第十四章 符号执行
（ Symbolic Execution ）

冯 洋

静态分析

- 静态分析使得我们可以**推导**（reason）程序所有可能的执行情况
  - **在程序正式部署前**，对程序执行情况提供一个分析
  - 已有很多静态分析工具
- 不同的静态分析工具，有不同的用途与特点
  - Completeness and Soundness
  - 商业工具研发的两个目的：
    - 降低false positive
    - 提升scalability

抽象

- 抽象使得我们可以对大规模程序的所有可能执行建模
  - 缺点：
    - 必须非常保守（conservative）（请思考为什么？）
    - 需要平衡精确性与可扩展性（scalability）
      - Flow-sensitive；context-sensitive；path-sensitivity…
- 静态分析的抽象通常与程序员对程序的抽象，有较大差异

测试

- 测试非常符合程序员的直觉
- 实践中，测试是使用最多的错误检测技术
- 缺点：
  - 每个测试用例，只能覆盖一种程序执行状态
- 请思考：能否用海量测试用例的执行来较好地覆盖所有的情况？

符号执行（Symbolic Execution）

- 实现逻辑推理自动化是人类社会的终极梦想之一，贯穿人类文明的发展历程。
- 发展历史：
  - 20世纪50-60年代，人工智能的符号学派对于逻辑推理自动化进行了探讨，开发了一些逻辑推理系统，但这些系统的推理能力普遍较弱。
  - 2000年左右，命题逻辑的可满足性问题（SAT）求解取得突破，普林斯顿大学的Sharad Malik团队开发了SAT求解器Chaff，首次实现了对大规模命题逻辑公式的求解，并且开始应用于工业界解决实际问题。
  - 几乎同时，各种特殊逻辑理论的判定算法的研究开始复兴，出现了一批早期的求解器，比如斯坦福大学的SVC和SteP、SRI的ICS等。

符号执行（Symbolic Execution）

- 发展历史：
  - 研究人员随后考虑了SAT和特殊理论判定算法的融合，由此提出了可满足性模理论问题(SMT) 。
  - SMT发展的里程碑包括：2003年开始组织每年一度的SMT研讨会(SMT Workshop)、2004年提出了 SMT-LIB作为SMT问题求解的输入格式标准、2005年创建了SMT竞赛(SMT-COMP)。
  - 迄今为止，SMT竞赛收集了超过100,000个测试用例。2010年之后出现了一批比较成熟的SMT求解器，比如美国微软的Z3、美国斯坦福大学和爱荷华大学的CVC4/ CVC5、美国斯坦福国际研究院的Yices等。

符号执行（Symbolic Execution）

- 应用场景：
  - 软件分析与验证：
  - 软件演绎验证归结为两个逻辑公式的蕴涵问题，然后可以编码为SMT的可满足性问题进行求解。微软基于Z3求解器开发了程序演绎验证工具Dafny[17]与Boogie[18]。
  - 软件的符号执行将路径约束编码为SMT公式，从而将路径可行性问题编码为SMT问题，如果SMT公式有解，则可以生成测试用例。斯坦福大学基于SMT求解器开发了符号执行工具Klee[19]，微软基于Z3求解器开发了测试用例生成工具Pex[20]。

符号执行（Symbolic Execution）

- King, James C. "Symbolic execution and program testing." *Communications of the ACM* 19, no. 7 (**1976**): 385-394.
- 核心思想：通过符号化变量，将程序执行过程一般化
- **符号执行器通过“符号化”执行程序，记录符号化后程序的各种状态**
- 请思考，如果遇到了分支或外部调用，如何处理？？？

符号执行（Symbolic Execution）

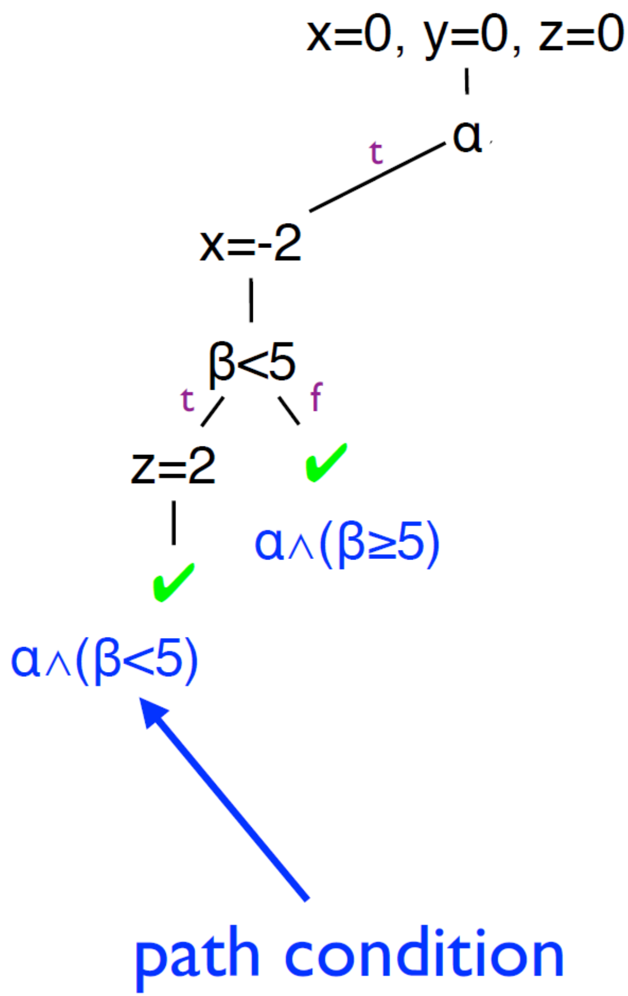
- 70、80年代：出现了基本算法混合不同理论，但求解能力有限
- 2000年前后：SAT(Boolean Satisfiability Problem，布尔可满足性问题)速度大幅提升，转为已SAT为中心的方法
  - 1999-：Eager方法，将SMT(Satisfiability Modulo Theories，**可满足性模理论**)问题编码成SAT问题
  - 2000-：Lazy方法，交互调用SAT求解器和各种专用求解器

符号执行（Symbolic Execution）

```
int a = α, b = β, 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)
```

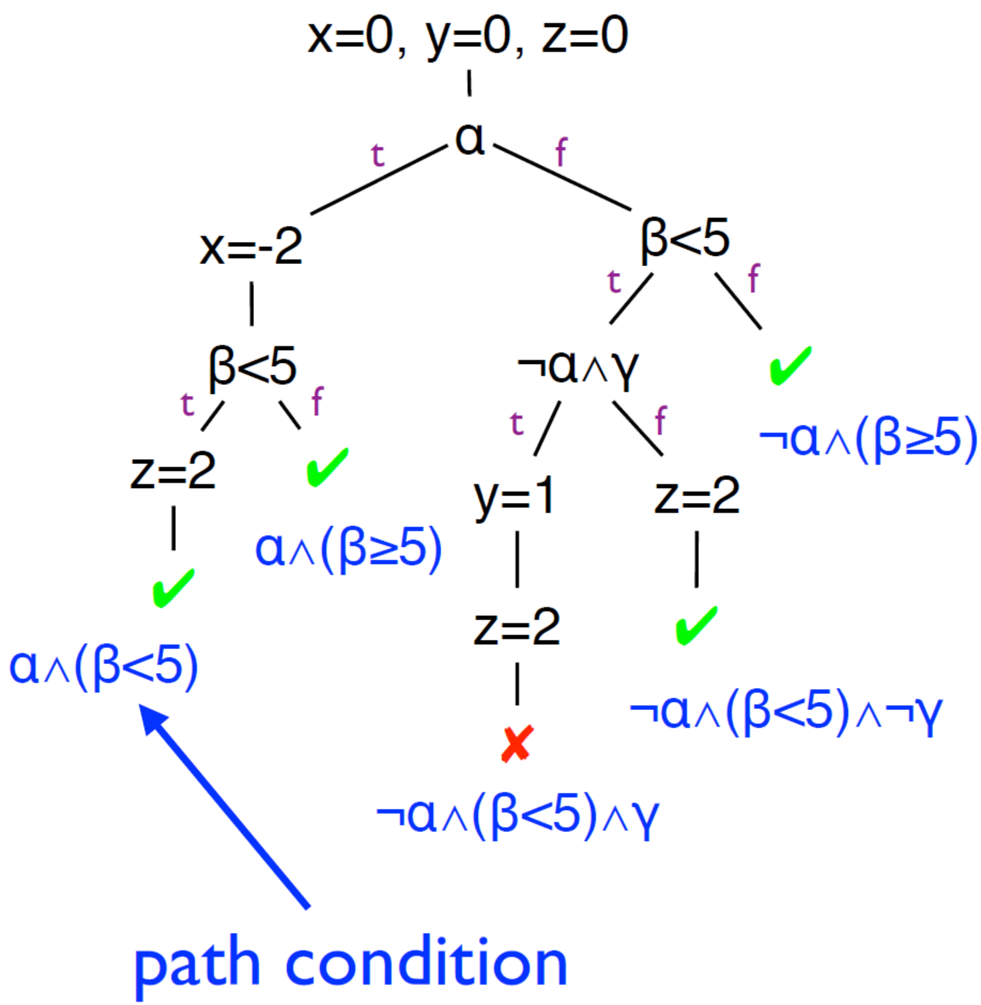
符号执行（Symbolic Execution）

```
int a = α, b = β, 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)
```



符号执行（Symbolic Execution）

```
int a = α, b = β, 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)
```



符号执行（Symbolic Execution）

- 符号执行过程中，需要考虑的核心问题：符号化之后形成的各种条件，如何满足？
  - 例如：某一个程序点如何达到？
    - 某个路径条件如何满足
  - 如果i是一个变量，那么a[i]是否访问越界？
    - 需要考虑 i < 0 或 i > a.length 条件是否满足？
  - 条件拟定之后，如何生成对应的实际输入？
- 如何解决以上问题？？？

符号执行（Symbolic Execution）

- SMT/SAT(Satisfiability Modulo Theories)
  - SAT = 布尔可满足性（SATisfiability）
  - SMT = Satisfiability modulo theory = SAT++
  - 目标：给予一个**约束集合**（Constraints Set），提供一个满足该集合中所有约束的实例
  - 实现示例：
    - Z3, Yices, STP

SMT

- SMT/SAT(Satisfiability Modulo Theories)
- SMT实例是一个first-order logic，在其中函数与谓词符号具有**额外的意义**
- 额外的意义根据使用的理论不同有所不同
  - 例如：线性不等式分析中，符号与谓词的额外意义可能是整数，+，－，×，≤
  - 常用到的理论：Uninterpreted functions; Linear real and integer arithmetic; Extensional arrays; Fixed-size bit-vectors; Quantifiers; Scalar types; Recursive datatypes, tuples, records; Lambda expressions; Dependent types

约束求解

- 给定一组约束，求
  - 这组约束是否可满足
  - （可选）如果可满足，给出一组赋值
  - （可选）如果不可满足，给出一个较小的矛盾集 unsatisfiable core
- 总的来说是不可判定的问题，但存在很多可判定的子问题
- 如

  a > 10

  b < 100 || b > 200

  a+b=30

  可满足：a=15, b=15

约束求解

- SAT solver：解著名的NP完全问题
- Linear solvers：求线性方程组
- Array solvers：求解包含数组的约束
- String solver：求解字符串约束
- SMT：综合以上各类约束求解工具

约束求解技术发展历史

- 约束求解历史上一直有两个特点
  - 速度慢
  - 约束求解算法分散发展，各自只能解小部分约束
- 进入2000年以来
  - SAT的求解速度得到了突飞猛进的进步
    - 理论上还无法完全解释SAT的高速求解
  - 以SAT为核心，各种单独的约束求解算法被整合起来，形成了SMT

约束求解技术发展历史

- SAT问题
  - 最早被证明的NP完全问题之一（1971）
  - 文字literal：变量x或者是x取反
    - 如$\neg x$
  - 子句clause：文字的析取（disjunction）
    - 如$x \vee \neg y$
  - 布尔赋值：从变量到布尔值上的映射
  - SAT问题：子句集上的约束求解问题
    - 给定一组子句，寻找一个布尔赋值，使得所有子句为真

约束求解技术发展历史

- 合取范式（Conjunctive Normal Form）
  - 合取范式：子句的合取
    - 如 $x \vee \neg y \wedge \neg x$
  - SAT问题通常是通过合取范式定义的
  - 任何命题逻辑公式可以表达为合取范式
  - 即：SAT问题可以求解任何命题逻辑公式

从SAT到SMT

- SAT问题回答某个命题逻辑公式的可满足性，如：

  $$A \wedge B \vee \neg C$$

- 但实际中的公式却往往是这样的：

  $$a + b < c \wedge f\ b > c \vee c > 0$$

- 如何判断这样公式的可满足性？
- 从逻辑学角度来看，$a + b < c$或者$f\ b > c$都是逻辑系统中 不包含的符号，需要知道他们的意思

从SAT到SMT

- 从逻辑学角度来看，$a + b < c$或者$f\ b > c$都是逻辑系统中 不包含的符号，需要知道他们的意思
- 理论(Theory)：
  - 理论用于对这类符号赋予含义
  - 理论包含一组公理和这组公理能推导出的结论
- 可满足性模理论Satisfiability Modulo Theories：
  - 给定一组理论，根据给定背景逻辑，求在该组理论解释下公式的可满足性

从SAT到SMT

- 常见理论举例：EUP（Equality with Uninterpreted Functions）
- 公理：
  - $ai = bi \Longrightarrow f(a1 \ldots an) = f(b1 \ldots bn)$
  - $a = b \Leftrightarrow \neg(a \neq b)$
  - 如：$a * (f(b) + f(c)) = d \wedge b * f(a) + f(c) \neq d \wedge a = b$
    f, *和+都看做是未定义的函数
  - **可通过以上公里直接推出矛盾**

从SAT到SMT

- 常见理论举例：EUF（Equality with Uninterpreted Functions）
- 算术

  $a+10<b$

  $2x + 3y + 4z = 10$
- 数组

  read(write(a, i, v), i)=v
- 位向量Bit Vectors

  $a[0] = b[1] \wedge a = c \wedge b[1] \neq c[0]$

从SAT到SMT

- 2000年前后：SAT速度大幅提升，转为已SAT为中心的方法
  - 1999-：Eager方法，将SMT问题编码成SAT问题
  - 2000-：Lazy方法，交互调用SAT求解器和各种专用求解器

Eager方法

- 将SMT问题编码成SAT问题
- 例：将EUF编码成SAT
  - $f(a) = c \wedge f\ b \neq c \wedge a \neq b$
- 引入符号替代函数调用
  - A替代$f\ a$，B替代$f(b)$
  - 原式变为
  - $A = c \wedge B \neq c \wedge a \neq b$
- 同时根据公理1添加约束
  - $a = b \rightarrow A = \mathrm{B}$

Eager方法

- 将SMT问题编码成SAT问题
- 例：将EUF编码成SAT
  - $f(a) = c \wedge f\ b \neq c \wedge a \neq b$
- 引入符号替代函数调用
  - A替代$f\ a$，B替代$f(b)$
  - 原式变为
  - $A = c \wedge B \neq c \wedge a \neq b$
- 同时根据公理1添加约束
  - $a = b \rightarrow A = \mathrm{B}$
- 引入布尔变量替代等式
  - $P_{A=c} \wedge \neg P_{B=c} \wedge P_{a \neq b}$
  - $P_{a=b} \rightarrow P_{A=B}$
- 同时为公里2添加约束
  - $P_{A=c} \wedge P_{B=c} \rightarrow P_{A=B}$
  - $P_{A=B} \wedge P_{B=c} \rightarrow P_{A=c}$
  - $P_{A=B} \wedge P_{A=c} \rightarrow P_{B=c}$
  - ……

Eager方法

- 依然存在一些问题
  - 很多理论存在专门的求解算法，如
    - EUF可以用一个不动点算法不断合并等价类求解
    - 线性方程组存在专门算法求解
  - 编码成SAT之后，SAT求解器无法利用这些算法
  - 模块化程度不高
    - 每种理论都要设计单独的编码方法
    - 不同理论混合使用时要保证编码方法兼容

Lazy方法

- 黑盒混合SAT求解器和各种理论求解器
- 理论求解器：
  - 输入：属于特定理论的公式组，组内公式属于合取关系
    - EUF公式组：
      - $f(a) = c$
      - $f(b) \neq c$
      - $a \neq b$
    - 线性方程组：
      - a+b=10
      - a-b=4
    - 输出：SAT或者UNSAT

Lazy方法

$$\underbrace{g(a) = c}_{1} \wedge \left(\underbrace{f(g(a)) \neq f(c)}_{-2} \vee \underbrace{g(a) = d}_{3} \right) \wedge \underbrace{c \neq d}_{-4}$$

- 生成如下公式到SAT求解器 : {1}, {-2, 3}, {-4}
- SAT求解器返回SAT和赋值{1, -2, -4}
- 生成如下公式组到EUF求解器:

  $g(a) = c$; $f(g(a)) \neq f(c)$; $c \neq d$
- EUF求解器返回UNSAT
- 生成如下公式到SAT求解器：{1}, {-2, 3}, {-4}, {-1, 2, 4}
- SAT求解器返回SAT和赋值{1, 2, 3, -4}
- EUF求解器返回UNSAT
- SAT求解器发现{1}, {-2, 3}, {-4}, {-1, 2, 4}, {-1, -2, -3, 4}不可满足

Lazy方法

- Lazy方法优点
  - 同时利用SAT求解器和理论求解器的优势
  - 模块化
    - 新的理论只需要实现公共接口就可以集成到SMT求解器中
  - 目前主流SMT求解器中普遍采用Lazy方法

Lazy方法

- Lazy方法的问题

$$\underbrace{a = b}_{1} \wedge \left(\underbrace{f(g(a)) \neq f(c)}_{-2} \vee \underbrace{g(a) = d}_{3}\right) \wedge \underbrace{b \neq a}_{-4}$$

SAT	EUF
{1, -2, 3, -4}	UNSAT
{1, -2, -3, -4}	UNSAT
{1, 2, 3, -4}	UNSAT
UNSAT	

谓词转化器语义

- 谓词转化器语义(Predicate Transformer Semantics)提供了一种语义支持，用以描述逻辑公式之间的转化
  - 最强后置条件语义（Strongest Post-condition Semantics）
    - 如果在程序c执行之前逻辑公式φ的结果是true，那么执行完毕之后逻辑公式ψ 则应该为true
    - 向前符号执行
  - 最弱前置条件语义（Weakest pre-condition semantics）
    - 如果在程序c执行之后逻辑公式φ的结果是true，那么执行之前逻辑公式ψ 则应该为true
    - 就是向后符号执行

谓词转化器语义

- 谓词转化器使得Hoare Logic运转
- Hoare Logic是一个推导系统
- 原子性（Axioms）及推导（Inference）规则被用于推导Hoare三元组，即{φ, c, ψ}
- 如果φ在程序c执行之前被满足，那么当c停止之后，ψ则需要被满足

符号执行的具体应用

literals

clauses

$\omega_1 = (b \vee c)$

$\omega_2 = (\neg a \vee \neg d)$

$\omega_3 = (\neg b \vee d)$

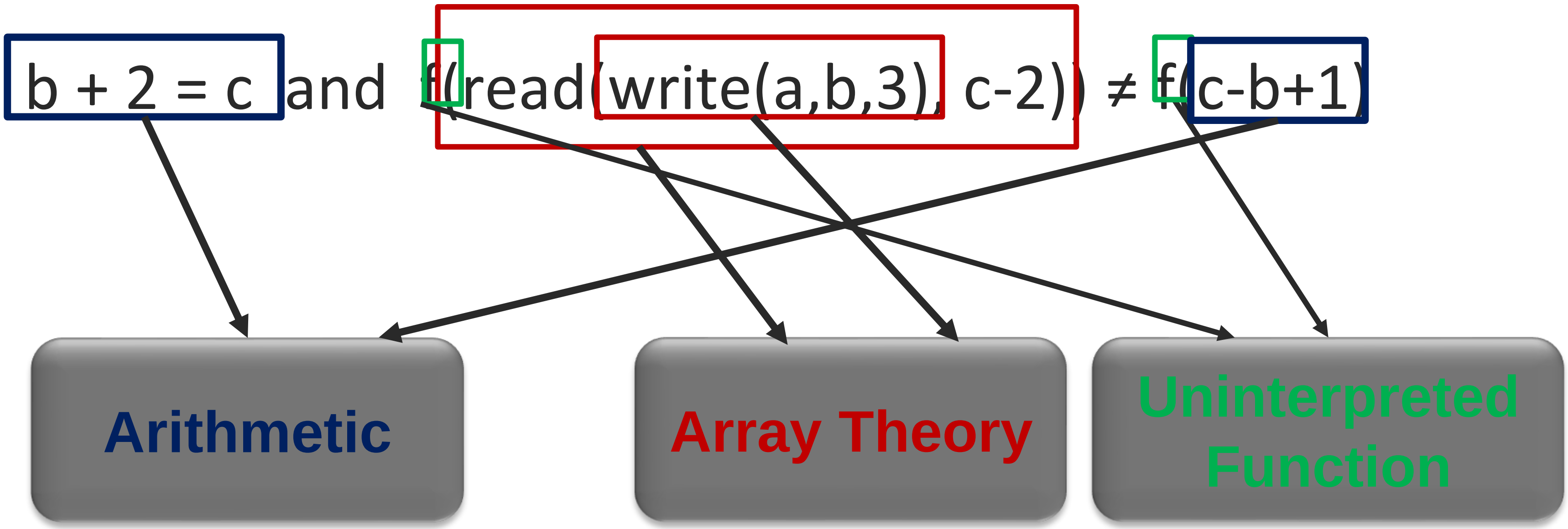
$\varphi = \omega_1 \wedge \omega_2 \wedge \omega_3$

$A = \{a=0, b=1, c=0, d=1\}$

SATisfying assignment!

- 给定一个若干条件语句组成的逻辑表达式，我们基于SAT求解器确定该表达式的解。

符号执行的具体应用



b + 2 = c and f(read(write(a,b,3), c-2)) ≠ f(c-b+1)
Arithmetic
Array Theory
Uninterpreted Function

符号执行的具体应用

- 一个简单的SMT求解例子

- b + 2 = c and f(read(write(a,b,3), c-2)) ≠ f(c-b+1)

[Substituting c by b+2]

- b + 2 = c and f(read(write(a,b,3), b+2-2)) ≠ f(b+2-b+1)

[Arithmetic simplification]

- b + 2 = c and f(read(write(a,b,3), b)) ≠ f(3)

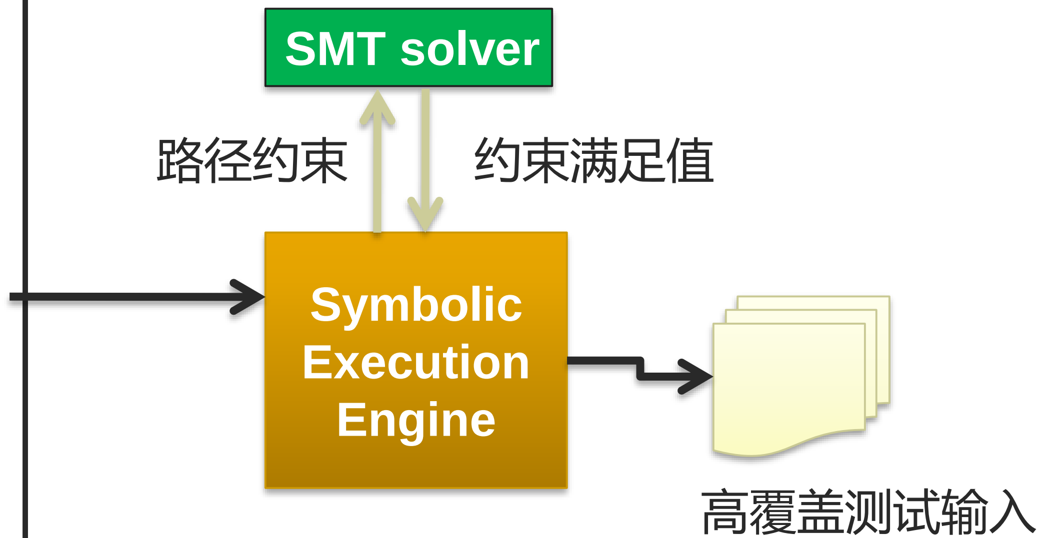
[Applying array theory axiom]

forall a,i,v:read(write(a,i,v), i) = v]

- b+2 = c and f(3) ≠ f(3) [**NOT SATISFIABLE**]

符号执行的例子

```
Void func(int x, int y){
        int z = 2 * y;
        if(z == x){
                if (x > y + 10)
                        ERROR
  }
}
int main(){
        int x = sym_input();
        int y = sym_input();
        func(x, y);
        return 0;}
```

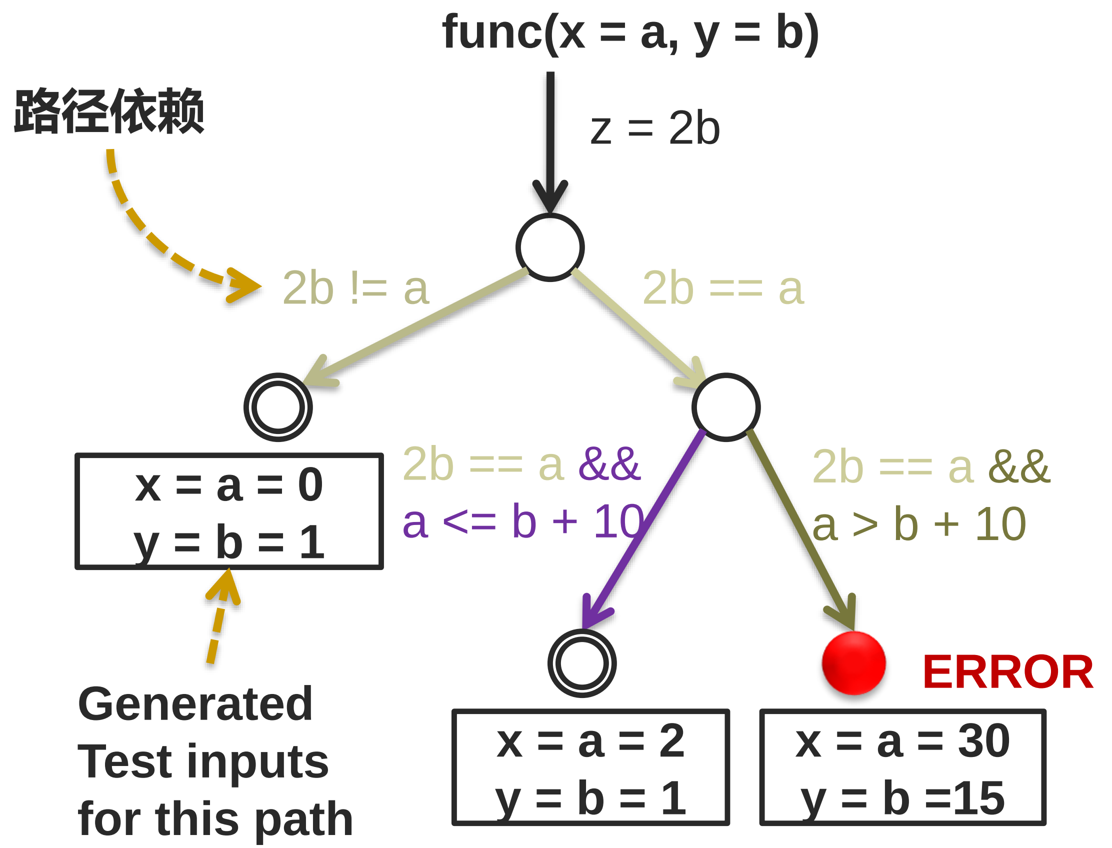


Symbolic Execution

符号执行的例子

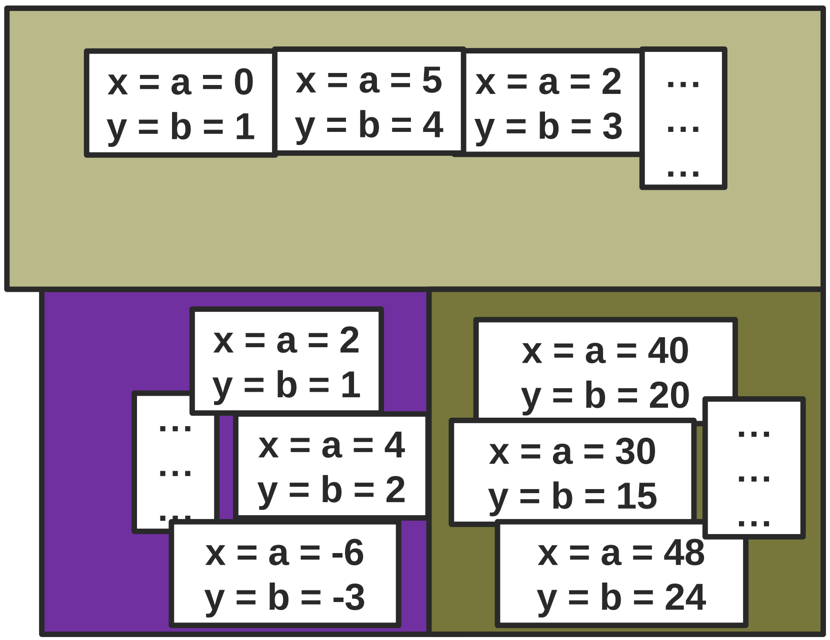
```
Void func(int x, int y){
    int z = 2 * y;
    if(z == x){
        if (x > y + 10)
            ERROR
    }
}
int main(){
    int x = sym_input();
    int y = sym_input();
    func(x, y);
    return 0;
}
```

符号执行技术具体怎么运行?

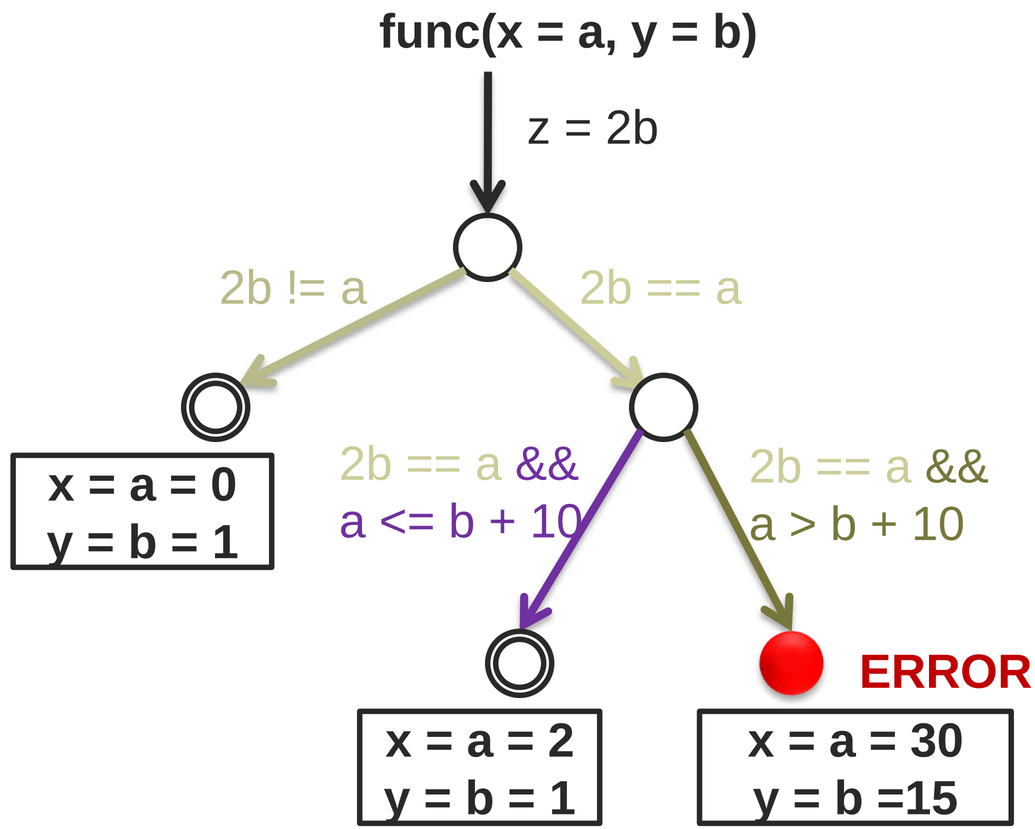


- 输入需要被标记为符号信息!

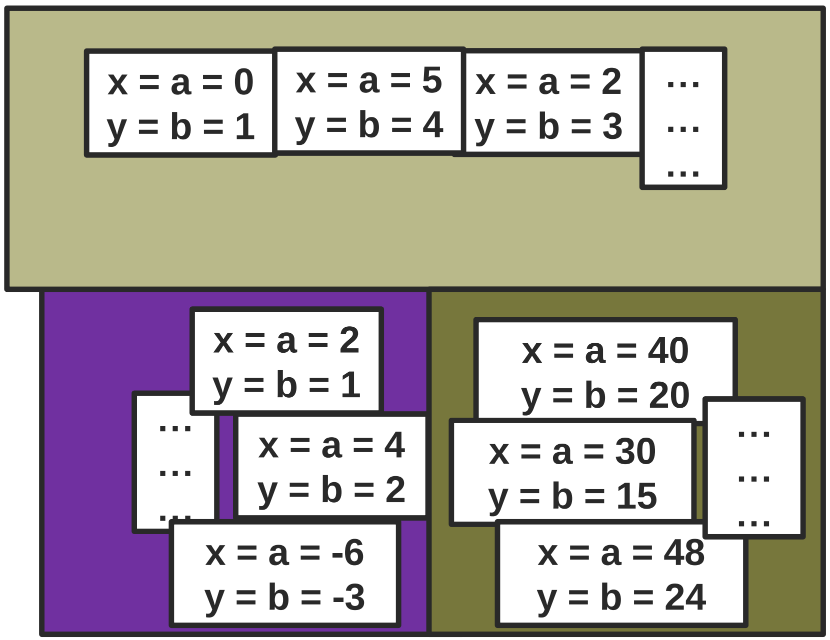
符号执行的例子



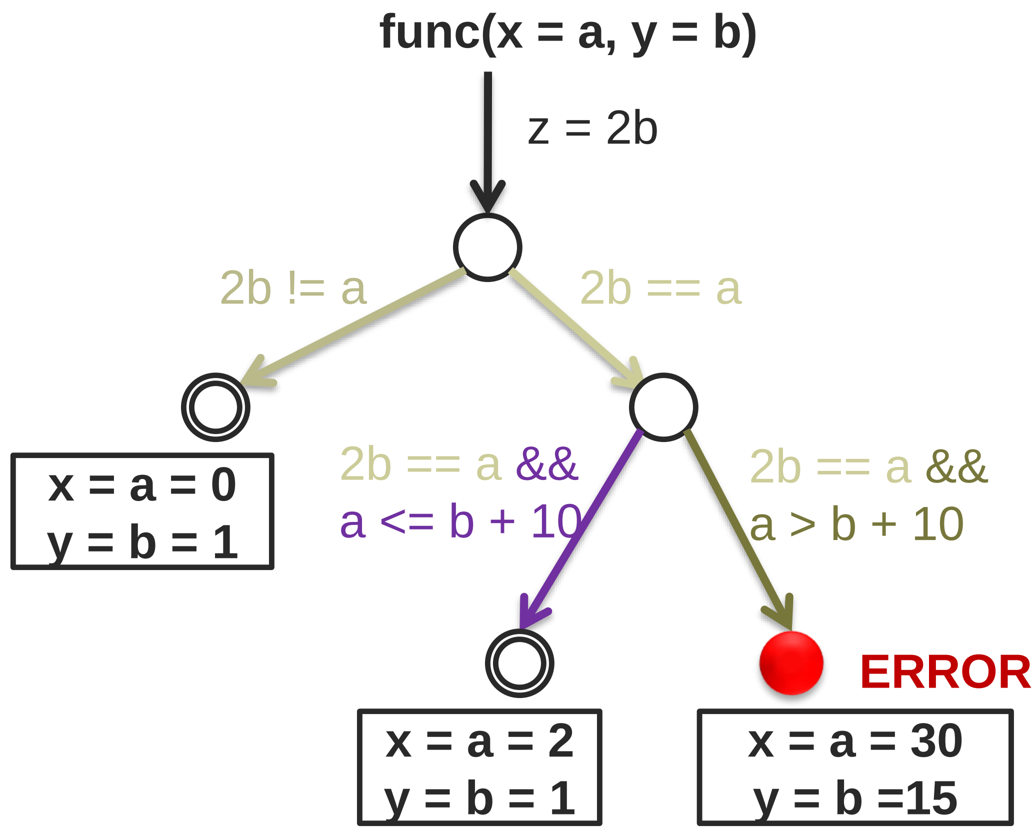
路径约束表明了输入域的等价类信息



符号执行的例子



路径约束表明了输入域的等价类信息



混合执行（Concolic Execution）

- 经典的符号执行存在的问题？
  - 循环与递归 – 无限的执行树
  - 路径爆炸 – 指数级增长的路径数量
  - 栈建模 – 符号执行的数据结构与指针
  - SMT Solver 的限制 – 用于处理复杂路径约束
  - 环境建模 – 用于处理本地的/系统的/库调用/文件操作/网络时间等
  - 覆盖问题 – 或许不能走到执行树的底层（特别是当代码中存在较多的循环的时候）

混合执行（Concolic Execution）

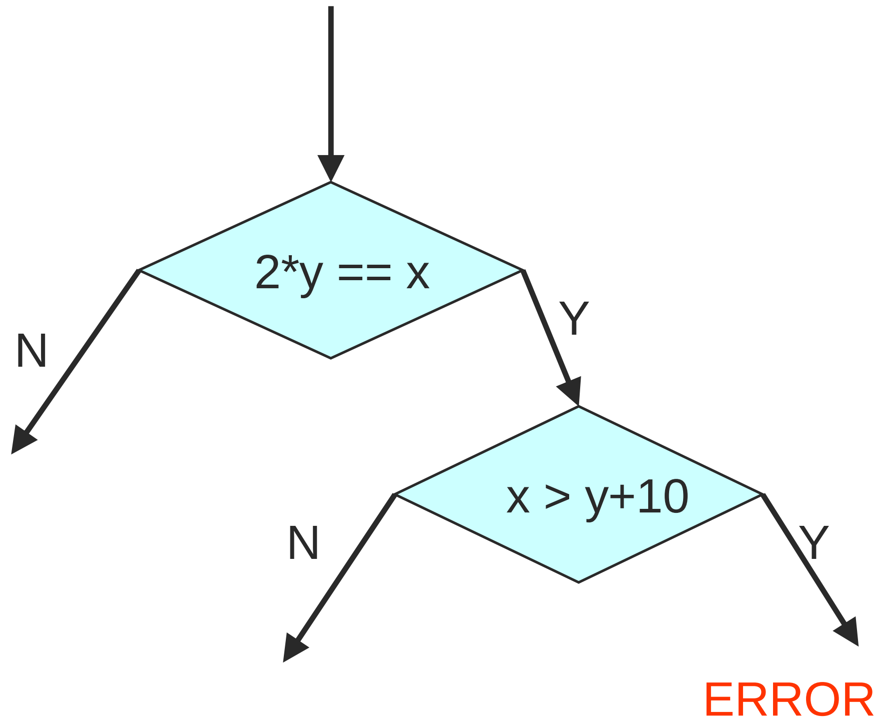
- 有没有解决方案？
- Concolic Execution
- Concolic = Concrete + Symbolic
  - 一种通过传统测试技术与自动化程序分析结合的技术
  - 也称为动态符号执行（Dynamic Symbolic Execution）
  - 目的是为了访问到程序执行树中比较深层的部分
  - 程序同时被采用符号（Symbolic）与具体（Concrete）两种方式执行
  - 通常通过一个随机的输入将程序进行执行
  - 对外部调用特别有效

混合执行（Concolic Execution）

- 混合执行的大体步骤
  - 基于一个随机输入启动程序的具体执行
  - 具体执行中收集路径约束
    - 例如：a && b && c
  - 后续执行中通过翻转最后一个条件来实现对应的符号结果
    - 例如：a && b && !c

混合执行（Concolic Execution）

```
void testme (int x, int y)
{
    z = 2*y;
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```



混合执行（Concolic Execution）

Concrete Execution

Symbolic Execution

```
void testme (int x, int y) {

z = 2* y;
if (z == x) {
        if (x > y+10) {
                ERROR;
        }
    }
}
```

concrete state	symbolic state	path condition
x = 22, y = 7	x = a, y = b	

混合执行（Concolic Execution）

Concrete Execution　　Symbolic Execution

```
void testme (int x, int y) {
z = 2* y;

if (z == x) {
        if (x > y+10) {
              ERROR;
        }
    }
}
```

concrete state	symbolic state	path condition
x = 22, y = 7, z = 14	x = a, y = b, z = 2*b	

混合执行（Concolic Execution）

Concrete Execution | Symbolic Execution

```
void testme (int x, int y) {
z = 2* y;

if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

concrete state	symbolic state	path condition
x = 22, y = 7, z = 14	x = a, y = b, z = 2*b	

混合执行（Concolic Execution）

Concrete Execution | Symbolic Execution

concrete state	symbolic state	path condition
		2*b != a
x = 22, y = 7, z = 14	x = a, y = b, z = 2*b	

```
void testme (int x, int y) {
    z = 2* y;

    if (z == x) {

        if (x > y+10) {
            ERROR;
        }
    }
}
```

混合执行（Concolic Execution）

Concrete Execution

Symbolic Execution

concrete state | symbolic state | path condition

```
void testme (int x, int y) {
    z = 2* y;

    if (z == x) {

        if (x > y+10) {
            ERROR;
        }
    }
}
```

Solve: 2*b == a
Solution: a = 2, b = 1

2*b != a

x = 22, y = 7, z = 14

x = a, y = b, z = 2*b

混合执行（Concolic Execution）

	Concrete Execution	Symbolic Execution	
	concrete state	symbolic state	path condition
	x = 2, y = 1	x = a, y = b	

```
void testme (int x, int y) {

    z = 2* y;
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }

}
```

混合执行（Concolic Execution）

Concrete Execution | Symbolic Execution

```
void testme (int x, int y) {
    z = 2* y;

    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

concrete state	symbolic state	path condition
x = 2, y = 1, z = 2	x = a, y = b, z = 2*b	

混合执行（Concolic Execution）

Concrete Execution | Symbolic Execution

```
void testme (int x, int y) {
    z = 2* y;
    if (z == x) {

        if (x > y+10) {
            ERROR;
        }

    }

}
```

concrete state	symbolic state	path condition
		2*b == a
		a < b + 10
x = 2, y = 1, z = 2	x = a, y = b, z = 2*b	

混合执行（Concolic Execution）

Concrete Execution | Symbolic Execution

```
void testme (int x, int y) {
    z = 2* y;
    if (z == x) {

        if (x > y+10) {
            ERROR;
        }

    }

}
```

concrete state	symbolic state	path condition
		2*b == a
		a < b + 10
x = 2, y = 1, z = 2	x = a, y = b, z = 2*b	

Solve: (2*b == a) ^ (a – b> 10)
Solution: a = 30, b = 15

混合执行（Concolic Execution）

Concrete Execution | Symbolic Execution

```
void testme (int x, int y) {
    z = 2* y;

    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

concrete state	symbolic state	path condition
x = 30, y = 15	x = a, y = b	

混合执行（Concolic Execution）

Concrete Execution

Symbolic Execution

```
void testme (int x, int y) {
    z = 2* y;
    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }
}
```

co cre

symbolic state

path condition

Program Error

2*b == a

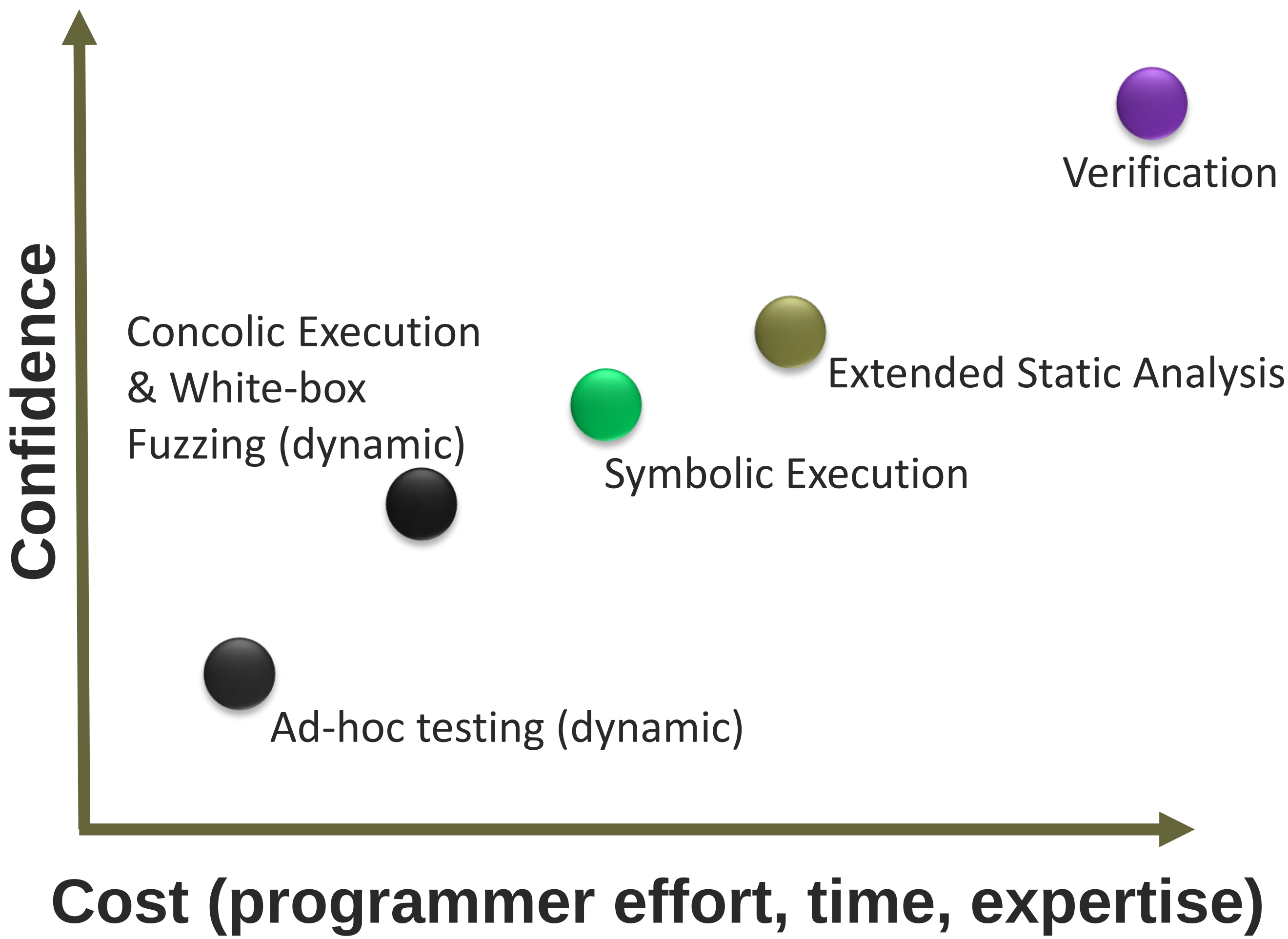
a > b+10

x = 30, y = 15

z = 30

x = a, y = b

程序验证方法对比


Verification
Concolic Execution
& White-box
Fuzzing (dynamic)
Extended Static Analysis
Symbolic Execution
Ad-hoc testing (dynamic)
Confidence
Cost (programmer effort, time, expertise)