第五部分 软件编码、测试与质量保障

- 5.1 软件编程
- 5.2 软件测试
- 5.3 白盒测试
- 5.4 黑盒测试
- 5.5 变异测试
- 5.6 性能测试

变异测试技术

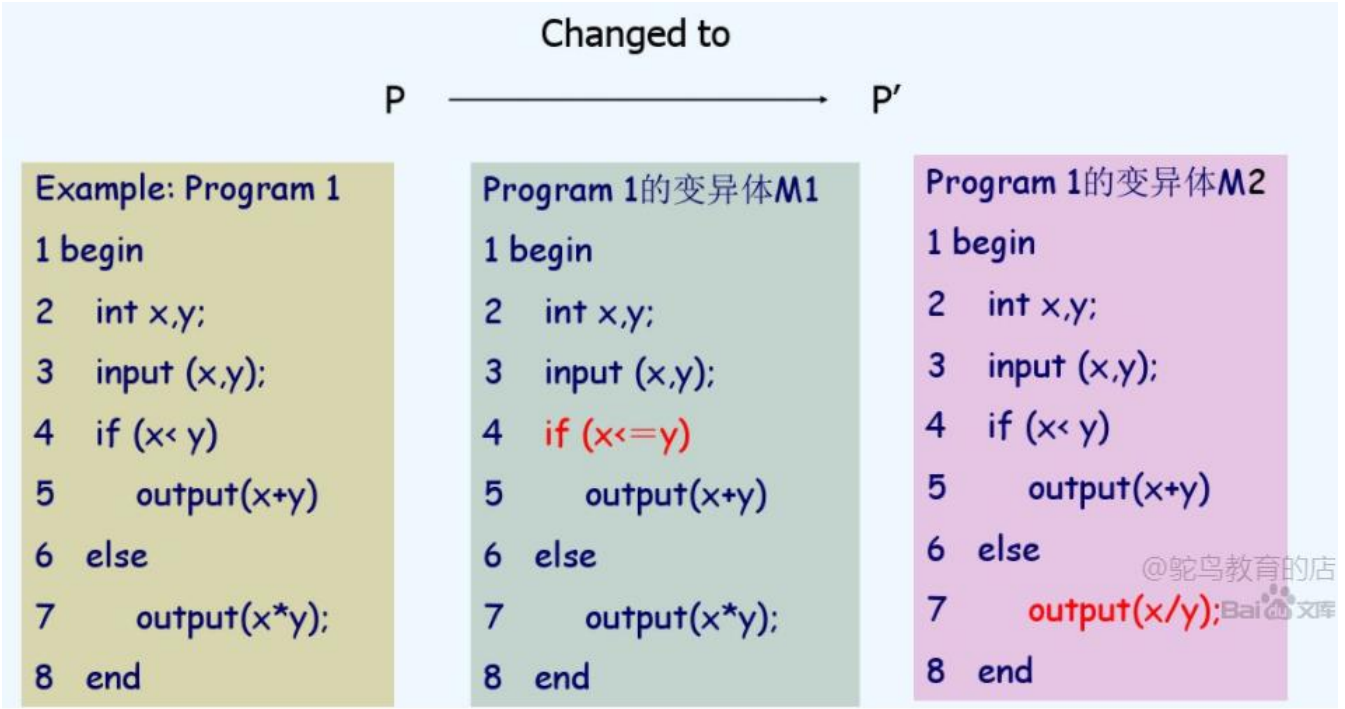
- 变异测试是一种对测试集的充分性进行评估的技术，以创建更有效的测试集。
- 变异测试与路径或数据流测试不同，没有测试数据的选择规则。
- 变异测试应该与传统的测试技术结合，而不是取代它们。

基本思想

- 给定一个程序P和一个测试数据集T，通过变异算子为P产生一组变异体M_i（合乎语法的变更），对P和M都使用T进行测试运行：
  - 如果某M_i在某个测试输入t上与P产生不同的结果，则该M_i被杀死；
  - 如果某M_i在所有的测试数据集上都与P产生相同的结果，则称其为活的变异体。
  - 接下来对活的变异体进行分析，检查其是否等价于P：
    - 对不等价于P的变异体M进行进一步的测试，直到充分性度量达到满意的程度。

程序变异概念（1）

- 假设程序P已使用测试T中的测试用例测试通过，而且没有错误。变异是一种轻微改变程序的操作。



程序变异概念（2）

- P’称为P的变异体
- 如果对于T中的测试t，有P(t)≠P’(t)，称作P’与P有区别，或者t杀死P’。
- 如果T中所有的测试t使得P(t)=P’(t)，称T不能区别P和P’。那么称在测试过程中P’是活的。
- 如果在程序P的输入域中不存在任何测试用例t使得P与P’区别，则称P’等价于P。
- 如果P’不等价于P，且T中没有测试能够将P’与P区别，则认为T是不充分的。
- 不等价而且是活的变异体为测试人员提供了一个生成新测试用例的机会，进而增强测试T。

为什么变异？

- 设想如下情况：
  - 程序员开发了一个程序P，按照某种测试充分性准则通过了测试，即将发布，这时会有人指出：程序中的表达式 count<max 还是 count<max+1
- 问题：为什么替代的方法不正确，或为什么替代的方法比所选的方法更好？
- 可能的答案：
  - 性能差别
  - 替代方法“不正确”，而现有方法是“正确的”或更好
    - 通过测试用例表明现有的方法与替代的方法是不同的，而且现有的方法是“正确的”
    - 表明替代方法与现有方法是等价的
- 变异是一种解决上述问题的系统的方法，变异测试可以发现程序中一些细微的错误——例如，表明替代的方法是不正确的

例：原子反应堆控制软件P

```
1.  enum dangerLevel {none,moderate,high,veryHigh};
2.  Procedure chechTemp(currentTemp, maxTemp)
3.  {  float currentTemp[3],maxTemp; int highCount=0;
4.     enum dangerLevel danger;
5.     danger=none;
6.     if(currentTemp[0]>maxTemp)
7.          highCount=1;
8.     if(currentTemp[1]>maxTemp)
9.          highCount=highCoun+1;
10.    if(currentTemp[2]>maxTemp)
11.         highCount=highCount+1;
12.    if (highCount==1) danger=moderate;
13.    if (highCount==2) danger=high;
14.    if (highCount==3) danger=veryHigh;
15.    returen(danger);
16. }
```

例：原子反应堆控制软件P'-M1

```
1.  enum dangerLevel {none,moderate,high,veryHigh};
2.  Procedure chechTemp(currentTemp, maxTemp)
3.  {  float currentTemp[3],maxTemp; int highCount=0;
4.     enum dangerLevel danger;
5.     danger=none;
6.     if(currentTemp[0]>maxTemp)
7.          highCount=1;
8.     if(currentTemp[1]>maxTemp)
9.          highCount=highCoun+1;
10.    if(currentTemp[2]>maxTemp)
11.         highCounthighCount+1;
12.    if (highCount==1) danger=moderate;
13.    if (highCount==2) danger=high;
14.    if (highCount==3) danger=none;    //if (highCount==3) danger=veryHigh;
15.    returen(danger);
16. }
```

例：原子反应堆控制软件P'-M2

```
1.  enum dangerLevel {none,moderate,high,veryHigh};
2.  Procedure chechTemp(currentTemp, maxTemp)
3.  {  float currentTemp[3],maxTemp; int highCount=0;
4.     enum dangerLevel danger;
5.     danger=none;
6.     if(currentTemp[0]>maxTemp)
7.          highCount=1;
8.     if(currentTemp[1]>maxTemp)
9.          highCount=highCoun+1;
10.    if(currentTemp[2]>maxTemp)
11.         highCounthighCount+1;
12.    if (highCount>=1) danger=moderate;  // if (highCount==1) danger=moderate;
13.    if (highCount==2) danger=high;
14.    if (highCount==3) danger=veryHigh;
15.    returen(danger);
16. }
```

强变异与弱变异

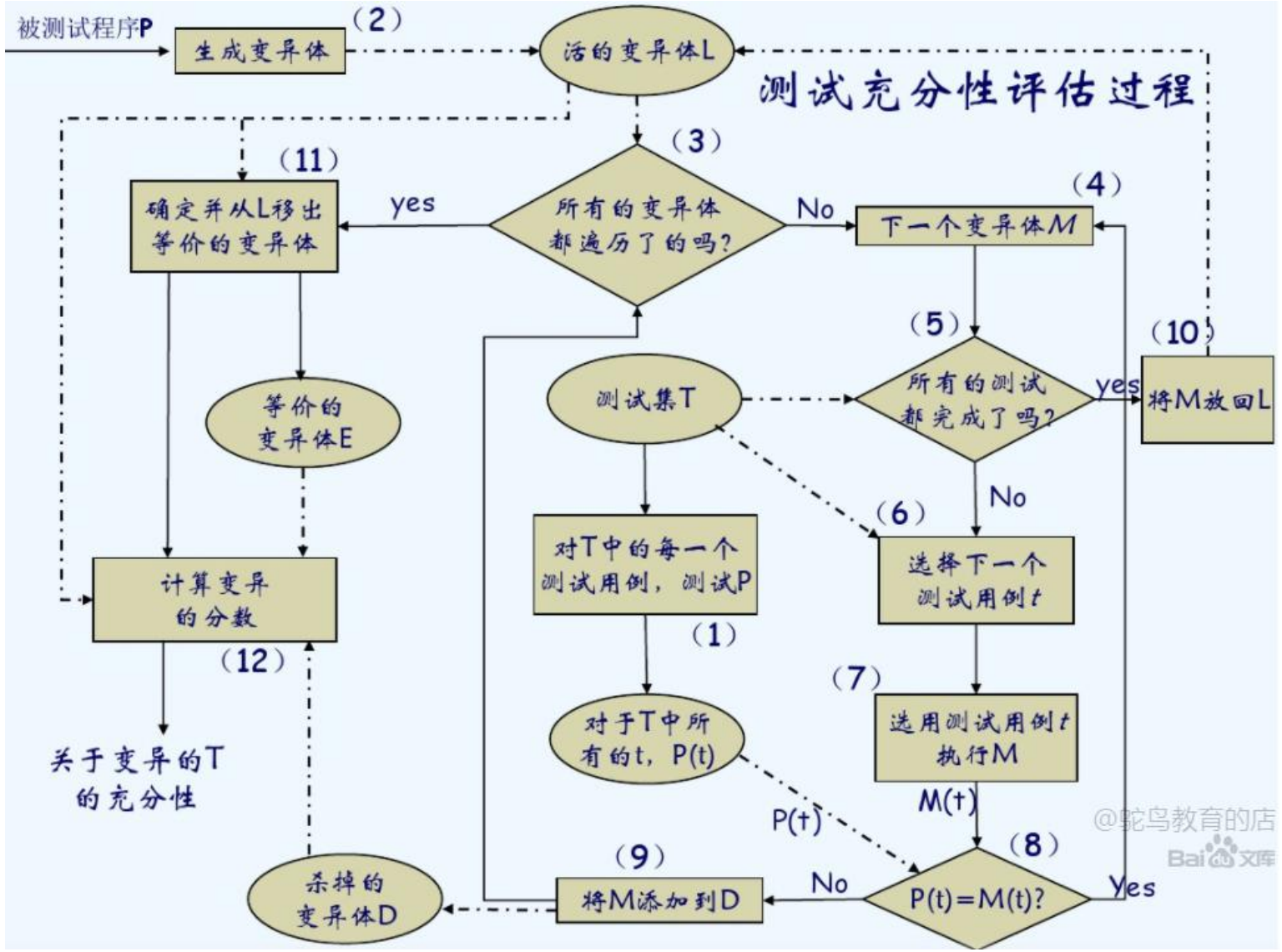
考查测试t:

< maxTemp=1200,

currenttemp=[1250,1389,1127] >

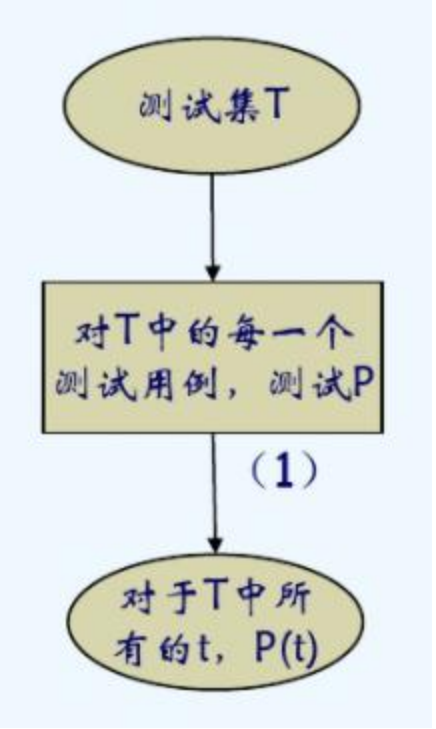
应用变异评估测试的充分性（1）

- 给定一个测试集□ 对于程序□ 这必须符合要求□，测试充分性评估程序如下：
  - 步骤1：创建一个□的突变体集合□. 令$M = \{M_1, M_2, ..., M_k\}$（□个突变体）
  - 步骤2：对于每个突变体M_i，查找□中是否存在□，使得：$M_i(t) \neq P(t)$. 如果存在这样的□，那么M_i被视为死亡，之后不再考虑
  - 步骤3：在第2步结束时，假设□$_1$(≤□）个突变体已经被杀死，并且（□−□$_1$）个突变体是活的
    - 情形1：（□−□$_1$)=0：□对于变异是充分的
    - 情形2：（□−□$_1$)>0，那么计算变异得分：$MS = k_1/(k-e)$，e是等价变异的数量，且有：$e \leq (k-k_1)$


测试充分性评估过程
被测试程序P
(2)
生成变异体
活的变异体L
(3)
所有的变异体
都遍历了的吗?
yes
No
(11)
确定并从L移出
等价的变异体
(4)
下一个变异体M
(5)
所有的测试
都完成了吗?
yes
(10)
将M放回L
No
测试集T
等价的
变异体E
(6)
选择下一个
测试用例t
对T中的每一个
测试用例，测试P
(1)
(7)
选用测试用例t
执行M
M(t)
对于T中所
有的t，P(t)
P(t)
(8)
P(t)=M(t)?
Yes
No
(9)
将M添加到D
杀掉的
变异体D
计算变异
的分数
(12)
关于变异的T
的充分性
@鸵鸟教育的店
Bai du 文库

测试充分性评估过程

- 第1步：程序执行
  - P(t)表示给定测试用例t，程序P的执行结果由P中变量的输出值表示（也可能与P的性能有关）
  - 如果P已经采用测试T测试通过，测试结果已保存至数据库中，则这一步可以跳过
  - 无论何种情况，第一步的结果是对于T中的所有t，P(t)保存至数据库

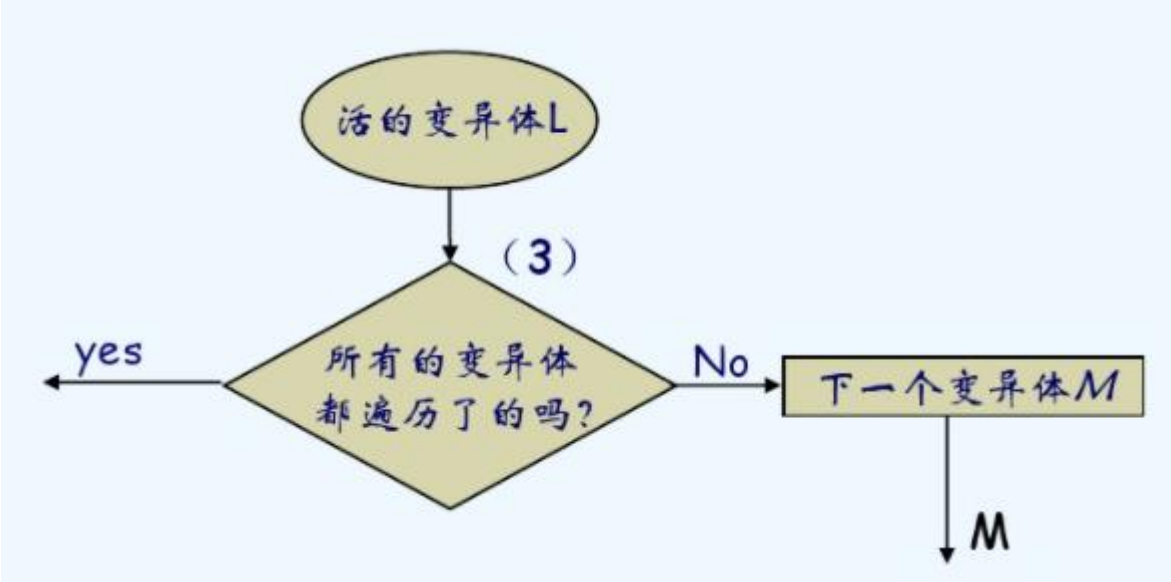


测试充分性评估过程

- 第2步：生成变异体
  - 例如“+”运算变成“-”运算，“×”运算变成“/”运算等
  - 系统的生成方法：通过变异算子生成
  - 第二步的结果是：活的变异体
    - 这些变异体还没有与程序P区分，即没有被杀死

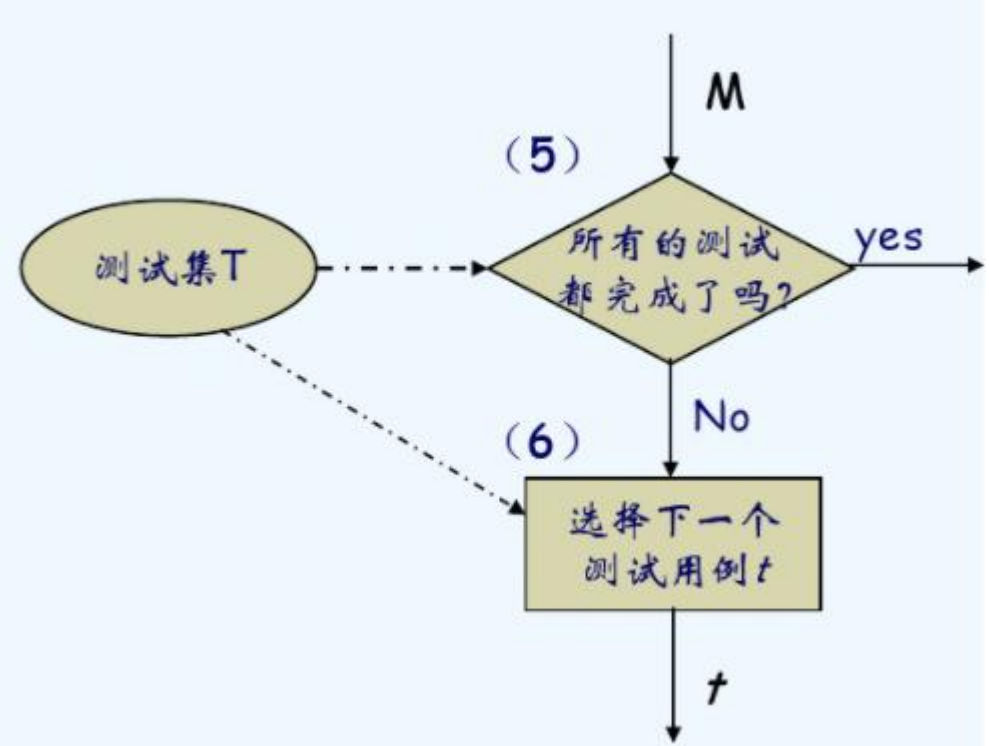
测试充分性评估过程

- 第3步和第4步：选择下一个变异体
  - 从L中选择，任意选择



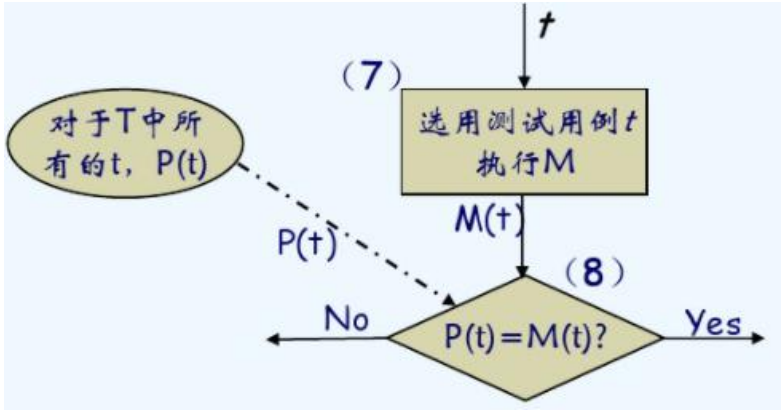
测试充分性评估过程

- 第5步和第6步：选择下一个测试用例
  - 是否存在测试t能够区分变异体与被测试程序P
  - 采用测试T中的测试用例执行变异体M
  - 结束：所有的测试用例执行完毕或者M被某个测试用例区别（杀掉）

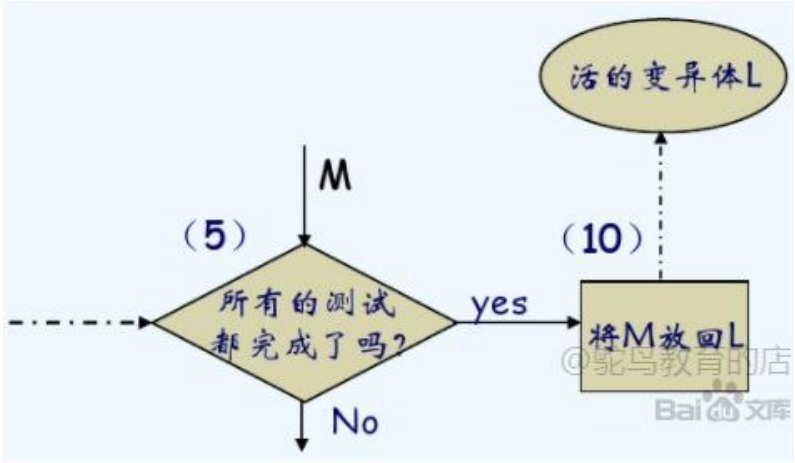


测试充分性评估过程

- 第7,8和9步：变异体执行和分类
  - 变异体执行的结果是否与P的执行结果相同或不同。



- 第10步：活变异体
  - 如果没有测试用例能够区分变异体与P，则该变异体存活，并被放回活变异体集合L中。

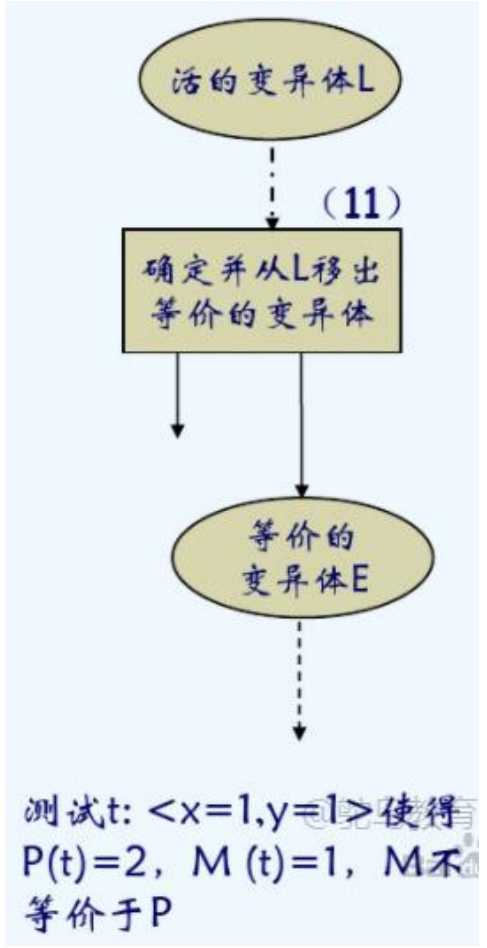


测试充分性评估过程

- 第11步：等价变异体
  - 如果对于程序P的输入域中的每一个输入，变异体M的执行结果等于P的执行结果，则认为M等价于P
  - 例：

```
Example: Program 1
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
```

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



测试充分性评估过程

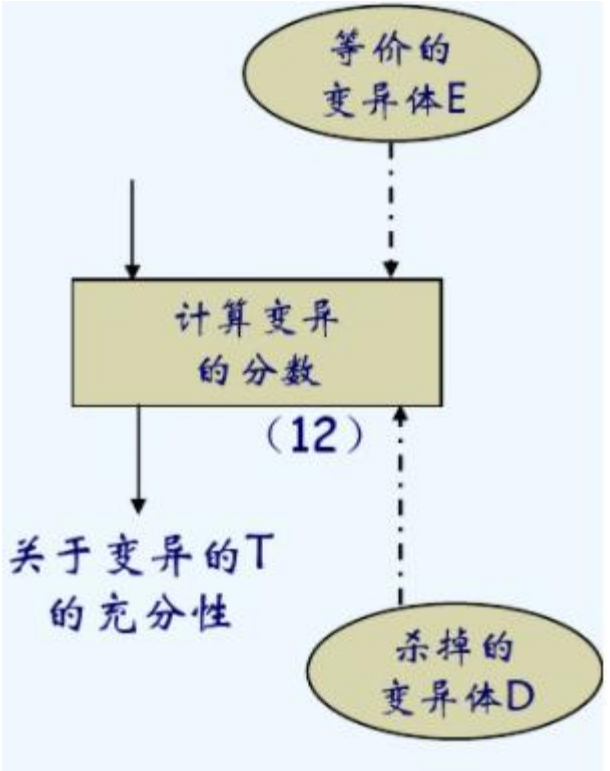
- 第12步：变异分数的计算
  - 量化评价指标：
    - =1代表相关于变异T是充分的
    - <1表示相关于变异T是不充分的
    - 可以通过增加额外的测试用例提高充分性（变异分数）
  - T的变异分数记为$MS(T)$

$MS(T) = |D|/(|L|+|D|)$ 或 $MS(T) = |D|/(|M|-|E|)$

其中：

$|D|$表示杀死的变异体数； |L|表示活的变异体数；

|E|表示等价的变异体数； |M|表示第2步生成的所有变异体数；



测试增强（例）

- P使用测试T并测试通过

{t1:<x=0, y=0>, t2:<x=0, y=1>, t3:<x=1, y=0>, t4:<x=-1, y=-2>}

- M使用T运行的结果与P相同，无法区分P和M
- 增加一个测试用例<x=1, y=1>，使P和M区别，表明增强了T

Example: Program 1

```
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
```

Program 1的变异体M

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

应用变异查错（1）

- 考虑如下本应该返回两个整数x和y之和的函数foo。显然，foo是不正确的。

```
int foo(int x, y){
return (x-y);      <-------   This should be return (x+y)
}
```

应用变异查错（2）

- 假设foo已经由测试集合T测试通过，T包含两个测试用例：

  {t1:<x=1, y=0>, t2:<x=-1, y=0>}

- 注意：foo在每一个测试用例上都返回了期望值，而且对于基于控制流和数据流的充分性准则来说T是充分的。
- 假设foo生成了三个变异体M1，M2和M3

M1:
```
int foo(int x, y){
return (x+y);
}
```

M2:
```
int foo(int x, y){
return (x-0);
}
```

M3:
```
int foo(int x, y){
return (0+y);
}
```

应用变异查错（3）

- 使用T执行每个变异体，直到变异体被杀死或执行完所有的测试。

  T={t1:<x=1, y=0>, t2:<x=-1, y=0>}

Test (t)	foo(t)	M1(t)	M2(t)	M3(t)
t1	1	1	1	0
t2	-1	-1	-1	0
		Live	Live	Killed

- 执行完所有的变异体后，2个活变异体，1个被杀死，计算变异分数需要对所有活的变异体判断等价

应用变异查错（4）

- 考察两个活的变异体：

M1:
```
int foo(int x, y){
return (x+y);
}
```

M2:
```
int foo(int x, y){
return (x-0);
}
```

```
int foo(int x, y){
return (x-y);
}
```

- 关注M1，一个测试能够区分M1和foo一定满足条件：

  $x-y \neq x+y$ 即 $y \neq 0$

因此，得到一个新的测试数据t3：〈x=1, y=1〉

应用变异查错（5）

- 使用t3执行foo得到foo(t3)=0，然而根据需求应该得到foo(t3)=2.因此，t3使得M1与foo有区别，同时发现了错误

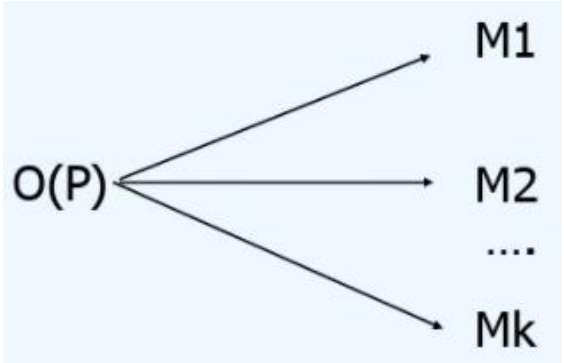
M1:
```
int foo(int x, y){
return (x+y);
}
```

M2:
```
int foo(int x, y){
return (x-0);
}
```

```
int foo(int x, y){
return (x-y);
}
```

变异算子（1）

- 变异算子O是一个函数，建立了被测试程序P与P的K(≥0)个变异体间的映射。



- 变异算子通过对被测试程序做简单的变化生成变异体。
- 变异算子对编程语言有依赖性，已经开发的与语言相关的变异算子有Fortran, C, Ada, Lisp 和 Java 等。
- 变异算子的设计基于经验和一些规则。

变异算子（2）

- 例如，“variable replacement”突变运算符用程序中声明的另一个变量替换变量名。“关系运算符替换”变异运算符用另一个关系运算符替换关系运算符。
- 一个变异操作符模拟了一个程序员可能犯的简单错误。
- 一些错误研究表明，程序员新手和专家都会犯简单的错误。例如，可能使用了x<y替代x<y+1。
- 虽然程序员也会犯“复杂的错误”，但变异操作符会模拟简单的错误。“耦合效应”解释了为什么只对简单错误建模。

变异算子（例）

Mutant operator	In P	In mutant
Variable replacement	z=x*y+1;	x=x*y+1; z=x*x+1;
Relational operator replacement	if (x<y)	if(x>y) if(x<=y)
Off-by-1	z=x*y+1;	z=x*(y+1)+1; z=(x+1)*y+1;
Replacement by 0	z=x*y+1;	z=0*y+1; z=0;
Arithmetic operator replacement	z=x*y+1;	z=x*y-1; z=x+y-1;

一阶和高阶变异体

- 通过“一次改变”获得的变异体被认为是一阶突变
- 通过两次改变获得的突变体为二阶突变体。同样，可以定义高阶突变体。例如，z=x+y的二阶突变体是x=z+y，其中变量替换算子应用了两次
- 在实践中仅产生一级突变体，原因有两个：（a）降低测试成本；（b）大多数高阶突变体通过与一阶突变体相关的测试被杀死。[见后面的耦合效应]

变异测试的理论基础

- 程序员的能力假设（Competent programmer hypothesis(CPH)）：被测试程序是由足够程序设计能力的程序员书写的，所产生的程序是接近正确的。
- 组合效应假设（Coupling Effect）：假设简单的程序设计错误和复杂的程序设计错误之间具有组合效应，即一个测试数据如果能够发现简单的错误，也可以发现复杂的错误。
- 以上两个假设来源于软件开发的实践，至今未达到正确性证明，但确定了变异测试的基本特征：通过变异算子对程序做一个较小的语法上的变动来产生一个变异体。

变异测试工具

- 免费提供的突变测试工具很少
  - Proteum for C from Professor Maldonado
  - muJava for Java from Professor Jeff Offutt
- 突变测试的典型工具提供以下特性
  - 变异算子的可选调色板
  - 对测试集T的管理
  - 在测试集T上执行被测程序并保存输出，以与在变异体上执行被测程序的结果进行比较
  - 变异体的生成

变异测试工具的特征（续）

- 使用用户识别的等效变异体执行变异和计算变异分数
- 增量变异测试：即允许将变异算子子集应用于被测程序的一部分
- Fortran的高级变异工具Mothra也使用DeMillo和Offutt的方法提供了自动测试生成

变异与系统测试

- 通常建议仅对相对较小的单元使用突变进行充分性评估
- 例如：
  - Java中的一个类
  - C语言中一个小的函数集合
- 然而，如果有一个好的工具，可以使用变异来评估系统测试的充分性
- 建议采用以下程序来评估系统测试的充分性

测试步骤

- 步骤1：确定对应用程序的安全运行至关重要的一组应用程序单元U。对U中的每个单元重复以下步骤。
- 步骤2：选择一小组变异算子。此选择最好由Eric Wong或Jeff Offutt定义的算子指导。
- 步骤3：将变异算子应用于所选单元

测试步骤

- 步骤4：使用上述产生的变异算子来访问T的充分性，若有必要，增强T
- 步骤5：对下一个单元重复步骤3和4，直到所有单元都考虑完毕
- 我们现在已经访问了T，并可能对其进行了增强。注意使用增量测试和约束变异（即：使用一组有限的高效变异算子）

变异测试的优缺点

- 排错能力强
  - 发现错误的能力较强——分析评估的结果
- 自动化程度高
  - 测试工具自动产生变异体，自动运行P和M，自动发现被杀死的变异体
- 灵活性高
  - 通过与测试提供工具的交互，有选择地使用功能变异算子
- 变异体与被测试程序的差别信息可以较容易地发现软件的错误

变异测试的优缺点（续）

- 可以完成语句覆盖和分支覆盖
  - 将每条语句或每个条件进行突变
- 需要大量的计算机资源完成充分性分析
  - n行程序产生$O(n^2)$变异体
  - 存储变异体的开销大
  - 变异体与被测试程序的等价判断需人工判定（判断两个程序是否等价是不可判定的命题）

小结

- 变异测试是评估和增强测试的最有力的技术。
- 变异就像任何其他测试评估技术，必须在良好工具的协助下逐步应用。
- 等价变异体的识别是一个不可判定的问题——类似于对基于控制流或数据流的测试评估中不可行路径的识别。

小结（续）

- 虽然变异测试通常被推荐用于单元测试，但如果仔细地逐步进行，它也可以用于评估系统测试的充分性
- 突变是一种强烈推荐的技术，用于确保高可用且安全的系统的质量