程序变异测试

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；
- 若某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。

Changed to

P ⟶ P′

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M1

```
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M2

```
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
```

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（distinguishes），或者t杀死（killed）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（live）.
- 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，而且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。

Slides from Aditya P.Mathur

为什么变异？

- A Scenario：
  - 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
  - 等价的
  - 通过测试用例表明现有的方法与替代的方法是“不同的”，哪个方法是“正确的”。
- 变异是一种解决上述问题的系统的方法，变异测试可以
  - 发现程序一些细微的错误
  - 或者：表明替代的方法是不正确的。

Slides from Aditya P.Mathur

应用变异查错（1）

- Consider the following function foo that is required to return the sum of two integers x and y. Clearly foo is incorrect.

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


This should be return (x+y)

Slides from Aditya P.Mathur

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T的两个测试用例上都Pass，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:
```
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);
}
```

Slides from Aditya P.Mathur

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，计算变异数需要对所有活的变异体判断等价。

Slides from Aditya P.Mathur

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);
}
```

对于M1：一个测试能够区分M1和foo一定满足条件：

x-y≠x+y implies y ≠0.

因此，得到一个新的测试数据t3: <x=1, y=1>

Slides from Aditya P.Mathur

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);
}
```

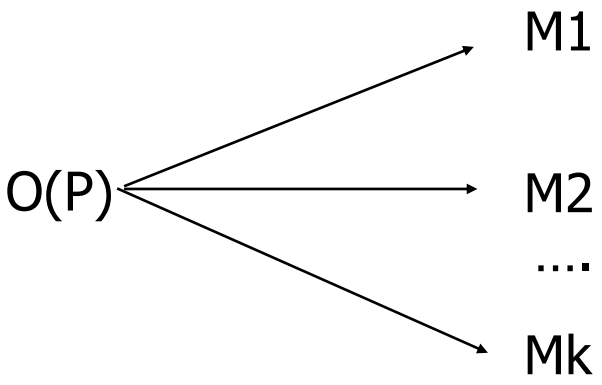
M2:
```
int foo(int x, y){
return (x-0);
}
```

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

Slides from Aditya P.Mathur

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。

Slides from Aditya P.Mathur

变异算子（2）

- A mutant operator models a simple mistake that could be made by a programmer
  - Several error studies have revealed that programmers--novice and experts--make simple mistakes. For example, instead of using x<y+1 one might use x<y
- While programmers make "complex mistakes" too, mutant operators model simple mistakes. The "coupling effect" explains why only simple mistakes are modeled.

- For example, the "variable replacement" mutant operator replaces a variable name by another variable declared in the program. An "relational operator replacement" mutant operator replaces relational operator with another relational operator

Slides from Aditya P.Mathur

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;

Slides from Aditya P.Mathur

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.     return(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.         highCount=highCount+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.     return(danger);
16. }
```

强变异

考查测试t：

< maxTemp=1200,

currenttemp=[1250,1389,1527] >

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.     return(danger);
16. }
```

弱变异

考查测试t：

< maxTemp=1200,

currenttemp=[1250,1389,1127] >

强变异和弱变异

- P和M执行到12行，二者的状态不同
  - P（danger,HighCount)=(none,2);
  - M2(danger,HighCount)=(moderate,2)
- 但是整个程序执行结束，二者的返回值是相同的，没有区别。

- 强变异：关心程序的外部行为
  - 如全局变量值的改变，数据文件等
  - P和M2在强变异下是等价的，
- 弱变异：关心程序的内部行为
  - 如状态。
  - 在给定的测试t，P和M2是有区别的，在弱变异下是不等价的。

应用变异评估测试的充分性（1）

- Given a test set T for program P that must meet requirements R, a test adequacy assessment procedure proceeds as follows:
  - Step 1: Create a set M of mutants of P. Let *M={M0, M1…Mk}*. Note that we have k mutants
  - Step 2: For each mutant Mi find if there exists a *t* in *T* such that *Mi(t) ≠P(t).* If such a *t* exists then *Mi* is considered killed and removed from further consideration.

Slides from Aditya P.Mathur

应用变异评估测试的充分性（2）

- Step 3: At the end of Step 2 suppose that k1（≤ k） mutants have been killed and (k-k1) mutants are live

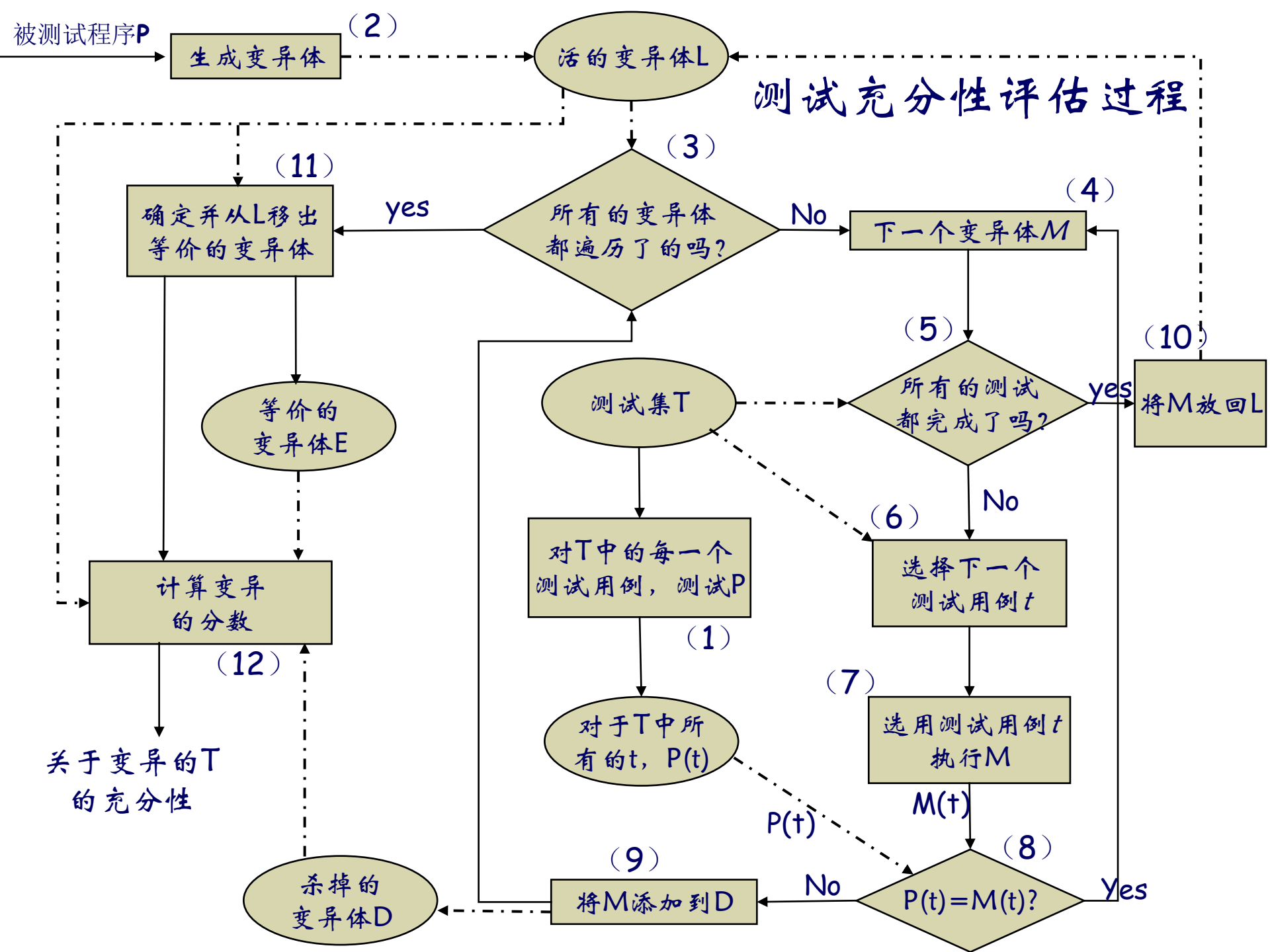
Case 1: (k-k1)=0: T is adequate with respect to mutation.

Case 2: (k-k1)>0 then we compute the mutation score (MS) as follows:

$$MS = k_1/(k-e)$$

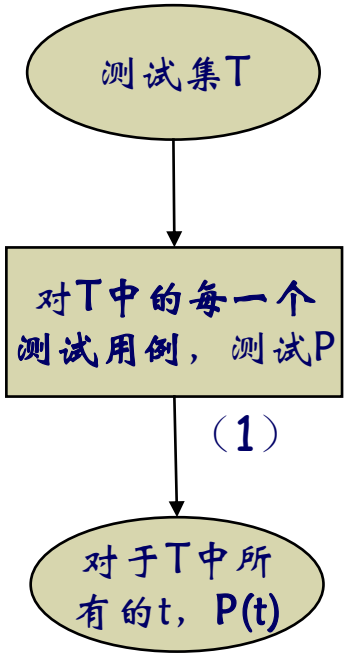
Where e is the number of **equivalent mutants**. Note: $e \leq (k-k_1)$.

Slides from Aditya P.Mathur


测试充分性评估过程
被测试程序P
生成变异体
（2）
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)?
No
Yes
（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

测试充分性评估过程

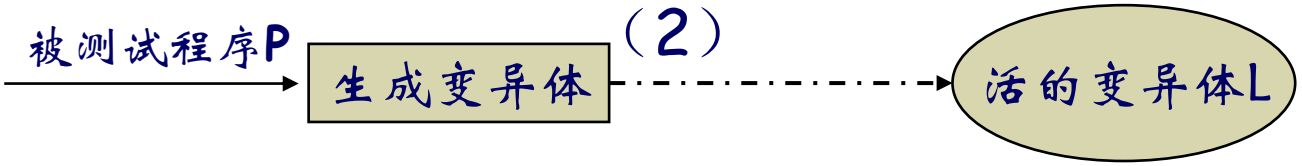
- 第1步：程序执行
  - P(t)表示给定测试用例t，程序P的**执行结果**：由P中变量的**输出**值表示（也可能与P的性能有关）
  - 如果P已经采用测试T测试通过，测试结果已保存至数据库中，则这一步可以跳过。
  - 不论何种情况，第一步的结果是对于T中的所有t，P(t)数据库



Slides from Aditya P.Mathur

测试充分性评估过程

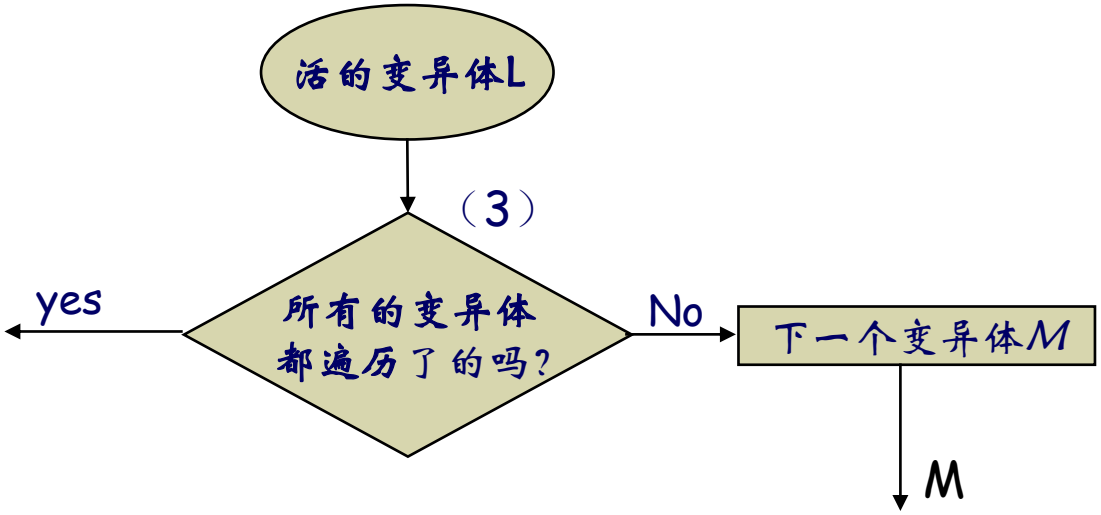
- 第2步：生成变异体
  - 例如“+”运算变成“−”运算，“×”运算变成“/”运算等
  - 系统的生成方法：通过变异算子生成
  - 第二步的结果是：**活的变异体**
    - 这些变异体还没有与程序P区分，即没有被杀死。



Slides from Aditya P.Mathur

测试充分性评估过程

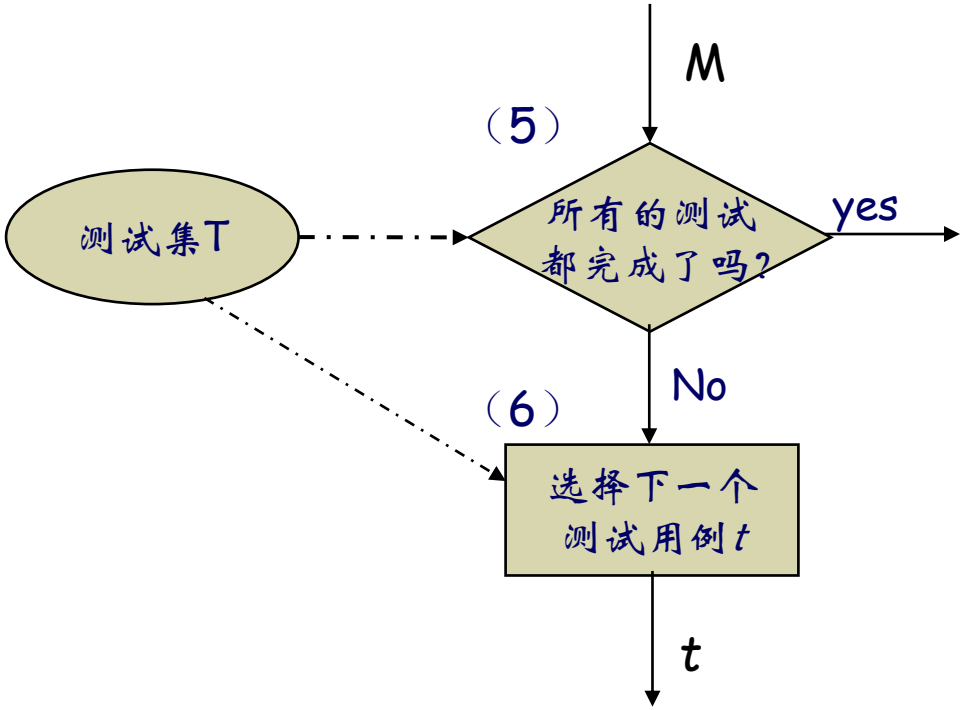
- 第3步和第4步：选择下一个变异体
  - 从L中选择，任意选择



Slides from Aditya P.Mathur

测试充分性评估过程

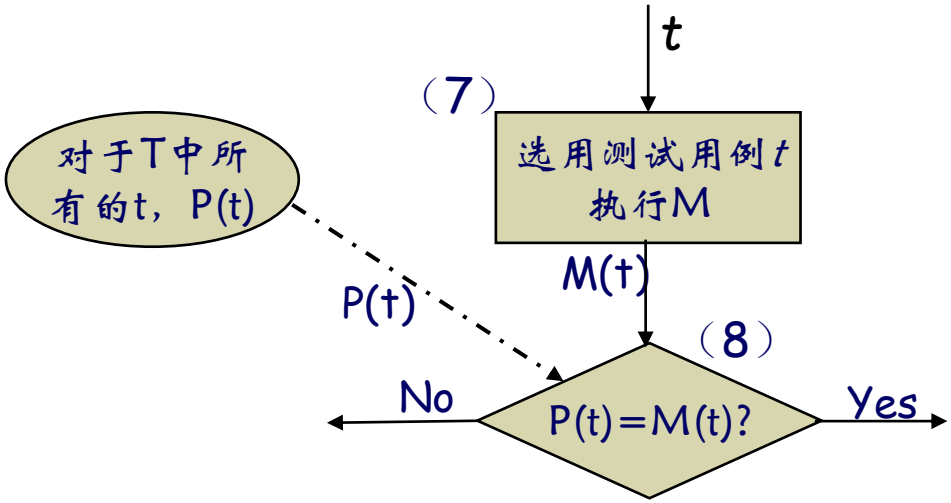
- 第5步和第6步：选择下一个测试用例
  - 是否存在测试t能够区分变异体与被测试程序P
  - 采用测试T中的测试用例执行变异体M。
  - 结束：所有的测试用例执行完毕或者M被某个测试用例区别（杀掉）。



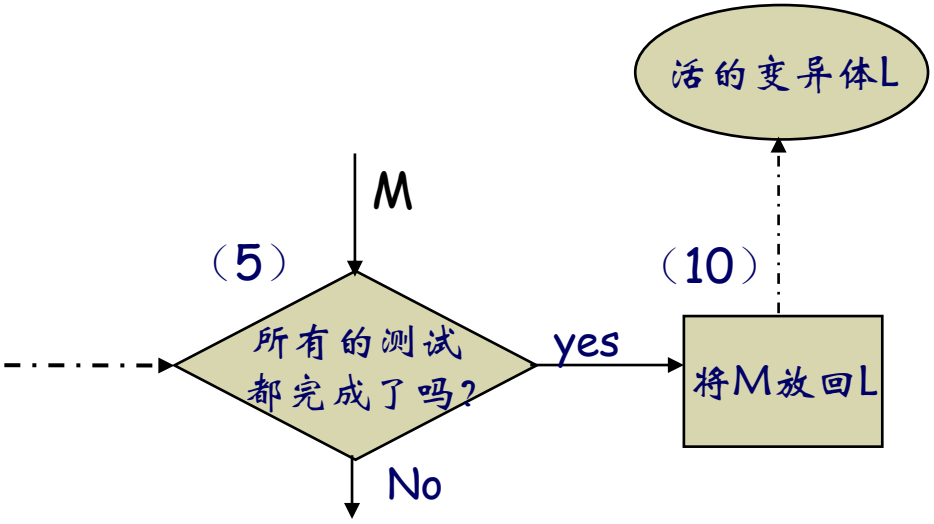
Slides from Aditya P.Mathur

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



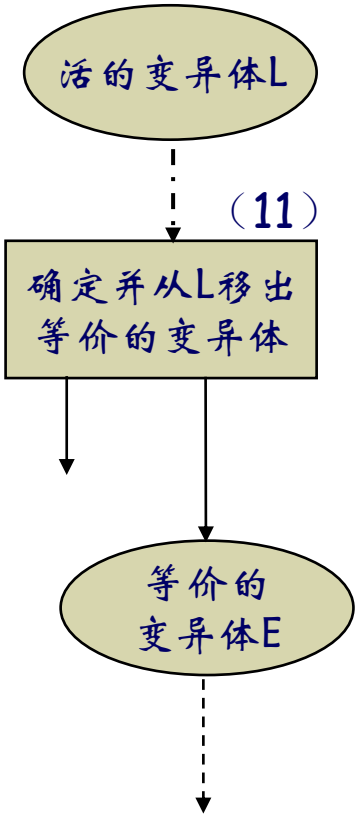
- 第10步：活变异体
  - 如果没有测试用例能够区分变异体与P，则该变异体存活，并被放回活变异体集合L中。



Slides from Aditya P.Mathur

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t: <x=1,y=1>使得P(t)=2，M (t)=1，M不等价于P

测试充分性评估过程

- 第12步：变异数的计算
  - 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变异数
  - T的变异数记为MS（T）

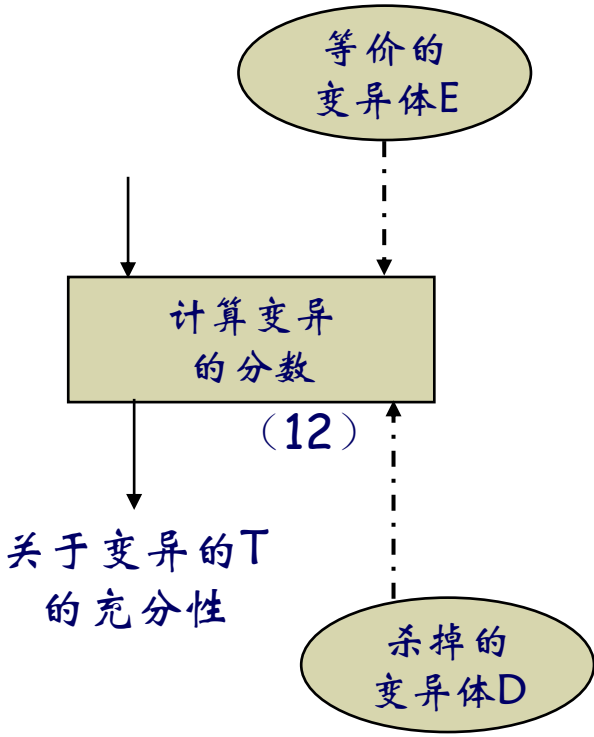
$$MS(T) = \frac{|D|}{|L| + |D|} \quad \text{或} \quad MS(T) = \frac{|D|}{|M| - |E|}$$

其中：|D|表示：杀死的变异体数

|L|表示：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



Slides from Aditya P.Mathur

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与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
```

变异测试的理论基础

- 两个假设来源于软件开发的实验，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。
  - 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。

DeMillo, Lipton, and Sayward

一阶和高阶变异体

- 一阶变异体: A mutant obtained by making exactly "one change" is considered first order
- 高阶变异体:
  - 二阶变异体A mutant obtained by making two changes is a second order mutant.
  - For example, z=x+y; is x=z+y;

  where the variable replacement operator has been applied twice
  - 更高阶：Similarly higher order mutants can be defined.
- **In practice only first order一阶mutants are generated** for two reasons:
  - (a) to lower the cost of testing and
  - (b) most higher order mutants are killed by tests adequate with respect to first order mutants. [See coupling effect later.]

变异测试工具

- There are few mutation testing tools available freely.
  - Proteum for C from Professor Maldonado
  - **muJava for Java from Professor Jeff Offutt.** （工具！）

- ***A typical tool*** for mutation testing offers the following features
  - A selectable palette of mutation operators.
  - Management of test set T
  - Execution of the program under test against T and saving the output for comparison against that of mutants.
  - Generation of mutants

Slides from Aditya P.Mathur

变异测试工具的特征（续）

- Mutant execution and computation of mutation score using user identified equivalent mutants
- Incremental mutation testing: i.e. allows the application of a subset of mutation operators to a portion of the program under test.
- Mothra, an advanced mutation tool for Fortran also provided automatic test generation using DeMillo and Offutt's method

Slides from Aditya P.Mathur

变异与系统测试

- Adequacy assessment using mutation is often recommended only for relatively small units.
  - e.g. a class in Java
  - a small collection of functions in C
- However, given a good tool, one can use mutation to assess adequacy of system tests.
- The following procedure is recommended to assess the adequacy of system tests.

Slides from Aditya P.Mathur

测试步骤

- Step 1: Identify a set U of application units that are critical to the safe and secure functioning of the application. Repeat the following steps for each unit in U.
- Step 2: Select a small set of mutation operators. This selection is best guided by the operators defined by Eric Wong or Jeff Offutt.
- Step 3: Apply the operators to the selected unit.

Slides from Aditya P.Mathur

测试步骤

- Step 4: Assess the adequacy of T using the mutants so generated. If necessary, enhance T.
- Step 5: Repeat Steps 3 and 4 for the next unit until all units have been considered.
- We have now assessed T, and perhaps enhanced it. Note the use of incremental testing and constrained mutation (i.e. use of a limited set of highly effective mutation operators).

Slides from Aditya P.Mathur

变异测试的优点

- 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
  - 通过与测试工具的交互，有选择地使用变异算子
- 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。
- 可以完成语句覆盖和分支覆盖
  - 将每条语句或每个条件用Trap语句代替

    trap 'command' signal

    - signal是要捕获的信号，command是捕获到指定的信号之后，所要执行的命令。

通过捕获 EXIT 信号，我们可以在 shell 脚本中止执行或从函数中退出时，输出某些想要跟踪的变量的值，并由此来判断脚本的执行状态以及出错原因，其使用方法是：

trap 'command' EXIT 或 trap 'command' 0

通过捕获 ERR 信号，我们可以方便的追踪执行不成功的命令或函数，并输出相关的调试信息，以下是一个捕获 ERR 信号的示例程序，其中的$LINENO 是一个 shell 的内置变量，代表 shell 脚本的当前行号。

以下是一个通过捕获 DEBUG 信号来跟踪变量的示例程序：

```
$ cat -n exp2.sh
     1  #!/bin/bash
     2  trap 'echo "before execute line:$LINENO, a=$a,b=$b,c=$c"
DEBUG
     3  a=1
     4  if [ "$a" -eq 1 ]
     5  then
     6     b=2
     7  else
     8     b=1
     9  fi
    10  c=3
    11  echo "end"
```

其输出结果如下：

```
$ sh exp2.sh
before execute line:3, a=,b=,c=
before execute line:4, a=1,b=,c=
before execute line:6, a=1,b=,c=
before execute line:10, a=1,b=2,c=
before execute line:11, a=1,b=2,c=3
end
```

- **Shell产生的三个伪信号**
- **EXIT** 从一个函数中退出或整个脚本执行完毕
- **ERR** 当一条命令返回非零状态时(代表命令执行不成功)
- **DEBUG** 脚本中每一条命令执行之前

变异测试的缺点

- 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

- Mutation testing is the most powerful technique for the assessment and enhancement of tests.
- Mutation, as with any other test assessment technique, must be applied incrementally and with assistance from good tools.
- Identification of equivalent mutants is an undecidable problem--similar the identification of infeasible paths in control or data flow based test assessment.

Slides from Aditya P.Mathur

小结（续）

- While mutation testing is often recommended for unit testing, when done carefully and incrementally, it can also be used for the assessment of system and other types of tests applied to an entire application.tests.

- Mutation is a highly recommended technique for use in the assurance of quality of highly available, secure, and safe systems.

Slides from Aditya P.Mathur

- FSE2014， 优秀论文

Are Mutants a Valid Substitute for Real Faults in Software Testing?

René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser

(University of Washington, USA; University of Waterloo, Canada; University of Sheffield, UK)