

基于约束的测试

Constraints Based Testing

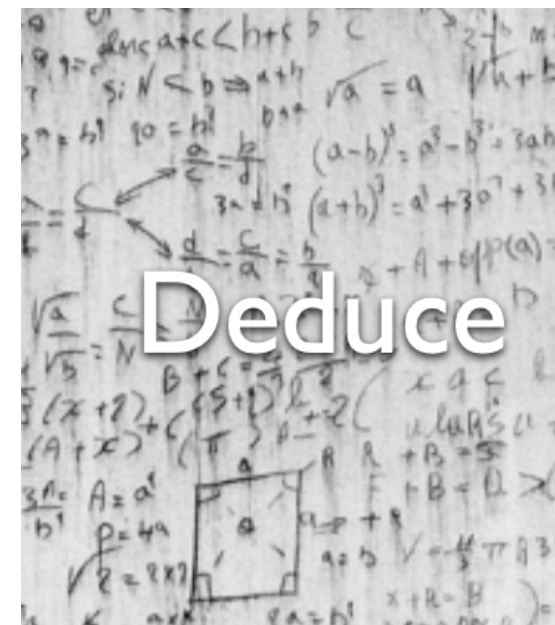
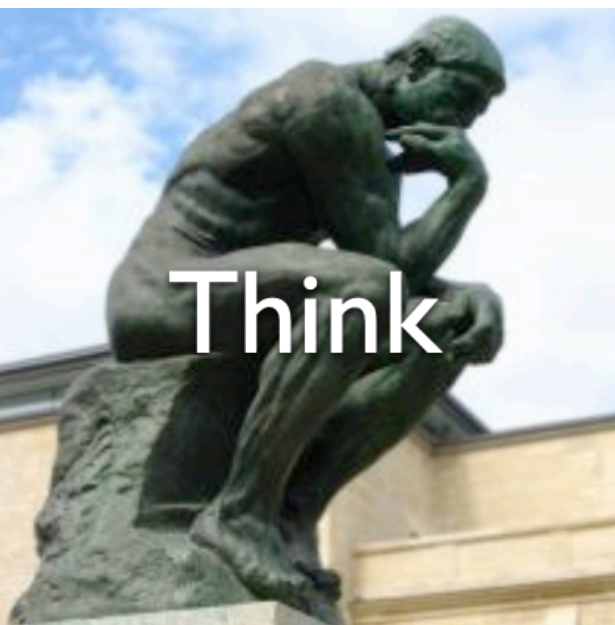
实现代码结构覆盖

给定一个待覆盖语句，如何设计一个测试用例使其恰好能执行到该语句？

```
# suppose x and y are in the range of [0, 99]
void function(int x, int y) {
    int z = 2 * y;
    if (x == z) {
        y = y + 2;
        if (x > y + 10) {
            assert(false);
        }
    }
}
```

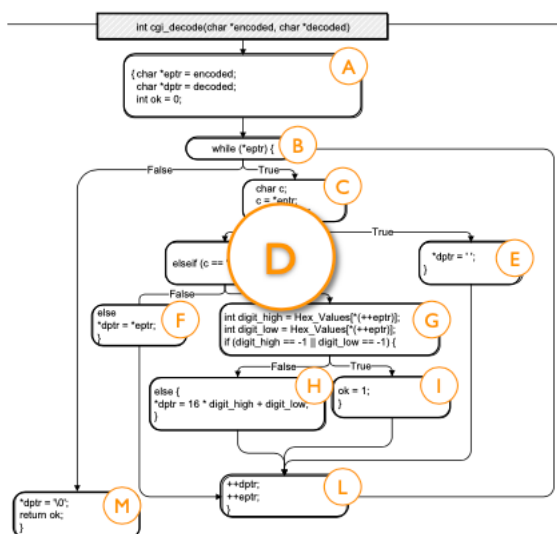
实现代码结构覆盖

给定一个待覆盖语句，如何设计一个测试用例使其恰好能执行到该语句？



基于约束的测试

将待测程序上的一个测试目标转换为某种约束表示形式 (Constraint Formula), 然后利用约束求解器 (Constraints Solver) 来自动化生成测试数据



Program and
Test Goal

$$(a > b \wedge c = 2) \vee (a > 5 \wedge b > a) \vee (a < b \wedge b < 5 \wedge a > 0)$$

Constraint
System



Constraint
Solver

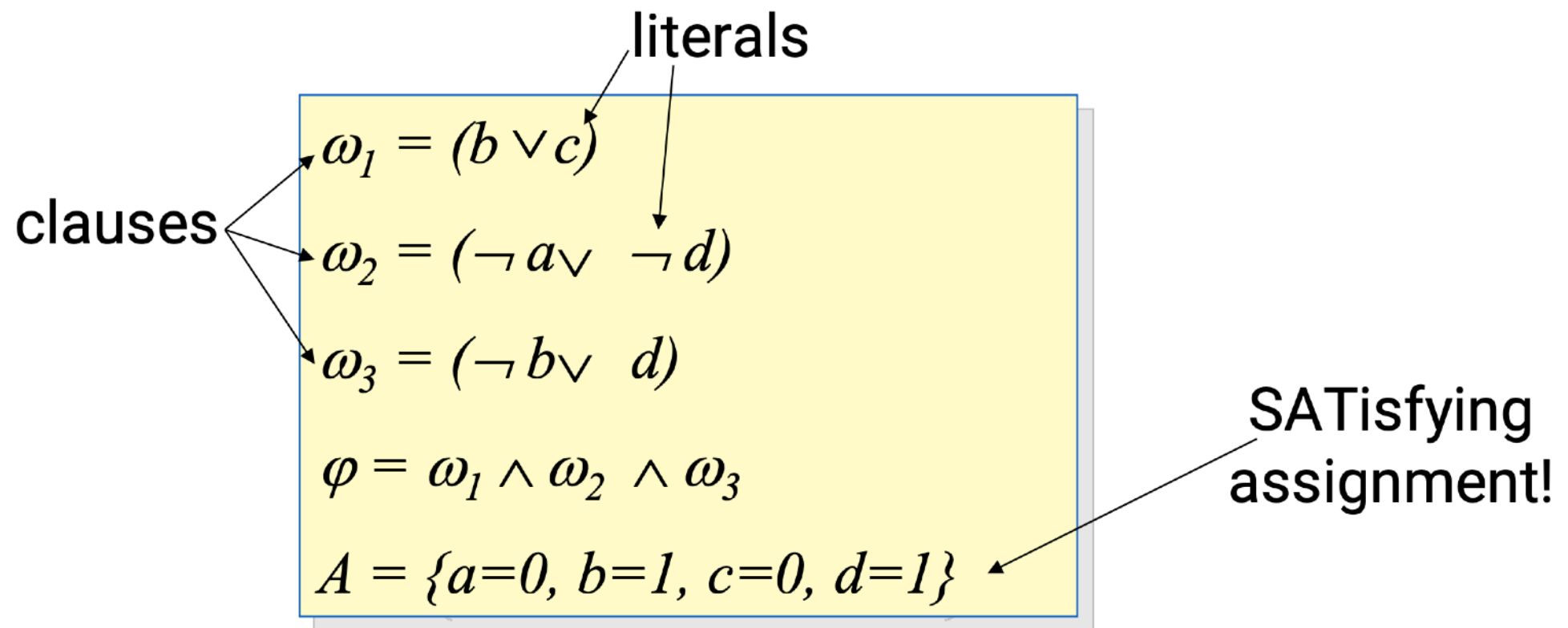
a = 1
b = 2
c = 3

Test
Data

SAT

Boolean Satisfiability Problem

给定一个 CNF 形式的命题公式，判断是否存在一组布尔变量赋值，
以使该公式为真 (Truth Assignment)



SMT

Satisfiability Modulo Theories

给定一个关于某个背景理论 (Background Theories) 的一阶逻辑公式，判断是否存在一组变量赋值，以使该公式为真 (Truth Assignment)

- 可以处理 Linear Integer/Real Arithmetic、Bit Vectors、Array 等 “理论”
- 一种比 SAT 更丰富的建模语言

$$x > 5 \wedge y \neq 3 \wedge z = 18$$

$$(x + y \leq 0) \wedge (x = 0) \wedge (\neg a \vee y = 0)$$

Symbolic Execution

使用符号值来表示程序变量和路径条件

Concrete

- 使用实际值 (actual value) 来执行程序
- 在语句中计算新的变量值
- 通过变量的值 (values of variables) 来表示程序状态

Symbolic

- 使用符号值 (symbolic value) 执行程序
- 在语句中计算新的符号表达式
- 通过由符号表达式 (symbolic expressions) 组成的条件来表示程序状态

Symbolic Execution

```
void function(int x, int y) {  
    int z = 2 * y;  
    if (x == z) {  
        y = y + 2;  
        if (x > y + 10) {  
            assert(false);  
        }  
    }  
}
```

```
x = 2, y = 1  
x = 2, y = 1, z = 2  
if (2 == 2)  
    y = 3  
    if (2 > 3 + 10)
```

END

Concrete

Symbolic Execution

```
void function(int x, int y) {  
    int z = 2 * y;  
    if (x == z) {  
        y = y + 2;  
        if (x > y + 10) {  
            assert(false);  
        }  
    }  
}
```

x = A, y = B
x = A, y = B, z = 2B
if (A != 2B) ?

END

Symbolic

Symbolic Execution

```
void function(int x, int y) {  
    int z = 2 * y;  
    if (x == z) {  
        y = y + 2;  
        if (x > y + 10) {  
            assert(false);  
        }  
    }  
}
```

$x = A, y = B$

$x = A, y = B, z = 2B$

if ($A == 2B$) ?

$x = A, y = B + 2, z = 2B$

...

Symbolic

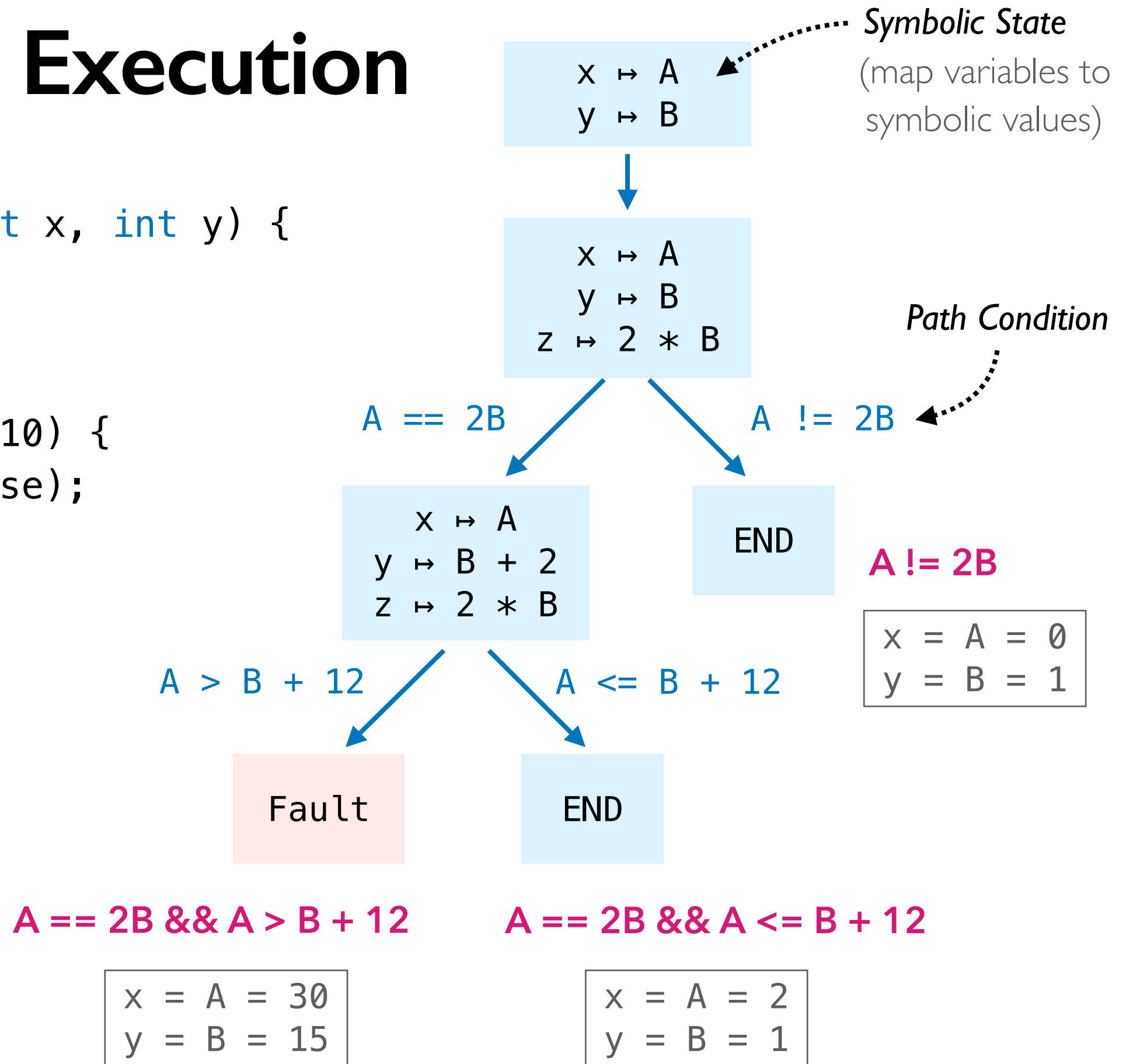
Symbolic Execution

Process

- 基于符号值对待测程序进行路径探索
- 沿执行路径收集输入需要满足的路径约束 (Path Conditions)
- 利用 Constraints Solver 生成能够触各条执行路径的具体测试输入
 - 如果约束可满足 ➡ Test Input
 - 如果约束不可满足 ➡ Infeasible Path

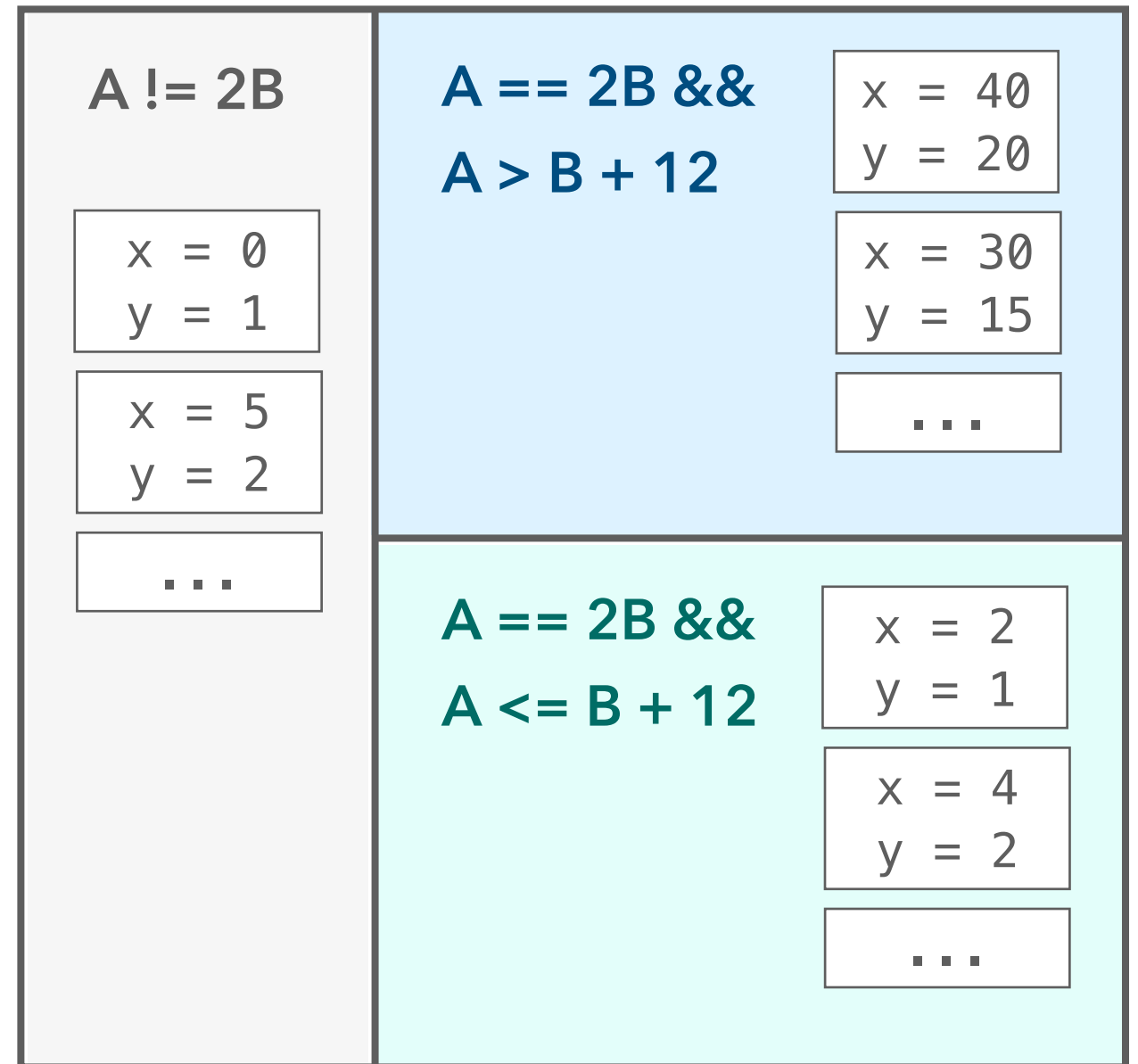
Symbolic Execution

```
void function(int x, int y) {  
  int z = 2 * y;  
  if (x == z) {  
    y = y + 2;  
    if (x > y + 10) {  
      assert(false);  
    }  
  }  
}
```



Symbolic Execution

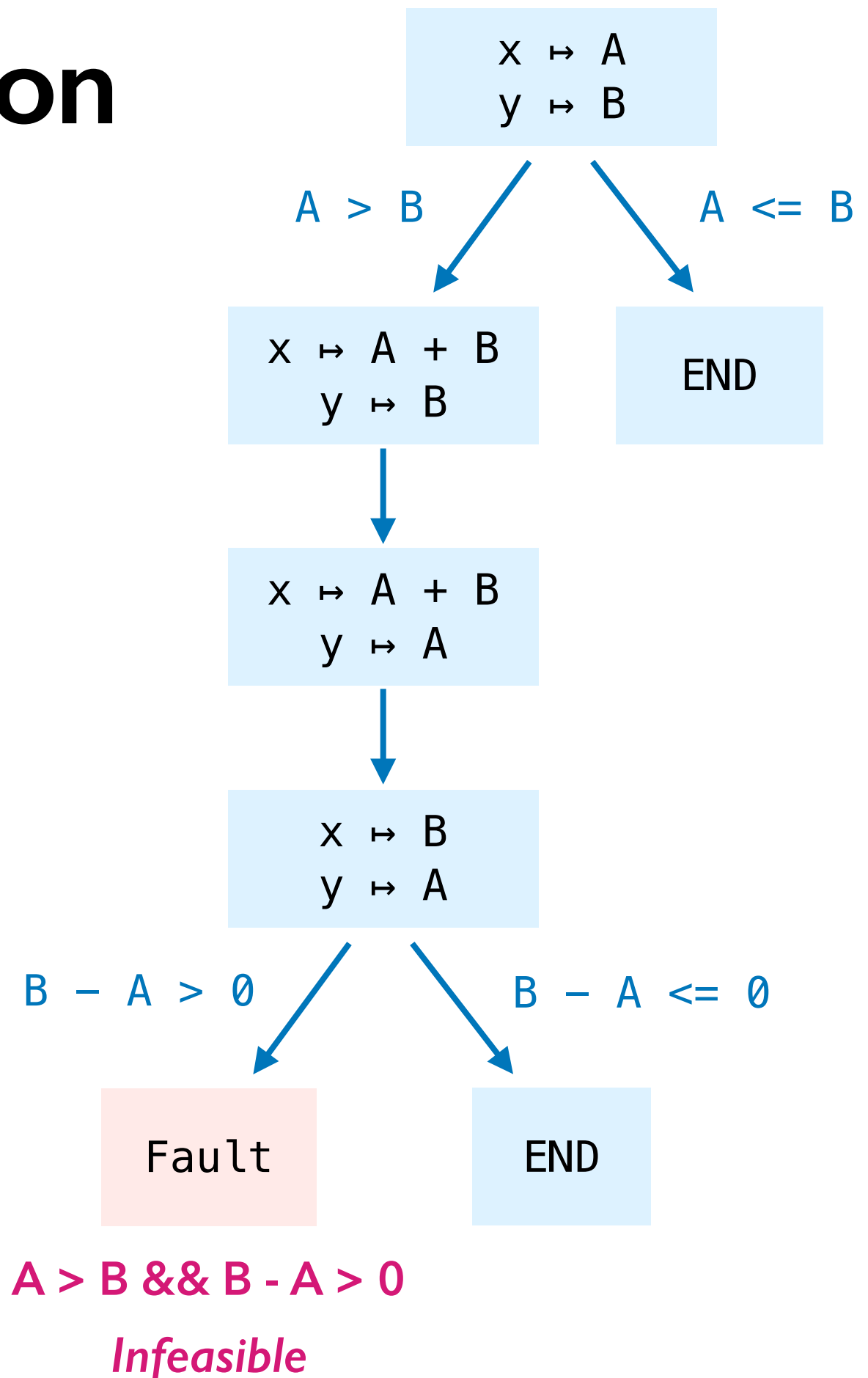
```
void function(int x, int y) {  
    int z = 2 * y;  
    if (x == z) {  
        y = y + 2;  
        if (x > y + 10) {  
            assert(false);  
        }  
    }  
}
```



通过 Path Constraints 刻画能够触发各执行路径的输入集合
(一种等价类的表现形式)

Symbolic Execution

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



[Getting Started](#)[Documentation](#)[Publications](#)[Getting Involved](#)[Releases](#)[Support](#)

The 5th [International KLEE Workshop on Symbolic Execution](#) is coming!

Join us on 16 October 2026 in Munich!

KLEE Symbolic Execution Engine

KLEE is a dynamic symbolic execution engine built on top of the [LLVM](#) compiler infrastructure, and available under the UIUC open source license. For more information on what KLEE is and what it can do, see the [OSDI 2008](#) paper.

Getting Started with KLEE

[Run KLEE via Docker](#)[Install with your package manager](#)[Build from source](#)[& other ways to get started](#)