黑盒测试与白盒测试

南京大学计算机学院

2025.9.15

小测试

A+B for Input-Output Practice (I)

Time Limit: 2000/1000 MS (Java/Others) Memory Limit: 65536/32768 K (Java/Others)
Total Submission(s): 68100 Accepted Submission(s): 26833

Problem Description

Your task is to Calculate a + b.
Too easy?! Of course! I specially designed the problem for acm beginners.
You must have found that some problems have the same titles with this one, yes, all these problems were designed for the same aim.

Input

The input will consist of a series of pairs of integers a and b, separated by a space, one pair of integers per line.

Output

For each pair of input integers a and b you should output the sum of a and b in one line, and with one line of output for each line in input.

Sample Input

```
1 5
10 20
```

Sample Output

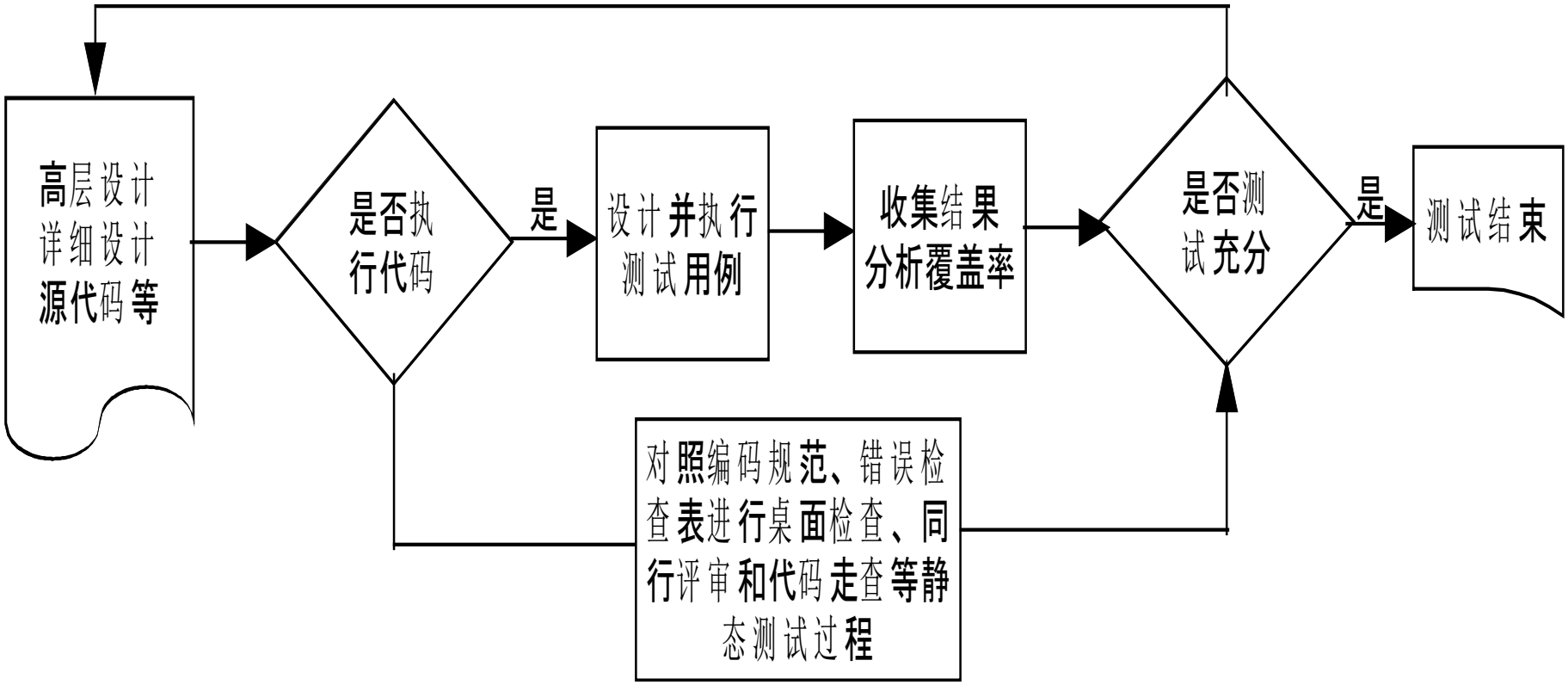
```
6
30
```

- 要求：记录时间，写好程序员姓名学号、评审员姓名学号及发现的问题
- 考察点：提前体验白盒测试与黑盒测试、同行评审等知识点

进一步分类

	静态测试	动态测试
白盒测试	1同行评审 2代码审查 3代码走查 4桌面评审	1语句覆盖；　2分支覆盖； 3条件覆盖；　4分支条件覆盖； 5条件组合；　6修改条件决策； 7 路径覆盖；　8数据流覆盖； 9线性序列跳转覆盖
黑盒测试	1文档审查	1等价类划分；　2边际值分析； 3因果图与决策表；4语法测试； 5故障推测法；　6状态转换测试

白盒测试过程


高层设计
详细设计
源代码等
是否执行代码
是
设计并执行测试用例
收集结果分析覆盖率
是否测试充分
是
测试结束
对照编码规范、错误检查表进行桌面检查、同行评审和代码走查等静态测试过程

白盒静态测试（四种形式）

- 白盒静态测试是一些评审过程，可以使几个程序员之间一个简单的会议，对软件的设计和编码进行详细和严格的审查，这些评审过程有以下四个特征：
- 发现问题。针对设计和代码，检查错误和遗漏，所有的批评都是对事不对人。
- 遵循一定的规则。例如，指定审查的代码量，会议时间和每个参会人员的责任等。
- 准备工作。评审会效果取决于参会人员的会前准备工作，每个人都要为会议作出贡献。很多问题都是在会前准备发现的，而不是在会议上。
- 会议报告。评审团要求就发现的问题写一个书面报告：多少问题，问题在哪里。

白盒静态测试（四种形式）

- 白盒静态测试通过各种评审，除了可以发现问题还有以下四个作用:
- 交流沟通的作用。通过评审会，程序员之间可以相互学习，项目经理可以了解项目进度，黑盒测试人员可能会得到一些有用信息，有利于进行黑盒测试。
- 提高编码和文档质量。程序的代码和相关的文档被同行和专家仔细阅读和评审，这对程序员是一个很大的压力，这使得他们在写程序及文档时会非常认真。
- 建立团队友谊。评审会使得大家相互了解对方的工作技能和难度，互相理解和尊重。
- 寻求解决方案。对于一些棘手的问题，评审会的讨论有可能提供部分解决思路。

同行评审（peer review）

- 同行评审，有时也称好友评审，不是很正式，就相当于“你把你的拿给我看看，我把我的拿给你看看”，通常由软件设计人员和编码人员加上一两个其它程序员或测试人员组成，要提高同行评审的效果，必须加强一般评审过程的四个特征。
- 不管怎样，聚到一起讨论程序代码还是有助于发现问题的。

代码审查（Code inspections）

- 代码审查是最正式的一种评审，编写受评审代码的程序员不能参与评审，所以要求每个参与人员都要受过专门的训练，能够学习和理解代码和相关的材料。
- 每个评审人员可以从用户、测试人员和产品支持人员等不同角度对代码进行评审，从而可以从不同方面发现错误。其中指定一个记录员和一名协调员负责整个审查过程的有效运行。
- 审查会议结束之后，评审人员与协调员就发现的问题写一个报告，指出解决相关问题的必要性工作。然后交给程序员进行整改，协调员对相关修改进行查证，并根据问题的大小决定是否进行下一轮审查。
- 代码审查可以有效发现程序中潜藏的错误，很多公司和开发团队都使用这种方法。

代码走查（Code walkthroughs）

- 代码走查更正式一点，写代码的程序员向由程序员和测试人员组成的五人左右的审查小组走查自己的程序。审查小组成员应该提早收到程序拷贝，认真阅读，写好评语和关心的问题。审查小组中至少要有一个资深的程序员。
- 程序员通过一行一行地，一个功能一个功能地阅读代码，解释代码做什么的以及原因，评审组听取汇报，并就可疑的地方进行提问。参与代码走查的人员比同行评审更多一点，因此，更要在发现错误、会议规则、提前准备和会议总结等环节下工夫。程序员最后将走查过程中发现的问题，及如何解决这些问题的初步计划写一个报告。

桌面检查（desk examination）

- 桌面检查可视为由单人进行的代码检查或代码走查，由一个人阅读程序，对照错误列表检查程序，对程序推演测试数据。
- 桌面检查一般效率比较低，主要因为桌面检查基本上没有约束，它违反了程序员不能检查自己程序的原则。改进的桌面检查由程序员之间各自交互检查相互间的程序，但即使是这样，其效果仍然不如代码审查或代码走查。

白盒动态测试

软件的白盒测试是对软件的过程性描述做细致的检查，这一方法是把测试对象看着一个打开的盒子，它允许软件测试员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。通过在不同点检查程序的状态，取得实际的状态，是否与预期的状态一致，故又称结构测试或逻辑驱动。

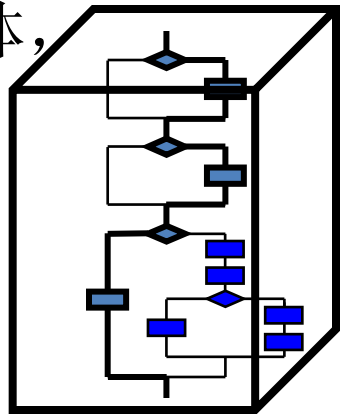
使用白盒测试，主要对模块进行如下的测试：

(1) 对程序模块的所有独立的执行路径至少执行一次。

(2) 对所有的逻辑判定，取值为“真”与取值为“假”的两种情况都能至少测试一次。

(3) 在循环的边界与运行界限内执行循环体，

(4) 测试内部数据结构的有效性。

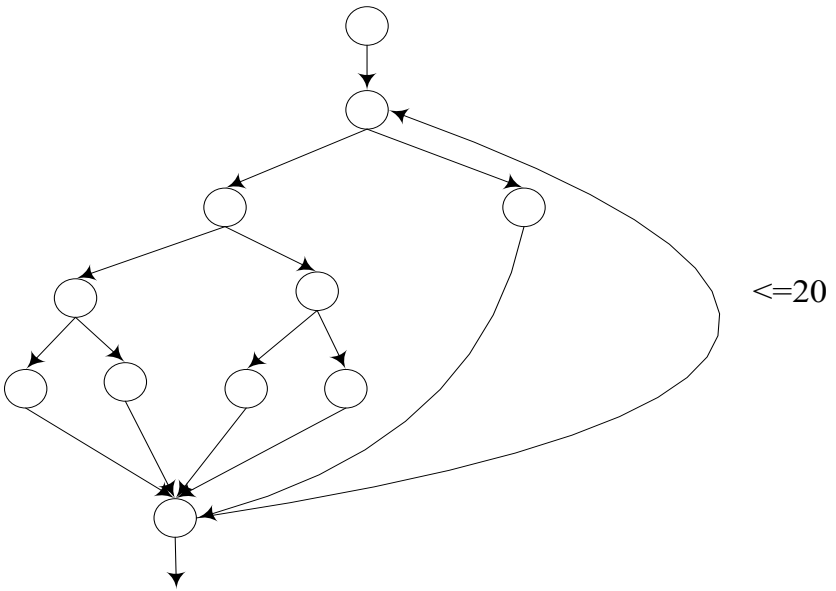


白盒动态测试的难度

如图所示的程序图，它对应了一个100行源代码的Pascal语言程序，其中包括了一个执行20次的循环，那么它所包含的不同路径高达5^{20}（$\approx 10^{14}$）条，若要对它进行穷举测试，即要设计测试用例，覆盖所有的路径。设有一个测试程序，对每条路径的测试需一毫秒，那么要完成测试约需3710年。

任何进行穷举测试都是一场达灾难，因此，我们的策略是在一定的开发周期和某种经济条件下，通过有限的测试尽可能多地发现错误。

测试只能发现错误，而不能保证程序没有错误。



白盒测试的测试用例设计

语句覆盖
判定覆盖
条件覆盖
判定/条件覆盖
修改决策条件测试（MC/DC）
条件组合覆盖
路径覆盖
数据流测试
LCSAJ Testing

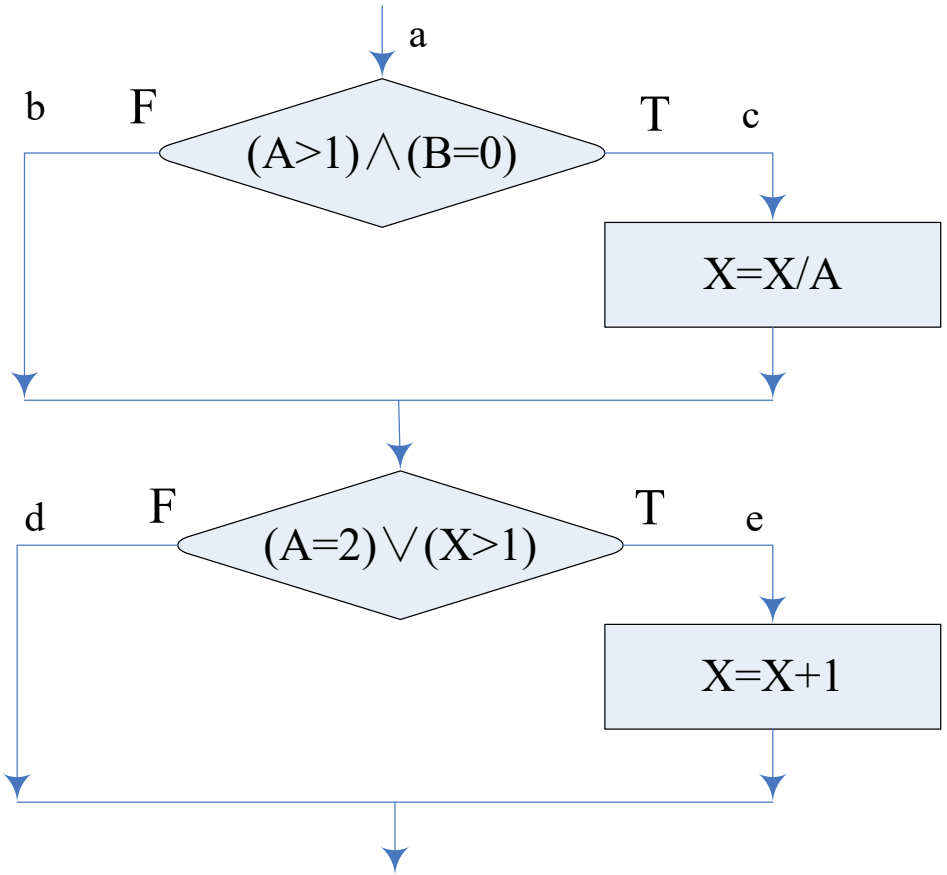
逻辑覆盖

有选择地执行程序中某些最有代表性的通路是对穷尽测试的唯一可行的替代办法。所谓逻辑覆盖是对一系列测试过程的总称

这组测试过程逐渐进行越来越完整的通路测试。测试数据执行（或叫覆盖）程序逻辑的程度可以划分成哪些不同的等级呢？从覆盖源程序语句的详尽程度分析，大致有前面9种不同的覆盖标准

逻辑覆盖

Input: A,B,X
Output: A,B,X
Algorithm is described as following
Test case (input, expected output)




```
#include<iostream>
Int main()
{
 using namespace std;
 int A,B,X;
Cout<<“please input A:”;
 cin>>A;
Cout<<“please input B:”;
 cin>>B;
Cout<<“please input X:”;
 cin>>X;
 if (A>0&B=0) X=X/A;
 if (A=2&X>1) X=X+1;
Cout<<“A=”<<A;
Cout<<“B=”<<B;
Cout<<“X=”<<X;
Return 0;
}
```

逻辑覆盖

L1(a→c→e)

=\{(A＞1)and(B＝0)\}and\{(A＝2)or(X/A＞1)\}

=(A＞1)and(B＝0)and(A＝2)or(A＞1)and(B＝0)and(X/A＞1)

=(A＝2)and(B＝0)or(A＞1)and(B＝0)and(X/A＞1)

L2(a→b→d)

=\{(A＞1)and(B＝0)\}and\{(A＝2)or(X＞1)\}

=\{(A＞1)or(B＝0)\}and\{(A＝2)and(X＞1)\}

=(A＞1)and(A＝2)and(X＞1)or(B＝0)and(A＝2)and(X＞1)

=(A≤1)and(X≤1)or(B≠0)and(A≠2)and(X≤1)

L3(a→b→e)

=\{(A＞1)and(B＝0)\}and\{(A＝2)or(X＞1)\}

=\{(A＞1)or(B＝0)\}and\{(A＝2)or(X＞1)\}

=(A＞1)and(X＞1)or(B＝0)and(A＝2)or(B＝0)and(X＞1)\}

= A≤1)and(X＞1)or(B≠0)and(A＝2)or(B≠0)and(X＞1)

L4(a→c→d)

=\{(A＞1)and(B＝0)\}and\{(A＝2)or(X/A＞1)\}

=(A＞1)and(B＝0)and(A≠2)and(X/A≤1)

语句覆盖

为了暴露程序中的错误，至少每个语句应该执行一次。语句覆盖的含义是，选择足够多的测试数据，使被测程序中每个语句至少执行一次。

测试用例的设计格式如下

【输入的(A,B,x),输出的(A,B,x)】

为子设计满足语句覆盖的测试用例是：

【(2,0,4),(2,0,3)】　　覆盖ace【L1】

判定覆盖

判定覆盖就是设计若干测试用例，运行所测程序，使得程序中每个判断的取真分支和取假分支至少经历一次。判定覆盖又称为分支覆盖。

例如对于给出的例子，如果选择路径L1和L2，就可得满足要求的测试用例：

【(2,0,4),(2,0,3)】　　覆盖ace【L1】

【(1,1,1),(1,1,1)】　　覆盖abd【L2】

如果选择路径L3和L4，还可得另一组可用的测试用例：

【(2,1,1),(2,1,2)】　　覆盖abe【L3】

【(3,0,3),(3,1,1)】　　覆盖acd【L4】

条件覆盖

条件覆盖就是设计若干个测试用例，运行所测程序，使得程序中每个判断条件的可能取值至少执行一次。

例如给出的例子中，事先可对所有条件的取值加以标记。例如，

对于第一个判断：条件 A＞1 取真值为T1，取假值为$\overline{T1}$

条件 B＝0 取真值为T2，取假值为$\overline{T2}$

对于第二个判断：条件 A＝2 取真值为T3，取假值为$\overline{T3}$

条件 x＞1 取真值为T4，取假值为$\overline{T4}$

条件覆盖

测　试　用　例	通过路径	条件取值	覆盖分支
【(2,0,4),(2,0,3)】	ace(L1)	T1 T2 T3 T4	c,e
【(1,0,1),(1,0,1)】	abd(L2)	$\overline{T1}$ T2 $\overline{T3}$ $\overline{T4}$	b,d
【(2,1,1),(2,1,2)】	abe(L3)	T1 $\overline{T2}$ T3 $\overline{T4}$	b,e

或

测　试　用　例	通过路径	条 件 取 值	覆盖分支
【(1,0,3),(1,0,4)】	abe(L3)	$\overline{T1}$ T2 $\overline{T3}$ T4	b,e
【(2,1,1),(1,0,1)】	abe(L3)	T1 $\overline{T2}$ T3 $\overline{T4}$	b,e

判定-条件覆盖

判定-条件覆盖就是设计足够的测试用例，使得判断中每个条件的所有可能取值至少执行一次，同时每个判断的所有可能判断结果至少执行一次。换言之，即是要求各个判断的所有可能的条件取值组合至少执行一次。

判定-条件覆盖也有缺陷。从表面上来看，它测试了所有条件的取值。但事实并非如此。因为往往某些条件掩盖了另一些条件。

判定-条件覆盖

测　试　用　例	通过路径	条件取值	覆 盖 分 支
【(2,0,4),(2,0,3)】 【(1,1,1),(1,1,1)】	ace(L1) abd(L2)	T1　T2　T3　T4 $\overline{T1}$　$\overline{T2}$　$\overline{T3}$　$\overline{T4}$	c,e b,d

条件组合覆盖

- 条件组合覆盖就是设计足够的测试用例，运行所测程序，使得每个判断的所有可能的条件取值组合至少执行一次。

记 ①A＞1，B＝0　作T1 T2,　属第一个判断的判定的取真分支；
②A＞1，B≠0　作T1 $\overline{T2}$,　属第一个判断的判定的取假分支；
③A≯1，B＝0　作$\overline{T1}$ T2,　属第一个判断的判定的取假分支；
④A≯1，B≠0　作$\overline{T1}$ $\overline{T2}$,　属第一个判断的判定的取假分支；
⑤A＝2，x＞1　作T3 T4,　属第二个判断的判定的取真分支；
⑥A＝2，x≯1　作T3 $\overline{T4}$,　属第二个判断的判定的取真分支；
⑦A≠2，x＞1　作$\overline{T3}$ T4,　属第二个判断的判定的取真分支；
⑧A≠2，x≯1　作$\overline{T3}$ $\overline{T4}$,　属第二个判断的判定的取假分支；

条件组合覆盖

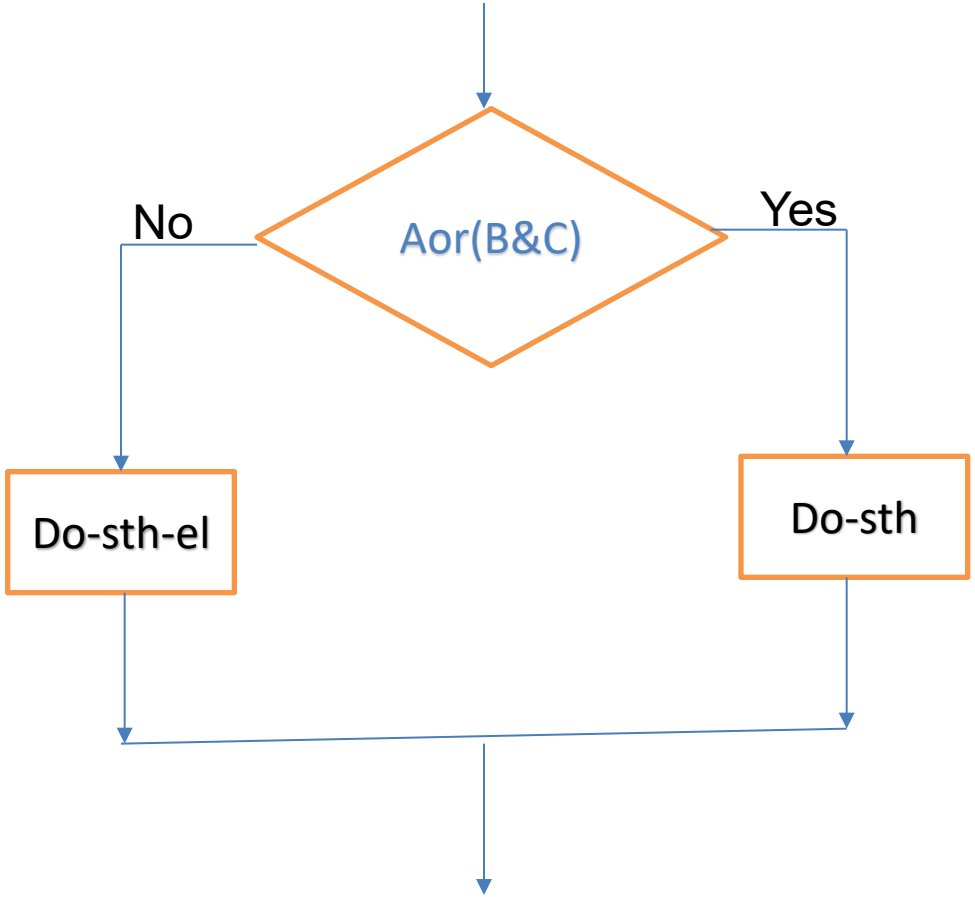
测 试 用 例	通过路径	条 件 取 值	覆 盖 分 支
【(2,0,4),(2,0,3)】	ace(L1)	T1 $\overline{T2}$ T3 $\overline{T4}$	①,⑤
【(2,1,1),(2,1,2)】	abe(L3)	T1 $\overline{T2}$ T3 $\overline{T4}$	②,⑥
【(1,0,3),(1,0,4)】	abe(L3)	$\overline{T1}$ T2 $\overline{T3}$ T4	③,⑦
【(1,1,1),(1,1,1)】	abd(L2)	$\overline{T1}$ $\overline{T2}$ $\overline{T3}$ $\overline{T4}$	④,⑧

修改决策条件测试

- 设计测试用例让每个条件变量独立改变判定语句的真假值。
- 这种测试方法的优点是：保证每个变量真假值出现一次、整个判定表达式出现真假值、让每个变量独立影响判定表达式的真假值、测试用例最好的情况下规模为n+1,最差情况下测试用例规模为2n，而条件组合覆盖需要2^n条测试用例，因此修改决策条件测试（MC/DC）是最实用的一种测试方法。

修改决策条件测试-example

```
if A or (B and C) then
      do_something;
else
      do_something_else;
end if;
```



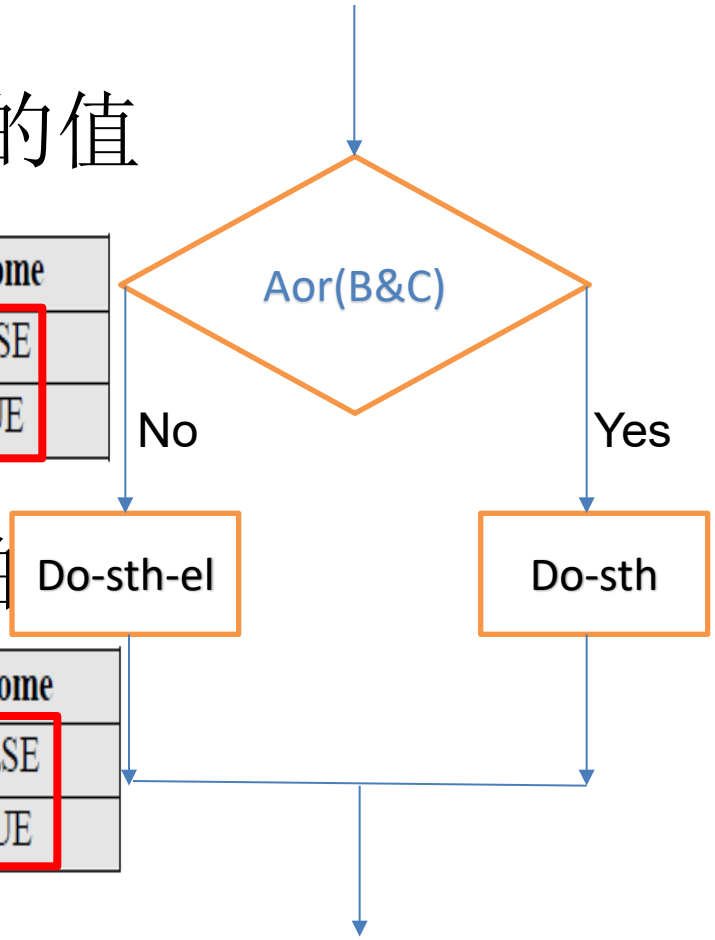
修改决策条件测试-example

- 条件A独立改变判定表达式的值

Case	A	B	C	Outcome
A1	FALSE	FALSE	TRUE	FALSE
A2	TRUE	FALSE	TRUE	TRUE

- 条件A独立改变判定表达式的

Case	A	B	C	Outcome
B1	FALSE	FALSE	TRUE	FALSE
B2	FALSE	TRUE	TRUE	TRUE



修改决策条件测试-example

- 条件C独立改变判定表达式的值

Case	A	B	C	Outcome
C1	FALSE	TRUE	TRUE	TRUE
C2	FALSE	TRUE	FALSE	FALSE

- 修改决策条件测试用例集

Case	A	B	C	Outcome
1 (A1,B1)	FALSE	FALSE	TRUE	FALSE
2 (A2)	TRUE	FALSE	TRUE	TRUE
3 (B2,C1)	FALSE	TRUE	TRUE	TRUE
4 (C2)	FALSE	TRUE	FALSE	FALSE

路径测试

- 路径测试就是设计足够的测试用例，覆盖程序中所有可能的路径。若还是以图8.6为例，则可以选择如下的一组测试用例来覆盖该程序段的全部路径。

测 试 用 例	通过路径	覆 盖 条 件
【(2,0,4),(2,0,3)】	ace(L1)	T1 T2 T3 T4
【(2,1,1),(2,1,2)】	abd(L2)	$\overline{T1}$ $\overline{T2}$ $\overline{T3}$ $\overline{T4}$
【(1,0,3),(1,0,4)】	abe(L3)	$\overline{T1}$ T2 $\overline{T3}$ T4
【(3,0,3),(3,0,1)】	acd(L4)	T1 T2 $\overline{T3}$ $\overline{T4}$

基本路径覆盖

- 基本路径测试就是这样一种测试方法，它是在程序控制流图的基础上，通过分析控制构造的环境复杂性，导出基本可执行路径集合，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一个可执行语句至少执行一次。

基本路径覆盖

1. 基本路径测试

基本路径测试法适用于模块的详细设计和源程序，其主要步骤如下：

(1) 以详细设计或源代码作为基础，导出程序的控制流图。

(2) 计算得到的控制流G的环路复杂性。

(3) 确定线性无关的基本路径集。

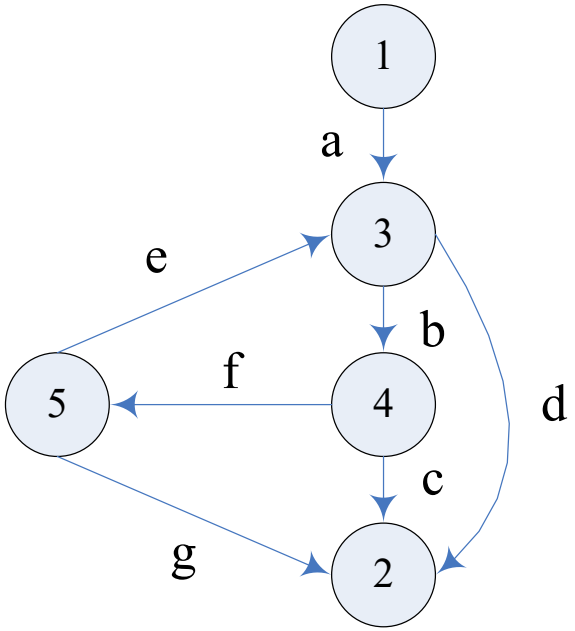
(4) 生成测试用例，确保基本路径集中每条路径的执行。

基本路径覆盖

2. 图形矩阵

图形矩阵是在基本路径测试中起辅助作用的软件工具，利用它可以实现自动地确定一个基本路径集。一个图形矩阵是一个方阵，其行/列数等于控制流图中的结点数。每行和每列依次对一个被标记的结点，矩阵元素对应到结点间的连接（即边）。

基本路径覆盖



控制流图

结点 \ 相连接点	1	2	3	4	5
1			**a**		
2					
3		**d**		**b**	
4		**c**			**f**
5		**g**	**e**		

图形矩阵

线性代码序列跳转测试（LCSAJ testing）

- Linear Code Sequence And Jump
- 线性代码序列跳转测试主要关注程序控制流的跳转，一个线性代码序列跳转表示为一个三元组：线性代码起点行数s、线性代码终点行数e和控制流跳转点的代码行数j，即（s,e,j）。
- 测试用例用来执行覆盖每个线性代码序列跳转，每个测试用例可以描述为三个部分：测试输入、被该测试用例执行的线性代码序列跳转、预期输出。
- 由于线性代码序列跳转测试的对象是线性代码序列跳转，所以线性代码序列跳转测试的覆盖率度量为：
- LCSAJ= $\frac{\text{执行过的线性代码序列跳转个数}}{\text{所有的线性代码序列跳转个数}} * 100\%$

Example

```
1     READ (Num);
2     WHILE NOT End of File DO
3           Prime := TRUE;
4           FOR Factor := 2 TO Num DIV 2 DO
5                 IF Num - (Num DIV Factor)*Factor = 0 THEN
6                       WRITE (Factor, ` is a factor of', Num);
7                       Prime := FALSE;
8                 ENDIF;
9           ENDFOR;
10          IF Prime = TRUE THEN
11                WRITE (Num, ` is prime');
12          ENDIF;
13          READ (Num);
14    ENDWHILE;
15    WRITE (`End of prime number program');
```

程序中包含的线性代码序列跳转及相应的条件

(2->3)	:	requires NOT End of File
(2->15)	: Jump	requires End of File
(4->5)	:	requires the loop to execute, i.e. Num DIV 2 is greater than or equal to 2
(4->10)	: Jump	requires the loop to be a zero-trip, i.e. Num DIV 2 is less than 2
(5->6)	:	requires the IF statement on line 5 to be true
(5->9)	: Jump	requires the IF statement on line 5 to be false
(9->5)	: Jump	requires a further iteration of the FOR loop to take place
(9->10)	:	requires the FOR loop to have exhausted
(10->11)	:	requires Prime to be true
(10->13)	: Jump	requires Prime to be false
(14->2)	: Jump	must always take place

各个线性代码序列跳转的执行情况

LCSAJ			
START LINE	FINISH LINE	JUMP TO LINE	TIMES EXECUTED
1	2	15	0 ***
1	4	10	0 ***
1	5	9	1
1	9	5	0 ***
1	10	13	0 ***
1	14	2	0 ***
2	2	15	1
2	4	10	1
2	5	9	0 ***
2	9	5	1
2	10	13	0 ***
2	14	2	0 ***
5	5	9	0 ***
5	9	5	0 ***
5	10	13	1
5	14	2	0 ***
9	9	5	0 ***
9	10	13	0 ***
9	14	2	1
10	10	13	0 ***
10	14	2	1
13	14	2	1
15	15	exit	1
Number of LCSAJs			23
Number executed			9
Number not executed			14
LCSAJ Coverage			39%

数据流测试（Data Tlow Testing）

- 所有定义覆盖（all-definitions）：设计测试用例执行所有变量定义点到某些使用点（计算使用或者谓词使用）的路径。
- 所有计算使用覆盖（all-c-uses）：设计测试用例执行每个变量定义点到其每个计算使用点的路径。
- 所有谓词使用覆盖（all-p-uses）：设计测试用例执行每个变量定义点到其每个谓词使用点的路径。
- 所有使用覆盖（all-uses）：设计测试用例执行每个变量定义点到其每个使用点（计算使用或者谓词使用）的路径。
- 所有定义使用对覆盖（all-du-paths）：设计测试用例执行所有变量定义点到每个使用点（计算使用或者谓词使用）的路径。

求平方根程序

```
procedure Solve_Quadratic(A, B, C: in Float; Is_Complex: out
                                  Boolean; R1, R2: out Float) is

-- Is_Complex is true if the roots are not real.
-- If the two roots are real, they are produced in R1, R2.

        Discrim : Float := B*B - 4.0*A*C;                          -- 1
        R1, R2: Float;                                             -- 2
    begin                                                          -- 3
        if Discrim < 0.0 then                                      -- 4
            Is_Complex := true;                                    -- 5
        else                                                       -- 6
            Is_Complex := false;                                   -- 7
        end if;                                                    -- 8
        if not Is_Complex then                                     -- 9
            R1 := (-B + Sqrt(Discrim))/ (2.0*A);                   -- 10
            R2 := (-B - Sqrt(Discrim))/ (2.0*A);                   -- 11
        end if;                                                    -- 12
    end Solve_Quadratic;                                           -- 13
```

求平方根程序中出现的变量及其分类

line	category		
	definition	c-use	p-use
0	A,B,C		
1	Discrim	A,B,C	
2			
3			
4			Discrim
5	Is_Complex		
6			
7	Is_Complex		
8			
9			Is_Complex
10	R1	A,B,Discrim	
11	R2	A,B,Discrim	
12			
13			

求平方根程序中变量的定义引用对

definition-use pair (start line -> end line)	variable(s)	
	c-use	p-use
0 ---> 1	A,B,C	
0 ---> 10	A,B	
0 ---> 11	A,B	
1 ---> 4		Discrim
1 ---> 10	Discrim	
1 ---> 11	Discrim	
5 ---> 9		Is_Complex
7 ---> 9		Is_Complex

所有定义覆盖（all-definition）测试用例

	All Definitions			INPUTS			EXPECTED OUTCOME		
test case	variable(s)	du-pair	subpath	A	B	C	Is_Complex	R1	R2
1	A,B,C	0 --> 1	0-1	1	1	1	T	unass.	unass.
2	Discrim	1 --> 4	1-4	1	1	1	T	unass.	unass.
3	Is_Complex	5 --> 9	5- 9	1	1	1	T	unass.	unass.
4	Is_Complex	7 --> 9	7- 9	1	2	1	F	-1	-1

所有计算使用覆盖（all-c-use）测试用例

	All-c-uses			INPUTS			EXPECTED OUTCOME		
test case	variable(s)	du-pair	subpath	A	B	C	Is_Complex	R1	R2
1	A,B,C	0 --> 1	0-1	1	1	1	T	unass.	unass.
2	A,B	0 --> 10	0-1-4-7-9-10	1	2	1	F	-1	-1
3	A,B	0 --> 11	0-1-4-7-9-10-11	1	2	1	F	-1	-1
4	Discrim	1 --> 10	1-4-7-9-10	1	2	1	F	-1	-1
5	Discrim	1 --> 11	1-4-7-9-10-11	1	2	1	F	-1	-1

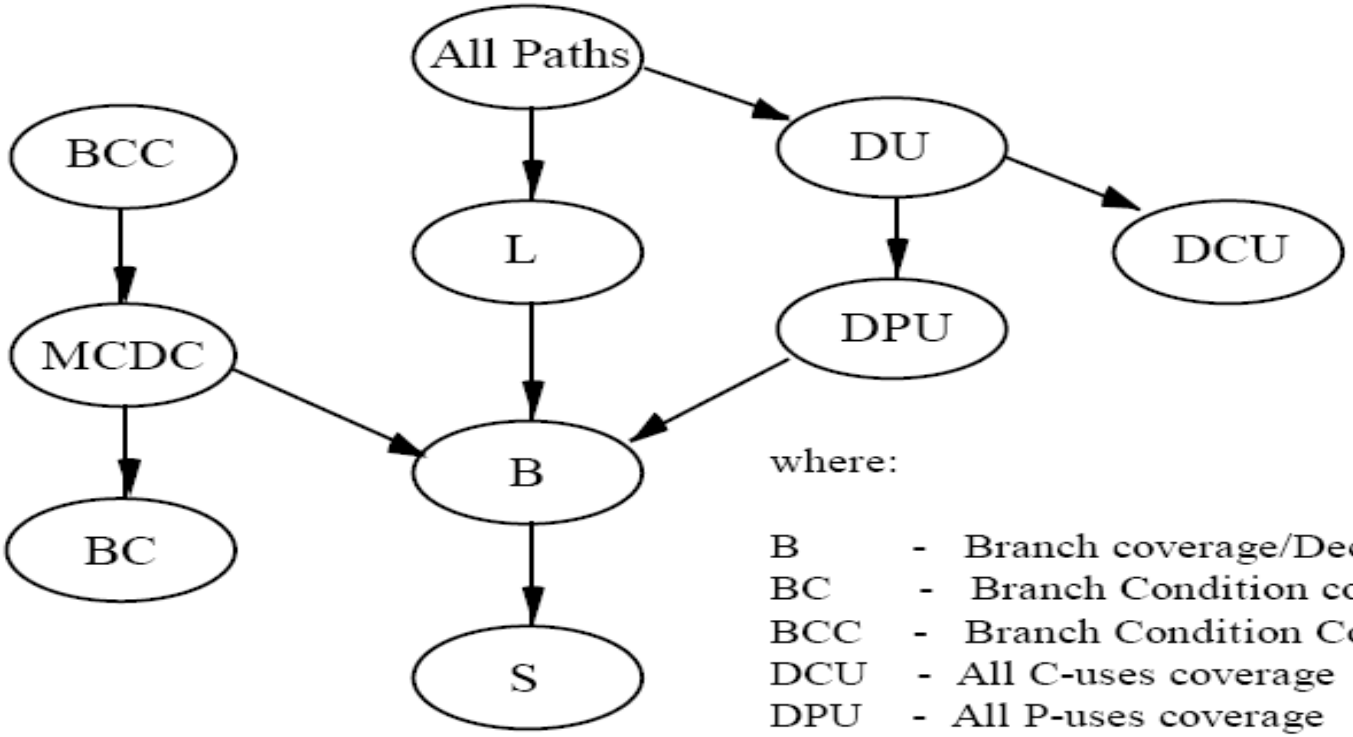
所有谓词使用覆盖（all-p-use）测试用例

	All-p-uses			INPUTS			EXPECTED OUTCOME		
test case	variable(s)	du-pair	subpath	A	B	C	Is_Complex	R1	R2
1	Discrim	1 --> 4	1-4	1	1	1	T	unass.	unass.
2	Is_Complex	5 --> 9	5- 9	1	1	1	T	unass.	unass.
3	Is_Complex	7 --> 9	7- 9	1	2	1	F	-1	-1

所有使用覆盖（all- use）测试用例

	All-uses / All du-paths			INPUTS			EXPECTED OUTCOME		
test case	variable(s)	d-u pair	subpath	A	B	C	Is_Complex	R1	R2
1	A,B,C	0 --> 1	0-1	1	1	1	T	unass.	unass.
2	A,B	0 --> 10	0-1-4-7-9-10	1	2	1	F	-1	-1
3	A,B	0 --> 11	0-1-4-7-9-10-11	1	2	1	F	-1	-1
4	Discrim	1 --> 4	1-4	1	1	1	T	unass.	unass.
5	Discrim	1 --> 10	1-4-7-9-10	1	2	1	F	-1	-1
6	Discrim	1 --> 11	1-4-7-9-10-11	1	2	1	F	-1	-1
7	Is_Complex	5 --> 9	5- 9	1	1	1	T	unass.	unass.
8	Is_Complex	7 --> 9	7- 9	1	2	1	F	-1	-1

Partial Ordering of Structural Test Coverage Criteria



where:

B - Branch coverage/Decision coverage
BC - Branch Condition coverage
BCC - Branch Condition Combination coverage
DCU - All C-uses coverage
DPU - All P-uses coverage
DU - All du-paths coverage
L - LCSAJ coverage
MCDC - Modified Condition Decision coverage
S - Statement coverage

分支-条件覆盖，基路径覆盖　　需要讨论（存在一个问题）

黑盒测试

所谓黑盒测试是指在完全不考虑程序的内部结构和处理过程的前提下，在程序接口进行的测试，它只检查程序功能是否能按照规格说明书的规定正常使用，程序是否能适当地接受输入数据产生正确的输出信息，并且保持外部信息的完整性。因此，又称为功能测试或数据驱动。

黑盒测试主要为发现以下几类错误：

(1) 是否有不正确和遗漏了的功能？

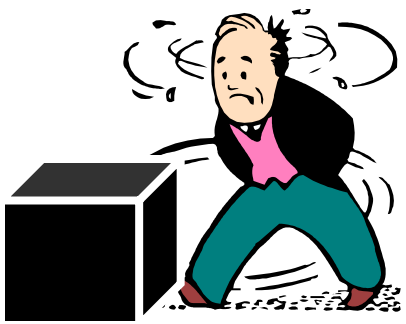
(2) 在接口上，输入是否正确地接受，是否输出正确的结果？

(3) 是否有数据结构错误或外部信息访问错误？

(4) 性能上是否满足要求？

(5) 是否初始化或终止性错误？

微信用户测试大讨论



黑盒测试的难度

例：设有一个程序需要三个整数型的输入数据，如果计算机字长16位，则每个整数可能取的值有2^{16}个，三个输入数据的各种可能值的排列组合共有

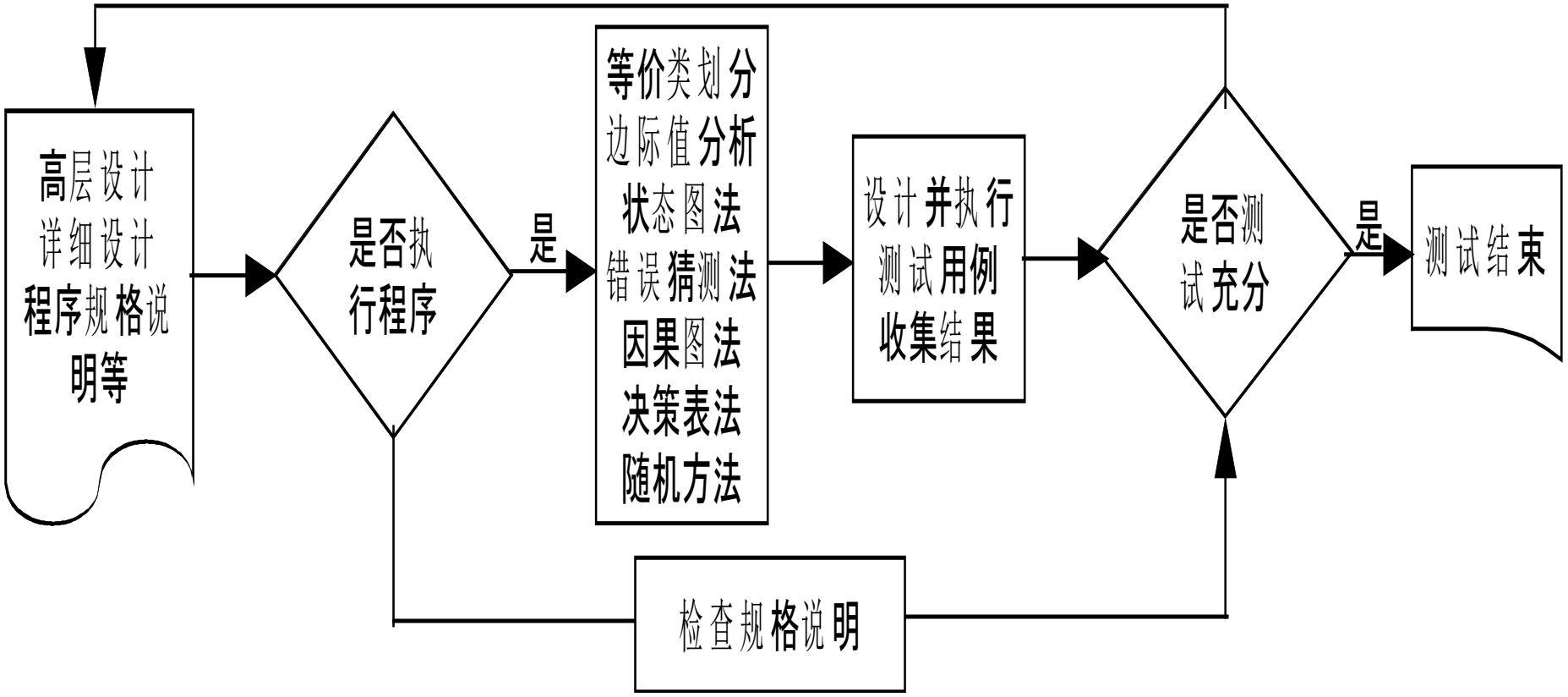
$$2^{16}\times 2^{16}\times 2^{16}=2^{48}\approx 3\times 10^{14}\text{种}$$

也就是说，大约需要把这个程序执行3×10^{14}次才能做到“穷尽”测试。假定每执行一次程序需要一毫秒，执行这么多次大约需要一万年；不仅测试时间长得叫人不可思议，测试得出的输出数据更是多得叫人完全无法分析。

黑盒测试技术

- 等价类划分
- 边界值分析
- 因果图分析
- 错误猜测
- 状态转换测试（state transition testing）
- 语法测试

黑盒测试流程


高层设计
详细设计
程序规格说明等
是否执行程序
是
等价类划分
边际值分析
状态图法
错误猜测法
因果图法
决策表法
随机方法
设计并执行测试用例
收集结果
是否测试充分
是
测试结束
检查规格说明

黑盒测试技术

- 等价类划分
- 边界值分析
- 因果图分析
- 错误猜测
- 状态转换测试（state transition testing）
- 语法测试

等价类划分

- 等价类划分是一种典型的黑盒测试方法，也是一种非常实用的重要测试方法。使用这一发时，完全不考虑程序内部结构，只依据程序的规格说明来设计测试用例。
- 它的指导思想是：把所有可能输入的数据划分成若干等价类，假定每类中的一个典型值在测试中的作用与这类中所有其他值的作用相同。然后可以从每个等价类中只取一组数据作为代表性数据用于测试，以便发现程序中的错误。

等价类划分

(1) 划分等价类

等价类的划分有两种不同情况：

(1) 合理等价类：输入数据满足程序模块的输入数据规范，是有意义的输入数据集合。使用合理等价类构造测试用例，主要检测程序模块是否实现了设计规格规定的功能和性能。

(2) 不合理等价类：输入数据不满足程序模块的输入数据规范，是无意义的输入数据集合。使用不合理等价类构造测试用例，主要检测程序模块是否能够拒绝无效数据输入，被测试对象在运行初始条件不具备时的可靠性如何。

等价类划分

2．等价类的划分原则：

⑴ 如果输入条件规定了取值范围，或值的个数，则可以确定一个有效等价类和两个无效等价类。例如：如果某输入条件规定输入数据的取值范围是：1到99，则有效等价类是[1，99]，两个无效等价类是“小于1和大于99的数”。

⑵ 如果输入条件规定输入值的集合，或者是规定了“必须如何”的条件，则可确立一个有效等价类和一个无效等价类。例如：在某些程序语言中对变量标识符规定为“以字母打头的串”，那么所有以字母打头的构成有效等价类，不以字母打头的归于无效等价类。

⑶ 如果输入条件是一个布尔量，则可以确定一个有效的等价类和一个无效的等价类。

等价类划分

⑷ 如果规定了数据的一组值，而且程序要对每个输入值分别进行处理。这时可为每一个输入值确定一个有效等价类，此外针对这组值确定一个无效等价类，它是所有不允许的输入值的集合。例如，在教师分房中规定对教授、副教授、讲师和助教分别计算分数，做相应的处理。因此，可以确定4个有效等价类为：教授、副教授、讲师和助教，以及一个无效等价类，它是所有不符合上述身份人员的输入值的集合。

⑸ 如果规定了输入数据必须遵守的规则，则可以确立一个有效等价类（符合规则）和若干个无效等价类（从不同角度违反规则）。例如，Pascal语言处理时规定“一个语句必须以分号‘；’结束”。这时可以确定一个有效等价类“以‘；’结束”，若干个无效等价类“以‘：’结束”、“以‘、’结束”、“以‘。’结束”等。

⑹ 如果确知，已划分的等价类中各元素，在程序中的处理方式是不同的，则应将此等价类进一步划分为更小的等价类。

等价类划分

3. 等价类划分的实施步骤：

第一步：划分等价类。划分等价类的依据是被测对象的功能说明/接口定义，构造如下表格。表格中的各等价类给予编号。

输入条件	有效等价类	无效等价类
…… ……	…… ……	…… ……

等价类划分

第二步：设计测试用例：

(1) 为每个等价类规定一个唯一的编号

(2) 设计一个测试用例，使之尽可能的覆盖尚未被覆盖的多个有效等价类。重复这一步，直到所有的有效等价类都被覆盖为止。

(3) 设计一个测试用例，使它覆盖一个而且只覆盖一个尚未被覆盖的无效等价类，重复该步骤，直至所有不合理等价类覆盖完毕。

边界值分析

- 经验表明，处理边界情况时程序最容易发生错误。因此针对各种边界情况设计测试用例。
- 使用边界值分析方法设计测试用例时，首先应该确定边界情况。通常输入等价类与输出等价类的边界，就是应该着重测试边界情况。应当选取正好等于、刚刚大于，或刚刚小于边界的值作为测试数据，而不是选取等价类的典型值或任意值作为测试数据。

边界值分析

选择测试用例的原则：

1) 如果输入条件规定了值的范围，则应取刚达到这个范围的边界的值，以及刚刚超越这个范围边界的值作为测试输入数据。例如，若输入值的范围是“-1.0～1.0”，则可选取“-1.0”、“1.0”、“-1.001”和“1.001”作为测试数据。
2) 如果输入条件规定了值的个数，则用最大个数、最小个数、比最大个数多1，比最小个数少1的数作为测试数据。例如：一个输入文件可有1～255个记录，则可以分别设计1个记录、255个记录以及0个记录和256个记录的输入文件。
3) 根据规格说明的每个输出条件。使用前面的原则1)。例如，某程序的功能是计算折扣量，最低折扣量是0元，最高折扣量是1050元。则设计一些测试用例，使它们恰好产生0元和1050元的结果。此外，还可考虑设计结果为负值或大于1050元的测试用例。

边界值分析

4) 根据规格说明的每个输出条件。使用前面的原则2)。例如，一个信息检索系统根据用户打入得命令，显示有关文献的摘要，但最多只显示4篇摘要。这时可设计一些测试用例，使得程序分别显示1篇，4篇，0篇摘要，并设计一个有可能使程序错误地显示5篇摘要的测试用例。

5) 如果程序的规格说明给出的输入域或输出域是有序集合(如有序表，顺序文件等)，则应选取集合的第一个元素和最后一个元素作为测试用例。

6) 如果程序中使用了一个内部数据结构，则应当选择这个内部数据结构的边界上的值作为测试用例。例如，如果程序中定义了一个数组，其元素下标的下界是0，上界是100，那么应选择达到这个数组小标边界的值，如0与100，作为测试用例。

7) 分析规格说明，找出其他可能的边界条件。

错误推测法

- 错误推测法在很大程度上靠直觉和经验进行。它的基本想法是列举出程序中可能有的错误和容易发生错误的特殊情况，并且根据它们选择测试方案。对于程序中容易出错的情况也有一些经验总结出来，
- 例如，输入数据为零或输出数据为零往往容易发生错误；如果输入或输出的数目允许变化（例如，被检索的或生成的表的项数），则输入或输出的数目为0和1的情况（例如，表为空或只有一项）是容易出错的情况。还应该仔细分析程序规格说明书，注意找出其中遗漏或省略的部分，以便设计相应的测试方案，检测程序员对这些部分的处理是否正确。

判定表

- 等价划分法和边界值分析法都只孤立地考虑各个输入数据的测试功效，而没有考虑多个输入数据的组合效应，可能会遗漏了输入数据易于出错的组合情况。选择输入组合的一个有效途径是利用判定表或判定树为工具，列出输入数据各种组合与程序应作的动作（及相应的输出结果）之间的对应关系，然后为判定表的每一列至少设计一个测试用例。

因果图法

1．因果图的适用范围

要检查输入条件的组合不是一件容易的事情，即使把所有输入条件划分成等价类，它们之间的组合情况也相当多。因此必须考虑使用一种适合于描述对于多种条件的组合，相应产生多个动作的形式来考虑设计测试用例，这就需要利用因果图。因果图方法最终生成的就是判定表。它适合于检查程序输入条件的各种组合情况。

因果图法

2．用因果图法生成测试用例的基本步骤

⑴分析软件规格说明描述中，哪些是原因(即输入条件或输入条件的等价类)，哪些是结果(即输出条件)，并给每个原因和结果赋予一个标识符。

⑵分析软件规格说明描述中的语义，找出原因和结果之间，原因与原因之间对应的关系。根据这些关系，画出因果图。

⑶由于语或环境限制，有些原因和原因之间，原因和结果之间的组合情况不可能出现。为表明这些特殊的情况，在因果图上用一些记号标明约束或限制条件。

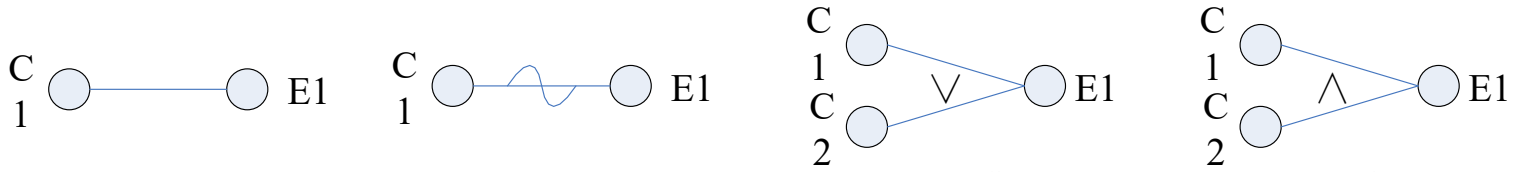
⑷把因果图转换成判定表。

⑸把判定表的每一列拿出来作为依据，设计测试用例。

因果图法

3. 在因果图中出现的基本符号

通常在因果图中用Ci表示原因，用Ei表示结果，其基本符号如下图所示。主要的原因和结果之间的关系有：



(a)恒等　(b)非　(c)或　(d)与

因果图法

(a)恒等：表示原因和结果之间一对一的对应关系。若原因出现，则结果出现。若原因不出现，则结果也不出现。

(b)非：表示原因和结果之间的一种否定关系。若原因出现，则结果不出现。若原因不出现，反而结果出现。

(c)或(∨)：表示若几个原因中有一个出现，则结果出现，只有当这几个原因都不出现时，结果才不出现。

(d)与(∧)：表示若几个原因都出现，则结果才出现。表示若几个原因中有一个不出现，结果就不出现。

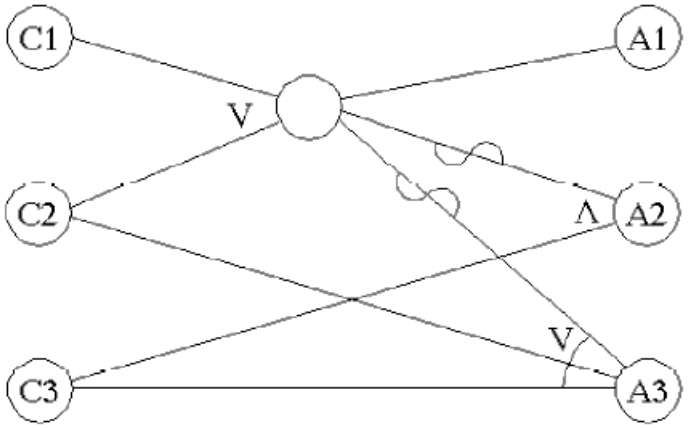
Example

If there are sufficient funds available in the account or the new balance would be within the authorised overdraft limit then the debit is processed. If the new balance would exceed the authorised overdraft limit then the debit is not processed and if it is a postal account it is suspended. Letters are sent out for all transactions on postal accounts and for non-postal accounts if there are insufficient funds available (i.e. the account would no longer be in credit).

Cause effect graph

C1	New balance in credit
C2	New balance overdraft, but within authorised limit
C3	Account is postal

A1	Process debit
A2	Suspend account
A3	Send out letter



This figure uses standard logical symbols for AND and OR (Λ and V respectively) and the wavy line ( ) indicates NOT. The arc groups those inputs affected by a logical symbol , where there are more than two inputs involved.

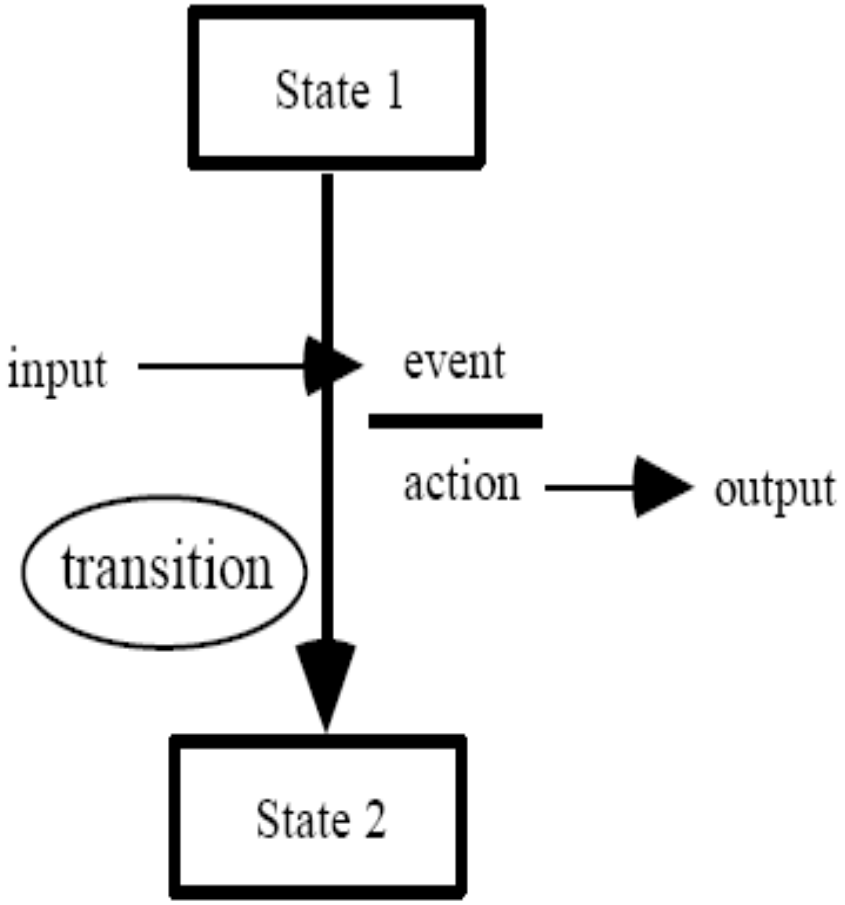
Rules:	1	2	3	4	5	6	7	8
C1: New balance in credit	F	F	F	F	T	T	T	T
C2: New balance overdraft, but within authorised limit	F	F	T	T	F	F	T	T
C3: Account is postal	F	T	F	T	F	T	F	T
A1: Process debit	F	F	T	T	T	T	*	*
A2: Suspend account	F	T	F	F	F	F	*	*
A3: Send out letter	T	T	T	T	F	T	*	*

Test cases

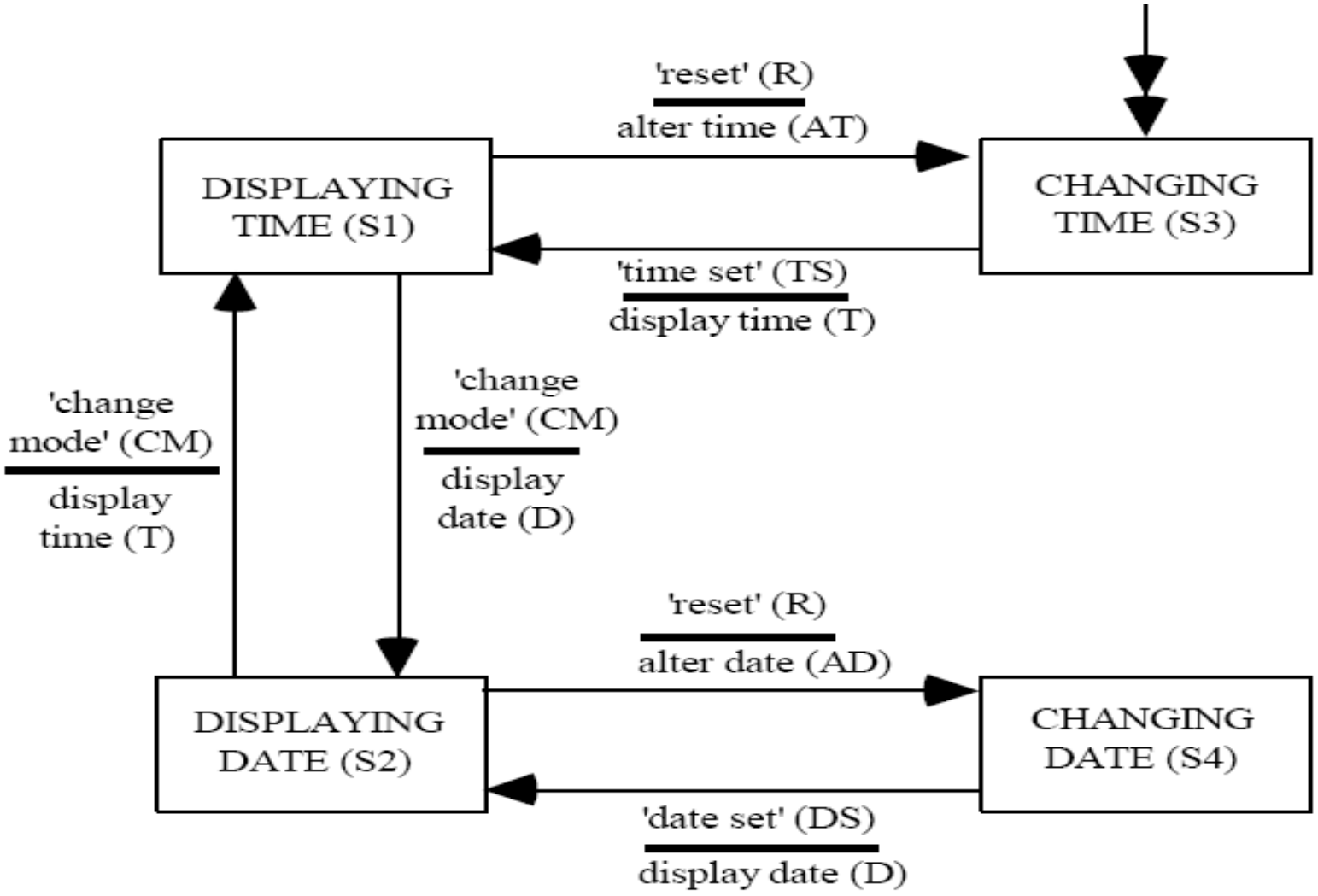
test case	CAUSES				EFFECTS	
	account type	overdraft limit	current balance	debit amount	new balance	action code
1	'c'	£100	-£70	£50	-£70	'L'
2	'p'	£1500	£420	£2000	£420	'S&L'
3	'c'	£250	£650	£800	-£150	'D&L'
4	'p'	£750	-£500	£200	-£700	'D&L'
5	'c'	£1000	£2100	£1200	£900	'D'
6	'p'	£500	£250	£150	£100	'D&L'

State transition testing

A state model is produced for the component to identify its states, transitions, and their events and actions. State transition diagrams (STD) are commonly used as state models and their notation is briefly illustrated opposite.



STD example


'reset' (R)
alter time (AT)
DISPLAYING TIME (S1)
CHANGING TIME (S3)
'time set' (TS)
display time (T)
'change mode' (CM)
display time (T)
'change mode' (CM)
display date (D)
'reset' (R)
alter date (AD)
DISPLAYING DATE (S2)
CHANGING DATE (S4)
'date set' (DS)
display date (D)

Test cases

Test Case	1	2	3	4	5	6	7	8	9	10
Start State	S1	S1	S1	S3	S3	S2	S2	S2	S4	S4
Input	CM	CM	R	TS	TS	CM	CM	R	DS	DS
Exp. Output	D	D	AT	T	T	T	T	AD	D	D
Next State	S2	S2	S3	S1	S1	S1	S1	S4	S2	S2
Input	CM	R	TS	CM	R	CM	R	DS	CM	R
Exp. Output	T	AD	T	D	AT	D	AT	D	T	AD
Finish State	S1	S4	S1	S2	S3	S2	S3	S2	S1	S4

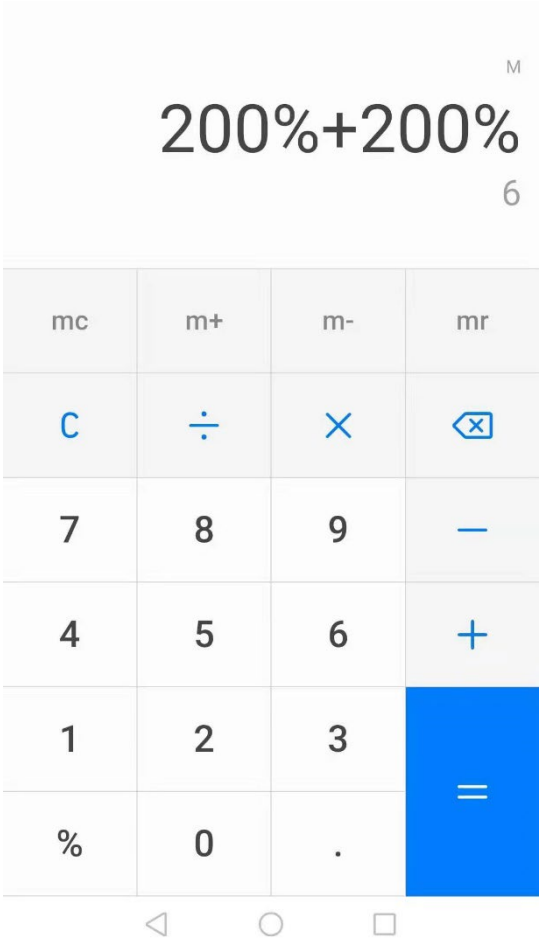
语法测试（Syntax Testing）

- 语法测试是一种不同的测试方法，它基于输入接口的语法变异生成测试用例，因此特别适合在对输入进行确认时发现不足之处。通过语法测试，可以确定合法输入串被接受，非法输入串被拒绝，并且没有输入串会引起系统失效。

语法测试（Syntax Testing）

- 语法测试主要针对三种类型的错误：
- 软件不识别正确的串；
- 软件接收不正确的串；
- 软件在接收一个串时崩溃。

计算器的bug



语法测试（Syntax Testing）

```
float  =  int "e" int.
int    =  ["+"|"-"] nat.
nat    =  {dig}.
dig    =  "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

test case	float_in	option(s) executed	check_res
1	3e2	opt_1	'valid'
2	+2e+5	opt_2	'valid'
3	-6e-7	opt_3	'valid'
4	6e-2	opt_4	'valid'
5	1234567890e3	opt_5	'valid'
6	0e0	opt_6	'valid'
7	1e1	opt_7	'valid'
8	2e2	opt_8	'valid'
9	3e3	opt_9	'valid'
10	4e4	opt_10	'valid'
11	5e5	opt_11	'valid'
12	6e6	opt_12	'valid'
13	7e7	opt_13	'valid'
14	8e8	opt_14	'valid'
15	9e9	opt_15	'valid'

语法测试（Syntax Testing）

```
float   =   int "e" int.
int     =   ["+"|"-"] nat.
nat     =   {dig}.
dig     =   "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
```

```
el_1    =   el_2 el_3 el_4.
el_5    =   el_6 el_7.
el_8    =   el_9.
el_10   =   el_11.
```

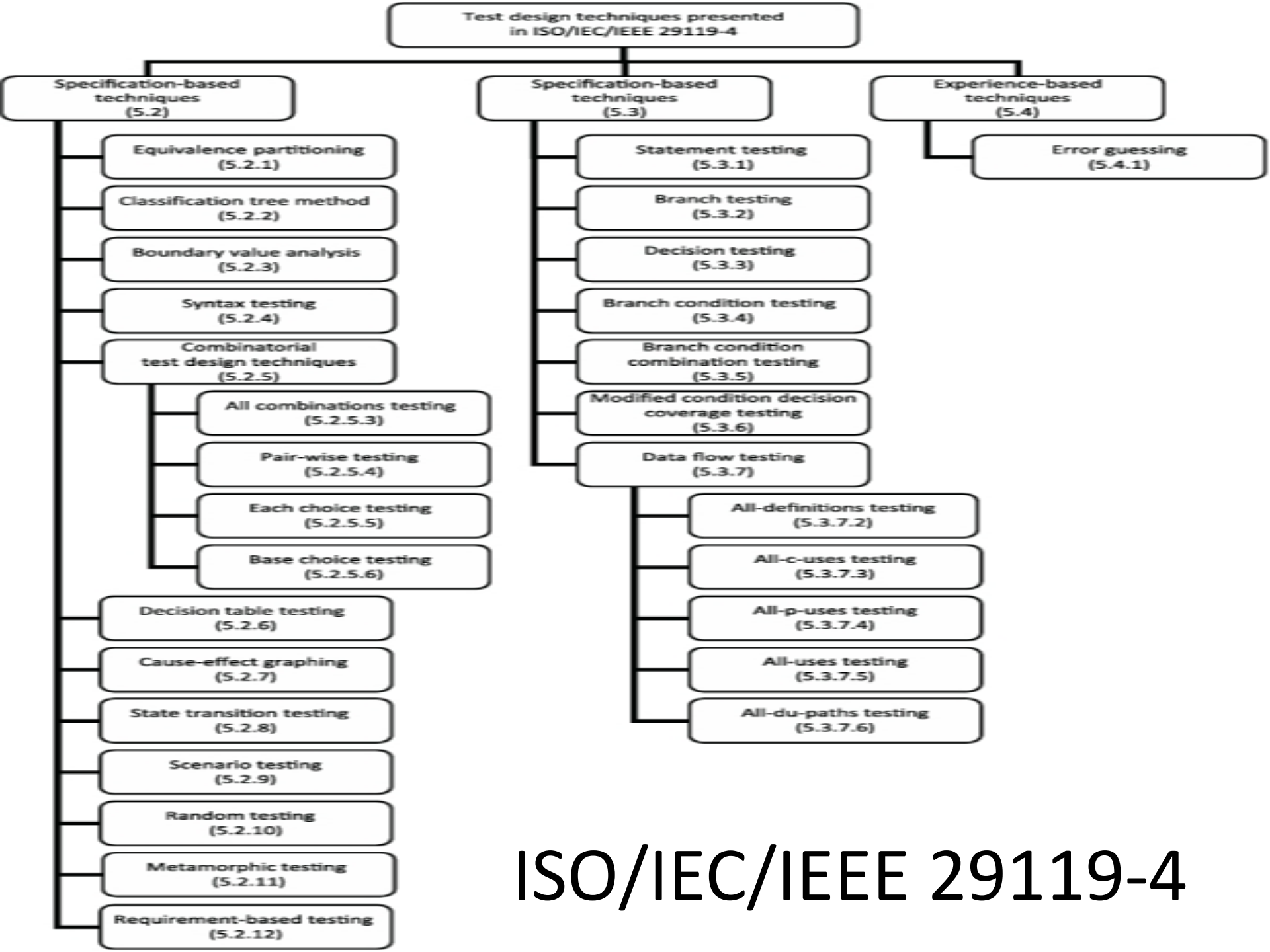
test case	float_in	mutation	element	check_res
1	xe0	m1	x for el_2	'invalid'
2	0x0	m1	x for el_3	'invalid'
3	0ex	m1	x for el_4	'invalid'
4	x0e0	m1	x for el_6	'invalid'
5	+xe0	m1	x for el_7	'invalid'
6	ee0	m2	el_3 for el_2	'invalid'
7	+e0	m2	el_6 for el_2	'invalid'
8	000	m2	el_2 for el_3	'invalid'
9	0+0	m2	el_6 for el_3	'invalid'
10	0ee	m2	el_3 for el_4	'invalid'
11	0e+	m2	el_6 for el_4	'invalid'
12	e0e0	m2	el_3 for el_6	'invalid'
13	+ee0	m2	el_3 for el_7	'invalid'
14	++e0	m2	el_6 for el_7	'invalid'
15	e0	m3	el_2	'invalid'
16	00	m3	el_3	'invalid'
17	0e	m3	el_4	'invalid'
18	y0e0	m4	y in el_1	'invalid'
19	0ye0	m4	y in el_1	'invalid'
20	0ey0	m4	y in el_1	'invalid'
21	0e0y	m4	y in el_1	'invalid'
22	y+0e0	m4	y in el_5	'invalid'
23	+y0e0	m4	y in el_5	'invalid'
24	+0yeo	m4	y in el_5	'invalid'

测试策略-1

- 对软件进行黑盒测试，一定要结合被测试软件的特点，选择性地使用以上方法，并将它们有机地结合起来。
- 例如按照待测试软件的输入输出条件进行等价类划分，通过在等价类中选择测试用例，可以首先将无限的测试域变为有限，同时等价类划分也是边界值分析方法的基础。应用边界值分析可以找出一些容易发现错误的测试输入，这一点在人们在实践中经常得到验证。如果要考虑输入条件的相互作用，就考虑组合测试等相关技术
- 应用错误猜测方法，结合测试人员的直觉和经验选择一些测试用例可以作为重要的补充。如果软件的功能规格说明中含有输入条件的组合情况，就应使用因果图分析方法；如果其中有非常明显的状态转换特征，就应采用状态转换测试方法；如果输入具有严格的语法，可以采用语法测试

测试策略-2

- 在实际测试过程中，往往先进行静态白盒测试，然后进行动态白盒测试方法；或者将这两个方面结合在一起使用，例如在静态白盒测试过程中设计动态测试用例，或者说在设计动态测试用例的过程中可以进静态白盒测试，这样可以设计出更具针对性的白盒测试用例。
- 在黑盒测试基础上增加白盒测试可以有效提高软件质量。



ISO/IEC/IEEE 29119-4

软件测试方法体系

基于代码
code

基于用户
user

基于故障
fault

基于输入
input

基于属性
attribute

基于性质
property

基于开发过程
stage

基于规格
specification

基于软件开发
方法及特性
Nature inspired

基于模型
model

基于专家经验
Intuition&Experience

？ Future？
更多新方法探索
更完整的理论

思考题

- 简述白盒静态测试的4个特征、4个功能和4种形式
- 简述白盒动态测试的9个逻辑覆盖层级和相互关系
- 简述黑盒测试的6种方法，黑盒测试的最优策略是什么？