第六章　变异与模糊测试

蔡彦

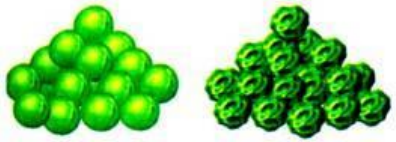
计算机科学国家重点实验室
中国科学院软件研究所

yancai@ios.ac.cn

第七章 变异与模糊测试（3学时）

1. 变异测试
2. 模糊测试

背景——从生物变异开始

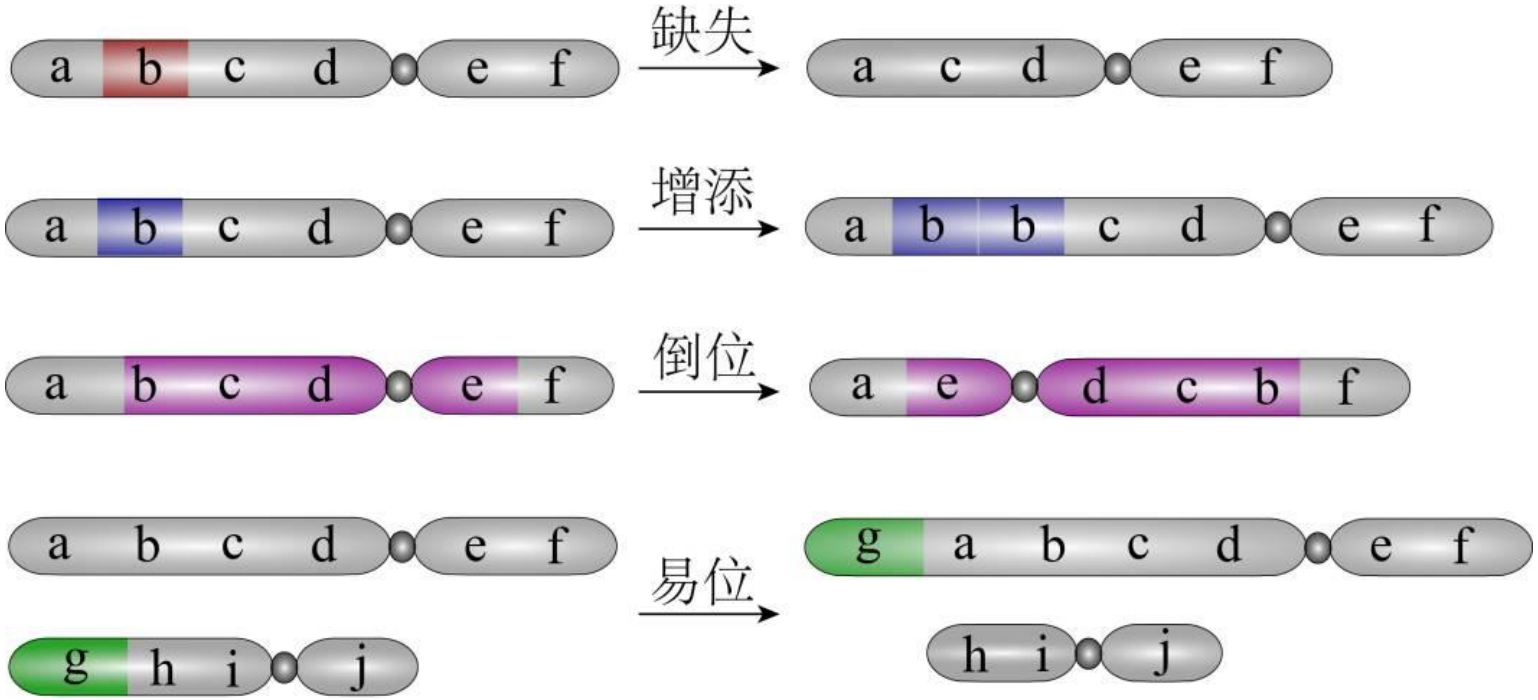


豌豆的圆粒与皱粒

番茄的红果与黄果

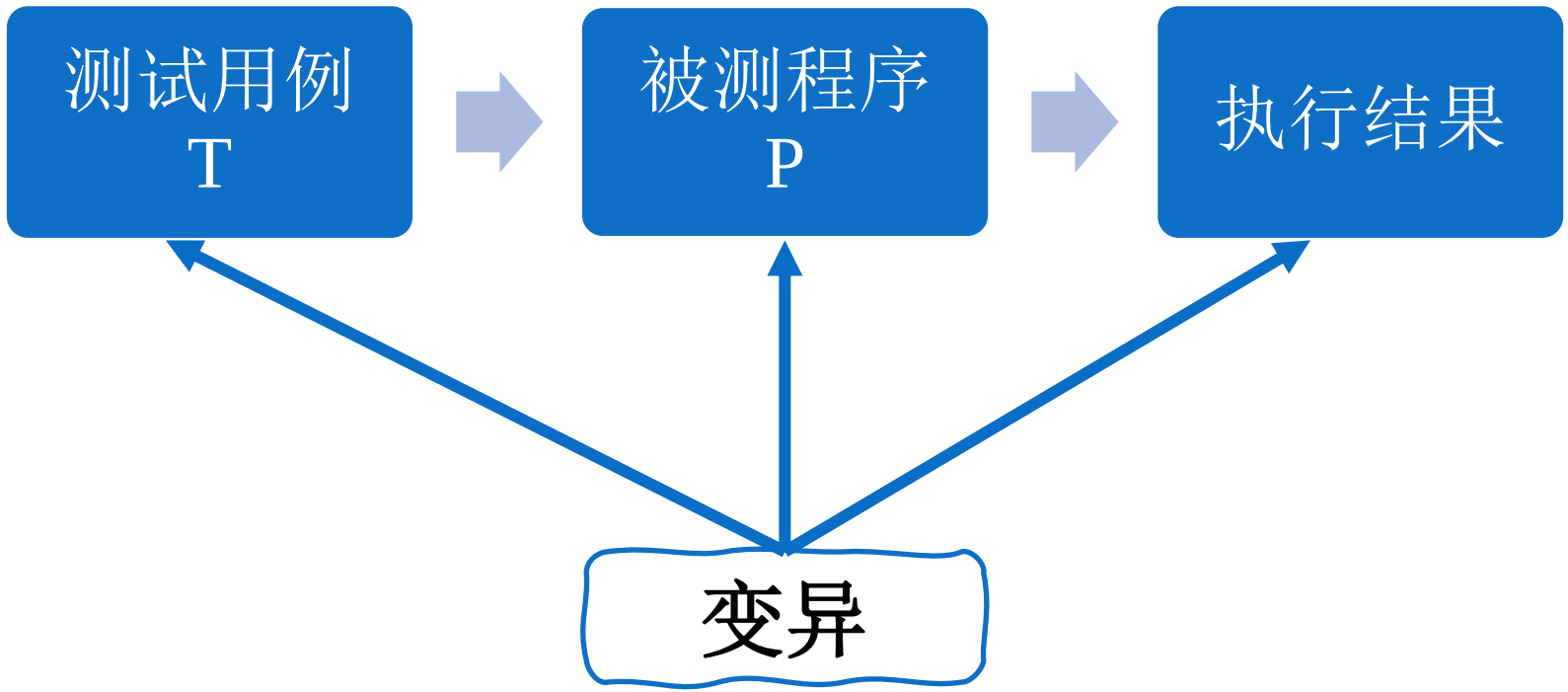
兔的白毛与黑毛

鸡的玫瑰冠与单冠



软件测试→变异测试

背景——变异与软件测试



背景——从测试覆盖率的局限性谈起

很多时候我们会用单元测试执行后的代码覆盖率来衡量测试的充分性和完整性，问题是有了很多测试用例，同时又有很高的白盒覆盖率，是否代码质量真的就高枕无忧了吗？

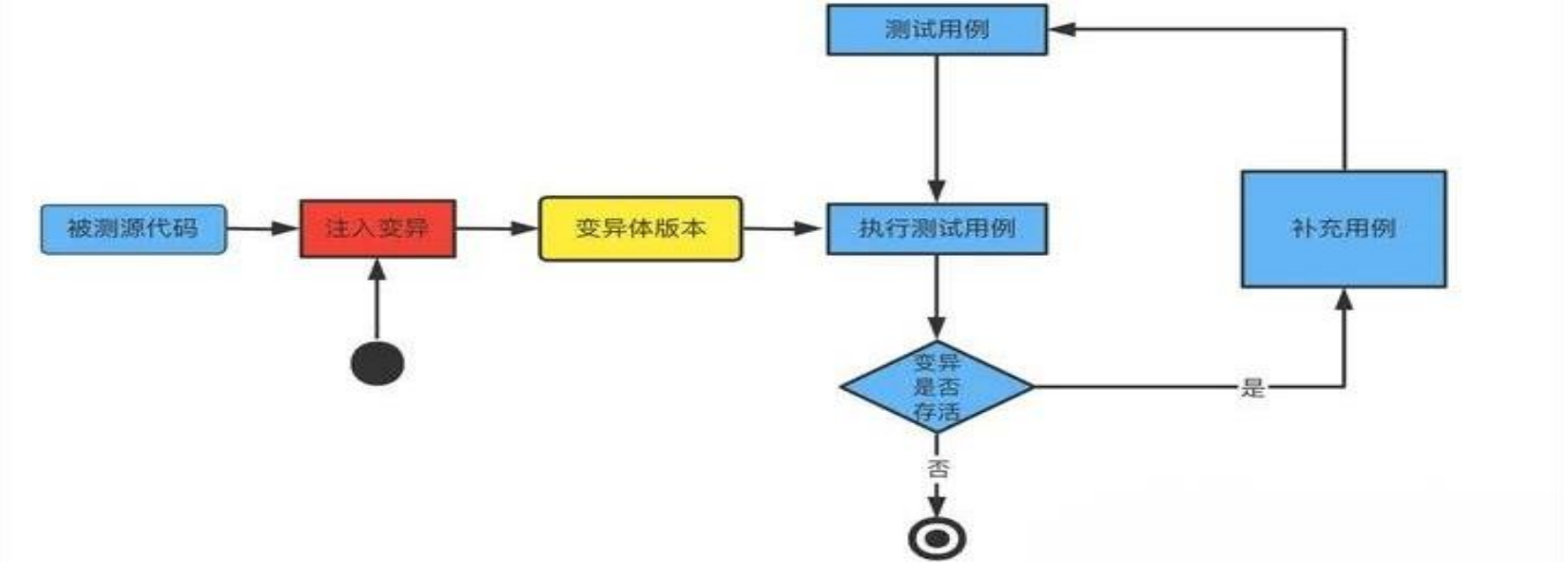
程序p	测试用例
...	...
def only_correct_data (a,b,c): return (a/(b-c));	def test_only_correct_data(self): #only tests with data that leads to correct results self.assertEqual(only_correct_data(1,2,3),-1) self.assertEqual(only_correct_data(2,3,1),1) self.assertEqual(only_correct_data(0,2,3),0)
...	...

背景——基本思想

- 变异测试是一种基于故障注入的测试技术，将错误代码插入到被测代码中，以验证当前测试用例是否可以发现注入的错误。该测试手段理论上属于白盒测试范畴。

- 变异测试的主要目的是为了验证测试用例的有效性，在注入变异后，测试用例能发现该错误，则表明用例有效的；反之，表明测试用例是无效的，需要补充该变异的测试用例。

- 变异测试有助于评估测试用例的质量，以帮助测试人员编写更有效的测试用例。测试人员设计的测试用例发现的变异体越多，表明其设计的测试用例质量就越高。



背景——起源

- 变异测试是一种 fault-based 的软件测试技术。这项技术已经广泛研究并使用了三十余年。它为软件测试贡献了一系列方法，工具，和可靠的结果。

Author	Title	Type	University	Year
Acree [2]	On Mutation	PhD	Georgia Institute of Technology	1980
Hanks [108]	Testing Cobol Programs by Mutation	PhD	Georgia Institute of Technology	1980
Budd [34]	Mutation Analysis of Program Test Data	PhD	Yale University	1980
Tanaka [228]	Equivalence Testing for Fortran Mutation System Using Data Flow Analysis	PhD	Georgia Institute of Technology	1981
Morell [164]	A Theory of Error-Based Testing	PhD	University of Maryland at College Park	1984
Offutt [194]	Automatic Test Data Generation	PhD	Georgia Institute of Technology	1988
Craft [48]	Detecting Equivalent Mutants Using Compiler Optimization Techniques	Master	Clemson University	1989
Choi [46]	Software Testing Using High-performance Computers	PhD	Purdue University	1991
Krauser [132]	Compiler-Integrated Software Testing	PhD	Purdue University	1991
Fichter [91]	Parallelizing Mutation on a Hypercube	Master	Clemson University	1991
Lee [141]	Weak vs. Strong: An Empirical Comparison of Mutation Variants	Master	Clemson University	1991
Zapf [261]	A Distributed Interpreter for the Mothra Mutation Testing System	PhD	Clemson University	1993
Delamaro [52]	Proteum - A Mutation Analysis Based Testing Environment	PhD	University of São Paulo	1993
Wong [248]	On Mutation and Data Flow	PhD	Purdue University	1993
Pan [197]	Using Constraints to Detect Equivalent Mutants	Master	George Mason University	1994
Untch [236]	Schema-based Mutation Analysis: A New Test Data Adequacy Assessment Method	PhD	Clemson University	1995
Ghosh [98]	Testing Component-Based Distributed Applications	PhD	Purdue University	2000
Ding [74]	Using Mutation to Generate Tests from Specifications	Master	George Mason University	2000
Okun [195]	Specification Mutation for Test Generation and Analysis	PhD	University of Maryland Baltimore	2004
Ma [148]	Object-oriented Mutation Testing for Java	PhD	KAIST University in Korea	2005
May [161]	Test Data Generation: Two Evolutionary Approaches to Mutation Testing	PhD	University of Kent	2007
Hussain [116]	Mutation Clustering	Master	King's College London	2008
Adamopoulos [4]	Search Based Test Selection and Tailored Mutation	Master	King's College London	2009

背景——基本假设

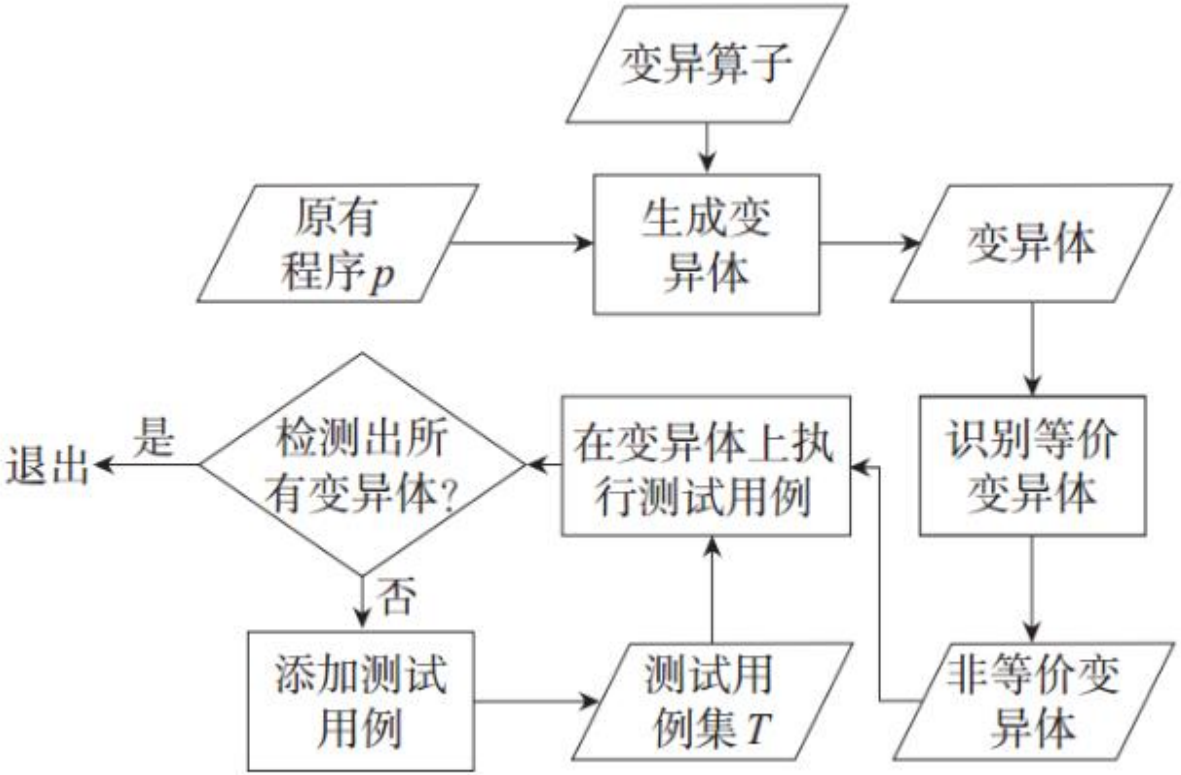
- 两大基本假设：变异测试旨在找出有效的测试用例，发现程序中真正的错误。在一个软件中，潜在BUG的数量是巨大的，通过生成突变体来全面覆盖所有的错误是不可行的。所以，传统的变异测试旨在寻找这些错误的子集，能尽量充分地近似描述这些BUG。于是，变异测试的理论基于两条假设：Competent Programmer Hypothesis (CPH) 和 Coupling Effect(CE)。

  - CPH：假设编程人员是有能力的，他们尽力去更好地开发程序，达到正确可行的结果，而不是搞破坏。它关注的是程序员的行为和意图。

  - CE：(耦合效应)更加关注在变异测试中错误的类别。一个简单的错误产生往往是由于一个单一的变异（例如句法错误），而一个庞大复杂的错误往往是由于多出变异所导致。复杂变异体往往是由诸多简单变异体组合而成。

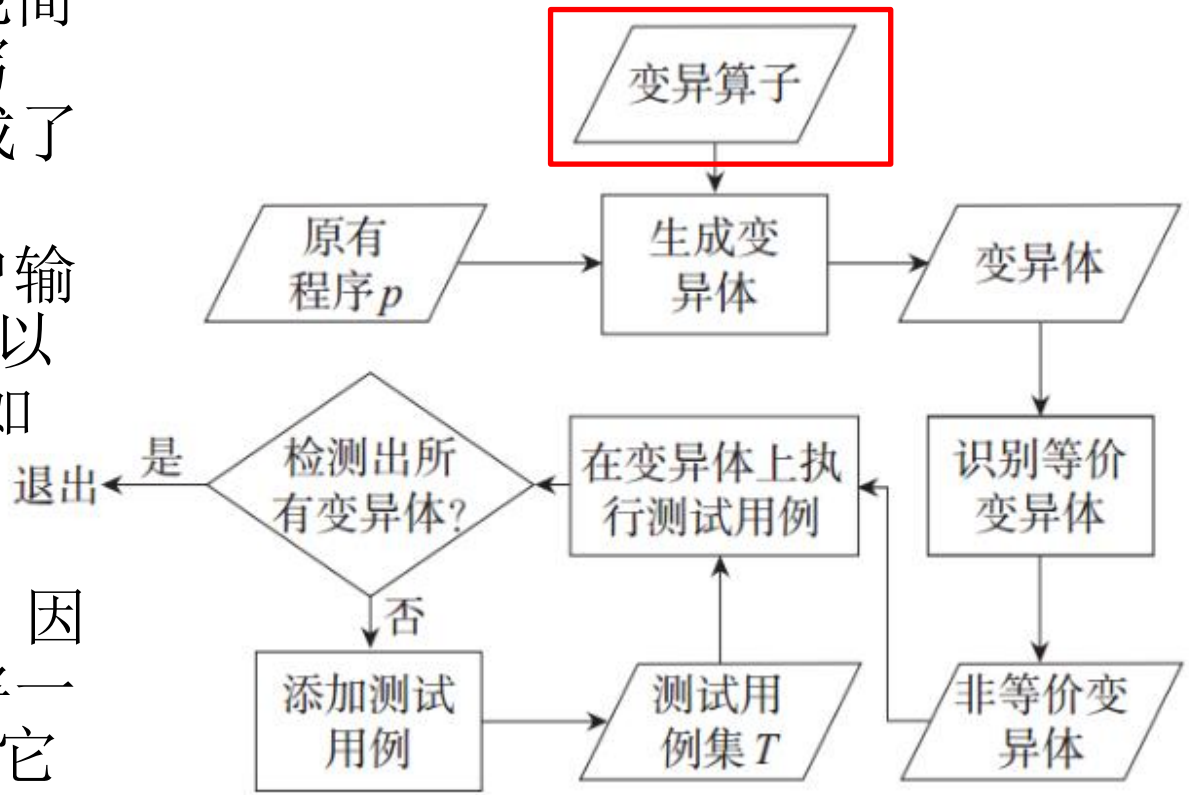
背景——概念定义

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。



原理——变异算子

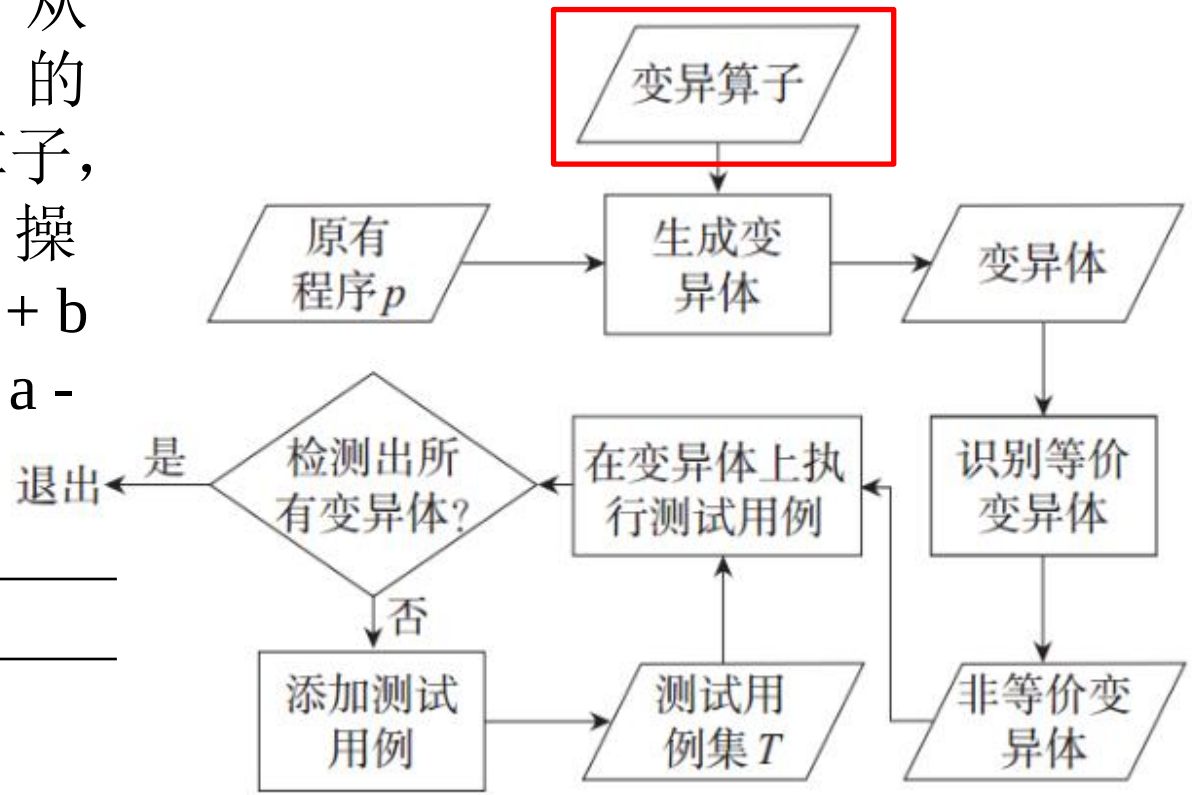
- 经验表明，人们在编程过程中都容易犯简单的非技术性错误，如应该在程序中写X<Y+1却因粗心漏掉后面的常数而写成了X<Y。
- 变异算子可以用来模拟那些典型的用户输入错误(如运算符或变量名使用错误等)以及强制使某些表达式满足一定的条件(如使表达式的值为0)。
- 变异算子是一个将原始程序(original program)转换成变异体(mutant)的规则，因此也称为变异规则或变异转换。可以将一个具体的变异算子O看成是一个函数，它将被测程序P映射为k个变异体。



原理——变异算子

在符合语法规则前提下， 变异算子定义了从原有程序生成差别极小程序（即变异体） 的转换规则。下表给出了一个典型的变异算子，该变异算子将“＋” 操作符变异为“-” 操作符。选择被测程序 p 中的条件表达式 a + b > c 执行该变异算子， 将得到条件表达式 a - b > c ， 并生成变异体 p’ 。

程序p	变异体p’
...	...
if (a+b>c)	if (a-b>c)
return true;	return true;
...	...



原理——变异算子

- 常用的变异算子有变量替换、运算符替换、常数替换和删除或插入整条语句等。

Mutant Operator	In P	In Mutant
变量替换	z=x*y+1	x=x*y+1
关系运算符替换	If (x<y)	If (x>y) If (x>=y) ...
算术运算符替换	z=x*y+1	z=x*y-1 z=x+y-1 ...
常量加减	z=x*y+1	z=(x+1)*y+1 z=x*(y+1)+1 ...
用0替换	z=x*y+1	z=0*y+1 z=0 ...

原理——变异算子的合理性

- 变异算子设计得合理有利于提高变异测试的效率。

对同一种程序设计语言而言，由不同的人所设计的变异算子也可能不同，那么如何对所设计的变异算子的合理性进行评价呢?

设Sl、S2是为同一种语言设计的两个不同的变异算子集，一般来说，如果由Sl生成的变异体比s2生成的变异体能够检测到更多故障，则认为Sl要比S2更加合理。

原理——变异算子相关研究

- Offutt和King在已有研究工作的基础上，于 1987年针对 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	CSR	用常量替代变量
10	DER	DO语句修改
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

原理——变异算子相关研究

•AAR、ACR、ASR、CAR、CNR、CRP、CSR、SAR、SCR、SRC和SVR属于操作数替换算子类，它们均表现为将一个操作数替换为另外一个合法的操作数，主要用于模拟程序员在编程过程中对变量名和数组引用名的误用这种场合，如变量的定义与引用、数组下标越界和非法的算术运算等计算型故障。

•ABS、AOR、LCR、ROR和UOI属于表达式修改算子类，这一类算子表现为用插入新的运算符或替换原有运算符来修改表达式，主要用于模拟程序员在编程时对表达式误用这种场合，如表达式内部用错算术运算符或关系运算符而导致分支型和循环型故障。

•DER、DSA、GLR、RSR、SAN和SDL属于语句修改算子类，它们用于修改整条语句，如删除动态内存释放语句可以模拟内存泄漏故障。

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	CSR	用常量替代变量
10	DER	DO语句修改
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

原理——变异算子相关研究

Mathur、DeMillo等人对c语言设计了一组变异算子，但由于C语言的语法要比Fortran语言复杂得多，因而设计出77个变异算子。可以将这些变异算子分成四大类：

1)语句变异(statement mutations)
2)运算符变异(operator mutations)
3)变量变异(variable mutations)
4)常量变异(constant mutations)

每大类下又细分为多个小类，如变量变异又可细分为标量变量引用替换、数组元素替换、指针引用替换、结构引用替换、数组下标变异和结构成员替换等，详细情况可以参考以下文献。

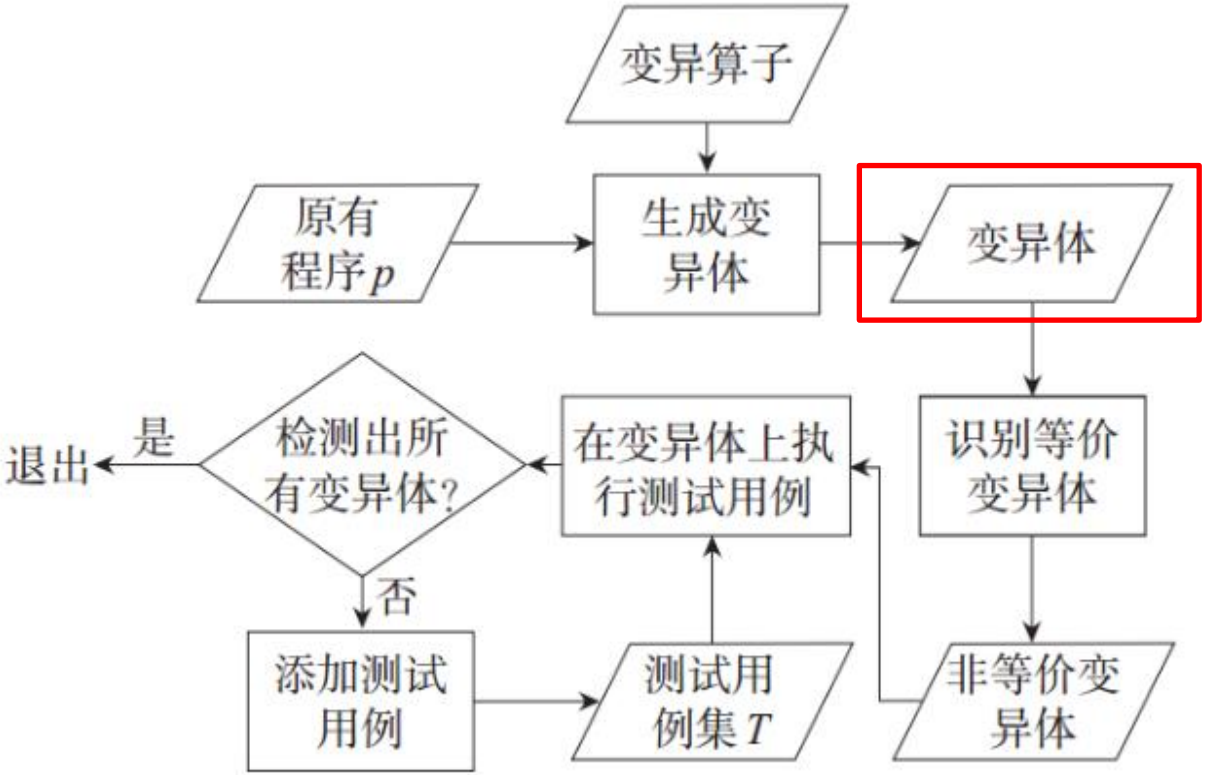
Agrawal, H., DeMillo, R.A., Hathaway, B.G., Hsu, W., Hsu, W., Krauser, E.W., Martin, R.J., Mathur, A.P., & Spafford, E.H. (2006). DESIGN OF MUTANT OPERATORS FOR THE C PROGRAMMING LANGUAGE.

Operator	Domain	Description	Page
CGCR	Constants	Constant replacement using global constants	63
CLSR	Constants	Constant for scalar replacement using local constants	63
CGSR	Constants	Constant for scalar replacement using global constants	63
CRCR	Constants	Required constant replacement	62
CLCR	Constants	Constant replacement using local constants	63
OAAA	‡	arithmetic assignment mutation	49
OAAN	‡	arithmetic operator mutation	49
OABA	†	arithmetic assignment by bitwise assignment	50
OABN	†	arithmetic operator by bitwise operator	50
OAEA	†	arithmetic assignment by plain assignment	50
OALN	†	arithmetic operator by logical operator	50
OARN	†	arithmetic operator by relational operator	50
OASA	†	arithmetic assignment by shift assignment	50
OASN	†	Arithmetic operator by shift operator	50
OBAA	†	Bitwise assignment by arithmetic assignment	50
OBAN	†	Bitwise operator by arithmetic assignment	50
OBBA	‡	Bitwise assignment mutation	49
OBBN	‡	Bitwise operator mutation	49
OBEA	†	Bitwise assignment by plain assignment	50
OBLN	†	Bitwise operator by logical operator	50
OBNG	†	Bitwise negation	52
OBRN	†	Bitwise operator by relational operator	50
OBSA	†	Bitwise assignment by shift assignment	50
OBSN	†	Bitwise operator by shift operator	50
OCOR	Casts	Cast operator by cast operator	53
OEAA	†	Plain assignment by arithmetic assignment	50
OEBA	†	Plain assignment by bitwise assignment	50
OESA	†	Plain assignment by shift assignment	50

原理——变异体

在完成变异算子设计后，通过在原有被测程序上执行变异算子可以生成大量变异体M，在变异测试中，变异体一般被视为含缺陷程序。

根据执行变异算子的次数，可以将变异体分为**一阶变异体**和**高阶变异体**等。



原理——变异体

在完成变异算子设计后，通过在原有被测程序上执行变异算子可以生成大量变异体M，在变异测试中，变异体一般被视为含缺陷程序。根据执行变异算子的次数，可以将变异体分为**一阶变异体**和**高阶变异体**等。

- 一阶变异体：在原有程序 p 上执行单一变异算子并形成变异体 p′，则称p′为p的一阶变异体。
- 高阶变异体：在原有程序 p 上依次执行多次变异算子并形成变异体 p′，则称 p′为 p 的高阶变异体。若在 p 上依次执行 k 次变异算子并形成变异体 p′，则称 p′为 p 的 k 阶变异体。

输入：a, x, y

```
1. z=x;
2. z=z+y;
3. if (a>0)
4.     return z;
5. else
6.     return 2*x+z;
```

测试用例	a=1	a=-1
原有程序	x+y	3x+y
变异体1	x+y+1	3x+y+3
变异体2	x+y-1	3x+y-1
变异体12	x+y	3x+y+2

两个测试用例：a=1和a=-1
其中，
变异体1将第一行变为z=++x; //一阶变异体
变异体2将第二行变为z=z+--y; //一阶变异体
变异体12合并了前两个变异体; //二阶变异体

原理——变异体的类型

在之前的定义中，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（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’是可存活的（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。

程序p	变异体p’
…	…
for (int i=0;i<10;i++)	for (int i=0;i!=10;i++)
sum+=a[i];	sum+=a[i];
…	…

原理——变异体的类型

- 导致变异体无法被杀死的原因主要有两个：
  - 测试数据集还不够充分，通过扩充测试数据集便能将该变异体杀死。
  - 该变异体在功能上等价于原始程序，称这类变异体为等价变异体(equivalent mutant)。

- 杀死变异体的过程一直执行到杀死所有变异体或变异充分度已经达到预期的要求。

- **变异充分度**:已杀死的变异体数目与所有已产生的非等价变异体数目的比值，即变异充分度=D／(M-E)，其中D为已经被杀死的变异体个数，M为所有已产生的变异体总数；E为所有已产生的变异体中与原来程序等价的变异体个数。

原理——等价变异体的检测

- 等价变异体检测是一个不可判定问题，因此需要测试人员借助手工方式予以完成。等价变异体在语法层次上有微小的差别，但是在语义层次上是一致的。有研究人员发现，在生成的大量变异体中，等价变异体所占比例一般介于10%~40%。在等价变异体的检测上，主要有两类方法。

1) 等价变异体**静态检测**法：该方法基于如下猜测，源代码在编译时借助优化规则可以生成语义等价代码；
2) 等价变异体**动态检测**法：Adamopoulos 等人提出一种基于遗传算法的协作演化法（即测试用例和变异体同时进行演化）来检测可能的等价变异体。他们通过设置合理的适应值函数， 确保当变异体是等价变异体时， 该函数可以返回一个很小的适应值。基于该适应值函数， 群体在演化过程中可以有效淘汰部分等价变异体， 同时将那些难以检测的变异体和检测能力强的测试用例均保留下来。

原理——变异体实例

- C语言程序求给定的两个整数x和y中的较大者，以及它的一个变异体。

C语言源程序：

```
1. int max (int x, int y){
2. int max_answer;
3. if (x>y)
4.     max_answer=x;
5. else
6.     max_answer=y;
7. return max_answer;
8. }
```

源程序变异体：

```
1. int max (int x, int y){
2. int max_answer;
3. if (x<y)
4.     max_answer=x;
5. else
6.     max_answer=y;
7. return max_answer;
8. }
```

测试数据(X=3,Y=4)能够将变异体杀死，而测试数据(x=3,y=3)不能将变异体杀死。

原理——变异体实例

- 变异体类型——变量间替换

源程序：

```
1. int min (int i, int j){
2. int min_value;
3. min_value=i;
4. if (j<i)
5.     min_value=j;
6. return min_value;
7. }
```

变异体1：

```
1. int min (int i, int j){
2. int min_value;
3. min_value=j;
4. if (j<i)
5.     min_value=j;
6. return min_value;
7. }
```

变异体2：

```
1. int min (int i, int j){
2. int min_value;
3. min_value=i;
4. if (j<min_value)
5.     min_value=j;
6. return min_value;
7. }
```

变异体3：

```
1. int min (int i, int j){
2. int min_value;
3. min_value=i;
4. if (j<i)
5.     min_value=i;
6. return min_value;
7. }
```

属于变量间替换变异，即用另外一个语法上合法的变量来替换当前变量。

原理——变异体实例

- 变异体类型——运算符间替换

源程序：

```
1. int min (int i, int j){
2. int min_value;
3. min_value=i;
4. if (j<i)
5.     min_value=j;
6. return min_value;
7. }
```

变异体4：

```
1. int min (int i, int j){
2. int min_value;
3. min_value=j;
4. if (j>i)
5.     min_value=j;
6. return min_value;
7. }
```

属于运算符间替换变异，即用另外一个语法上合法的运算符来替换当前的运算符。

原理——变异体实例

- 变异体类型——等价变异体的出现

源程序：

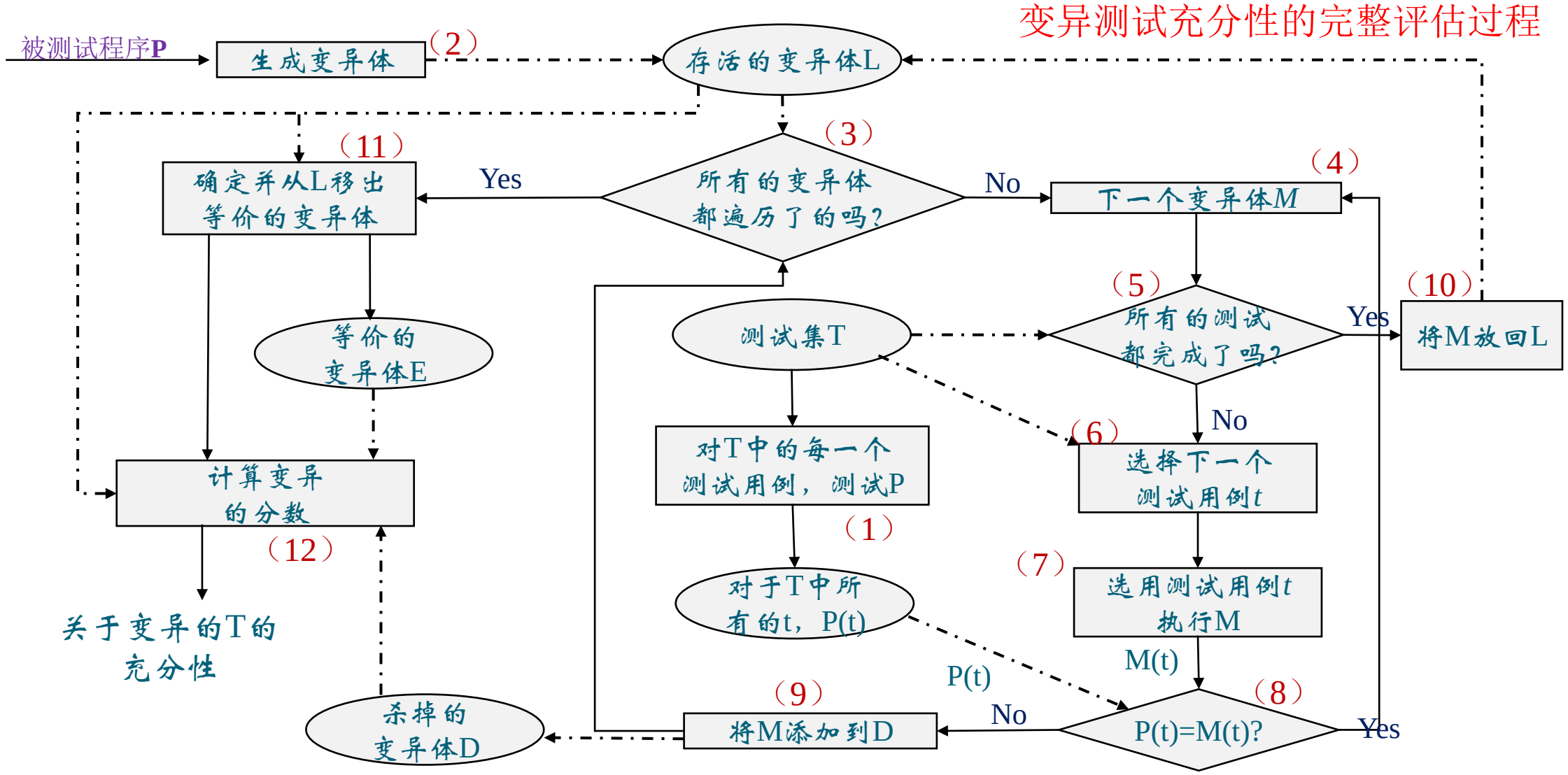
```
1. int min (int i, int j){
2. int min_value;
3. min_value=i;
4. if (j<i)
5.     min_value=j;
6. return min_value;
7. }
```

变异体5：

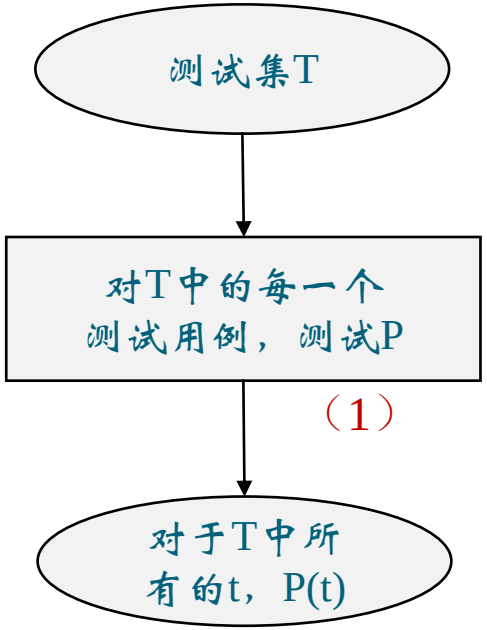
```
1. int min (int i, int j){
2. int min_value;
3. min_value=j;
4. if (j<min_value)
5.     min_value=j;
6. return min_value;
7. }
```

用变量min_value来替换变量i，而在其前的一条语句中变量i的值被赋值给了变量min_value，此时变量i和变量min_value有相同的值。因此，该变异体是一个等价变异体。

变异测试充分性


变异测试充分性的完整评估过程
被测试程序P
生成变异体
(2)
存活的变异体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
等价的
变异体E
(5)
所有的测试
都完成了吗?
Yes
(10)
将M放回L
测试集T
No
(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
对于T中所
有的t，P(t)
M(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

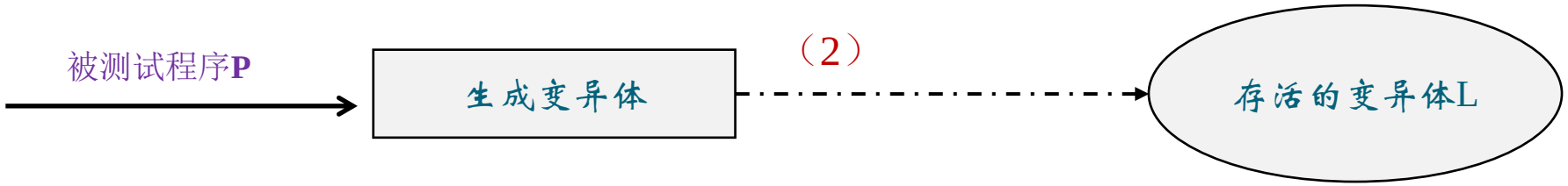
变异测试充分性（1）



第1步：执行程序

- 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，得到对应的P(t)数据库

变异测试充分性（2）



第2步：生成变异体

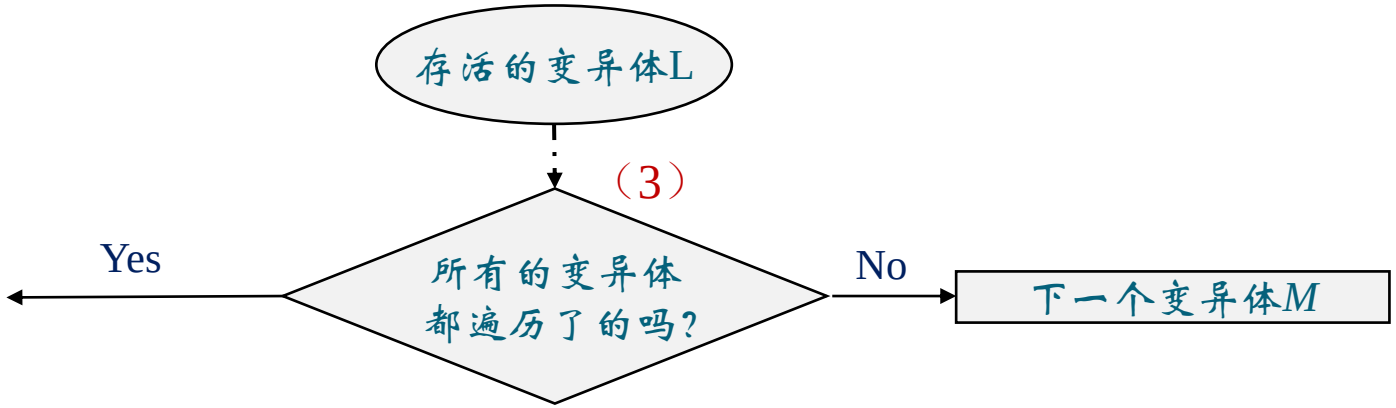
例如：“＋”运算变成“－”运算，“×”运算变成“/”运算等

系统的生成方法：通过变异算子生成

第二步的结果是：尚且存活的变异体

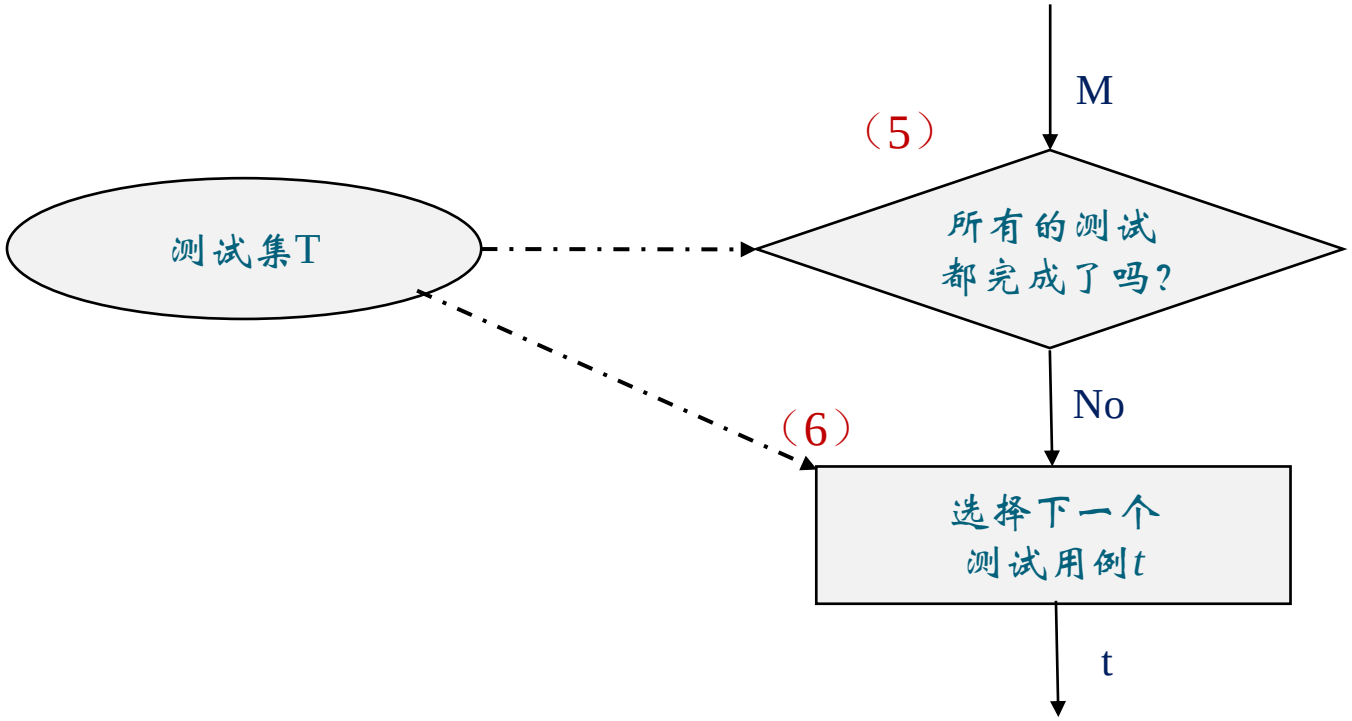
这些变异体还没有与程序P区分，即暂时没有被杀死。

变异测试充分性（3-4）



第3步和第4步：选择下一个变异体。从存活的变异体集合L中任意选择；

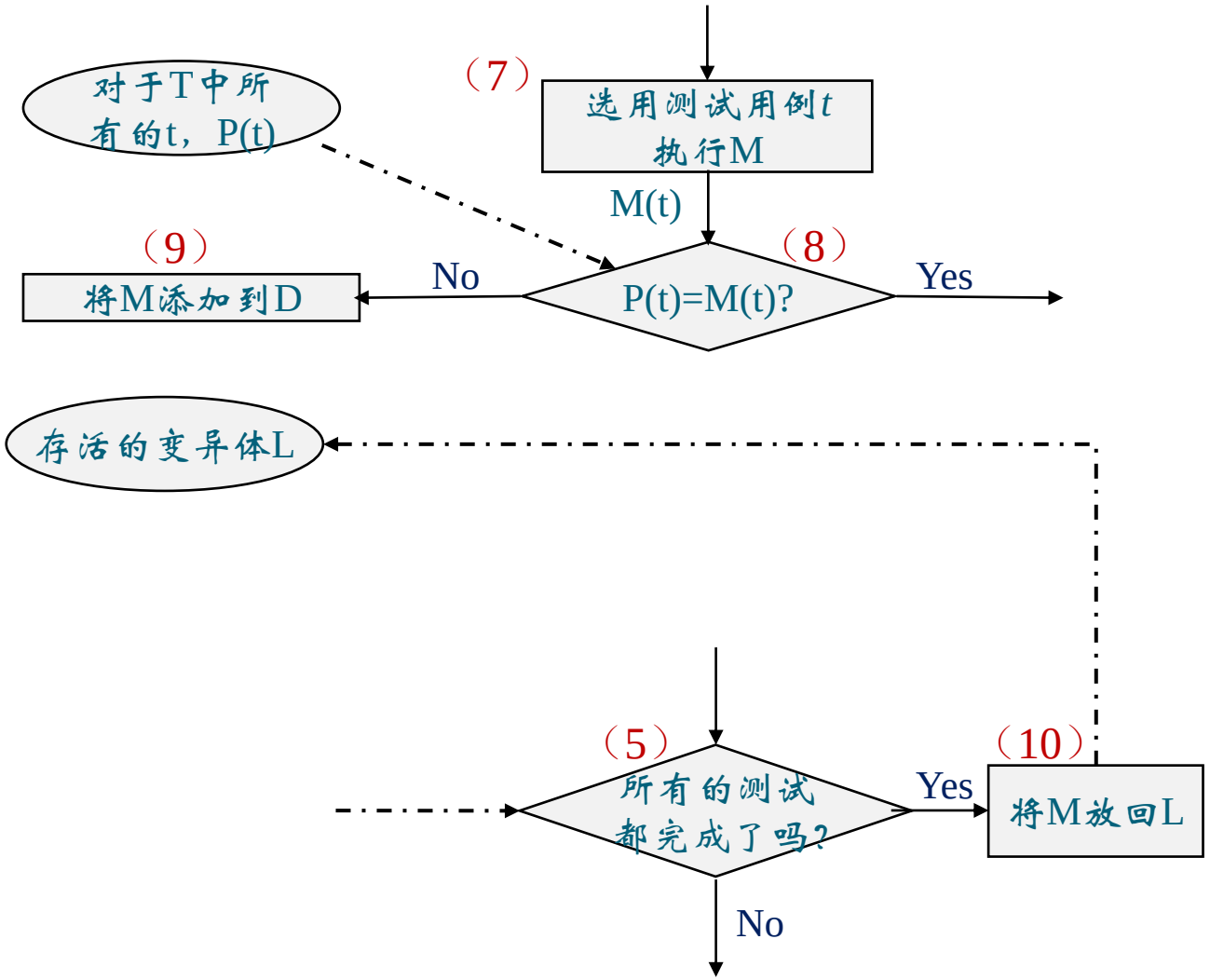
变异测试充分性（5-6）



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）。

变异测试充分性（7-10）



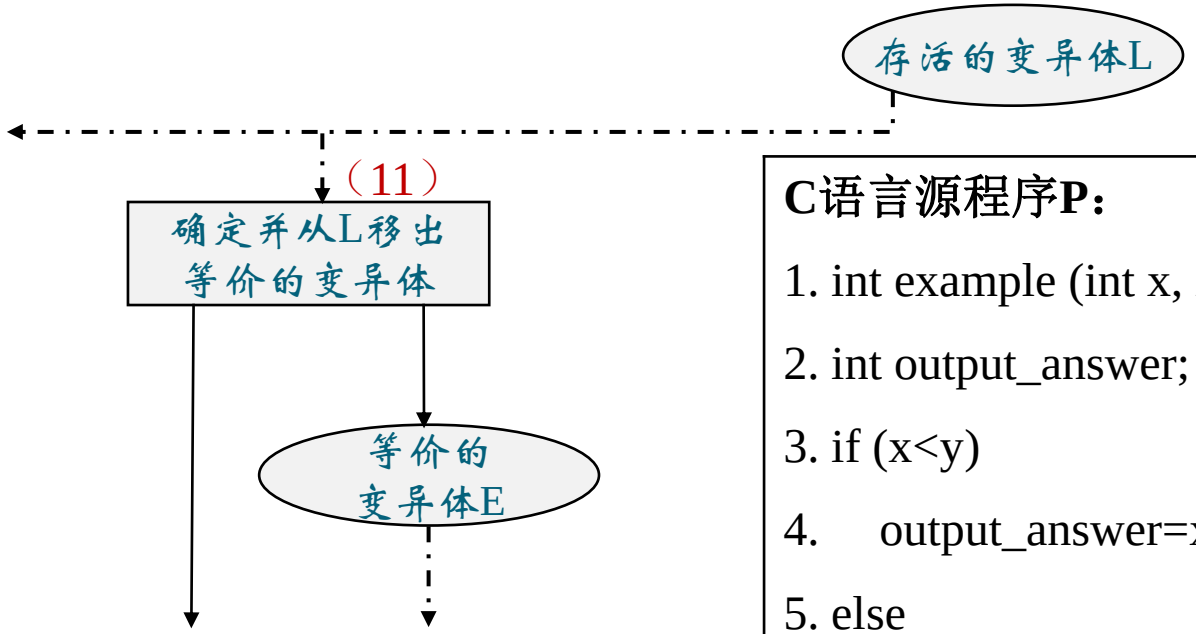
- 第7-9步：变异体执行和分类

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：存活变异体

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

变异测试充分性（11）



● 第11步：等价变异体的判断

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。

C语言源程序P：

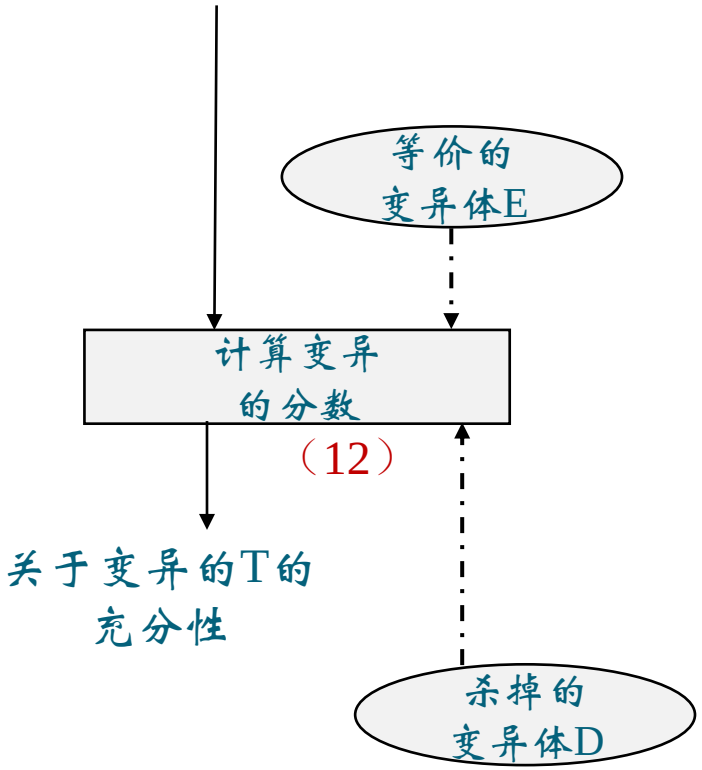
```
1. int example (int x, int y){
2. int output_answer;
3. if (x<y)
4.     output_answer=x+y;
5. else
6.     output_answer=x*y;
7. return output_answer;
8. }
```

源程序变异体M：

```
1. int example (int x, int y){
2. int output_answer;
3. if (x<y+1)
4.     output_answer=x+y;
5. else
6.     output_answer=x*y;
7. return output_answer;
8. }
```

测试输入t: <x=1,y=1>使得P(t)=2，M(t)=1，M不等价于P

变异测试充分性（12）



● 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）

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

其中，|D|表示杀死的变异体数，
|L|表示存活的变异体数，
|M|表示流程图第2步中生成的所有变异体数，
|E|表示等价变异体数；

变异测试充分性（13）拓展

C语言源程序P：

```
1. int example (int x, int y){
2. int output_answer;
3. if (x<y)
4.     output_answer=x+y;
5. else
6.     output_answer=x*y;
7. return output_answer;
8. }
```

源程序变异体M：

```
1. int example (int x, int y){
2. int output_answer;
3. if (x<y+1)
4.     output_answer=x+y;
5. else
6.     output_answer=x*y;
7. return output_answer;
8. }
```

- 拓展：测试增强
- 源程序P使用测试集T

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

并通过测试

- 变异体M使用测试集T运行结果与源程序P相同，无法区分M和P
- 增加一个测试数据t5:<x=1, y=1>，成功区分M和P，表明增强了测试集T

变异测试应用——case1

C语言源程序P：

```
1. int foo (int x, int y){
2.     return (x-y);
3. }
```

return (x+y);

- case1：考虑左侧这个程序实例

该程序本身的目的应该是计算并返回两数之和，那么显然此时的代码是错误的；

假设foo已经由测试集合T测试通过，T包含以下两个测试用例：

{ 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正确的期望值，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。

变异测试应用——case1

C语言源程序P：

```
1. int foo (int x, int y){
2.     return (x-y);
3. }
```

return (x+y);

- case1：考虑左侧这个程序实例

该程序本身的目的应该是计算并返回两数之和，那么显然此时的代码是错误的；

考虑foo生成的三个变异体：

变异体M1：

```
1. int foo (int x, int y){
2.     return (x+y);
3. }
```

变异体M2：

```
1. int foo (int x, int y){
2.     return (x-0);
3. }
```

变异体M3：

```
1. int foo (int x, int y){
2.     return (0+y);
3. }
```

变异测试应用——case1

考虑foo生成的三个变异体：

变异体M1：

```
1. int foo (int x, int y){
2.      return (x+y);
3. }
```

变异体M2：

```
1. int foo (int x, int y){
2.      return (x-0);
3. }
```

变异体M3：

```
1. int foo (int x, int y){
2.      return (0+y);
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个存活变异体，1个被杀死，计算变异数需要对所有活的变异体判断等价。

变异测试应用——case1

考虑源程序P和两个存活的变异体：

变异体M1：

```
1. int foo (int x, int y){
2.      return (x+y);
3. }
```

变异体M2：

```
1. int foo (int x, int y){
2.      return (x-0);
3. }
```

C语言源程序P：

```
1. int foo (int x, int y){
2.      return (x-y);
3. }
```

关注变异体M1：

一个测试数据如果能够区分M1和源程序P，其一定满足条件：

$x-y \neq x+y$ 即 $y \neq 0$

于是产生新的一组测试数据：t3: <x=1, y=1>

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与源程序P产生区别，同时发现了错误。

变异测试实践——case2

考虑现实的变异测试工具Mutpy（Python突变测试模块）：

Python源程序calculate.py：

```
1. def mul(x, y):
2.     return x*y;
```

Python测试程序test_calculate.py：

```
1. from unittest import TestCase
2. from calculate import mul
3.
4. class CalculatorTest(TestCase):
5.         def test_mul(self):
6.                 self.assertEqual(mul(2,2),4)
```

参考链接 https://github.com/mutpy/mutpy

变异测试实践——case2

变异测试工具Mutpy（Python突变测试模块）：

Python源程序calculate.py：

```
1. def mul(x, y):
2.     return x*y;
```

Python测试程序test_calculate.py：

```
1. from unittest import TestCase
2. from calculate import mul
3.
4. class CalculatorTest(TestCase):
5.     def test_mul(self):
6.         self.assertEqual(mul(2,2),4)
```

命令行输入

python.exe F:\ANACONDA\Scripts\mut.py --target calculate --unit-test test_calculate -m

```
[*] Start mutation process:
   - targets: calculate
   - tests: test_calculate
[*] 1 tests passed:
   - test_calculate [0.00000 s]
[*] Start mutants generation and execution:
   - [#   1] AOR calculate:
----------------------------------------------------------------------
  1: def mul(x, y):
- 2:     return x * y
+ 2:     return x / y
----------------------------------------------------------------------
[0.00103 s] killed by test_mul (test_calculate.CalculatorTest)
   - [#   2] AOR calculate:
----------------------------------------------------------------------
  1: def mul(x, y):
- 2:     return x * y
+ 2:     return x // y
----------------------------------------------------------------------
[0.00000 s] killed by test_mul (test_calculate.CalculatorTest)
   - [#   3] AOR calculate:
----------------------------------------------------------------------
  1: def mul(x, y):
- 2:     return x * y
+ 2:     return x ** y
----------------------------------------------------------------------
[0.00000 s] survived
[*] Mutation score [10.02176 s]: 66.7%
   - all: 3
   - killed: 2 (66.7%)   - survived: 1 (33.3%)
   - incompetent: 0 (0.0%)
   - timeout: 0 (0.0%)
```

变异测试工具

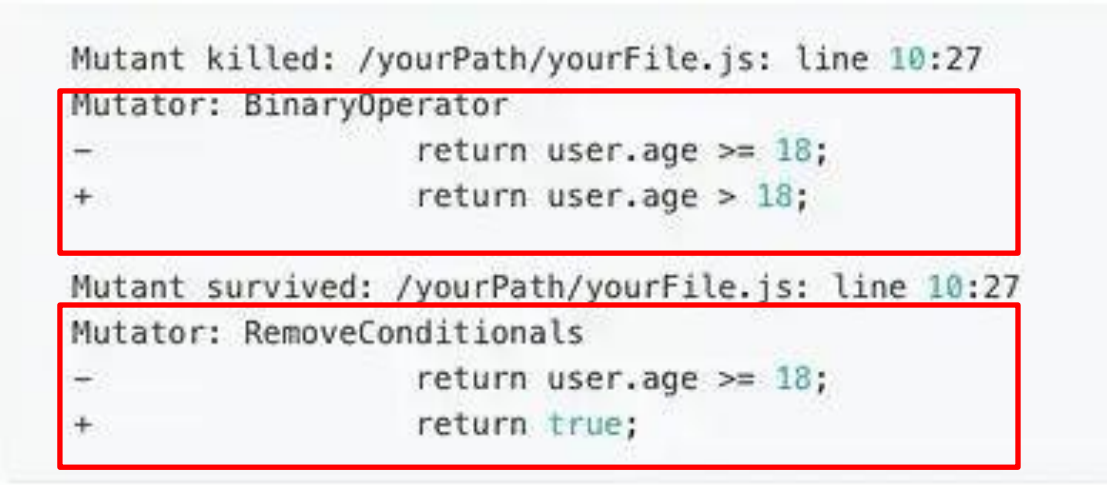
1. Stryker，是一个针对JavaScript语言的变异测试框架，可以帮助提高单元测试的质量。变异测试的工作原理是通过对代码进行小的修改（称为“变异”），然后运行单元测试以查看哪些修改没有被测试捕获，这可以帮助揭示代码覆盖率的盲点。

在Node.js环境中，你可以使用npm（Node包管理器）来安装：

npm install --save-dev @stryker-mutator/core @stryker-mutator/mocha-runner @stryker-mutator/javascript-mutator

进一步需要创建一个Stryker配置文件。这个文件名通常为stryker.conf.js，并且应该位于项目的根目录下。在这个文件中，可以定义Stryker应该如何运行测试和创建变异。

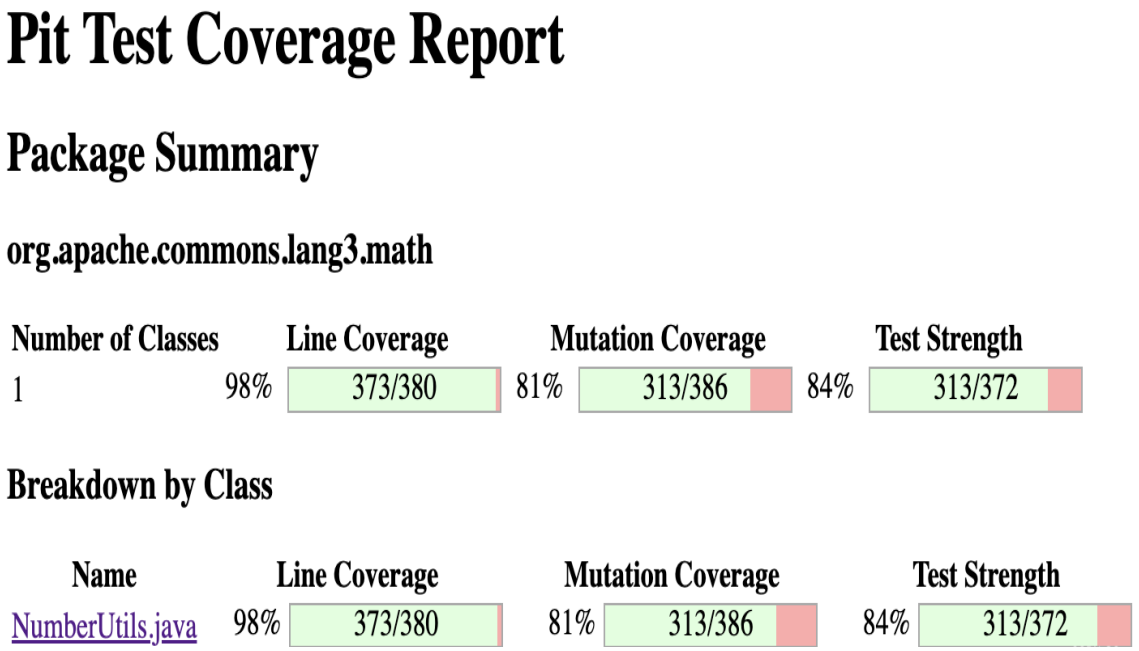
```
// stryker.conf.js
module.exports = function(config){
  config.set({
    mutator: "javascript",
    packageManager: "npm",
    reporters: ["clear-text", "progress"],
    testRunner: "mocha",
    transpilers: [],
    coverageAnalysis: "off",
    mutate: ["src/**/*.js"],});};
```



变异测试工具

2. Jumble，针对Java的变异测试工具，简单的非图形开源工具，将文本文件转换为可以研究文件格式的版本，直接在源代码级别运行，并加快变异测试的过程，其支持的有限变异运算符集包括：条件、二进制运算符、增量、内联常量、类池常量、返回值和开关语句等；

3. PITest，针对Java的变异测试工具，是目前最高水平的变异测试系统，其在字节码级别对程序进行变异，并支持多种使用方式，包括command line/ant/maven等等，其官网有详细的使用文档；



Pit Test Coverage Report

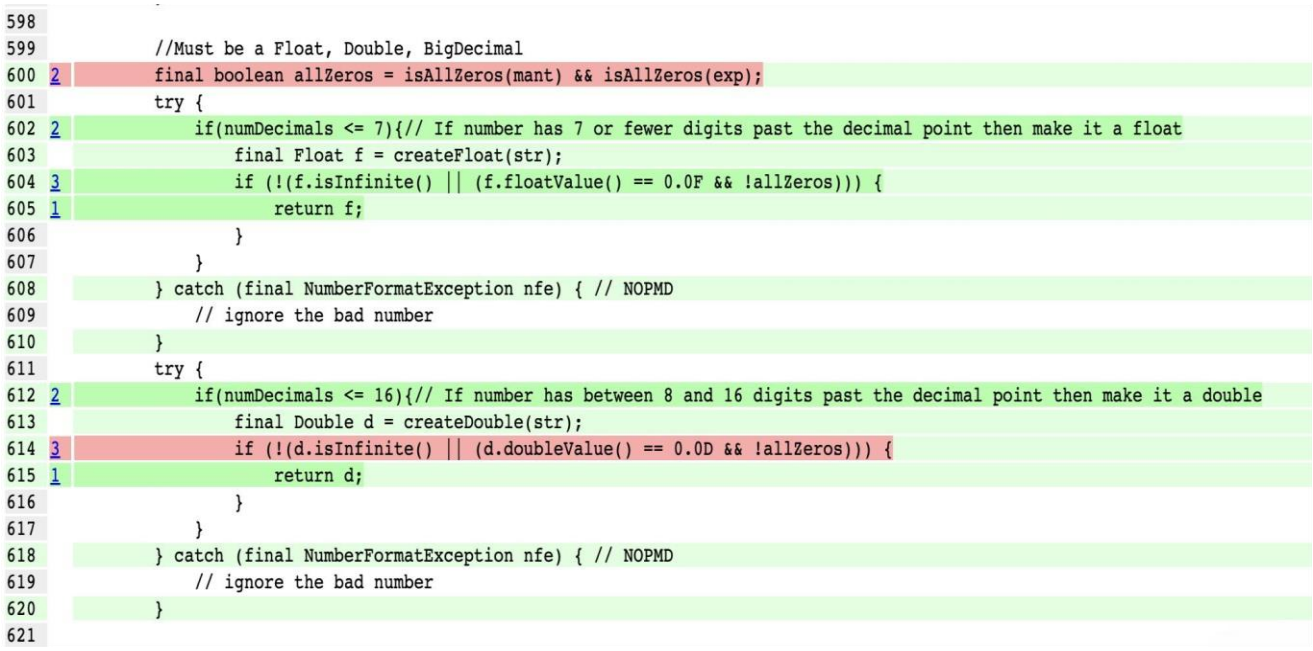
Package Summary

org.apache.commons.lang3.math

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	98%	373/380	81%	313/386	84%	313/372

Breakdown by Class

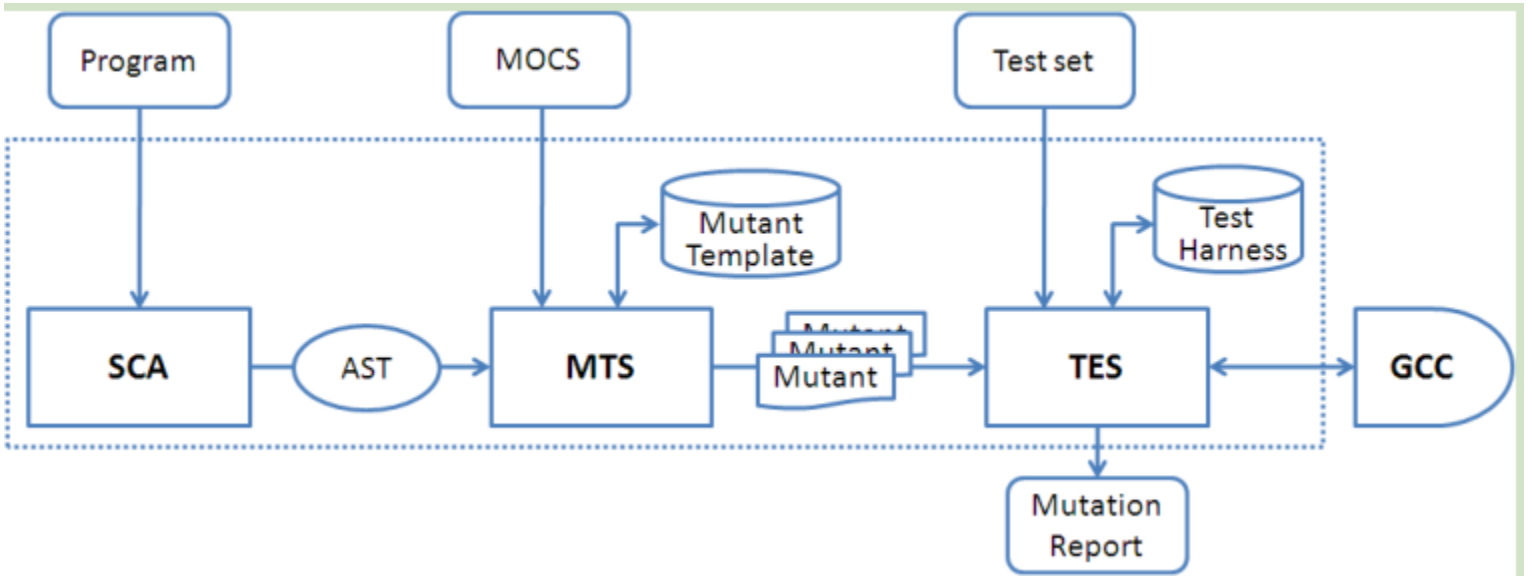
Name	Line Coverage		Mutation Coverage		Test Strength	
NumberUtils.java	98%	373/380	81%	313/386	84%	313/372



```
598
599         //Must be a Float, Double, BigDecimal
600 2       final boolean allZeros = isAllZeros(mant) && isAllZeros(exp);
601         try {
602 2           if(numDecimals <= 7){// If number has 7 or fewer digits past the decimal point then make it a float
603                 final Float f = createFloat(str);
604 3               if (!(f.isInfinite() || (f.floatValue() == 0.0F && !allZeros))) {
605 1                   return f;
606                 }
607             }
608         } catch (final NumberFormatException nfe) { // NOPMD
609             // ignore the bad number
610         }
611         try {
612 2           if(numDecimals <= 16){// If number has between 8 and 16 digits past the decimal point then make it a double
613                 final Double d = createDouble(str);
614 3               if (!(d.isInfinite() || (d.doubleValue() == 0.0D && !allZeros))) {
615 1                   return d;
616                 }
617             }
618         } catch (final NumberFormatException nfe) { // NOPMD
619             // ignore the bad number
620         }
621
```

变异测试工具

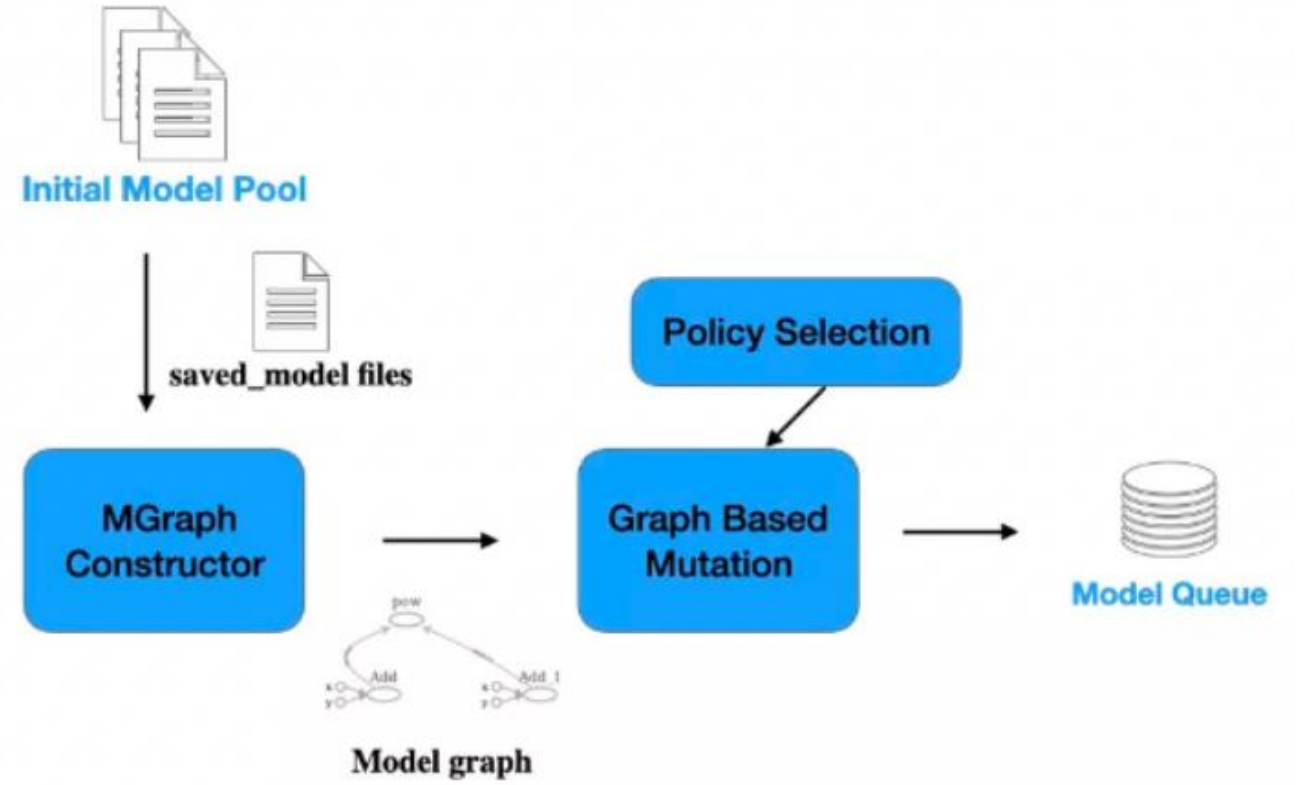
4. Milu，一个可定制、运行时优化、高阶的C语言变异测试工具；之前的工具是将所有可能的变异算子应用到测试中，而MiLu可以指定应用到测试中的变异算子；为减少运行时开销，MiLu使用“测试治理(test harness)”技术将变异体和测试集嵌入到测试程序中。Muli的基本功能：根据指定变异算子生成变异体 + 根据给定的测试集执行变异程序 + 报告变异scores（一个衡量测试集充分性的量）



5. Mutpy 是 Python 中的一个突变测试工具，它生成突变体并计算突变分数。它支持标准的 unittest 模块，生成 YAML/HTML 报告，并有丰富多彩的输出。但对较新版本的Python支持力度不大，需要对源码进行修改后才能继续使用。

变异测试发展——拥抱AI

AI技术已经被广泛应用于各行各业。一方面，AI技术领域火热的研究趋势和高效的开发迭代速度可以应对日益剧增的需求。另一方面，技术的频繁更新和开发的高速迭代对底层的计算框架的安全性带来了冲击。如何确保AI计算框架的代码开发质量成为了代码开发过程中必不可少的一部分。



AI Model-Mutator

变异测试成本

•相关研究表明，一个软件模块所能产生的变异体数正比于程序中数据对象的数量与数据对象的引用次数之乘积。

•变异测试需要运行大量的变异体，每一个变异体至少需要执行一个测试用例以便将其杀死，这就需要大量的计算能力。因此，即使是对一小规模的软件模块，也将产生大量的变异体。
比如仅有4行的Min函数，经过Mothra变异系统可以生成44个变异体。
三角形分类程序Trityp的代码规模为30行，变异后能产生951个变异体。

•由于每个变异体都至少要执行一个测试用例，一般需要运行多个(最多的时候需要运行完测试用例集中的所有测试用例)，变异测试在这个步骤需要巨大的计算开销。正是由于难以接受的巨大计算开销阻碍了变异测试在软件工业界的广泛应用。

变异测试成本——弱变异

•传统的变异测试系统如Mothra比较的是原始程序和变异体的最终输出结果，需要完全执行完整个程序，因此，被称为强变异(strong mutation)。

•弱变异由于避免了完全执行整个程序，从而节省了计算开销，提高了变异测试的效率，改进了变异测试实用性。

•主要思想：判断杀死变异体时比较的是原始程序和变异体的内部状态，即比较点紧接着程序的变异部分之后，只需运行部分语句，不需要执行完整个程序。弱变异有四个比较点：

1)表达式级比较点(expression weak)，在对变异部分之后最内层表达式第一次求值时进行比较。

2)语句级比较点(statement weak)，在第一次执行完变异语句之后进行比较。

3)基本块级比较点1(basic block weak / 1)，在第一次执行完包含变异语句的基本块之后进行比较。

4)基本块级比较点2(basic block weak / n)，这种情况主要针对于带有循环的变异，在第一次循环时还无法将变异体杀死。因此，需要在每次执行完包含变异语句的基本块后都进行比较，直到将变异体杀死。

变异测试总结——优劣势

- 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

- 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）

Q & A

问答