sigma \rangle}\ \textit{big-whilefalse}$$

Symbolic Execution: A Generalization of Testing

- What input values of a,b,c will cause the assert to fail?

```
01 int x = y = z = 0;
02 if (a) {
03   x = -2;
04 }
05 if (b < 5) {
06   if (!a && c) {
07     y = 1;
08   }
09   z = 2;
10 }
11 assert(x + y + z != 3);
```

line	g	E

Symbolic Execution: A Generalization of Testing

- What input values of a,b,c will cause the assert to fail?

```
01 int x = y = z = 0;
02 if (a) {
03   x = -2;
04 }
05 if (b < 5) {
06   if (!a && c) {
07     y = 1;
08   }
09   z = 2;
10 }
11 assert(x + y + z != 3);
```

line	g	E
0	true	$a \mapsto \alpha, b \mapsto \beta, c \mapsto \gamma$
1	true	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
2	$\neg\alpha$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
5	$\neg\alpha \wedge \beta \geqslant 5$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
9	$\neg\alpha \wedge \beta \geqslant 5 \wedge 0+0+0 \neq 3$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$

Symbolic Execution: A Generalization of Testing

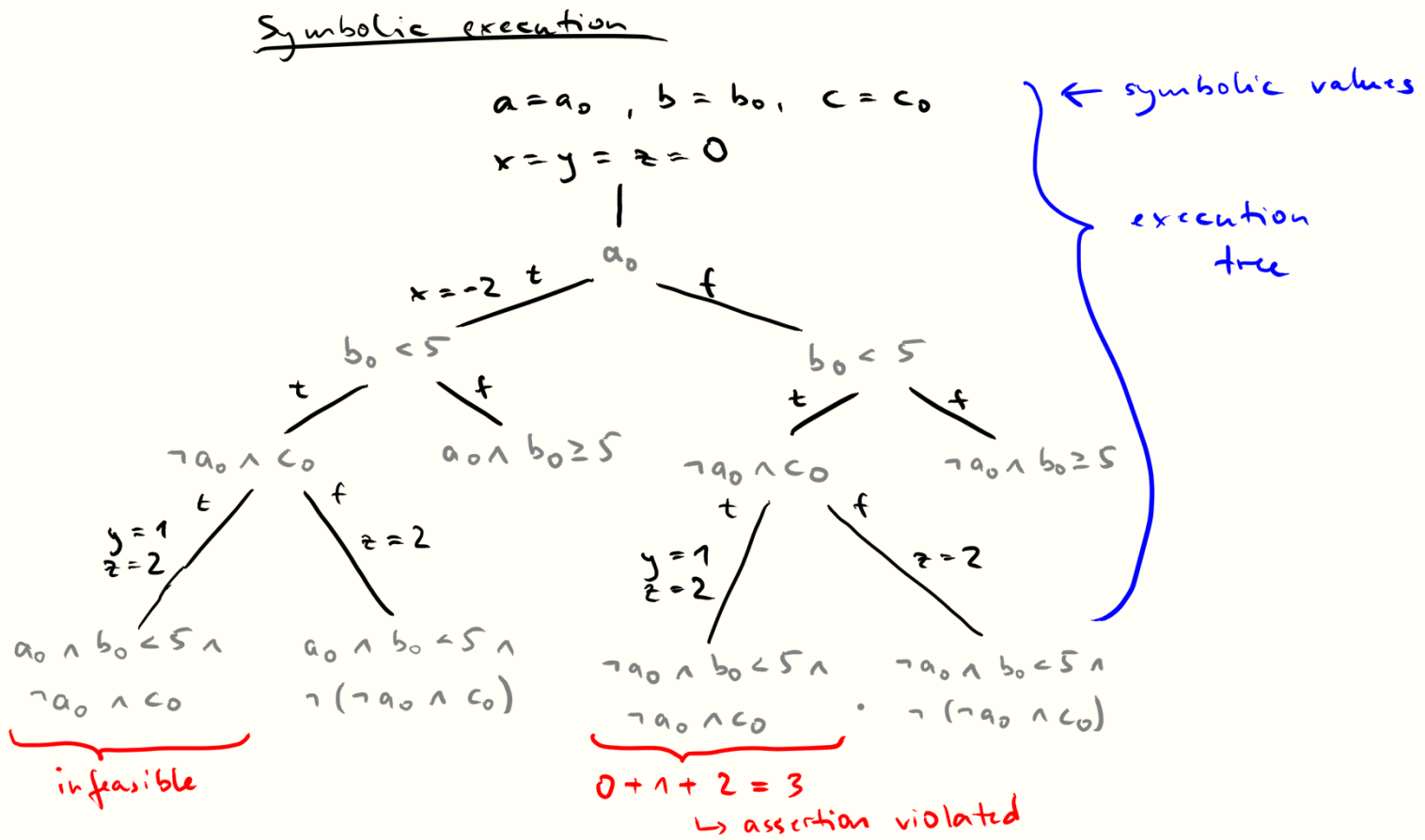
- What input values of a,b,c will cause the assert to fail?

```
01 int x = y = z = 0;
02 if (a) {
03   x = -2;
04 }
05 if (b < 5) {
06   if (!a && c) {
07     y = 1;
08   }
09   z = 2;
10 }
11 assert(x + y + z != 3);
```

line	g	E
0	true	$a \mapsto \alpha, b \mapsto \beta, c \mapsto \gamma$
1	true	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
2	$\neg\alpha$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
5	$\neg\alpha \wedge \beta < 5$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
6	$\neg\alpha \wedge \beta < 5 \wedge \gamma$	$\ldots, x \mapsto 0, y \mapsto 1, z \mapsto 0$
6	$\neg\alpha \wedge \beta < 5 \wedge \gamma$	$\ldots, x \mapsto 0, y \mapsto 1, z \mapsto 2$
9	$\neg\alpha \wedge \beta < 5 \wedge \neg(0 + 1 + 2 \neq 3)$	$\ldots, x \mapsto 0, y \mapsto 1, z \mapsto 2$

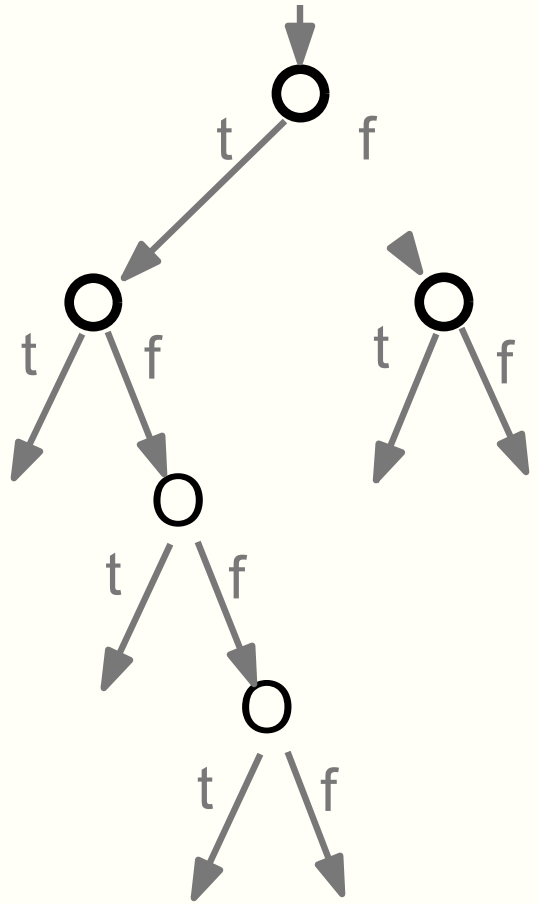
Execution Tree

```
01 int x = y = z = 0;
02 if (a) {
03   x = -2;
04 }
05 if (b < 5) {
06   if (!a && c) {
07     y = 1;
08   }
09   z = 2;
10 }
11 assert(x + y + z != 3);
```



Execution Tree

- **All possible execution paths**
  - Binary tree
  - Nodes: Conditional statements
  - Edges: Execution of sequence on non-conditional statements
  - Each path in the tree represents an equivalence class of inputs



符号执行

- **状态：**
  - 当前变量的符号值；下一条待执行的语句；当前状态的路径约束
- **不断遍历状态**
  - 遇上结束状态时用SMT求解器判断规约是否被满足
  - 不满足规约，表示当前路径不满足规约
    - 程序不满足规约=任意路径不满足规约
    - 程序满足规约=所有路径满足规约
- **符号执行**
  - 符号执行迭代过程中的任意状态表示了沿着某条路径执行特定步数之后的任意精确执行踪迹集合，是所有执行的下近似
  - 路径无穷多时符号执行做不到验证的目的
  - 通常遍历有限次循环来模拟

符号执行时的路径规约求解时机

- **无需对不可达路径继续探索或验证**
- **Eager evaluation：**
  - 在分支的时候就判断路径的可达性
  - 对同一条路径可能调用更多次，但探索的路径总数会减少
- **Lazy evaluation：**
  - 只对完整路径判断，和之前的方法等价
  - 如果路径不可达，lazy evaluation的约束不会更简单，但有可能更容易产生冲突，导致解得更快

Applications of Symbolic Execution

- **General goal: Reason about behavior of program**
- **Basic applications**
  - Detect infeasible paths
  - Generate test inputs
  - Find bugs and vulnerabilities
- **Advanced applications**
  - Generating program invariants
  - Prove that two pieces of code are equivalent
  - Debugging
  - Automated program repair

Problems of Symbolic Execution

- **Loops and recursion: Infinite execution trees**
- **Path explosion:**
  - Number of paths is exponential in the number of conditionals
- **Environment modeling:**
  - Dealing with native/system/library calls "if (file.read()==x)"
- **Solver limitations:**
  - Dealing with complex path conditions
    - "$x^5 + 3x^3 == y$"
    - "p->next->value == x"
- **Heap modeling:**
  - Symbolic representation of data structures and pointers

Examples

Consider: What if we could not (or did not want to) analyze double as if it is an external function?

```
1  int double (int v) {
2      return 2*v;
3  }
4
5  void bar(int x, int y) {
6      z = double (y);
7      if (z == x) {
8          if (x > y+10) {
9              ERROR;
10         }
11     }
12 }
```

Consider: What if our solver cannot handle non-linear arithmetic or modulo?

```
1  int foo(int v) {
2      return v*v%50;
3  }
4
5  void baz(int x, int y) {
6      z = foo(y);
7      if (z == x) {
8          if (x > y+10) {
9              ERROR;
10         }
11     }
12 }
```

Concolic Execution

❑Concolic Execution:

- ➢Concrete execution + symbolic execution
- ➢Perform concrete and symbolic execution side-by-side
- ➢Gather path constraints while program executes
- ➢After one execution, negate one decision, and re-execute with new input that triggers another path

1. Instrument program to collect path constraints during concrete execution
2. Run program with concrete inputs (initially random) to collect path constraint g
3. Negate last clause in g and solve for model
4. If SAT, then get satisfying assignment as new input and repeat from 2
5. If UNSAT, then pop off last clause and repeat from 3

❑Concolic testing:

- ➢Use concolic execution for program testing

Concolic Execution: Example 1

```
1   int double (int v) {
2       return 2*v;
3   }
4
5   void bar(int x, int y) {
6       z = double (y);
7       if (z == x) {
8           if (x > y+10) {
9               ERROR;
10          }
11      }
12  }
```

1. Input: x=0, y=1
   - Path: (2*y != x)
   - Next: (2*y == x) :: **SAT**
2. Input: x=2, y=1
   - Path: (2*y == x) && (x <= y+10)
   - Next: (2*y == x) && (x > y+10) :: **SAT**
3. Input: x=22, y=11
   - Path: (2*y == x) && (x > y+10)
   - **Bug found!!**

Concolic Execution: Example 2

```
1   int foo(int v) {
2       return v*v%50;
3   }
4
5   void baz(int x, int y) {
6       z = foo(y);
7       if (z == x) {
8           if (x > y+10) {
9               ERROR;
10          }
11      }
12  }
```

1. Input: x=22, y=7
   - Path: (49 != x). // y*y%50 = 49%50 = 49
   - Next: (49 == x) :: **SAT**
2. Input: x=49, y=7
   - Path: (49 == x) && (x > y+10)
   - **Bug found!!**

Concolic Execution

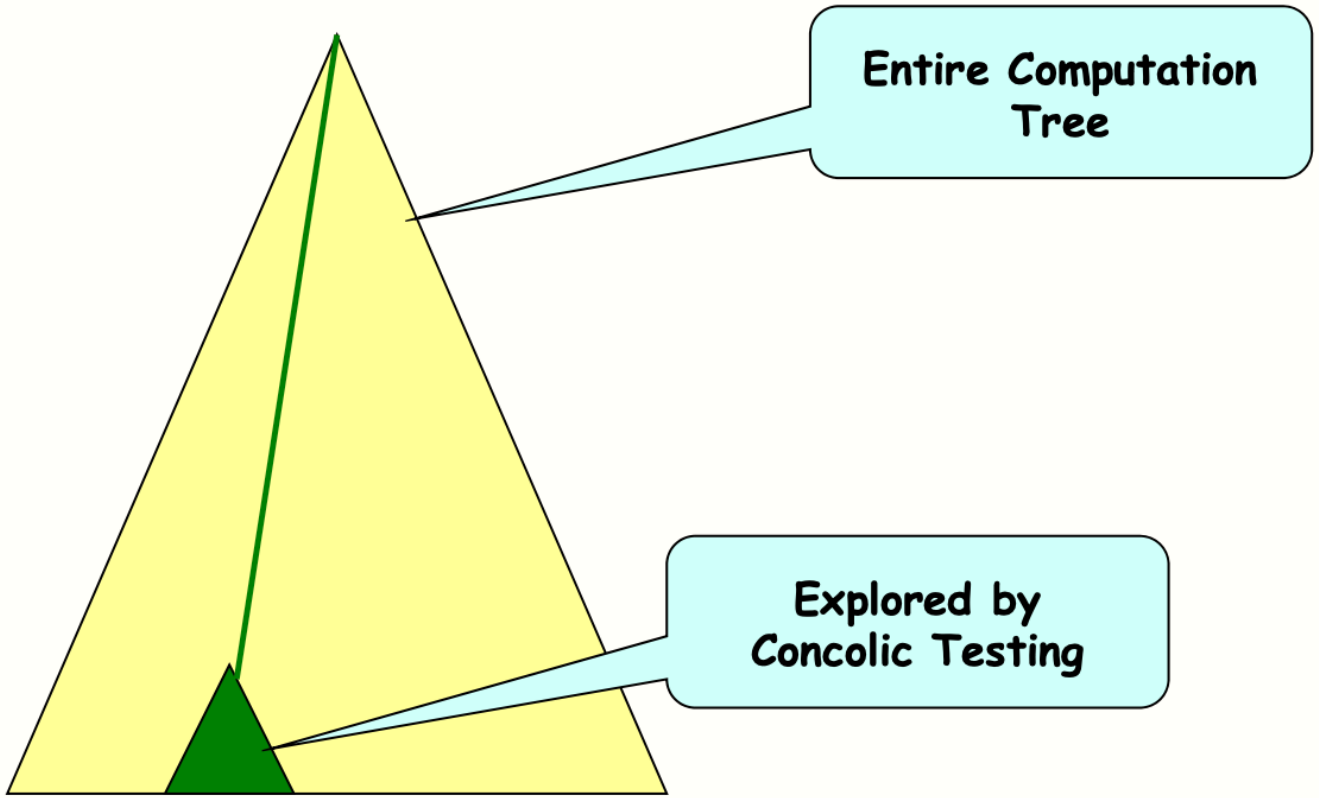
- **Key advantage: Always have a concrete input in parallel**
  - When constraint cannot be modeled (e.g. external function, features not handled by solver), replace with concrete value.
- **Soundness:**
  - Concrete replacement is a true under-approximation
  - Unsound Example:

```
1  int foo(int v) {
2      return v*v%50;
3  }
4
5  void baz(int x, int y) {
6      z = foo(y);
7      if (z == x) {
8          if (x > y+10) {
9              ERROR;
10         }
11     }
12 }
```

1. Input: x=0, y=8
   - Path: (14 != x) // y*y%50 = 64%50 = 14
   - Next: (14 == x) :: **SAT**
2. Input: x=14, y=8
   - Path: (14 == x) && (x <= y+10)
   - Next: (14 == x) && (x > y+10) :: **SAT**
3. Input: x=14, y=2
   - Path: (4 != x)
   - **Unsoundness!**

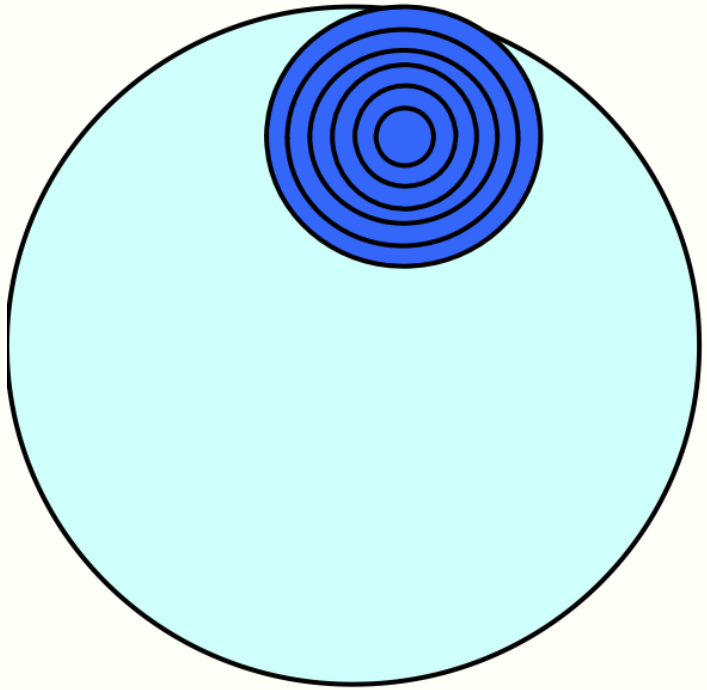
Concolic Execution: Limitations

- **Path Space of a Large Program is Huge**
  - Path Explosion Problem

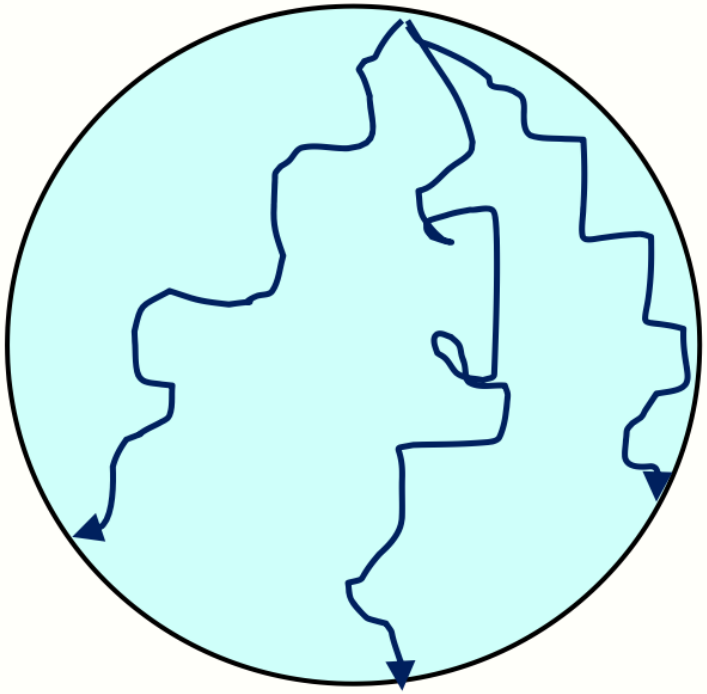


Concolic Execution: Limitations

- Concolic Testing vs Random Testing



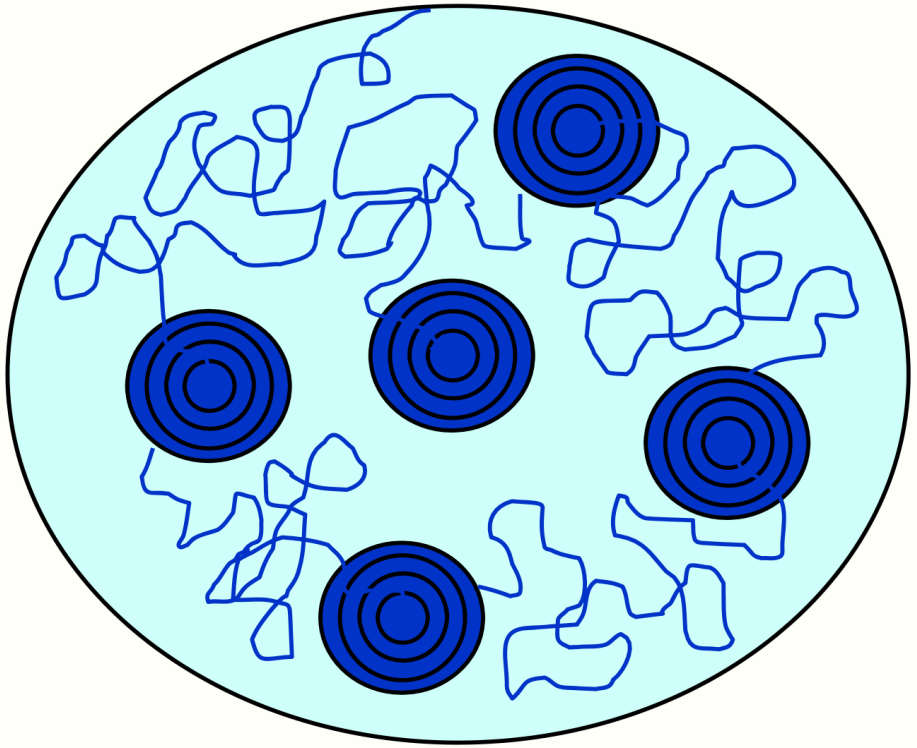
Concolic: Broad, shallow



Random: Narrow, deep

Hybrid Concolic Testing

- **Interleave Random and Concolic Testing**
  - 4X more coverage than random testing
  - 2X more coverage than concolic testing



Deep, broad search

```
while (not required coverage) {
    while (not saturation)
        perform random testing;
    Checkpoint;
    while (not increase in coverage)
        perform concolic testing;
    Restore;
}
```

Concolic Execution：Summary

- **替换变量为具体值的方法不保证完整性**
  - 可满足的约束可能变得不可满足
  - (a>b+10)Λ（b*b)%50=a中，如果b=0，则不可满足
- **但效果一定优于静态符号执行**
  - 替换只在原约束无法求解的情况下进行
- **为什么将具体值代入执行而不是在约束无法求解的时候替换为随机值?**
  - 因为有可能拿不到完整的路径约束
  - 比如路径约束有可能是foo(x)==0，但foo的实现代码拿不到，foo也不能脱离系统状态单独调用

(Optional) 作业：绘制execution tree

- Draw the execution tree for this function. How many nodes and edges does it have?

```
function f(x,y) {
  var s = "foo";
  if (x < y) {
    s += "bar";
    print s;
  }
  if (y === 23) {
    print s;
  }
}
```

(Optional)作业：符号执行工具使用

- **Popular Symbolic/Concolic Tools**
  - DART (Directed Automated Random Testing)
  - CUTE (Concolic Unit Testing Engine)
  - KLEE (“dynamic symbolic execution”), C语言
  - SAGE (Scalable, Automated, Guided Execution aka “whitebox fuzzing”)
  - Java PathFinder, SymbolicPathFinder, JBSE
  - Angr
  - PyExZ3
  - Jalangi
  - …
- **选择任意一款符号执行工具，学习使用，并撰写使用报告**