西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

第四章　白盒测试方法

授课教师：　郑炜

第四章 白盒测试方法

- 4.3 路径分析与测试
- 4.4 数据流测试分析

4.4.1 测试充分性基础

4.4.2 测试充分性准则的度量

4.4.3 测试集充分性的度量

4.4.4 数据流概念

4.4.5 基于数据流的测试充分性准则

- 4.5 变异测试
  - 4.5.1 变异和变体
  - 4.5.2 强变异和弱变异
  - 4.5.3 用变异技术进行测试评价
  - 4.5.4 变异算子
  - 4.5.5 变异算子的设计
  - 4.5.6 变异测试的基本原则

4.1.1 基本块

基本块：

如果P是一个由过程式程序设计语言（如C语言）编写的程序，那么只有一个入口块和出口块的连续语句序列就可以被认为是一个基本块。

一个基本块只有唯一的入口块和出口块，这个入口块即为基本块的第1条语句，出口块是最后一条语句。程序的控制从入口块进入，从出口块退出，除此之外程序不能在基本块其他点退出或是中止。如果基本块仅有一条语句，那么认为入口和出口是重合的。

4.1.1 基本块

例4.1：示例程序

```
1    begin
2    int x,y,temp;
3    float z;
4    input(x,y);
5    if(x>y)
6    temp=x+y;
7    else
8    temp=x-y;
9    z=temp+5
10   end
```

（1）第1行到第5行就构成了一个基本块，第1行是唯一的入口块，第5行是唯一的出口块。

（2）还有某些程序分析工具把单条过程调用语句当作一个单独的基本块。在这样的定义下，可以把例4.1中的input语句也当作一个基本块。

（3）函数调用本身常常被当作基本块，由于这些调用语句会造成控制程序从当前执行的函数转移到别的地方，很容易导致程序的非正常终止。若无特别说明，则认为函数调用和其他顺序语句是相同的，即认为它们都不会引起程序中止。

4.1.2 流图的定义与图形表示

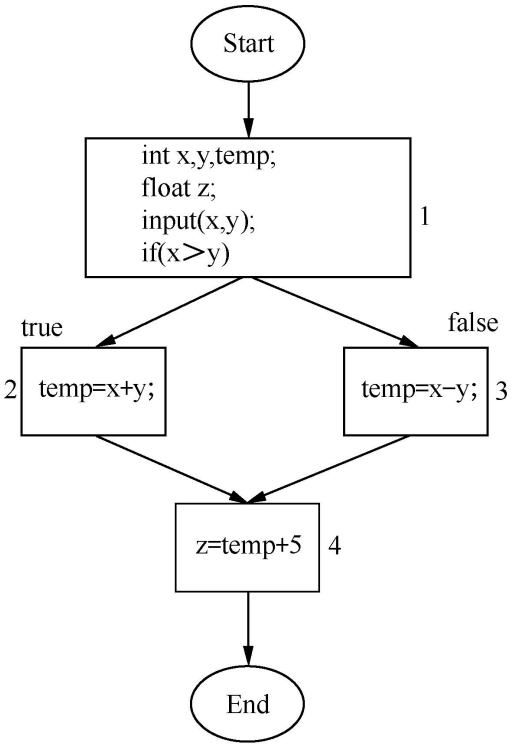
流图

G=(N,E)表示流图G，其中N是节点的有限集合，E是有向边的有限集合。Star出发节点，End终止节点。

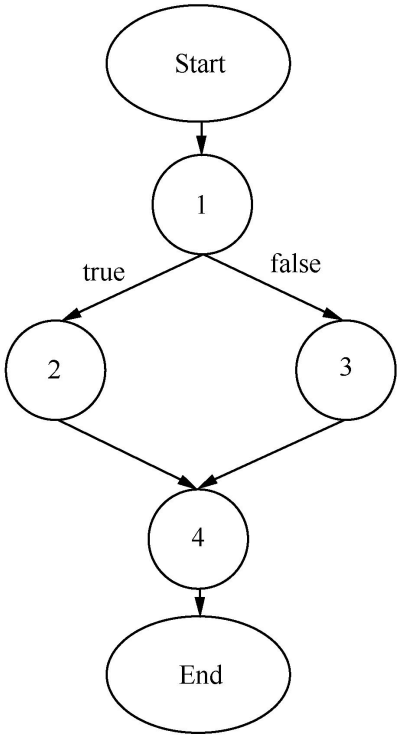
- 节点
  - 椭圆或矩形框
  - 表示基本块
- 箭头
  - 控制流的方向
- 例4.1的图形表示

N={Start,1,2,3,4,End}

E={(Start,1),(1,2),(1,3),(2,4)(3,4),(4,End)}



（a）显示基本块中的语句



（b）省去基本块中的语句

4.2.1 测试覆盖率

- 测试覆盖率：用于确定测试所执行到的覆盖项的百分比，其中覆盖项是指作为测试基础的一个入口或属性，如语句、分支、条件等。

- 测试覆盖率是对测试充分性的表示，它可以作为在测试分析报告中的一个可量化的指标依据，一般认为测试覆盖率越高，测试效果越好。但是测试覆盖率并非测试的绝对目标，而只是一种手段。

4.2.1 测试覆盖率

- **常用的测试覆盖评测**

1. 基于需求的测试覆盖

基于需求的测试覆盖在测试生命周期中要评测多次，并在测试生命周期的里程碑处提供测试覆盖的标识（如已计划的、已实施的、已执行的和成功的测试覆盖）。

2. 基于代码的测试覆盖

基于代码的测试覆盖评测测试过程中已经执行的代码的多少，与之相对的是要执行的剩余代码的多少。基于代码的测试覆盖可以建立在控制流（语句、分支或路径）或数据流的基础上。

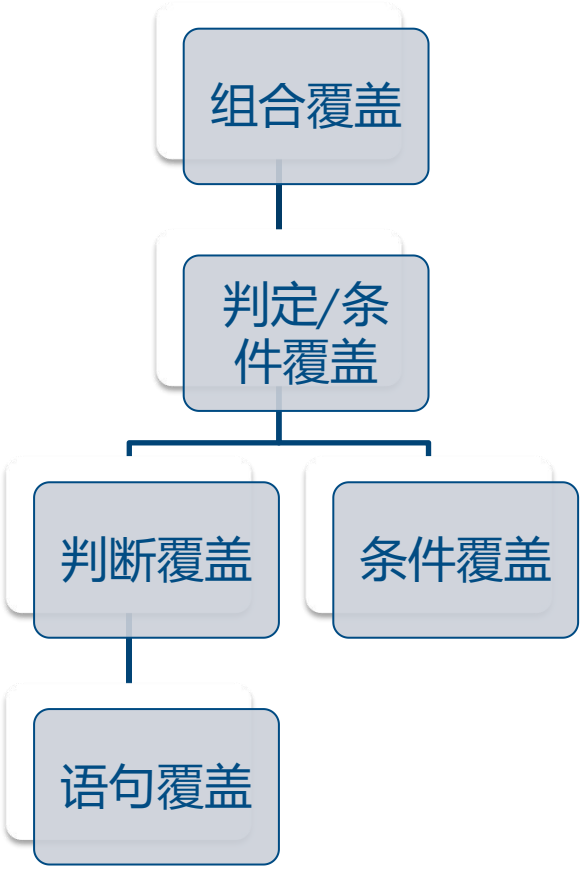
4.2.2 逻辑覆盖

覆盖类型	覆盖标准
语句覆盖	每行可执行语句至少执行一次
判断覆盖	每个真假分支至少被执行一次
条件覆盖	每个判断中的每个条件的可能取值至少满足一次
条件/判断组合覆盖	每个判断中的每个条件的可能取值至少满足一次 每个判断至少获得一次“真”和一次“假”。
多条件覆盖	每个判断中的每个条件的各种可能组合至少出现一次

4.2.2 逻辑覆盖

覆盖类型	覆盖标准
修正条件覆盖	• 每个程序模块的入口和出口点都至少要被调用一次，每个程序的判定到所有可能的结果值要至少转换一次； • 程序的判定被分解为通过逻辑操作符“and”和“or”连接的布尔条件，每个条件对于判定的结果值是独立的
组合覆盖	每个判定的所有可能的条件取值都至少出现一次。
路径覆盖	所有可执行路径至少执行一次

4.2.3 逻辑覆盖准则之间的包含关系



●测试覆盖准则

1.错误敏感测试用例分析（Error Sensitive Test Cases Analysis，ESTCA）

- 错误敏感测试用例分析规则

[规则1] 对于A rel B（rel可以是<、=或>）型的分支谓词，应适当地选择A与B的值，使得测试执行到该分支语句时，A < B、A = B和A > B的情况分别出现一次。

[规则2] 对于A rel C（rel可以是>或<，A是变量，*C*是常量）型的分支谓词，当rel为<时，应适当地选择*A*的值，使*A* = *C*−*M*（*M*是距离*C*最小的容器容许正数，*A*和*C*均为整型时，*M* = 1）。同样，当rel为>时，应适当地选择*A*，使*A* = *C* + *M*。

[规则3] 对外部输入变量赋值，使其在每一测试用例中均有不同的值与符号，并与同一组测试用例中其他变量的值与符号不一致。

4.2.4 测试覆盖准则

- **测试覆盖准则**

2.线性代码序列与跳转（Linear Code Sequence and Jump，LCSAJ）。

- 层次LCSAJ覆盖准则

第1层：语句覆盖。

第2层：分支覆盖。

第3层：LCSAJ覆盖，即程序中的每个LCSAJ都至少在测试中经历过一次。

第4层：两两LCSAJ覆盖，即程序中的每两个相连的LCSAJ组合起来在测试中都要经历一次。

……

第n+2层：每n个首尾相连的LCSAJ组合在测试中都要经历一次。

在实施测试时，若要实现上述的层次LCSAJ覆盖，需要产生被测程序的所有LCSAJ。

- 4.3 路径分析与测试
- 4.4 数据流测试分析
  - 4.4.1 测试充分性基础
  - 4.4.2 测试充分性准则的度量
  - 4.4.3 测试集充分性的度量
  - 4.4.5 基于数据流的测试充分性准则

路径测试

- 路径测试（Path Testing）是指根据路径设计测试用例的一种技术，经常用于状态转换测试中。
- 基本路径测试法是在程序控制流图的基础上，通过分析控制构造的环路复杂性，导出基本可执行的路径集合，从而设计测试用例的方法。
- 设计出的测试用例要保证在测试中程序的每个可执行语句至少执行一次借助漏洞扫描工具测试

基本路径测试步骤

（1）画出程序的控制流图。

（2）计算程序的圈复杂度，导出程序基本路径集中的独立路径条数，这是确定程序中每个可执行语句至少执行一次所需的测试用例数量的上界。

（3）导出基本路径集，确定程序的独立路径。

（4）根据步骤（3）中的独立路径，设计测试用例的输入数据和预期输出结果。

示例程序：例4.2

```
void Sort ( int iRecordNum, int iType )
1   {
2       int x=0;
3       int y=0;
4       while ( iRecordNum-- > 0 )
5       {
6           if ( iType==0 )
7             {x=y+2; break; }
8           else
9             if ( iType==1 )
10              x=y+10;
11            else
12              x=y+20;
13      }
14  }
```

（1）画出程序的控制流图，如图4-2

（2）计算程序的圈复杂度

圈复杂度的定义：

$$V(G)= E - N + 2P$$

*E*表示控制流图的边的数量；
*N*表示控制流图的节点数；
*P*表示控制流图中相连接的部分

$$V(G)=10-8+2=4$$

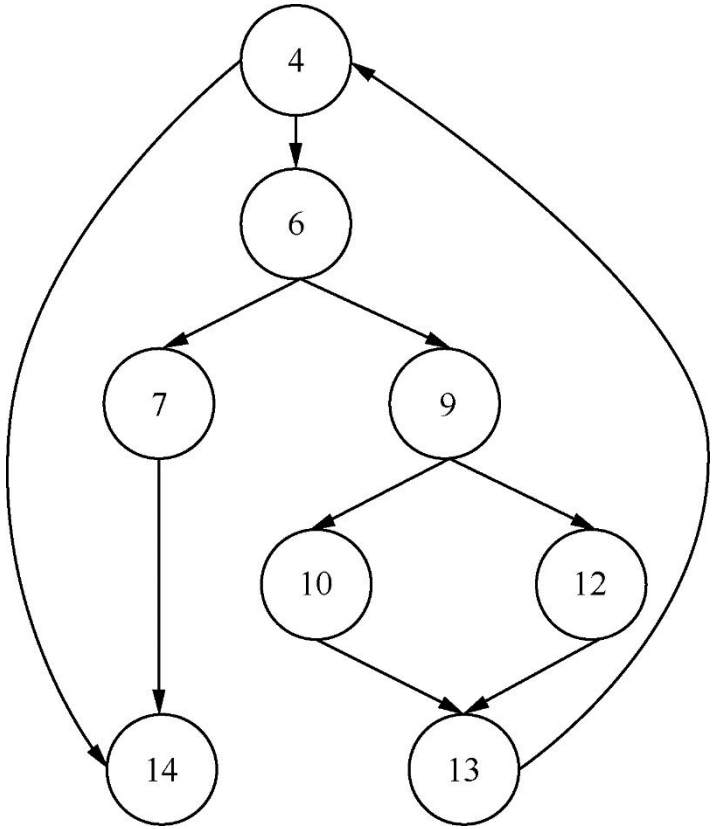


图4-2 例4.2的控制流图

(3) 导出独立路径

路径1：4→14
路径2：4→6→7→14
路径3：4→6→9→10→13→4→14
路径4：4→6→9→12→13→4→14

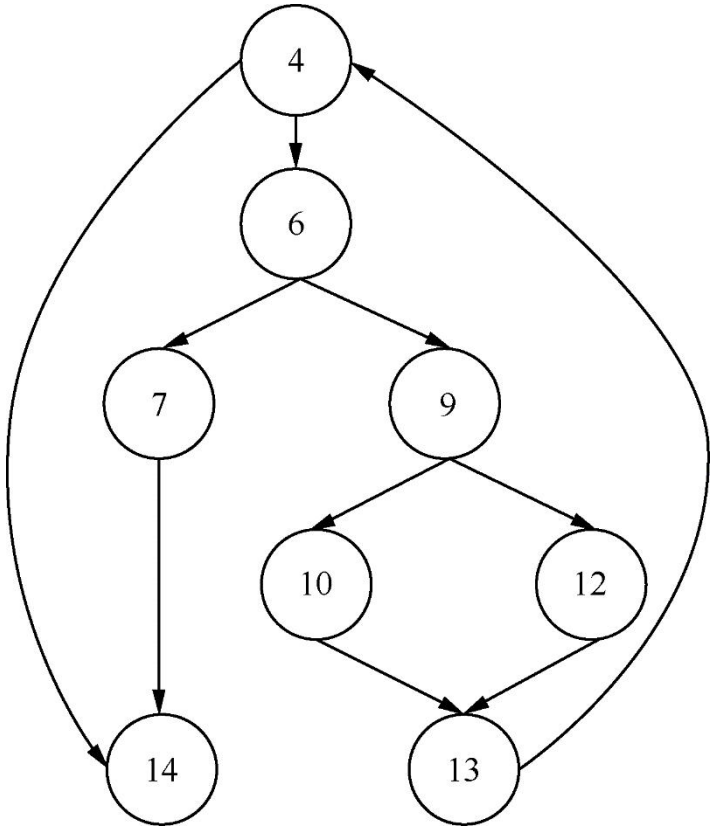


图4-2 例4.2的控制流图

(4) 设计测试用例

	输入数据	预期输出
测试用例1	irecordnum = 0 itype = 0	x = 0 y = 0
测试用例2	irecordnum = 1 itype = 0	x = 0 y = 0
测试用例3	irecordnum = 1 itype = 1	x = 10 y = 0
测试用例4	irecordnum = 1 itype = 2	x = 0 y = 20

- 4.3 路径分析与测试
- 4.4 数据流测试分析

4.4.1 测试充分性基础

4.4.2 测试充分性准则的度量

4.4.3 测试集充分性的度量

4.4.4 数据流概念

4.4.5 基于数据流的测试充分性准则

4.4.1测试充分性基础

测试充分性准则描述有谓词和度量函数两种形式。

- 谓词形式的充分性准则描述将充分性定义为一个谓词，用于确定测试数据必须具备什么性质才是一个彻底的测试；
- 度量函数形式的充分性准则将充分性描述为测试的充分程度，是一种更广义的充分性准则定义。

4.4.1测试充分性基础

- **定义4.1 谓词形式的充分性准则**

设P为被测程序，D是一可数集合，为P的输入域，对$d \in D$，$P(d)$为程序的输出；S是$D \to P(D)$二元映射关系的一个子集，表示软件规格说明的集合；T是测试数据集合的集合，则谓词形式的软件测试充分性准则C是一个定义在$P \times S \times T$上的谓词，即

$$C: P \times S \times T \to \{\text{true,false}\}$$

$C(p,s,t)$=true表示根据充分性准则C和规格说明s，用测试数据集t测试程序p是充分的，否则是不充分的。

- **定义4.2 度量函数形式的充分性准则**

一个测试数据的充分性准则M是$P \times S \times T$到区间［0,1］上的映射，用数学形式可表示为$M: P \times S \times T \to$［0,1］。$M(s, p, t)=r$表示根据规格说明$s$和充分性准则$M$，用测试数据集$t$ 来测试程序p的充分度为r，显然$0 \le r \le 1$，r越大说明测试越充分。

4.4.2测试充分性准则的度量

1. 揭错能力：

- 揭错能力是测试充分性准则有效性的最直接的度量之一，如果使用充分性准则A比使用充分性准则B可以发现更多的软件错误，则认为准则A的有效性要高于准则B。
- 揭错能力可由缺陷检测效力（Fault Detection Effectiveness，FDE）和缺陷检测概率（Fault Detection Probability，FDP）来表示。

4.4.2测试充分性准则的度量

1. 揭错能力：

- 为了对FDE和FDP进行定量描述，首先定义一个有效性矩阵E，如表4-2所示。
- 设软件P包含了$F_1,\ldots, F_m$，共m个缺陷，根据测试准则C，生成$T_1, T_2,\ldots, T_n$，共n个测试集，则n个测试集对m个缺陷的检测结果构成了测试准则C对软件P的有效性矩阵$E(m, n)$，其中，$e_{i,j}=0$表示测试集T_j未能检测出缺陷F_i，$e_{i,j}=1$表示测试集T_j能检测出缺陷F_i。

4.4.2测试充分性准则的度量

1. 揭错能力：

表4-2：测试准则C对程序P的有效性矩阵E

Fault（缺陷）	Test Suite（测试集）						*FDP*（缺陷检测概率）
	T_1	T_2	…	T_j	…	T_n	
F_1	1	1	…	…	…	0	$2/n$
F_2	0	1	…	…	…	0	$1/n$
…	0	0	…	…	…	1	…
F_i	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…
F_m	1	0	…	…	…	0	…
FDE	$2/m$	…	…	…	…	$1/m$	

4.4.2测试充分性准则的度量

1. 揭错能力：

- **定义4.3 缺陷检测效力**

给定测试集T和软件P，T对P的缺陷检测效力定义为T所检测到的缺陷占P所包含缺陷的比率，即

$$FDE_j = \sum_{i=1}^{m} \frac{e_{i,j}}{m}$$

- **定义4.4 缺陷检测概率**

对软件P所包含的缺陷类型Fi，满足测试准则C的所有测试集中能检测到该缺陷的测试集占整个测试集的比率，即

$$FDP_i = \sum_{i=1}^{n} \frac{e_{i,j}}{n}$$

2. 软件可靠性：

- 如果程序P经过满足测试充分性准则A测试后的可靠性要比经过满足测试充分性准则B测试后的可靠性高，则认为测试充分性准则A比测试充分性准则B具有更好的有效性。
- 软件可靠性是衡量软件质量的最重要要素和软件开发的最终目标，但软件可靠性指标本身就是一个很复杂的问题，利用可靠性对测试充分性准则的有效性进行比较是非常困难的。
- 通过充分性准则C测试后，为了表示软件P可获得的软件可靠性程度，则需要综合考虑缺陷的发生频率和测试准则的检测效力。用缺陷的潜在失效距离（Potential Failure Distance，PFD）对满足某种充分性准则测试后的软件进行可靠性度量。

2. 软件可靠性：

- **定义4.5 潜在失效距离**

对软件P，其对测试准则C的潜在失效距离定义为根据测试准则C对P进行充分测试后，该程序仍可能失效的概率为

$$PFD = \sum_{i=1}^{m} P_{\text{occur\_}i} \times (1 - P_{\text{detect\_}i})$$

其中，$P_{\text{occur_}i}$表示实际操作中导致软件失效的概率；$P_{\text{detect_}i}$表示通过某充分性准则测试后错误被移除的概率。

$$P_{\text{occur\_}i} = \frac{|D_i|}{|D|}$$

其中，$|D_i|$为会引发缺陷i的输入空间的大小，$|D|$表示整个输入空间的大小。

3. 测试开销：

软件测试是软件开发中开销较大的一项工程，测试开销和所选择的测试充分性准则密切相关，其实质是比较使用测试准则所用的代价，最为简单的对测试开销的度量是选择某个测试准则*C*时所需要的最少测试*T*。研究人员认为，使用一个测试准则进行测试的开销包括生成一组满足测试准则*C*的测试用例集的开销、执行测试的开销，以及检测输出结果的开销。

4.4.3 测试集充分性的度量

- 覆盖域

一个测试集的充分性由一个有限集来度量。针对相对应的测试准则 C，根据软件的需求或者软件的代码导出一个有限集，把这个有限集记为C_e，也就是所谓的覆盖域。

- 测试覆盖有限集

当对某个测试集T的充分性进行测试时，给定一个有n个元素的有限集C_e，$n \geq 0$。如果说测试集T覆盖有限集C_e，是指对于有限集C_e中的每个元素e，在测试集T中都至少有一个测试用例能够对它进行测试。

- 测试集的充分性

假如测试集T覆盖了有限集C_e中的所有元素，则认为对于测试准则C来说测试集T是充分的；而假如测试集T只覆盖了有限集C_e中的k个元素（$k<n$），则认为对于测试准则C来说测试集T是不充分的。

4.4.3 测试集充分性的度量

- 测试覆盖率

测试集T对测试准则C的充分度用分数k/n表示，这个分数也被称为是T对C的覆盖率。

- 测试充分性准则

要确定有限集C_e中的每一个元素e是否都被测试集T测试到了，这需要依赖于元素e的程序P。如果程序P中的每条路径都被遍历至少一次，则认为测试集T针对测试准则C是充分的。

但是充分的测试集可能不能发现软件中最明显的错误，例如，在对例4.3进行编程时，若给出示例程序，这个程序显然是错误的；若以测试准则C测试测试集T={t:x≤2,y≥3}，虽然测试集T相对于程序P是充分的，但显然这个程序并不正确。

4.4.3 测试集充分性的度量

例4.3 考虑编写程序sumProject，其需求如下。

R_1：当$x < y$时，求x与y之积，并在标准输出设备上输出x与y之积。

R_2：当$x \geq y$时，求x与y之和，并在标准输出设备上输出x与y之和。

示例程序如下：

```
1 begin
2     int x,y;
3     input(x,y);
4     sum=x+y;
5     output(sum);
6 end
```

一个充分的测试集有可能不能发现软件中最明显的错误，但是这丝毫不影响测试充分性度量的价值。

4.4.4 数据流概念

数据流

控制流测试是面向程序的结构，数据流测试面向的是程序中的变量。

- **变量的定义**

一个变量在程序中的某处出现使数据与该变量绑定

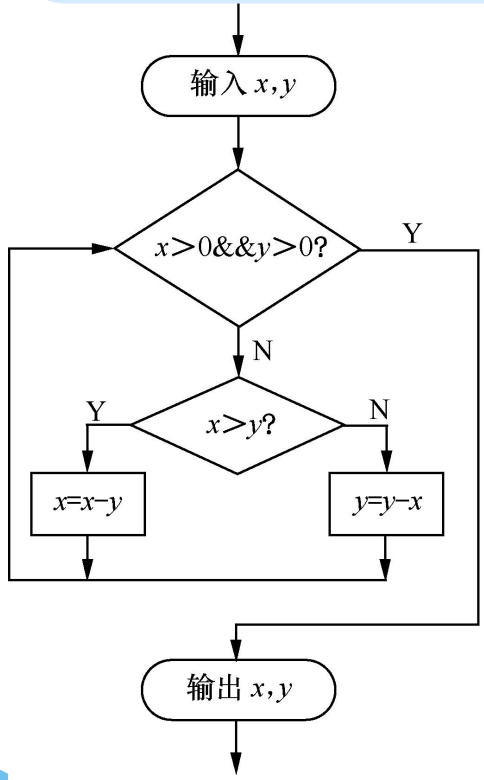
- **变量的引用**

一个变量在程序中的某处出现使之与该变量绑定的内容被引用

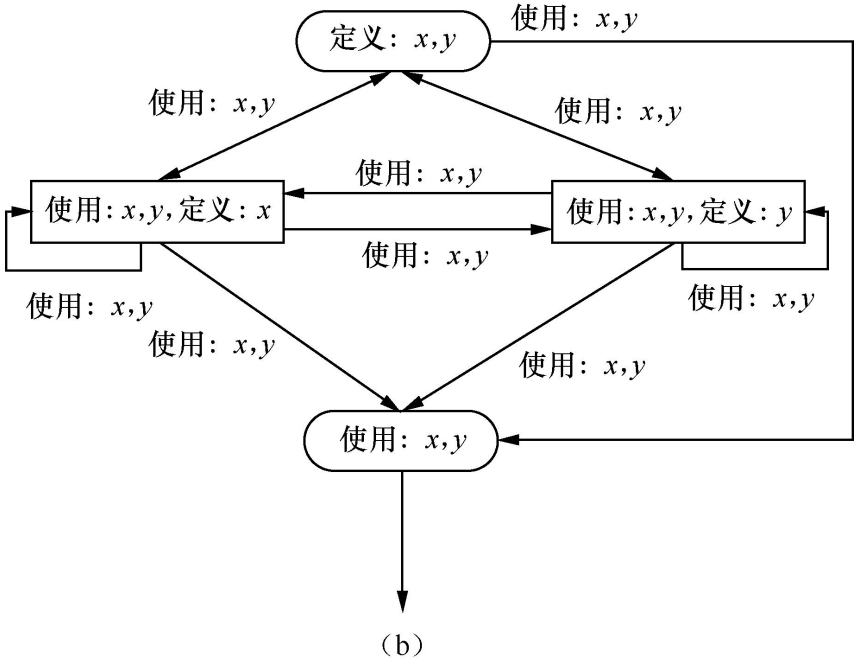
- 计算性引用：用于计算新数据或输出结果，用c-use表示；
- 谓词性引用：用于计算判断控制转移方向的谓词，用p-use表示。
- 若一个变量被引用前在它出现的程序块内无其定义性出现，则该引用称为全局性引用；否则称为局部性引用。

4.4.4 数据流概念

为方便引用，可将之前论述过的控制流图进行改造，即去掉控制流图中的判断框，把其中的谓词放在边上，这种图称为具有数据信息的控制流图。例如，图4-3（b）所示的就是一个计算最大公约数的具有数据信息的控制流图，而图4-3（a）所示的则是该程序的控制流图。



（a）



（b）

4.4.4 数据流概念

- 变量的定义传递到引用定义

定义4.8 ：$<n_1,n_2,n_3,\cdots,n_k>$是具有数据流信息的控制流图G_p中的一条路径，x是程序中的一个变量。

- 如果节点n_i（$i=1,2,\cdots,k$）都不含x的定义性出现，则该路径称为对于x来说是无定义的。

让n_0,n_k为G中的节点，n_0中包含变量x的定义性出现，n_k中包含变量x的计算性引用出现。

- 从节点n_0到节点n_k的路径$<n_1,n_2,n_3,\cdots,n_k>$称为一条将x在n_0中的定义传递到n_k中的计算性引用的路径，
- 如果路径对x来说是无定义的，则称x在n_0中的定义可传递到n_k中的计算性引用。

4.4.4 数据流概念

■ 全局/局部变量的使用

一个变量可能在同一个基本块中被定义、使用和重定义。考虑如下含有3条语句的基本块：

```
p = y+z;
x = p+1;
p = z*z;
```

- 变量*p*的第1个定义是局部的，变量*p*的第2个定义是全局的，变量*p*的第1个使用也是局部使用。
- 变量*y*和变量*z*的c-use是全局使用，
- 变量*x*的定义也是全局的。

注意：这里所讲的全局变量和局部变量，是针对程序中的基本块而言，这与传统的全局和局部概念有所不同。传统上，全局变量是在函数或模块的外部声明的，而局部变量是在函数或模块的内部声明的。

4.4.4 数据流概念

数据流图

程序的数据流图（def-use图），它刻画了程序中的变量在不同的基本块之间的定义流，与程序的控制流图相似。CFG中的节点、边以及所有的路径都在数据流图中保留了下来；程序的数据流图可以从它的CFG中导出。

设$G=(N, E)$为程序P的CFG，其中，N是节点集合，E是边集合。CFG中的节点对应于P中的基本块，假设程序P含有$k>0$个基本块，

b_1、b_2、…、b_k	基本块
def_i	定义在基本块i中的变量的集合
c-use_i	在基本块i中有c-use的变量的集合
p-use_i	在基本块i中有p-use的变量的集合
$d_i(x)$	变量x在节点i中的定义
$u_i(x)$	变量x在节点i中的使用

程序中的变量声明、赋值语句、输入语句和传址调用参数都可以用来定义变量。

4.4.4 数据流概念

- 数据流图

考虑如下基本块b，它包含两条赋值语句和一个函数调用语句：

```
p=y+z;
foo (p+q, number); //传值参数
A[i]=x+1;
if (x>y) {…};
```

由这个基本块，得到

- $\text{def}_b = \{p, A\}$，
- $\text{c-use}_b = \{y, z, p, q, \text{number}, x, i\}$，
- $\text{p-use}_b = \{x, y\}$。

4.4.4 数据流概念

- 数据流图

采用下面的过程，可以根据程序*P*及其CFG构造出其数据流图。

```
p=y+z;
foo (p+q, number); //传值参数
A[i]=x+1;
if (x>y) {…};
```

- **步骤1** 计算*P*中每个基本块*i*的def_i、c-use_i和p-use_i。
- **步骤2** 将节点集*N*中的每个节点*i*与def_i、c-use_i和p-use_i关联起来。
- **步骤3** 针对每个具有非空p-use集并且在条件*C*处结束的节点*i*，当条件*C*为真时，执行的是边(i, j)，当条件*C*为假时，执行的是边(i, k)，分别将边(i, j)、(i, k)与条件*C*、!*C*关联起来。

4.4.4 数据流概念

- **def-use对**

  def-use对勾画了变量的一次特定的定义和使用，程序的数据流图会包括该程序所有的def-use对。

- 一个def-use对由一个变量在某个基本块中的定义和该变量在另一个基本块中的使用构成。
- 一般来说，主要关心两种类型的def-use对：一种是定义及其c-use构成的def-use对；另一种是定义及其p-use构成的def-use对。分别用集合*dcu*和*dpu*来描述这两类def-use对。
- 对一个变量而言，总有一个*dcu*集合和一个*dpu*集合。

CU和PU的计算

设CU、PU分别表示程序P中定义的所有c-use总数量、p-use总数量。设$v=\{v_1,v_2,\ldots,v_n\}$表示程序P中所有变量的集合，N表示程序P中节点集合，d_i表示v_i的定义次数，$1\le i\le n$，$0\le d_i\le|N|$。CU和PU的计算如下：

$$CU=\sum_{i=1}^{n}\sum_{j=1}^{d_i}\left|dcu(v_i,n_j)\right|$$

$$PU=\sum_{i=1}^{n}\sum_{j=1}^{d_i}\left|dpu(v_i,n_j)\right|$$

其中n_j是变量v_i第j次被定义时所在的节点，$|S|$表示集合S中元素的个数。

- 覆盖率计算

1.测试集T针对(P,R)的c-use覆盖率计算如下：

$$\frac{CU_c}{CU - CU_i}$$

2.测试集*T*针对(*P*,*R*)的p-use覆盖率计算如下：

$$\frac{PU_c}{PU - PU_i}$$

3.测试集T针对(P,R)的all-use覆盖率计算如下

$$\frac{CU_c + PU_c}{(CU + PU) - (CU_i + PU_i)}$$

- 4.5 变异测试
  - 4.5.1 变异和变体
  - 4.5.2 强变异和弱变异
  - 4.5.3 用变异技术进行测试评价
  - 4.5.4 变异算子
  - 4.5.5 变异算子的设计
  - 4.5.6 变异测试的基本原则

4.5.1 变异和变体

- 原子性

用*P*表示原始被测程序，*M*表示轻微变更程序*P*后得到的程序，则可以把程序*M*称为程序*P*的变体（Mutation），程序*P*是程序*M*的父体（Parent）。

父体：

```
1 begin
2    int x,y;
3    input(x,y);
4    if(x<y)
5       output(x+y);
6    else
7       output(x*y)
8 end
```

变体：

```
1 begin
2    int x,y;
3    input(x,y);
4    if(x<y)
5       output(x-y);
6    else
7       output(x*y)
8 end
```

4.5.1 变异和变体

- 一阶变体

在对程序进行测试的时候，仅经过一次变更而得到的变体称为一阶变体。

- 高阶变体

同样地，二阶变体就是经过了两次简单变更而得到的变体，三阶变体就是经过了三次简单变更而得到的变体，依此类推。对一个一阶变体再进行一次简单变更就可以得到二阶变体，也就是说，一个n阶变体可以由一个(n−1)阶变体进行一次简单变更而得到。高于一阶的变体叫作高阶变体。

在实际中，使用最多的还是一阶变体，主要原因有两个：一个是在数量上，高阶变体的数量远多于一阶变体的数量，大量的变体会影响到充分性评价的可量测问题；另一个是涉及耦合效应。

4.5.2 强变异和弱变异

- 强变异检测

给定被测程序P和基于程序P生成的一个变体M，若测试用例t在程序P和变体M上的输出结果不一致，则称测试用例t可以强变异检测到变体m

- 弱变异检测

假设被测程序 P 由 n 个程序实体构成集合$S=\{s1,s2,\ldots,sn\}$。通过对程序实体sm执行变异算子生成变体M，若测试用例t在程序P和变体M中的程序实体sm上执行后的程序状态出现不一致，则称测试用例t可以弱变异检测到变体M

4.5.3用变异技术进行测试评价

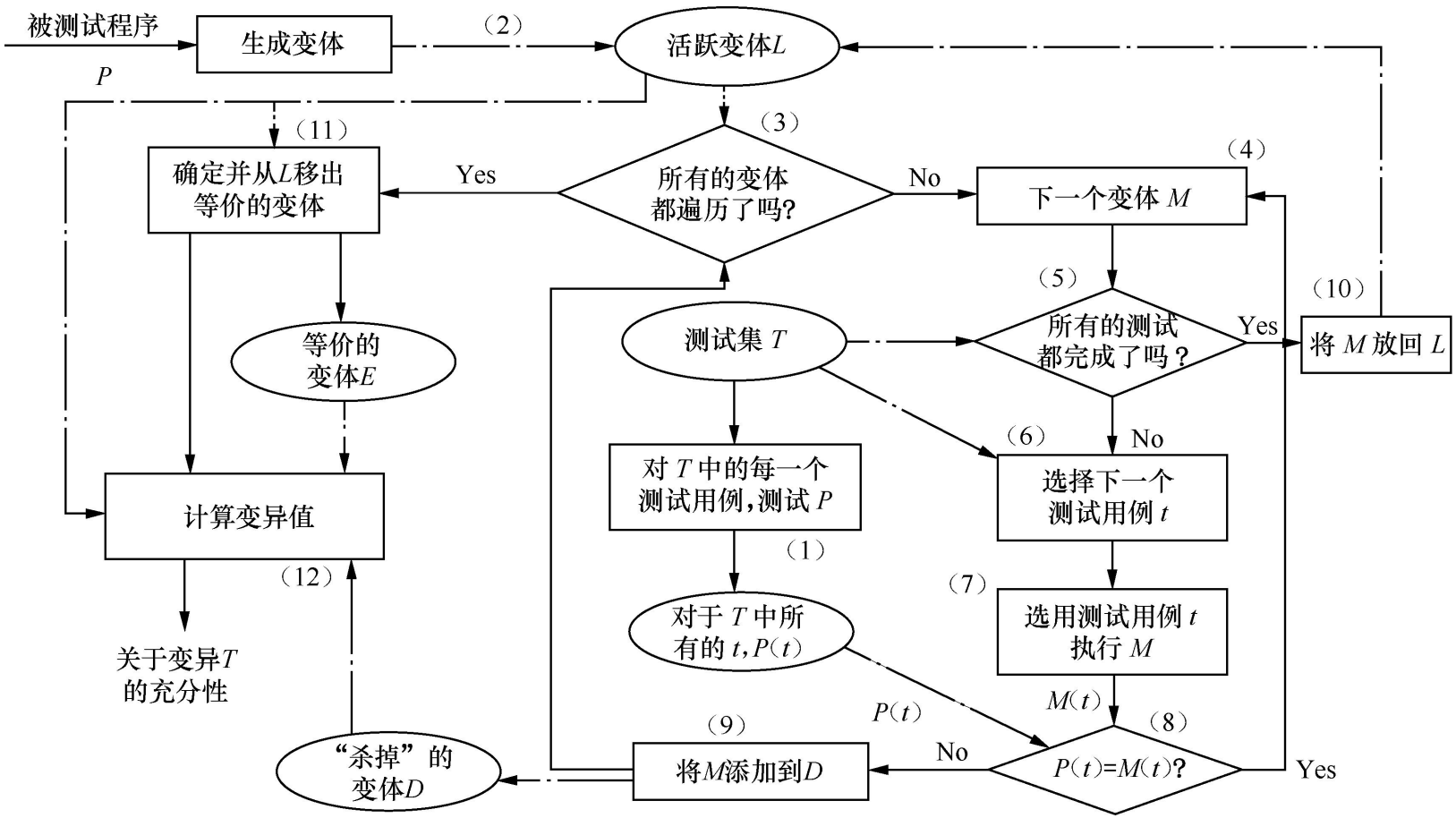


图4-4 利用变异技术对测试集进行充分性评价的过程

4.5.3用变异技术进行测试评价

图4-4采用了变异技术对测试集进行充分性评价的步骤，共涉及12个具体的步骤。

步骤 1 执行程序

评价测试集T针对于（P，R）充分性过程的第1步就是针对测试集T中的每个测试用例t执行程序P。P(t)代表程序P执行测试用例t时输出变量的集合。

步骤 2 生成变体

步骤2结束时得到一个变体集合L，其中包含的变体被称为活跃变体，因为这些变体都还没有与原来的程序区分开。

步骤3和步骤4 选取下一个变体

选择下一个将要考虑的变体，这个变体必须是以前没有考虑过的。循环选取集合L中的变体，直到集合T中的每个变体都被选取过为止，被选取过的变体会被从集合L中剔除。

步骤5和步骤6 选取下一个测试用例

选取变体M后，下一个目标就是从测试集T中找到一个测试用例，将变体与原程序分开。

循环结束条件：所有的测试用例都执行完了，或变体M被某个测试用例发现，与原程序区别开了

步骤7～步骤9 变体的执行和分类

对于已经被选出的变体M和准备执行的测试用例t，在步骤7中使用测试用例t执行变体M；在步骤8中，检查针对测试用例t执行变体M产生的结果与执行程序P产生的结果是否相同。

若存在测试用例t（t∈T），在变体M与原有程序P上的执行结果不一致，则称该变体M相对于测试用例集T是可杀除变体。若不存在任何测试用例t（t∈T），在变体M与原有程序P上的执行结果一致，则称该变体M相对于测试用例集T是可存活变体。

步骤7～步骤9在活跃变体中区分可杀除变体和可存活变体的过程。

步骤10 活跃变体

当测试集T中没有测试用例能将变体M与其父体程序P区分开来时，变体M被放回到活跃集合L中。

步骤11 等价变体

执行完所有变体之后，要检查是否还存在活跃变体，即检查变体集合L是否为空。一部分可存活变体通过设计新的测试用例可以转换成可杀除变体，剩余的可存活变体则可能是等价变体。若变体M与原有程序P在语法上存在差异，但在语义上与P保持一致，则称M是P的等价变体。

步骤12 变异值的计算

一般用下面的公式计算测试用例集T的变异值。

$$MS(T)=\frac{|D|}{|L|+|D|}$$

注意：集合L只包含活跃变体，而且这些变体和原程序都不等价。这个变异值总是在0和1之间。

同时可以用M表示第2步中生成的变体总数，则计算公式可以被表示为：

$$MS(T)=\frac{|D|}{|M|-|E|}$$

4.5.4 变异算子

- 变异算子

定义了从原有程序生成差别极小程序（即变体）的转换规则。1987年，奥佛特（Offutt）和金（King）针对FORTRAN77首次定义了22种变异算子

序号	变异算子	描述
1	AAR	用一数组引用替代另一数组引用
2	ABS	插入绝对值符号
3	ACR	用数组引用替代常量
4	AOR	算术运算符替代
5	ASR	用数组引用替代变量
6	CAR	用常量替代数组引用
7	CNR	数组名替代
8	CRP	常量替代
9	CER	用常量替代变量
10	DER	DO语句修改

4.5.4 变异算子

序号	变异算子	描述
11	DSA	DATA语句修改
12	GLR	GOTO标签替代
13	LCR	逻辑运算符替代
14	ROR	关系运算符替代
15	RSR	RETURN语句替代
16	SAN	语句分析
17	SAR	用变量替代数组引用
18	SCR	用变量替代常量
19	SDL	语句删除
20	SRC	源常量替代
21	SVR	变量替代
22	UOI	插入一元操作符

4.5.5 变异算子的设计

- 4项准则

（1）语法正确性

一个变异算子必须产生一个语法正确的程序。

（2）典型性

一个变异算子必须能模拟一个简单的共性错误。当然，在真正的程序中错误常常并不简单，但是变异算子只能对简单的错误进行模拟，很多这样简单的错误集合在一起，就能够构成一个复杂错误。

（3）最小性和有效性

变异算子的集合应该是最小且有效的集合。

（4）明确定义

必须明确定义变异算子的域和范围。变异算子的域和范围都依赖于具体的编程语言，在定义变异算子范围时要考虑所有语法保真替换。

4.5.6 变异测试的基本原则

传统变异测试一般通过生成与原有程序差异极小的变体来充分模拟被测软件的所有可能缺陷。其可行性基于两个原则，也是两个重要的假设：熟练程序员假设（Competent Programmer Hypothesis，CPH）和耦合效应假设（Coupling Effect Hypothesis，CEH）

- **假设1（熟练程序员假设）** 假设熟练程序员因编程经验较为丰富，编写出的有缺陷代码与正确代码非常接近，仅需做小幅度代码修改就可以完成缺陷的移除。

- **假设2（耦合效应假设）** 若测试用例可以检测出简单软件缺陷，则该测试用例也易于检测到更为复杂的软件缺陷。简单软件缺陷是仅在原有程序上执行单一语法修改形成的软件缺陷，而复杂软件缺陷是在原有程序上依次执行多次单一语法修改形成的软件缺陷。

本章小结

- **白盒测试**

  语句覆盖、判定覆盖、条件覆盖、条件/判定组合覆盖、多条件覆盖、修正判定条件覆盖、组合覆盖和路径覆盖

- **控制流测试/数据流测试**

  控制流测试是面向程序的结构，数据流测试面向的是程序中的变量

- **变异测试**

  这些所谓的变异，是基于良好定义的变异操作