软件分析与测试
第三次课：测试准则

严俊

中国科学院大学
University of Chinese Academy of Sciences
ISCAS 中国科学院软件研究所
Institute of Software, Chinese Academy of Sciences

测试的要素

- 对被测软件行为的描述
- 对被测软件功能的正确性的描述（Oracle）
- 测试充分性—测试什么时候终止
  - →Coverage criteria

测试准则 Test Criterion

- 功能
  - 作为测试可以结束的终止条件;
  - 作为软件质量的一个度量，一个测试集合代表了一个质量等级
  - 用于生成测试用例。如果两个测试集合符合相同的测试准则，我们称他们为等效的。
- 一般定义为某种覆盖（Coverage）

黑盒测试中的准则

- **检查程序的接口和性能**
  - 尽可能少的测试用例集合来保证对系统的充分测试;
  - 检查所有功能点。
    部分错误不是白盒测试能解决的，如白盒测试不能发现功能缺失的错误。
- **边界值分析（Boundary Value Analysis）**
  - 主要目的是查找域错误 。
  - 先选取边界上一个点作为测试用例。如果这条边界在子域内，再选 一个外点（也就是“临近”边界，但是不在子域中的点）作为测试用例；否则选一个内点作为测试用例。

组合测试中的覆盖

- 按照如下目标设计测试用例

- 给定的一个小正整数 t（称为组合强度，一般为2或者3），测试用例应该覆盖任意 t 个参数的所有取值组合

- 条件表达式测试

Traffic Collision Avoidance System (TCAS)

```
bool Non_Crossing_Biased_Descend()
{
    int upward_preferred;
    bool result;

    upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
    if (upward_preferred)  {
        result = Own_Below_Threat() && (Cur_Vertical_Sep >= MINSEP) &&
                 (Down_Separation >= ALIM());
    }
    else  {
        result = !(Own_Above_Threat()) || ((Own_Above_Threat()) &&
                 (Up_Separation >= ALIM()));
    }
    return result;
}
```

条件表达式

- （布尔）逻辑公式
- Decision 判定/判断
- Condition 条件

- 例： S = (x1∨x2) ∧ (x3∨x4)

  四个条件

Condition

- NASA的定义
  - A decision without a Boolean operator is a condition. If a condition appears more than once in a decision, each occurrence is a distinct condition.
  - 比如(A ∧ B) ∨ (A ∧ C)，实际上是4个condition， A(1), B, A(2), C
  - 这个定义是不合适的，因为没法构造测试用例分别测试A(1)， A(2)

Decision coverage and Condition coverage

- Decision coverage (DC) requires two test cases for each decision: one for a true outcome and another for a false outcome.

- Condition coverage (CC) requires that each condition in a decision take on all possible outcomes at least once.

条件表达式测试

- **判定覆盖DC**
  - 对于每个判断，至少有两个测试数据使其在运行中分别取真以及假
  - 两个测试用例

S = (x1∨x2)∧(x3∨x4)

判定覆盖测试集

x1	x2	x3	x4	S
0	0	0	0	0
1	0	1	0	1

- **条件覆盖CC**
  - 每个判断中的每个条件，至少有两个测试数据使其在运行中分别取真、假值
  - 两个测试用例

条件覆盖测试集

x1	x2	x3	x4	S
0	1	0	1	1
1	0	1	0	1

C/DC测试效果

- $S_0 = (x1 \vee x2) \wedge (x3 \vee x4)$
  满足CC，DC，C/DC的测试集

x1	x2	x3	x4	S
0	0	0	0	0
1	1	1	1	1

- 假定表达式误写为 $S_1 = (x1 \vee x2) \wedge (x3 \vee \neg x4)$
  - S_0 和S_1 的测试结果相同
  - 原因：没有测试每个条件独立对判定取值的影响

常见错误类型

- ORF (Operator Reference Faults)
  - 例如 and 写成 or
- VNF (Variable Negation Faults)
  - 变量误写为它的否定
- ENF (Expression Negation Faults)
  - 表达式误写为它的否定

测试用例的区分性

- $S = a \wedge b$
- 测试集

  $t_1=(1, 1)$

  $t_2=(1, 0)$

  $t_3=(0, 1)$

故障类型	SUT	t_1	t_2	t_3
正确表达式	$a \wedge b$	1	0	0
单布尔运算符故障	$a \vee b$	1	1	1
	$a \wedge \neg b$	0	1	0
	$\neg a \wedge b$	0	0	1
多布尔运算符故障	$a \vee \neg b$	1	1	0
	$\neg a \vee b$	1	0	1
	$\neg a \wedge \neg b$	0	0	0
	$\neg a \vee \neg b$	0	1	1

MC/DC

- MC/DC（修正的判断条件覆盖，Modified Condition/Decision Coverage）
  - 测试用例要满足 C/DC
  - 每个条件要能够独立地影响判断的值：对于每个判断的每一个条件，至少有两个测试用例 --这个条件取不同的真假值，同时这个判断也取不同的值
- 用于航空航天等安全关键软件的测试

$$S = a \wedge b$$

TC	a	b	s
t_1	1	1	1
t_2	1	0	0
t_3	0	1	0

MC/DC – ex.1

对每一个条件，都能找到一**对测试用例**，满足三点：

1. **只改变这个条件的取值**
2. 其他条件保持不变（或至少不影响结果）
3. **最终判定结果发生翻转**

比如 `x`：

`t2` 和 `t3` 里，`y` 和 `z` 的组合保证 `(y || z)` 为真，唯一变化的是 `x`，而整个 `x && (y || z)` 的结果确实随 `x` 翻转。

这就证明：**x 对判定结果有独立影响。**

```
bool x, y, z;
int example() {
  int w;
  if(x && (y || z))
    w = 0;
  else w = 1;
  if(w == 0) return 1;
  else return 0;
}
```

TABLE I: Outcomes of conditions and decisions

expression	t_1	t_2	t_3	t_4	t_5
x	F	T	F	T	T
y	T	T	T	F	F
z	T	F	F	T	F
$x \wedge (y \vee z)$	F	T	F	T	F

t2 and t3 → x;
t2 and t5 → y;
t4 and t5 → z.

MC/DC – ex.2

- S = (x1∨x2) ∧ (x3∨x4)
- 5个测试用例
  - {t1, t3} 覆盖条件x1
  - {t1, t2} 覆盖条件x2
  - {t4, t5} 覆盖条件x3
  - {t2, t4} 覆盖条件x4
- 测试用例{t2, t4} 区分
  - S_0 = (x1∨x2) ∧ (x3∨x4)
  - S_1 = (x1∨x2) ∧ (x3∨ ¬x4)

用例	x1	x2	x3	x4	S
t1	0	0	0	1	0
t2	0	1	0	1	1
t3	1	0	0	1	1
t4	0	1	0	0	0
t5	0	1	1	0	1

MC/DC测试集大小

- 一般情况测试用例数[n+1, 2n]
- 对很多表达式，最优测试集大小就是n+1

Ling Yang, et al., Generating Minimal Test Set Satisfying MC/DC Criterion via SAT Based Approach. ACM SAC 2018

一种（常见的）特殊情况

- 对于一个只含有（与、或、非运算）的n变量布尔表达式，每个变量（条件）只出现一次，那么它的MC/DC最优测试集大小为n+1。

- 证明：（数学归纳法)
  - n=1 成立
  - 假设表达式$S_1(x_1,x_2,\ldots,x_j)$有大小为j+1的测试集
    $TS_1 = \{t_{1,1}, t_{1,2},\ldots, t_{1,j+1}\}$,
    表达式$S_2(x_{j+1},x_{j+2},\ldots,x_{j+k})$有大小为k+1的测试集
    $TS_2 = \{t_{2,1}, t_{2,2},\ldots, t_{2,k+1}\}$
  - 对于表达式$S= S_1 \wedge S_2$，不妨假定$S_1(t_{1,1})=1$, $S_2(t_{2,1})=1$,构造集合
    $T_1 = \{t_{1,1}t_{2,1}, t_{1,2}t_{2,1}, \ldots, t_{1,j+1}t_{2,1}\}$
    $T_2 = \{t_{1,1}t_{2,1}, t_{1,1}t_{2,2}, \ldots, t_{1,1}t_{2,k+1}\}$
    显然集合T_1可以MC/DC 覆盖S的前j个参数，集合T_2可以MC/DC 覆盖S的后k个参数。那么 $T = T_1 \cup T_2$是表达式S的MC/DC测试集，它的大小为t+1+k+1-l = t+k+1 （有一个公共元素$t_{1,1}t_{2,1}$ ）。
  - 类似可以证明$S= S_1 \vee S_2$的情况。

MUMCUT

- 针对判断表达式中的更多错误设计，由三个子准则构成

  当整个判定为真的时候，每个条件有没有机会“ 单独成为关键角色 ”

  - MUTP (Multiple Unique True Point)
  - MNTP (Multiple Near False Point)

  整个判定为 false，但只要把某一个条件翻转，判定就会变成 true

  - CUTPNFP (Corresponding Unique True Point and Near False Point Pair)

  对每一个条件，你不仅要有一个 Unique True Point，还要有一个与之高度对应的 Near False Point，而且这两个用例之间只允许该条件发生变化

- 不足之处
  - 过于复杂，不易设计测试用例
  - 要求条件表达式具有特定的形式

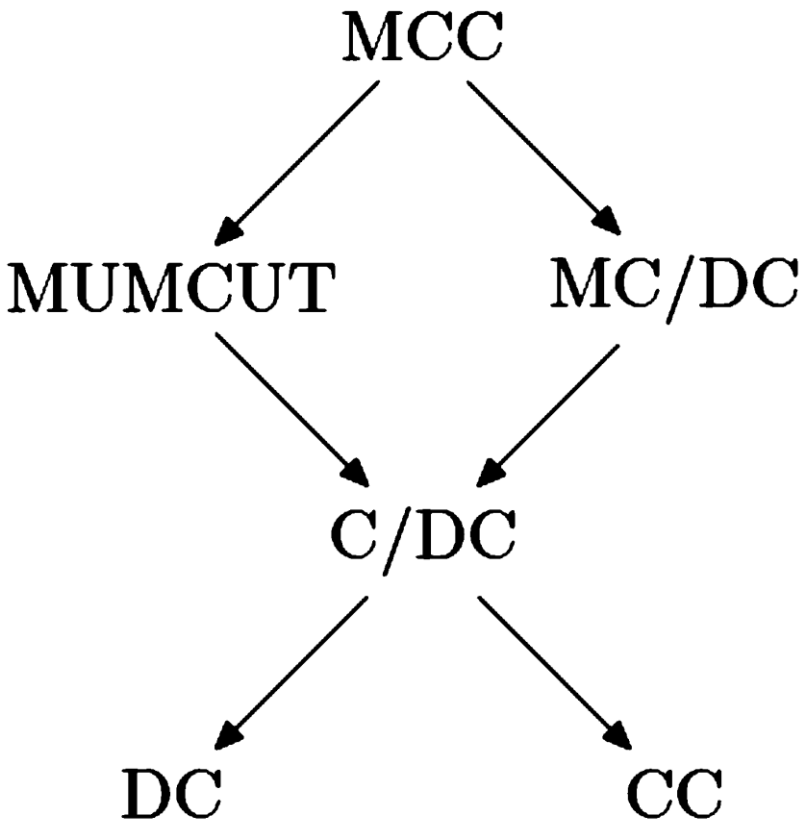
覆盖准则之间的包含关系

- 一个准则 A 包含 (Subsume) 准则 B, 当且仅当满足 A 的测试数据集同时是满足 B 的测试数据集。

- 如果 A 包含 B , 那么测试准则 A 比 B 有更强的查错能力。

条件表达式覆盖准则之间的关系

- 一个准则 A 包含 (Subsume) 准则 B, 当且仅当满足 A 的测试数据集同时是满足 B 的测试数据集.
- 如果 A 包含 B , 那么测试准则 A 比 B 有更强的查错能力.



控制流测试

图模型

- 软件工程（软件测试）中，最常用的模型
- 一些常见的图模型
  - 控制流图 CFG (Control Flow Graph)
    - statements & branches, methods & calls, etc.
  - 状态机 FSMs and statecharts
    - states and transitions
  - Use cases

有向图（Directed Graph）

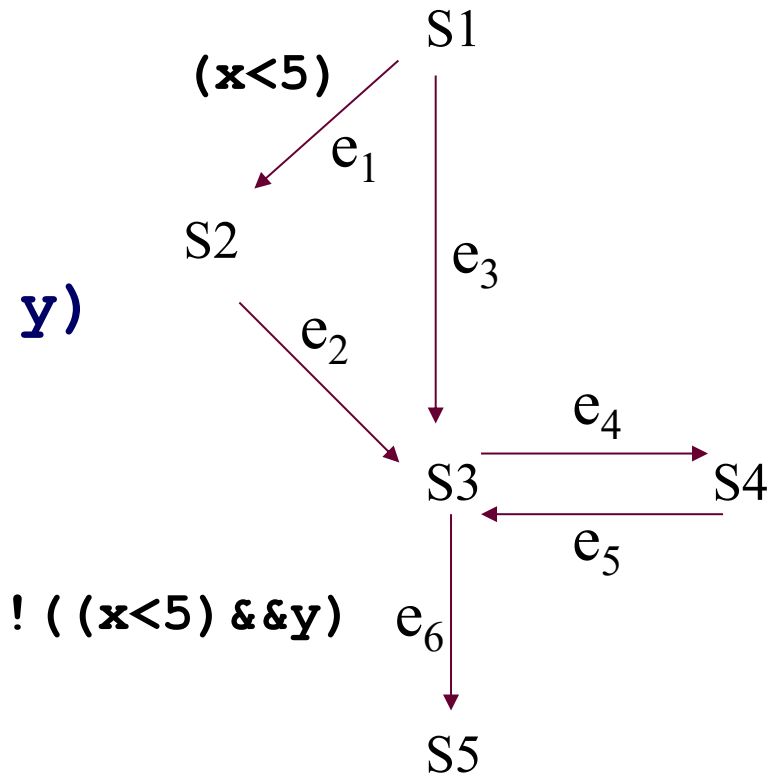
- A set N of nodes, N is not empty
- A set N_0 of initial nodes, N_0 is not empty
- A set N_f of final nodes, N_f is not empty
- A set E of edges, $E \subseteq N \times N$
- Path: A sequence of nodes $p = [n_1, n_2, \ldots, n_k]$
  - Each pair of nodes is an edge
  - Length: (k-1) - the number of edges
- Subpath: A subsequence of nodes in p
- *Reach*(n): Subgraph that can be reached from n

控制流图

- **控制流图（CFG）**
  - 图中每个点（基本块）代表程序中的一系列顺序运算
  - 每条边代表两个节点之间的一个转移
- **控制流图有三种基本结构**
  - 顺序结构
  - 分支结构（**if-then-else, switch-case**）
  - 循环结构（**while, do-while, for**）
- goto语句会破坏程序的结构
  - 程序中尽量避免使用break, continue 等

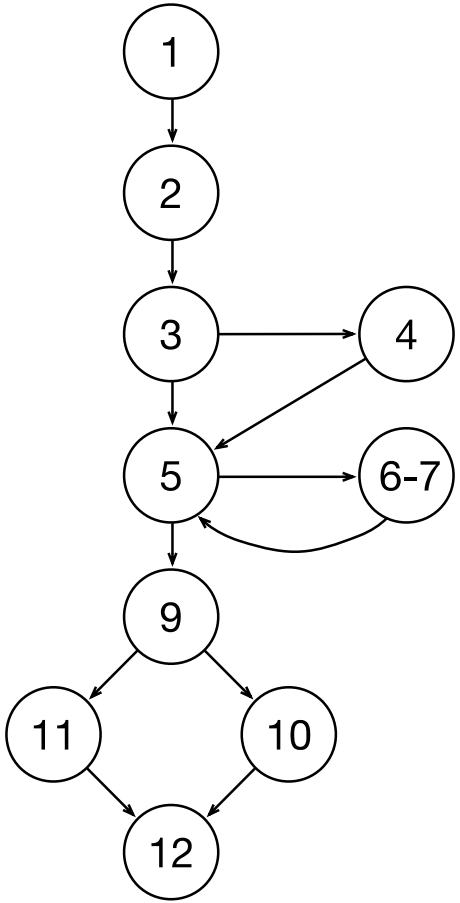
Example. A program and its flow graph

```
          int x, y;
/*S1*/    if (x < 5)
/*S2*/      y = 2;
/*S3*/    while ((x < 5)  && y)
/*S4*/      { x++; y--; }
/*S5*/    return;
```



Another Example

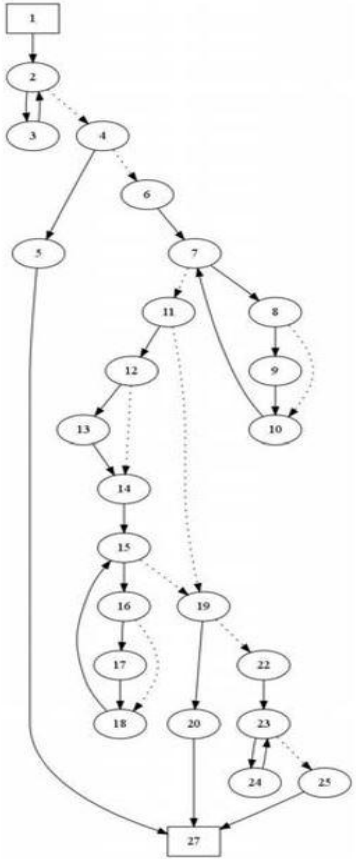
```
1 int maxsum (int maxint, value) {
2     int result = 0, i = 0 ;
3     IF (value < 0)
4     THEN value = - value ;
5     WHILE ((i < value)
        AND (result <= maxint))
6      {   i = i + 1 ;
7          result = result + i ;
8      }
9      IF (result <= maxint)
10    THEN OUTPUT (result);
11    ELSE OUTPUT ("too large");
12 }
```



Yet Another Example (real program)

```
int getop(s, lim)
    char s[];
    int  lim;
{
    int i, c;
    while ((c = getchar()) == ' ' || c == '\t' || c == '\n') ;
    if (c != '.' && (c < '0' || c > '9'))
        return(c);
    s[0] = c;
    for(i = 1; (c=getchar()) >= '0' && c <= '9'; i++)
        if (i < lim) s[i] = c;
    if (c == '.') {if (i < lim)    return(c); …}
    …
}
```

Test Paths




```
1:  c=getchar();
2:  !(((c==32 || c==9) || c==10))
3:  c=getchar();
4:  !((c!=46 && (c<48 || c>57)))
5:  RETURN: c;
6:  s[0]=c; c=getchar(); i=1;
7:  !((c>=48 && c<=57))
8:  !(i<lim)
9:  s[i]=c;
10: c=getchar(); _temp_var0=i; i=i+1;
11: !(c==46)
12: !(i<lim)
13: s[i]=c;
14: c=getchar(); _temp_var1=i; i=i+1;
15: !((c>=48 && c<=57))
16: !(i<lim)
17: s[i]=c;
18: c=getchar(); _temp_var2=i; i=i+1;
19: !(i<lim)
20: s[i]=0; RETURN: 1000;
22: NOP
23: !((c!=10 && c!=-1))
24: c=getchar();
25: _temp_var3=lim-1; s[_temp_var3]=0; RETURN: 9999;
27: END
```

Fig. 1. getop()

Selected paths from its flow graph

- **Path 1**

  1 → 2 → 4 → 5 → 27.

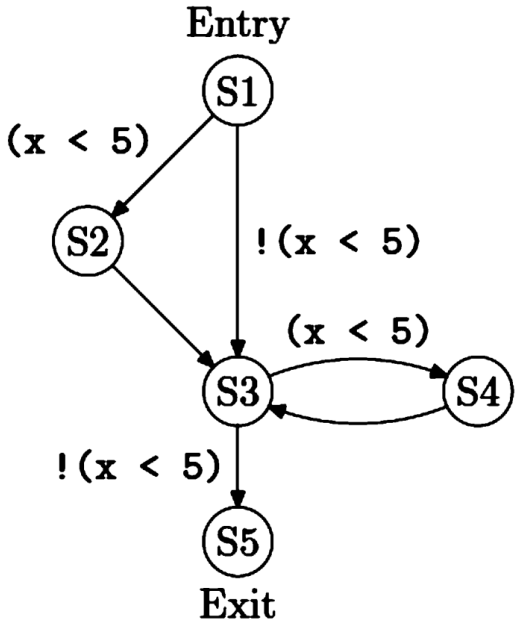
- **Path 2**

  1 → 2 → 4 → 6 → 7 → 11 → 19 → 20 → 27.

测试路径

- CFG上的路径
  - 完整路径：*A path that starts at an initial node and ends at a final node*
  - 例：S1-S2-S3-S4-S3-S5
- 不可行路径
  - S1-S3-S4-S3-S5
- 不可达点（死代码）
  - 所有从起点到该点的路径都是不可行的

```
void f(int x）{
/*S1*/ if (x < 5）
/*S2*/ y = 2;
/*S3*/ while (x < 5）
/*S4*/ x++;
/*S5*/ return;
}
```



基于控制流图的结构测试

- 找到一组测试用例（带有输入数据），以覆盖控制流图中的一些基本元素。

- Rationale: it is impossible to detect a fault in some piece of code by testing, if that code is never executed.

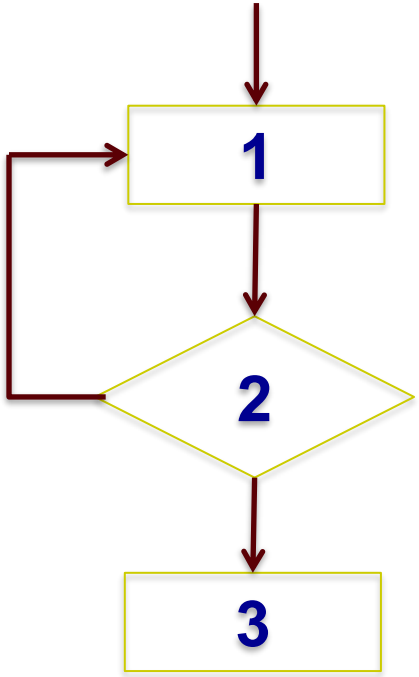
覆盖准则

- 语句覆盖（Statement Coverage）
  - 程序中的所有语句都被执行
  - 覆盖控制流图中所有的节点
- 分支覆盖（Branch Coverage）
  - 控制转移都得到了覆盖
  - 测试路径覆盖了控制流图中所有的边
  - 对应于迁移条件表达式的判定覆盖
- 路径覆盖（Path Coverage）
  - 覆盖控制流图中所有的完整路径
  - 进一步定义可行路径覆盖(Feasible Path Coverage)
  - （可行）路径数量可能是无限的

语句与分支覆盖

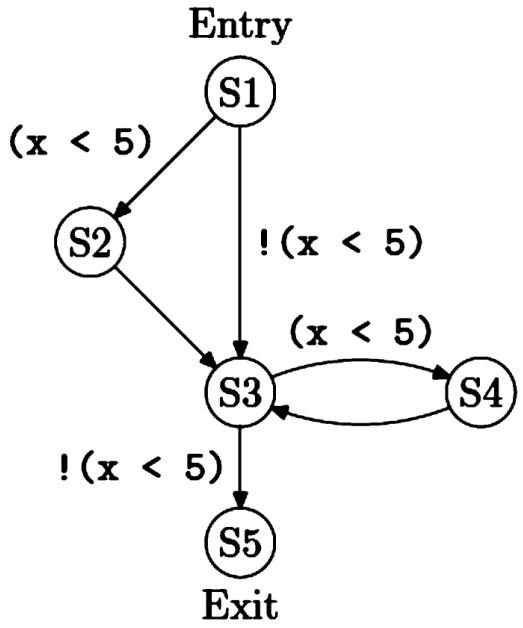
Statement coverage: Path 1, 2, 3

Branch coverage: Path 1, 2, 1, 2, 3

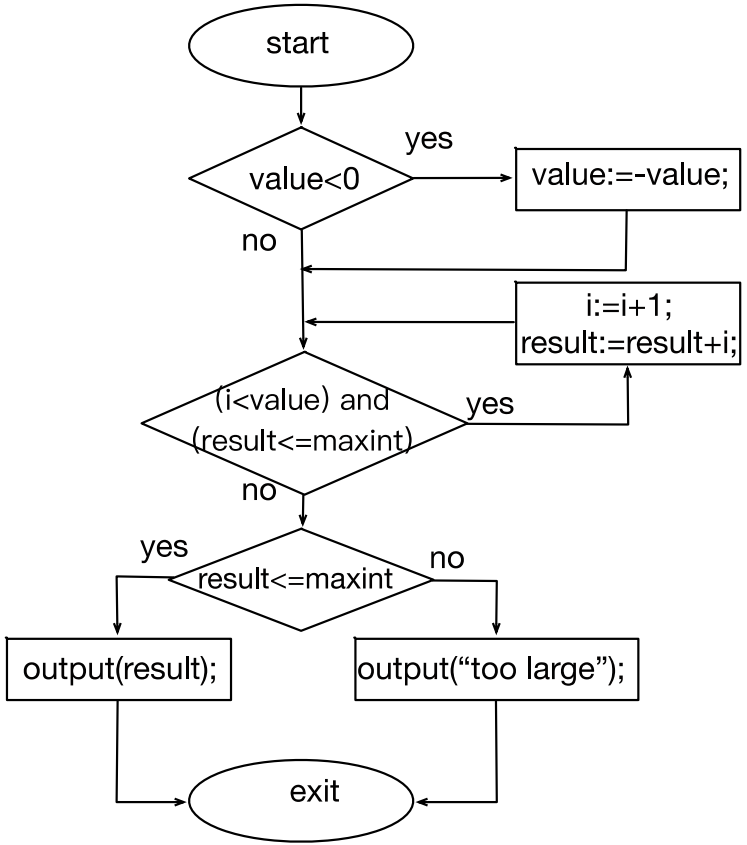


- 语句覆盖
  S1-S2-S3-S4-S3-S5

- 分支覆盖（两条路径）
  S1-S2-S3-S4-S3-S5
  S1-S3-S5



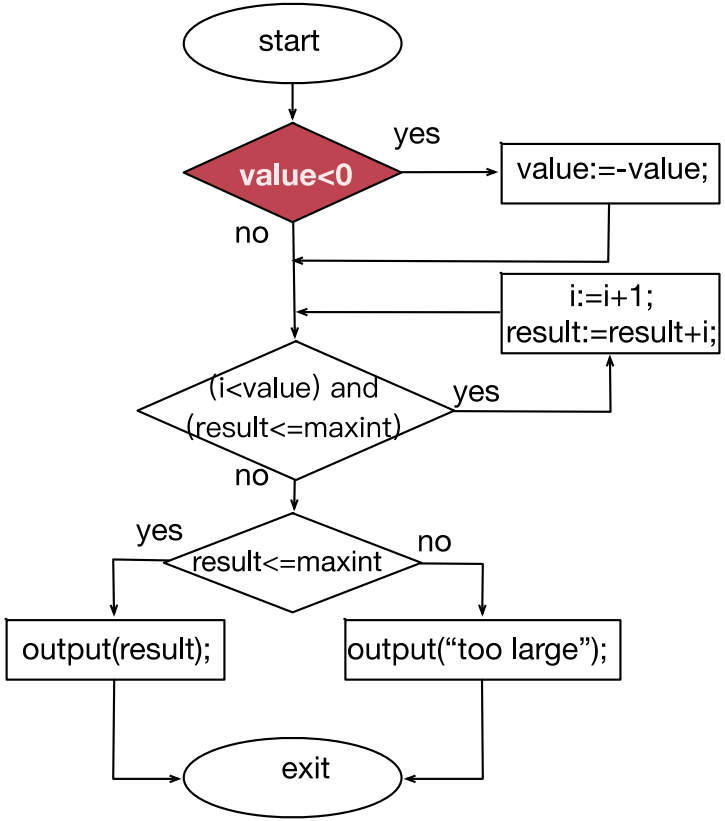
Example: Statement Coverage



Test cases for 100% statement coverage:

maxint	value
10	-1
0	-1

Example : Branch Coverage



Test cases for 100% branch coverage:

maxint	value
10	2
0	-1

路径覆盖准则

- 长度*n*准则（Length-*n*）
  - 测试路径集合覆盖所有长度不超过*n* 的子路径
  - Length-0 对应语句覆盖, Length-1 对应分支覆盖
- 简单路径覆盖（Simple Path Coverage）和初等路径覆盖（Elementary Path Coverage）
  - a path that has no repeated occurrence of any *edge* is called a simple path in graph theory
  - a path that has no repeated occurrences of any *node* is called an elementary path.
- Level-*i* 覆盖
  - The criterion starts with testing all elementary paths from the begin node to the end node. Then, if there is an elementary subpath or cycle that has not been exercised, the subpath is required to be checked at the next level. This process is repeated until all nodes and edges are covered by testing.

先测试所有从开始节点到结束节点的初等路径；
如果发现还有没被覆盖的初等子路径或循环，就提升一个 level，把它们纳入测试；
如此反复，直到所有节点和边都被覆盖。

循环体测试

- 循环 *K* 次准则
  - 最常用：0-1循环覆盖准则
- 循环体有三种:
  - 简单循环 (Simple Loops）
  - 嵌套循环 (Nested loops）
  - 串联循环 (Concatenated Loops）
- 对每个循环分别执行循环 *K* 次准则.

程序复杂度

- **Size:**

  LOC 是最直观的代码行数，但它又被细分成 SLOC（源代码长度）和 LLOC（逻辑代码长度）

  - **length:**
    - **LOC: SLOC (source program length)、LLOC (logic)**
    - **#components: number of files, classes**
  - **Amount of functionality: functional points (specification-based)**
- **Structure:**
  - **Control flow -- McCabe's**
  - **Data flow**
  - **Modularity**

圈复杂度 (Cyclomatic Complexity)

- 欧拉公式：G是连通平面图，那么
  $\varphi = e - n + 2$
  φ 是图中不同区域的个数 – 圈
  e 是G中的边数
  n 是G中的节点数
- 图的圈复杂度由$V(G) = e - n + 2p$给出，其中
  e 是G中的边数
  n 是G中的节点数
  p 是G中的连通分量数

圈复杂度 (Cyclomatic Complexity)

- 程序的控制流图 → 圈复杂度
  - 增加一个分支，控制流图的区域数+1
  - 如果所有分支都是二叉的，那么 *V*(*G*)是判定节点数+1
- 圈复杂度代表了程序的逻辑复杂度
- 经验表明，程序中可能存在的Bug数和圈复杂度有着很大的相关性
- 测试应该和圈复杂度关联

基本路径测试

- 路径向量$A = \langle a_1, a_2, \ldots, a_k \rangle$， a_i 表示编号为 i 的有向边在路径 A 中出现的次数
- 路径B称为路径 $A_1, A_2, \ldots, A_n$ 的线性组合，如果存在常数 $\lambda_1, \lambda_2, \ldots, \lambda_n$ 使得 $B = \lambda_1 A_1 + \lambda_2 A_2 + \ldots + \lambda_n A_n$
- 对于一个测试路径集合来说，如果流图中的任意一个完整路径都是测试集合中路径的线性组合，那么这个测试集合满足基本路径覆盖（Basis Path Coverage）。

基本路径测试

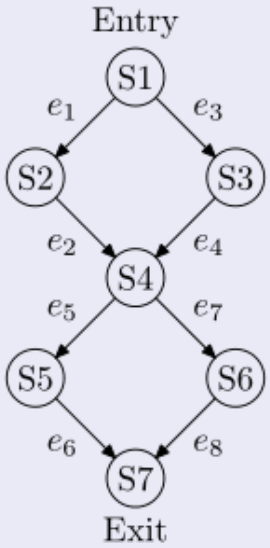
- 最优测试集合
  - 各条测试路径是线性无关的
  - 对应于线性空间的基(basis)
  - 含有 $V(G) = |E| - |N| + 2$ 条线性独立的完整路径

例

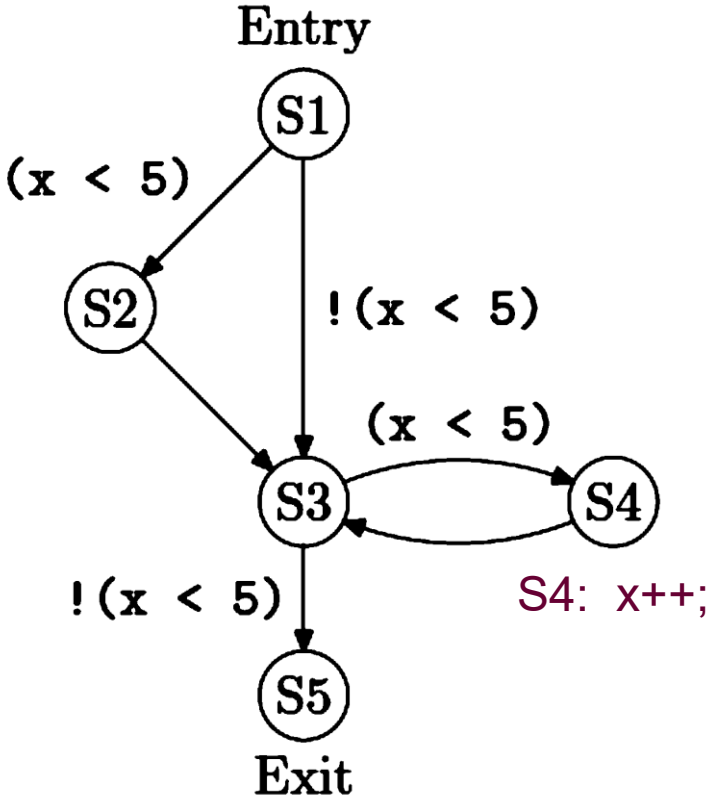
$$p_1 = e_1 e_2 e_5 e_6 = \langle 1,1,0,0,1,1,0,0 \rangle$$
$$p_2 = e_1 e_2 e_7 e_8 = \langle 1,1,0,0,0,0,1,1 \rangle$$
$$p_3 = e_3 e_4 e_5 e_6 = \langle 0,0,1,1,1,1,0,0 \rangle$$
$$p_4 = e_3 e_4 e_7 e_8 = \langle 0,0,1,1,0,0,1,1 \rangle$$

任意一条路径都可以由其他三条路径线性组合得出, 如

$$p_4 = -p_1 + p_2 + p_3$$



思考



- 左图中的程序，其基本路径集合是什么？

- S1-S3-S5
- S1-S2-S3-S4-S3-S5
- S1-S2-S3-S4-S3-S4-S3-S5

基于程序文本的测试

- **LCSAJ**
  - 线性代码序列与跳转（Linear Code Sequence And Jump）
  - 一组顺序执行的代码, 以控制流跳转为其结束点
  - 几个首尾相接, 且第一个 LCSAJ 起点为程序起点，最后一个 LCSAJ 终点为程序终点的 LCSAJ 串就组成了程序的一条路径.
- **TER$_n$**
  - 第一层n=1是语句覆盖;
  - 第二层是分支覆盖;
  - 第三层就是LCSAJ覆盖，即程序中的每一个LCSAJ都至少在测试中经历过一次;
  - 而TER$_{n+2}$要求每n个首尾相连的LCSAJ组合在测试中都要经历一次.

LCSAJ - Example

```
 1:#include <stdlib.h>
 2:#include <string.h>
 3:#include <math.h>
 4:#define MAXCOLUMNS 26
 5:#define MAXROWS 20
 6:#define MAXCOUNT 90
 7:#define ITERATIONS 750
 8:int main(void){
 9:   int count = 0, totals[MAXCOLUM
10:   memset(totals, 0, MAXCOLUMNS *
11:   while(count < ITERATIONS){
12:      val = abs(rand()) % MAXCOLUM
13:      totals[val] += 1;
14:      if(totals[val] > MAXCOUNT){
15:         totals[val] = MAXCOUNT;
16:      }
17:      count++;
18:   }
19:   return(0);
20:}
```

来看一个极简程序：

```c
1: x = 0;
2: y = 1;
3: if (x < 5)
4:     y = y + 1;
5: z = y;
```

现在我们从"源代码执行"的角度拆。

- 第 1、2 行：顺序执行，没有任何跳转
- 第 3 行：这是一个**控制流跳转点**（if）

于是，从第 1 行开始，到第 3 行为止，就构成了一个 **LCSAJ**：

👉 **LCSAJ₁ = ⟨1, 3, true⟩ 或 ⟨1, 3, false⟩**

含义是：

"从第 1 行开始，线性执行到第 3 行，然后根据条件跳转"

如果条件为 true，会跳到第 4 行；

如果为 false，会跳到第 5 行。

接着看 true 分支：

```c
4: y = y + 1;
5: z = y;
```

这里又是一段**线性顺序代码**，直到程序结束，没有跳转，于是：

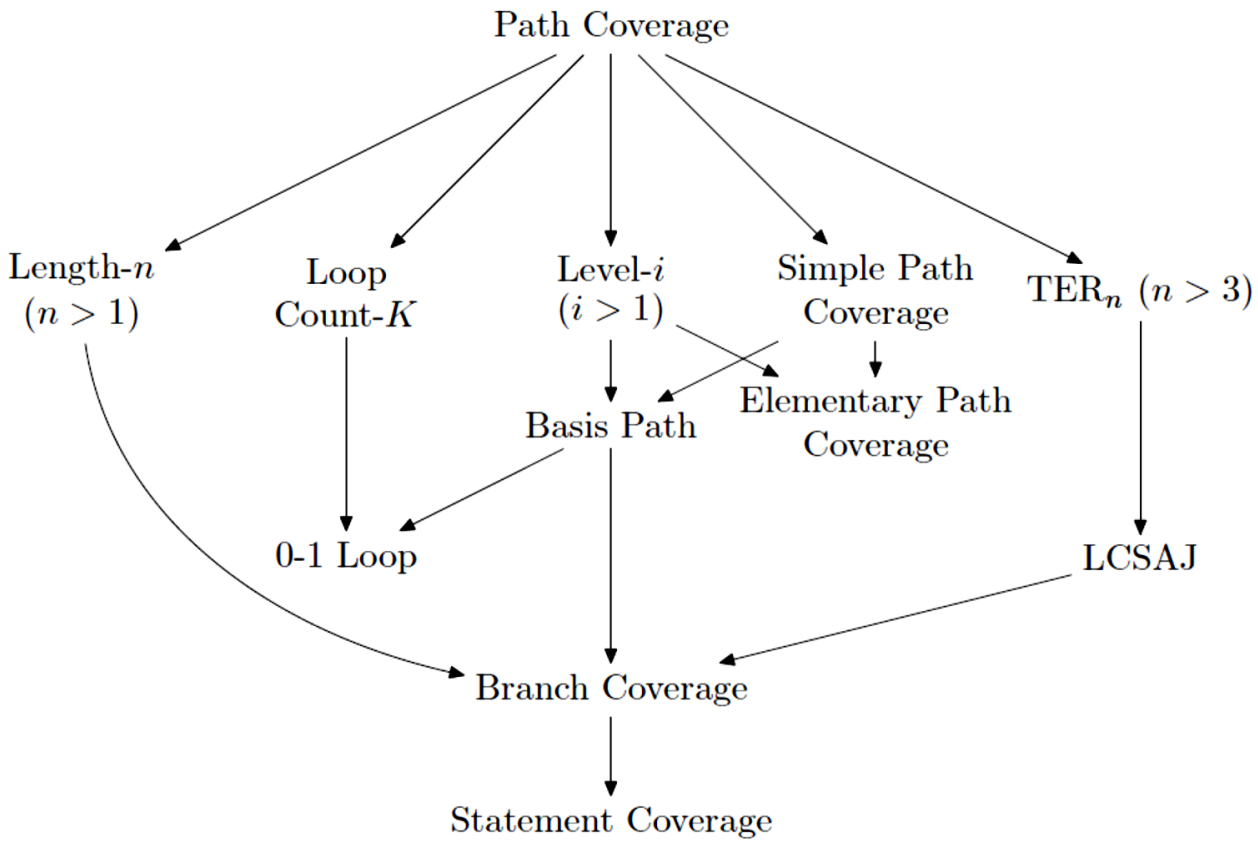
👉 **LCSAJ₂ = ⟨4, 5, exit⟩**

false 分支则是：

👉 **LCSAJ₃ = ⟨5, 5, exit⟩**

Start	End	Jump To
8	11	19
8	14	17
8	18	11
11	11	19
11	14	17
11	18	11
17	18	11
19	19	-1

控制流测试准则之间的关系



数据流测试

数据流测试--基本概念

- 定义性出现（Definition Occurrence）
  y = x + z
  `scanf("%d %d", &x, &y)` 定义变量 x和y
- 引用性出现（Use Occurrence）
  - 计算性引用（c-use）
    y = x + z
  - 谓词性引用（p-use）
    if (x1 < x2)
- Def of a variable at line l_1 and its use at line l_2 constitute a def-use pair. (l_1 and l_2 can be the same.)
- 无定义的（Def-Clear）
- du-path：定义到引用之间是def-clear的

Rapps和Weyuker的数据流测试准则

- 定义覆盖准则（All-Defs Criterion） 存在一条测试路径覆盖从变量的定义性出现传递到某一个引用性出现
- 引用覆盖准则（All-Uses Criterion） 对于每一个变量的每一个定义性出现，每一个能够可行的传递到的引用，存在一条测试路径覆盖该传递和引用
- 定义-引用覆盖准则（All-DU-Paths Criterion） 所有的定义到引用的传递路径都检查 （在循环上使用0-1准则）。
- 计算性引用覆盖（All-C-Uses），谓词性引用覆盖（All-P-Uses）

Example 1

```
1.  begin
2.  int x,y; float z;
3.  input(x,y);
4.  z = 0;
5.  if (x != 0)
6.    z = z+y;
7.  else z = z-y;
8.  if (y != 0)
    // should be (y != 0 &&
    x != 0)
9.    z = z/x;
10. else z = z*x;
11. output(z);
12. end
```

满足分支覆盖的测试用例

	x	y	z
t_1	0	0	0.0
t_2	1	1	1.0

变量z的定义：4，6，7，9，10
变量z的使用：6，7，9，10

Example 1

```
1.  begin
2.  int x,y; float z;
3.  input(x,y);
4.  z = 0;
5.  if (x != 0)
6.    z = z+y;
7.  else z = z-y;
8.  if (y != 0)
9.    z = z/x;
10. else z = z*x;
11. output(z);
12. end
```

对于变量Z的du-path覆盖测试用例

	x	y	z	DU pairs covered
t_1	0	0	0.0	(4,7), (7,10)
t_2	1	1	1.0	(4,6), (6,9)
t_3	0	1	0.0	(4,7), (7,9)
t_4	1	0	1.0	(4,6), (6,10)

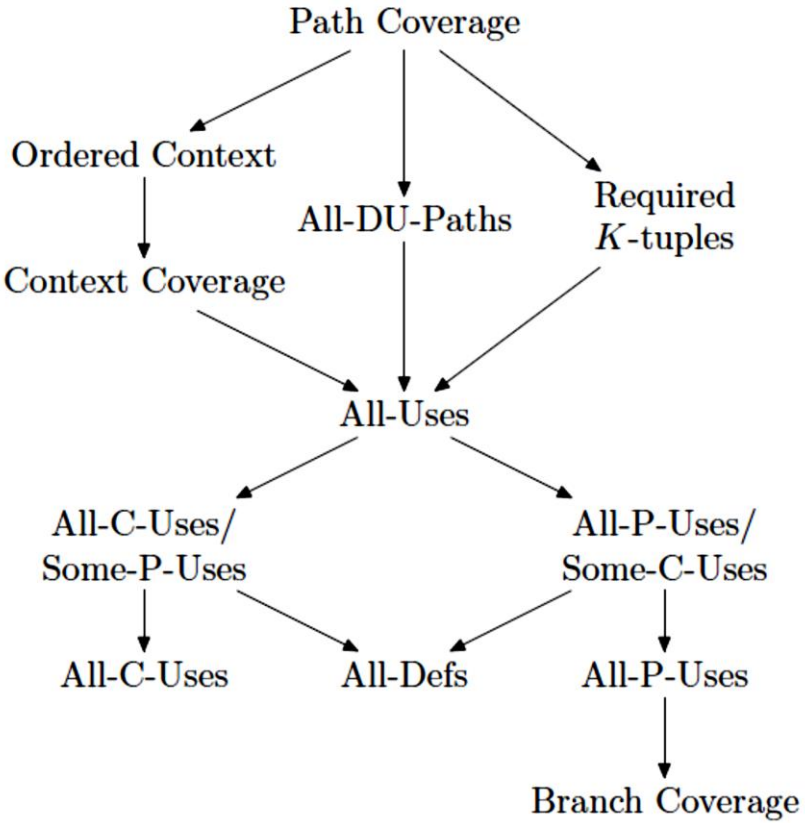
(l_1,l_2): *z* is defined in l_1 and used in l_2.

Ntafos的数据流测试准则

考虑数据在程序中的传播过程

- *k*元数据交互链（*k-dr* interaction），交互路径 $n_1, n_2, \ldots, n_k$
  如果节点n_i上有变量x_i的定义，节点n_{i+1}上有x_i的引用性出现，该引用被用来计算赋予x_{i+1}的数据。从定义到引用，再定义到新的变量的过程不断重复，使得数据在变量之间传播。
- *k*元数据交互链覆盖准则（Required *k*-Tuples criterion）
- 2元数据交互链覆盖准则即为引用覆盖准则。

数据流测试准则之间的关系



- 关于覆盖率

Hierarchy (Aditya P. Mathur)

- 需求 (requirement)
- 函数 (function/method)
- 语句 (statement)
- 判定 (decision)
- 数据流 (data-flow)
- …

错误检测率与覆盖率增长的关系（Gittens）

- 经验研究
  - 商用软件，221,629行，221个模块，
  - 代码块覆盖
- 饱和效应
  - 在系统测试中，覆盖率达到51%-60%之前，错误检测率与覆盖率相关
  - 回归测试中，覆盖率达到61%-70%之前，错误检测率与覆盖率相关

覆盖率统计工具：llvm-cov

- 编译
  - clang -fprofile-instr-generate -fcoverage-mapping test.cc -o test
- 运行
  - LLVM_PROFILE_FILE=“test.profraw" ./test
- 对profraw文件索引
  - llvm-profdata merge -sparse test.profraw -o test.profdata
- 生成覆盖率报告
  - llvm-cov show ./test -instr-profile=test.profdata
  - llvm-cov report -show-region-summary=false ./test -instr-profile=test.profdata

覆盖率报告

- 第一列为程序的行号
- 第二列为该行代码被覆盖的次数
- Branch (line : column) 标记了分支的位置
- [True:10 , False:1]表示满足条件和不满足条件的覆盖次数
- 标红的行表示未被测试用例覆盖

```
  1|       |#include <stdio.h>
  2|       |
  3|       |int main(void)
  4|      1|{
  5|      1|    int i,total;
  6|      1|    total = 0;
  7|     11|    for(i=0;i<10;i++)
------------------
|  Branch (7:13): [True: 10, False: 1]
------------------
  8|     10|        total += i;
  9|      1|    if(total != 45)
------------------
|  Branch (9:8): [True: 0, False: 1]
------------------
 10|      0|        printf("Failure\n");
 11|      1|    else
 12|      1|        printf("Success\n");
 13|       |    /*else
 14|       |        printf("just test\n");*/
 15|      1|    return 0;
 16|      1|}
 17|       |
```

覆盖率总结

- Functions为程序中所含函数数量
- Missed Functions 为未被覆盖的函数
- Executed 为函数覆盖率
- Lines 代码总行数
- Missed Lines 未被覆盖的行的数量
- Cover 行覆盖率
- Branches 条件总数
- \- Missed Branches 未被覆盖了条件的数量

Filename	Functions	Missed Functions	Executed	Lines	Missed Lines	Cover	Branches	Missed Branches	Cover
/home/hurx/test/test.c	1	0	100.00%	11	1	90.91%	4	1	75.00%
TOTAL	1	0	100.00%	11	1	90.91%	4	1	75.00%

覆盖率总结报告

覆盖率与找错

- 测试即使达到100%的语句/分支覆盖率，也可能会漏掉很多错误。
- 例.
  - int a[10], i, j;
  - INPUT(i);
  - if (i > 4) j = i-1;
  - else j = i+1;
  - a[j] = j; // 可能越界

两个测试用例
1. i = 4
2. i = 5

Example 2

```
float x, y, z = 0.0; //def z
int count;
input(x, y, count);
do {
  if (x <= 0)
    if (y >= 0) {
      z = y*z+1;
    }
    else
      z = 1/x ;
  }
  y = x*y + z;
  count = count -1;
} while (count > 0);
output(z);
```

T1:

<x= -2, y=2, count=2>

T2:

<x= 2, y=2, count=1>

这两个测试用例满足100%语句、分支、MC/DC覆盖

巧合正确性

- Coincidental Correctness
- 执行了语句/路径，但未发现其中的错误。

- 例：
  - y = x * 2;
  - y = x ^ 2;

    - 如果输入数据 x==2, …

参考文献

- Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software unit test coverage and adequacy. ACM Comput. Surv. 29, 4 (December 1997), 366-427.
- 朱鸿、金凌紫，《软件质量保障与测试》，科学出版社，1997。
- A. P. Mathur, Foundations of Software Testing, 2008.
- Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, Zhendong Su: A Survey on Data-Flow Testing. ACM Comput. Surv. 50(1): 5:1-5:35 (2017)

参考文献

- Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software unit test coverage and adequacy. ACM Comput. Surv. 29, 4 (December 1997), 366-427.
- 朱鸿、金凌紫，《软件质量保障与测试》，科学出版社，1997。
- A. P. Mathur, Foundations of Software Testing, 2008.
- Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, Zhendong Su: A Survey on Data-Flow Testing. ACM Comput. Surv. 50(1): 5:1-5:35 (2017)