第 3 章 动态测试技术

第3章思维导图

3.1 黑盒测试技术

3.1.1 边界值分析法

我们知道，函数可以理解为从一个集合（函数的定义域）映射到另一个集合（函数的值域），定义域和值域可以是其他集合的叉积。任何程序都可以看作是一个函数，程序的输入构成函数的定义域，程序的输出构成函数的值域。定义域测试是著名的功能性测试方法之一。这种形式测试的重点是从输入变量的定义域来进行分析并设计出测试用例，但实际上，也可以根据被测程序本身的特点基于变量的值域来分析并设计测试用例。从定义域或值域来分析并设计测试用例往往能互相补充，其基本思想均源于函数。

1. 基本边界值分析

为了便于理解，先讨论具有两个变量 x_1 和 x_2 的函数 F。如果函数 F 对应一个程序，那么输入的两个变量 x_1 和 x_2 的值应该存在取值的边界，其边界值要根据程序的需求来确定，变量的边界值可能是显示的，也可能是隐含的，如果是隐含的则需要根据实际情况进行分析。这里假设变量 x_1 和 x_2 有如下边界：

$$a \leqslant x_1 \leqslant b$$

$$c \leqslant x_2 \leqslant d$$

边界值分析关注的是输入变量的边界，依据边界来设计测试用例。边界值测试的基本原理是程序的错误或缺陷可能出现在输入变量的极限值附近。例如，程序中循环语句的循环次数可能会多一次或少一次，就涉及边界值问题；超市销售系统中的食品保质日期是一个边界值问题；银行系统每天的取款限额也是一个边界值问题。在我们的生活中边界值问题比比皆是。

基本边界值分析的基本思想是在输入变量的取值区间内取最小值、略高于最小值、正常值、略低于最大值和最大值 5 个值。边界值分析也是基于一种关键假设，这种假设称为“单缺陷”假设，即由于缺陷导致的程序失效极少是由两个（或多个）缺陷的同时作用引起的，也就是程序的失效极少是由于两个（或多个）变量在其边界值附近取值引起的，而是由单个变量在其边界值附近取值引起的。

基本边界值分析的测试用例设计规则是：通过使其中的一个变量分别取最小值（min）、比最小值大的值（或略高于最小值，min＋）、位于或接近中间的正常值（nom），以及比最大值小的值（或略低于最大值，max－）和最大值（max）这 5 个值，其他变量都取正常值，每个变量分别取一次。下面是两个变量的基本边界值分析的测试用例的输入组合：

$$\{<x_{1\text{nom}},x_{2\text{min}}>,<x_{1\text{nom}},x_{2\text{min+}}>,<x_{1\text{nom}},x_{2\text{nom}}>,<x_{1\text{nom}},x_{2\text{max}}>,$$

$$<x_{1\text{nom}},x_{2\text{max}}>,<x_{1\text{min}},x_{2\text{nom}}>,<x_{1\text{min+}},x_{2\text{nom}}>,<x_{1\text{nom}},x_{2\text{nom}}>,$$

$$<x_{1\text{max}},x_{2\text{nom}}>,<x_{1\text{max}},x_{2\text{nom}}>\}$$

以上为 10 个测试用例的输入，实际上只要考虑 9 个就可以了，因为当两个变量都取位于或接近中间的正常值时的测试用例有两个，这两个测试用例在实际的测试过程中的效果是相同的，一般不会有新发现。就程序的执行路径而言，这两个测试用例执行的路径相同即也不会发现新错误或缺陷，因此可以省略其中之一。

那么对于 n 个变量的被测程序，基本边界值分析的测试用例数为：对于有 n 变量程序，每次使除一个以外的所有其他变量取正常值，使剩余的那个变量分别取最小值、略高于最小值、

位于或接近中间的正常值、略低于最大值和最大值,对每个变量都重复进行一次。这样,对于一个 n 变量函数,基本边界值分析法会产生 $4n+1$ 个测试用例。

基本边界值分析法可以采用两个步骤:分析变量数和变量的值域。分析变量数,可以根据所测试的程序本身进行分析,例如,在机票订购系统中的查询航班功能,输入的变量可能有出发地、目的地、出发时间、人数、时间段共 5 个变量;确定变量的值域取决于变量本身的性质,例如,对于万年历中的日期处理有月份(m)、天(d)和年(y)三个变量,对于变量 d 和变量 m 无论是定义成枚举类型还是其他数值类型均能很容易地确定其值域;而对于变量 y,可以根据所测试程序实际情况指定一个"人工"值域。值域确定后就可以根据变量的值域取最小值、略高于最小值、正常值、略低于最大值和最大值了。对于"人工"指定的值域要根据具体的情况去考虑,甚至可以取该变量类型允许的最大值和最小值。

边界值分析对布尔变量没有什么意义,布尔取值为 True 和 False,其余三个值不明确。布尔变量可以用后面论述的决策表方法进行测试。

逻辑变量也可以用"遍历"边界值分析来设计测试用例。例如在 ATM 例子中,银行业务处理类型是逻辑变量,其只有三个值:存款、取款和查询。密码也是一个逻辑量,假设进入某系统的密码为 4 位,那么"遍历"所有可能的组合则很困难。所以设计测试用例时根据情况决定。

基本边界值分析具有局限性。如果被测程序有多个独立变量,这些变量也是物理量,则很适合用边界值分析。这里的关键词是"独立"和"物理量"。例如,万年历中的月份、日期和年三个变量之间具有依赖关系,虽然三个变量具有物理量的性质,但边界值分析没有考虑到变量之间的依赖,这样用边界值分析法设计的测试用例其测试效果则不佳。物理量准则决定了物理量的实际含义,对用例的设计很重要。例如,变量(物理量)表示温度、压力、空气速度、迎角、负载等,则对于边界值分析极为重要。如监控系统监控的温度范围;医疗分析系统使用的步进电机确定要分析的样本传送带的位置等都是物理量的例子。物理量便于确定变量的值域。

2. 健壮性边界分析

健壮性边界分析是基本边界值分析的一种简单扩展。除了变量的 5 个边界值分析取值以外,还要取一个略高于最大值的值(max+),以及取一个略低于最小值的值(min−),以测试超过边界极值时系统会有什么表现。

基本边界值分析的大部分讨论都直接适用于健壮性边界分析。健壮性边界分析最有意思的部分不是输入,而是程序的预期输出。当物理量超过其最大值或小于其最小值时程序会出现什么情况呢?如果变量代表飞机机翼的迎角,超出值域范围可能会使飞机失速;如果变量代表电梯的负荷能力,当超出规定的重量时会出现什么情况?健壮性边界分析主要的价值是观察程序的例外处理情况。

健壮性边界分析的测试用例个数分析与基本边界值类似,其理论测试用例数为 $6n+1$,其中 n 为变量的个数。

3. 最坏情况边界分析

在基本边界值分析方法中,我们提及边界值测试分析采用了"单缺陷"假设。除了这种"单缺陷"假设之外,还有所谓的"多缺陷"假设的情况,也就是程序的失效是由于两个(或多个)变量值在其边界值附近取值共同引起的,而不是单个变量在其边界值附近取值引起的。

当我们关心多个变量取极值时程序可能会出现失效的情况,这在电子电路分析中叫作"最坏情况测试",在这里也使用这种思想来讨论最坏情况的边界分析来设计测试用例。其方法是:对每个变量,首先取包含最小值、略高于最小值、正常值、略低于最大值和最大值 5 个值构

成一个集合，然后对这些集合进行笛卡儿积计算，生成的新集合中的每个元素均是一个测试用例的输入。

对于两个变量 x_1 和 x_2 的情况如下：

$$A=\{x_{1\min},x_{1\min+},x_{1\mathrm{nom}},x_{1\max-},x_{1\max}\}$$

$$B=\{x_{2\min},x_{2\min+},x_{2\mathrm{nom}},x_{2\max-},x_{2\max}\}$$

$$\begin{aligned}A\times B=\{&<x_{1\min},x_{2\min}>,<x_{1\min},x_{2\min+}>,<x_{1\min},x_{2\mathrm{nom}}>,<x_{1\min},x_{2\max-}>,\\&<x_{1\min},x_{2\max}>,<x_{1\min+},x_{2\min}>,<x_{1\min+},x_{2\min+}>,<x_{1\min+},x_{2\mathrm{nom}}>,\\&<x_{1\min+},x_{2\max-}>,<x_{1\min+},x_{2\max}>,\cdots\}。\end{aligned}$$

笛卡儿积生成的新集合共有 25 个元素，故有 25 个测试用例集合。

集合 A 和 B 的笛卡儿积中的元素就是测试用例的输入。最坏情况测试显然更彻底，因为基本边界值分析的测试用例是最坏情况边界值分析测试用例集合的真子集。最坏情况测试还意味着花费更多的工作量，即 n 变量函数的最坏情况测试，会产生 5^n 个测试用例，n 为变量的个数。

从以上的分析中，看出诸如测试用例的输入组合$<x_{1\min+},x_{2\min+}>$，这里 x_1 和 x_2 分别取了值域的最小值，这是“多缺陷”假设的体现。

最坏情况边界分析与基本边界值分析一样，两者也有相同的局限性，特别是独立性要求方面的局限性。最坏情况边界分析的最佳运用是物理变量本身存在大量交互的情况，或者在程序失效的代价极高的情况下采用。

除了上述方法之外，还有一种更为极端的边界值分析方法，即健壮最坏情况边界值分析。其测试用例的设计是对每个变量分别取比最小值小、最小值、略高于最小值、正常值、略低于最大值、最大值、略高于最大值共 7 个值构成一个集合，然后对这些变量的取值集合进行笛卡儿积计算，生成的新集合中的每个元素均是一个测试用例的输入。使用健壮最坏情况边界分析的测试用例个数为 7^n，n 为变量的个数。

4. 边界值分析设计测试用例的原则

用边界值分析设计测试用例应遵循以下几条原则：

(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，以及刚刚超过这个范围边界的值作为测试输入数据。

(2) 如果输入变量规定了值的个数，则用最大个数、最小个数、比最小个数少 1、比最大个数多 1 的数作为测试数据。

(3) 边界值分析同样适用于输出变量，根据规格说明的每个输出条件，使用前面的原则(1)和(2)。

(4) 如果程序的规格说明给出的输入域或输出域是有序集合，则应选取集合的第一个元素和最后一个元素来设计测试用例。

(5) 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边界上的值来设计测试用例。

(6) 分析规格说明，找出其他可能的边界条件。

(7) 分析变量的独立性，以确定边界值分析法的合理性。

(8) 在取中间值或正常值时，只要取接近取值范围中间的值就可以了。

(9) 在取比最小值小的值时，根据情况可以取多个，可以取负值、0 和小数。

(10) 在取比最大值大的值时，根据情况可以取多个，当最大值非指定时，根据业务具体分析。

3.1.2 等价类测试法

使用等价类作为功能性测试的基础有两个方面考虑：希望所设计的测试用例既比较完备，同时又避免测试用例的冗余。边界值测试方法不能很好地解决这两个方面的问题，即研究使用边界值分析法设计的测试用例，很容易看出测试用例存在大量冗余，再进一步仔细研究，还会发现测试用例的设计存在严重漏洞，其原因主要是没有考虑到同一个变量的多区间或多含义性，也没有考虑到不同变量之间的依赖关系。等价类测试法从另外一种角度来设计测试用例，其用例设计也使用了“单缺陷”假设和“多缺陷”假设的思想。

1. 等价类的基本思想

在前面的关系概念中讨论过满足等价关系的元素构成等价类。等价类面临的问题是如何对变量(输入或输出变量)划分等价类，同时要分析和考虑等价类划分的粒度问题，根据变量划分成的等价类构成了不同的子集，这些子集的并即是变量的整个集合或全集。这对于测试用例的设计有两点非常重要的意义：子集并成整个集合或全集提供了测试用例设计的完备性；而子集之间的互不相交可保证测试用例设计的一种形式上的无冗余。由于子集是由等价关系决定的，因此子集内的所有元素或所有点在所研究的业务领域内具有共同的性质。等价类测试的思想是通过对每个等价类中取一个元素或一个点来作为测试用例，如果等价类划分合理，则可以大大降低测试用例数量和测试用例之间的冗余。例如，根据输入的三条边判断输出的三角形类型的例子中，应该设计一个测试用例，其输出结果是一个等边三角形的情况，这样的测试用例我们可能选择一个三元组(5.5,5.5,5.5)作为测试用例的输入，也可以选择诸如(6,6,6)和(100,100,100)这样的测试用例输入。直觉告诉我们，程序对这些测试用例的执行过程和第一个测试用例是相同的，因此，其他两个测试用例是冗余的。再如，对于具有不同账户类型的银行系统进行测试也存在类似的等价类问题。如果结合白盒测试(结构性测试)来理解，会看到具有同样账务类型的测试用例在执行时程序的“处理”是相同的，映射到白盒测试去理解就是“遍历相同的执行路径”。

等价类测试的关键就是依据等价关系划分等价类。用一个简单的例子说明等价类的划分问题。为了便于理解，这里讨论一个有两个变量 x_1 和 x_2 的程序 P。输入变量 x_1 和 x_2 有以下边界以及边界内的区间：

$a \leqslant x_1 \leqslant e$，区间为$[a,b)$，$[b,c)$，$[c,d)$，$[d,e]$。

$f \leqslant x_2 \leqslant h$，区间为$[f,g)$，$[g,h]$。

其中，方括号和圆括号分别表示闭区间和开区间的端点。x_1 和 x_2 的无效区间是：$x_1 < a$，$x_1 > e$，以及 $x_2 < f$，$x_2 > h$，如图 3-1 所示。

2. 弱一般等价类测试

上面的例子中两个变量 x_1 和 x_2，根据其范围做出图 3-1 所示的标识分析，从图中可以看出标识为 1、2、3、4、5、6、7、8 的范围的区域均可以理解为一个有效等价类，因为在这些不同的区间内其所有的点具有同样的特性，即符合等价关系的定义，如在区间 1 中的所有点同时符合 $a < x_1 < b$，$g < x_2 < h$。

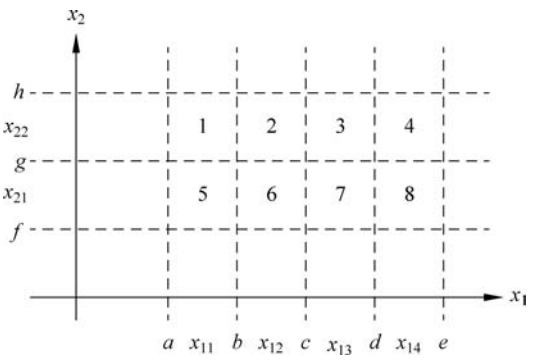


图 3-1 弱一般等价类测试用例分布

弱等价类测试用例的设计基于如下因素考虑：

- 基于“单缺陷”假设；
- 测试用例的个数是变量划分区间最多的那个变量的有效区间个数；
- 测试用例的选取应该考虑分布的均匀。

根据以上分析，对于有两个变量 x_1 和 x_2 的程序 P 的弱等价类测试用例的个数为 4 个，测试用例分布可以是图 3-1 中标号为 1、6、3、8 或者是标号为 5、2、7、4 的区域（有效等价类）的任一组。这组 x_1 和 x_2 的值的组合构成弱一般等价类的测试用例的输入。

3. 强一般等价类测试

强一般等价类测试与弱一般等价类测试的不同主要在于强等价类测试是基于“多缺陷”假设，需要从不同的输入或输出变量划分的有效等价类中或区间中取一个值分别构成集合，这些不同变量取值构成的集合的笛卡儿积中的每个元素就对应一个强一般等价类的测试用例的输入。下面是针对图 3-1 进行的分析。

变量 x_1 和 x_2 在其有效区间构成的集合是：

$$A=\{x_{11},x_{12},x_{13},x_{14}\}$$

$$B=\{x_{21},x_{22}\}$$

$$A\times B=\{<x_{11},x_{21}>,<x_{11},x_{22}>,<x_{12},x_{21}>,<x_{12},x_{22}>,<x_{13},x_{21}>,<x_{13},x_{22}>,<x_{14},x_{21}>,<x_{14},x_{22}>\}$$

在 A 和 B 的笛卡儿积（$A\times B$）中的每个元素均对应图 3-1 中的一个有效等价类区域，所以，其测试用例应该覆盖 1,2,…,8，这 8 个区域（等价类），在每个区域内任一个点构成一个测试用例的输入。这 8 个区域就是 x_1 和 x_2 这两个变量的划分构成的有效等价类。

强一般等价类测试具有一定的完备性：一是保证测试用例覆盖所有的有效等价类；二是输入或输出变量每个有效区间或每个有效等价类之间的每个组合均能取一个测试用例。

通过例子可以看到，“好的”等价类测试的关键是等价关系划分的选择，最好的情况是每个等价类内具有被“相同处理”的特性或等价类内的点在我们研究的业务领域内具有同样的性质。特别强调的是，等价类划分既可以基于输入变量进行，也可以基于输出变量进行。

4. 弱健壮等价类测试

弱健壮等价类测试是在弱一般等价类测试的基础上考虑了无效等价类的情况。测试用例的设计思想仍然是考虑了“单缺陷”假设。弱健壮等价类测试的等价类划分原则与前面的等价类方法相同。其测试用例由两个部分构成：

- 弱一般等价类部分的测试用例。
- 额外弱健壮部分的测试用例。对于 n 个变量而言，在这 n 个变量中每次取一个变量，分别取这个变量的所有可能的无效值和其他 $n-1$ 个变量的有效值组合来构成测试用例的输入，保证如此取法涉及每个变量即每个变量取一次。

对于上面的例子，即有两个变量 x_1 和 x_2 的程序 P，如图 3-2 所示，其变量 x_1 和 x_2 的无效区间均为两个，即 $x_1>e$，$x_1<a$ 和 $x_2>h$，$x_2<f$。

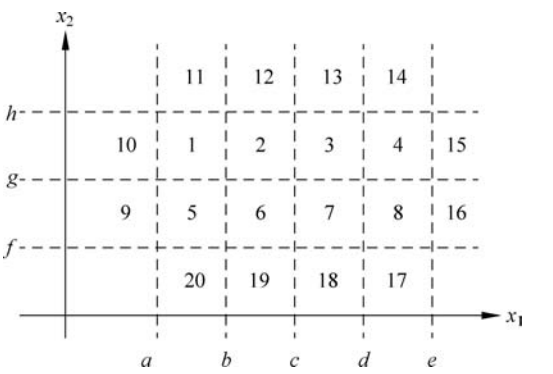


图 3-2　弱健壮等价类测试用例分布 1

图 3-2 中弱健壮等价类测试用例来自以下区域,包含两个部分:弱一般的部分,即在 1、6、3、8 这 4 个区域内或在 5、2、7、4 这 4 个区域内分别取一个测试用例,加上弱健壮部分即在 9、10、11、12、13、14、15、16、17、18、19、20 这 12 个区域内分别取一个测试用例,构成测试用例集的测试用例输入或输出组合。

健壮等价类测试主要测试输入或输出变量无效或例外的情况,在实际的测试中,需求规约中一般没有表达对无效或例外的处理,为无效等价类的测试用例设计带来不便,但测试人员应该积极地理解需求,努力划分可能的无效等价类,以找出程序对无效或例外情况处理的正确性。

5. 强健壮等价类测试

强健壮等价类测试是在强一般等价类测试的基础上考虑无效等价类的情况。测试用例的设计思想仍然考虑多缺陷假设。强健壮等价类测试的等价类划分原则与前面的等价类方法相同,其测试用例由两个部分构成:

- 强一般等价类部分的测试用例。
- 额外强健壮部分的测试用例:是在“额外弱健壮部分的测试用例”的基础上进一步考虑 n 个变量中两个或两个以上变量或所有变量都无效的情况下等价类内的取值。即在 n 个变量划分的等价类(包含有效等价类和无效等价类)中,分别取值构成测试用例,在这些测试用例中 n 个变量的取值可以有一个变量取值来源于该变量的无效等价类,也可能存在其中的两个或 n 个变量的取值全部来源于无效等价类。

实际上强健壮等价类的测试用例输入对应于每个变量区间取值(包括有效区间和无效区间)构成集合的笛卡儿积,笛卡儿积中的每个元素就是一个测试用例的输入组合。值得注意的是,这些元素可能是输入变量的组合,即是测试用例的输入;也可能是输出变量的组合,即是测试用例的输出。针对以上例子,如图 3-3 中标有数字的区域均是测试用例的取值区域,其中的 1、2、3、4、5、6、7、8 为强一般等价类部分的测试用例取值区域;9、10、11、12、13、14、15、16、17、18、19、20、21、22、23、24 为额外强健壮部分的测试用例取值区域。

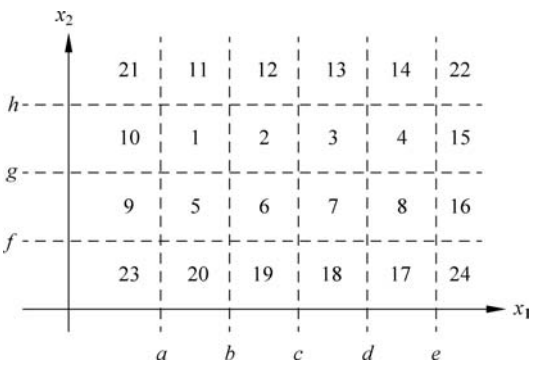


图 3-3 强健壮等价类测试用例分布 2

6. 等价类方法设计原则

等价类是指某个输入域或输出域的子集合。在该子集合中,各个输入数据对于发现程序中的错误或缺陷都是等效的。因此,可以把全部输入或输出数据合理地划分为若干等价类,在每个等价类中取一个数据作为测试的输入条件或输出条件,就可以用少量代表性的测试数据取得较好的测试结果。等价类划分有两种不同的情况:有效等价类和无效等价类。有效等价类是指对于程序的规格说明来说是合理的、有意义的输入数据或输出数据构成的集合,利用有效等价类可检验程序是否实现了规格说明中所规定的功能和性能。无效等价类指与有效等价类的定义恰巧相反。设计测试用例时,要同时考虑这两种等价类。因为软件不仅要能接收合理的数据,也要能经受意外的考验,这样的测试才能确保软件具有更高的可靠性。

下面给出 6 条确定等价类方法的设计原则:

- 在输入或输出条件规定了取值范围或取值个数的情况下,可以确立一个有效等价类和两个无效等价类。

- 在输入或输出条件规定了输入或输出值的集合或者规定了“必须如何”的条件的情况下，可以确立一个有效等价类和一个无效等价类。
- 在输入或输出条件是一个布尔量的情况下，可确定一个有效等价类和一个无效等价类。
- 在规定了输入数据的一组值（假定 n 个），并且程序要对每个输入值分别处理的情况下，可确立 n 个有效等价类和一个无效等价类。
- 在规定了输入数据必须遵守的规则的情况下，可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）。
- 在确知已划分的等价类中各元素在程序中的处理方式的不同，则应再将该等价类进一步地划分为更小的等价类。
- 一个输入条件或一个输出条件均可能划分成多个有效等价类和多个无效等价类。

7. 举例

三角形问题：输入三角形的三条边 a、b、c，程序的输出是这三条边确定的三角形类型。如果 a、b 和 c 满足两边之和大于第三边，且三条边相等，则程序的输出是等边三角形。如果 a、b 和 c 满足两边之和大于第三边，且恰好有两条边相等，则程序的输出是等腰三角形。如果 a、b 和 c 满足两边之和大于第三边，且没有两条边相等，则程序输出的是不等边三角形。如果 a、b 和 c 不满足两边之和大于第三边，则程序输出的是非三角形。

（1）等价类设计方法一：根据输入变量划分等价类。

根据输入变量划分等价类主要是等价类划分的粒度问题，划分的粒度大了，测试用例的设计会有漏洞；粒度太小，则测试用例会有冗余。

第一次尝试，将等价类做如下划分：

$D_1=\{<a,b,c>: a=b=c\}$

$D_2=\{<a,b,c>: a=b, a\neq c\}$

$D_3=\{<a,b,c>: a=c, a\neq b\}$

$D_4=\{<a,b,c>: b=c, a\neq b\}$

$D_5=\{<a,b,c>: a\neq b, a\neq c, b\neq c$，且任意两边之和大于第三边$\}$

分析一下这样的划分，可以看出等价类 D_2、D_3、D_4 的划分粒度过大，这些等价类可能包括构成三角形和不构成三角形的情况，设计出的测试用例会有漏洞。所以必须进行第二次划分尝试，将 D_2、D_3 和 D_4 分别划分成 D_{21}、D_{22}、D_{31}、D_{32} 及 D_{41}、D_{42}。具体划分的结果如下：

$D_1=\{<a,b,c>: a=b=c\}$

$D_{21}=\{<a,b,c>: a=b, a\neq c, a+b>c\}$

$D_{22}=\{<a,b,c>: a=b, a\neq c, a+b\leqslant c\}$

$D_{31}=\{<a,b,c>: a=c, a\neq b, a+c>b\}$

$D_{32}=\{<a,b,c>: a=c, a\neq b, a+c\leqslant b\}$

$D_{41}=\{<a,b,c>: b=c, a\neq b, b+c>a\}$

$D_{42}=\{<a,b,c>: b=c, a\neq b, b+c\leqslant a\}$

$D_5=\{<a,b,c>: a\neq b, a\neq c, b\neq c$，且任意两边和大于第三边$\}$

根据以上划分结果，可以得出如表 3-1 所示的测试用例，这里可以不考虑弱和强的情况，也不考虑 a、b 和 c 为负数和 0 的情况。

表 3-1　依据第二次尝试划分的测试用例

用 例 编 号	a	b	c	对应的等价类	预 期 输 出
$T_1(D_1)$	10.5	10.5	10.5	D_1	等边三角形
$T_2(D_{21})$	20.11	20.11	30	D_{21}	等腰三角形
$T_3(D_{22})$	20	10	8	D_{22}	非三角形
$T_4(D_{31})$	2000	1500	1500	D_{31}	等腰三角形
$T_5(D_{32})$	33	80	33	D_{32}	非三角形
$T_6(D_{41})$	60	35	35	D_{41}	等腰三角形
$T_7(D_{42})$	90	40	40	D_{42}	非三角形
$T_8(D_5)$	11	7	17	D_5	不等边三角形

第二次等价类划分尝试后应该是基本合理的。但是还可以进行第三次划分尝试,即将 D_{22},D_{32} 和 D_{42} 划分为:

$D_{221}=\{<a,b,c>:\ a=b,a\neq c,a+b<c\}$

$D_{222}=\{<a,b,c>:\ a=b,a\neq c,a+b=c\ \}$

$D_{321}=\{<a,b,c>:\ a=c,a\neq b,a+c<b\}$

$D_{322}=\{<a,b,c>:\ a=c,a\neq b,a+c=b\ \}$

$D_{421}=\{<a,b,c>:\ b=c,a\neq b,b+c<a\}$

$D_{422}=\{<a,b,c>:\ b=c,a\neq b,b+c=a\}$

等价类这样划分的结果更加合理,将非三角形的情况中的两边之和小于或等于第三边,分为两边之和小于第三边和等于第三边两种情况,其测试用例的个数达到了 11 个,具体测试用例这里略。

另外,在实际测试用例的设计时应该考虑 a、b 和 c 的值为无效情况的等价类或用边界值的方法设计一些测试用例做补充,即 a、b 和 c 的值为负值和 0 的情况。

(2) 等价类设计方法二,根据输出变量或输出的结果划分等价类。

根据三角形 4 种可能出现的输出:非三角形、不等边三角形、等腰三角形和等边三角形,设计如下的输出(值域)等价类:

$R_1=\{<a,b,c>:$ 有三条边 a、b 和 c 的等边三角形$\}$

$R_2=\{<a,b,c>:$ 有三条边 a、b 和 c 的等腰三角形$\}$

$R_3=\{<a,b,c>:$ 有三条边 a、b 和 c 的不等边三角形$\}$

$R_4=\{<a,b,c>:$ 三条边 a、b 和 c 不构成三角形$\}$

按照弱一般等价类测试用例的设计思想,弱一般等价类共有 4 个测试用例,与强一般等价类的测试用例个数相同,如表 3-2 所示。

表 3-2　三角形程序的弱一般和强一般测试用例

用 例 编 号	a	b	c	预 期 输 出
R_1	6	6	6	等边三角形
R_2	3.3	3.3	4.4	等腰三角形
R_3	7	8	9	不等边三角形
R_4	11	5	5	非三角形

虽然等价类的划分是以输出结果进行的，但是在考虑等价类的健壮情况时，还是要从输入变量入手。这里为了便于问题的讨论，假设 a、b、c 的范围为 $0 < a < 800$，$0 < b < 800$，$0 < c < 800$。那么根据弱健壮等价类设计思想，三角形问题的额外弱健壮等价类测试用例部分如表 3-3 所示，当然还应该考虑 a、b 和 c 为 0 的情况。

表 3-3　额外弱健壮测试用例

用例编号	a	b	c	预期输出
W_1	−1	22	25.5	a 取值不在范围之内
W_2	801	3.3	3.3	a 取值不在范围之内
W_3	0	5.4	8.0	a 不能为零
W_4	15	−2	9	b 取值不在范围之内
W_5	11	803	5	b 取值不在范围之内
W_6	15	0	10.3	b 不能为零
W_7	10	2.5	−5	c 取值不在范围之内
W_8	19.3	790	880	c 取值不在范围之内
W_9	25.1	29	0	c 不能为零

根据额外强健壮等价类设计思想，变量 a、b、c 可能一个无效，可能两个无效，也甚至可能三个全无效。基于数学的组合知识得到额外的强健壮等价类测试用例数为：$C_3^1C_2^1+C_3^2C_2^1C_2^1+C_3^3C_2^1C_2^1C_2^1=26$，这里没考虑 a、b 和 c 为零的情况。表 3-4 是部分额外的强健壮等价类测试用例。

表 3-4　部分额外的强健壮等价类测试用例

用例编号	a	b	c	预期输出
SW_1	−1	22	25.5	a 取值不能为负
SW_2	3.3	801	3.3	b 取值不在范围之内
SW_3	15	9	−2	c 取值不能为负
SW_4	−5.3	803	5	a 不能为负，b 取值不在范围之内
SW_5	10	−2.5	−7	b，c 不能为负
SW_6	−19.3	799	807	a 不能为负，c 取值不在范围之内
SW_7	−1.3	−2.7	−0.01	a、b、c 取值不能为负
SW_8	809	−3.33	−4	a 取值不在范围之内，b、c 不能为负

通过上面例子的第二种设计方法得到三角形程序按照输出变量划分等价类的测试用例结果：

- 弱一般等价类测试用例数：4；
- 强一般等价类测试用例数：4；
- 弱健壮等价类测试用例数：10；
- 强健壮等价类测试用例数：30（没考虑 a、b 和 c 为 0 和负值的情况）；如果考虑 a、b 和 c 为 0 的情况，测试用例数更多。

值得进一步思考的是，对于所测程序，如果每个变量（输入或输出）均能划分成不同的等价类，其弱、强等价类方法设计测试用例的思路是一致的，重要的是应灵活掌握。例如，万年历程

序中的变量 year、month 和 day 就可以分别划分成不同的等价类。

3.1.3 错误推测法

错误推测法就是基于经验和直觉推测程序中所有可能存在的各种错误或缺陷,有针对性地设计测试用例的方法。

错误推测法的基本思想是列举出程序中所有可能的错误或缺陷和容易发生错误或缺陷的特殊情况,根据这些推测来设计测试用例。错误推测法本身不是一种测试技术,而是一种可以应用到所有测试技术中产生更加有效测试的一种技能,例如,设计一些非法、错误、不正确和垃圾数据进行输入测试是很有意义的。如果软件要求输入数字,就输入字母。如果软件只接受正数,就输入负数。如果软件对时间敏感,就输入异常的时间,如在公元 3000 年是否还能正常工作。再如,在单元测试时曾发现并列出的许多在模块中经常出现的错误、以前软件测试中曾经发现的错误等,这些就是经验的总结。另外,错误推测法常常会考虑输入数据和输出数据为 0 的情况,或者输入表格为空格或输入表格只有一行,这些都是容易发生错误的情况,可根据这些情况设计测试用例。

总之,用好错误推测法应该具备如下条件:

- 充分地理解业务;
- 具有开发和测试的实际经验;
- 掌握全面的测试技术。

以下是三个具体的例子:

(1) 测试两位加法计算器中错误推测法的测试用例举例,如表 3-5 所示。

表 3-5 两位加法计算器错误推测法测试用例

错误推测	测试用例	测试结果
边界数据	−99+99	正常计算
规定范围内数据相加	−40+60	正常计算
没有输入数据	空+空	错误提示
输入小数	2.3+10	错误提示
非法字符	A+b	错误提示

(2) 对一个排序程序,根据推测可以列出可能存在错误或缺陷的几种情况:

- 输入表为空。
- 输入表中只有一行。
- 输入表中所有的行都具有相同的值。
- 输入表已经是排序的。

(3) 对一个采用两分法的检索程序,根据推测可以列出可能存在错误或缺陷的几种情况:

- 被检索的表格只有一行。
- 表格的行数恰好是 2 的幂次(如 16)。
- 表格的行数比 2 的幂次多 1 或少 1(如 15、17)。

表 3-6 列出常用错误推测法的部分功能测试用例库,供参考。

表 3-6　常用错误推测法的部分功能测试用例库

部分功能测试用例库	
1. 输入验证	
输入验证主要包括数字输入验证、字符输入验证、输入字符长度验证、必填项验证等	数字输入验证：分别输入数字（正数、负数、零值、单精度、双精度）、字符串、空白值、空值、临界数值。不合法的输入，系统给出必要的判断提示信息
	字符输入验证：分别输入单字节字符、双字节字符、大小写字符、特殊字符、空白值、空值。对于不合法的输入，系统给出必要的判断提示信息
	日期、时间输入验证：分别输入任意字符、任意数字、非日期格式的数据、非正确日期（错误的闰年日期）、空值、空白值。对于不合法的输入，系统给出必要的判断提示信息。注：有些系统会不允许输入当日以后或者以前的日期、时间；有些系统会通过诸如 JavaScript 来自动填写日期时间，这时需要注意是否能人工主观填写输入
	多列表选择框：测试是否能多选，列表框中的数据是否能显示完全。当列表框中的数据过多时，需要对数据有一定格式的排序
	单列表下拉框：测试是否能手工输入，下拉框中的数据是否能显示完整。当下拉框中的数据很多时，需要对数据有一定格式的排序。如果下拉框中数据值过多，可能会超出显示范围，此种情况数据不能够被接收
	大文本输入框（textArea）：虽然它能够满足大数据量的输入，但最好能够显式地标明输入字符的长度限制，并且应该结合“字符输入验证”进行。需要注意的是，应该允许标点的存在
	文件输入框输入验证：该输入框主要用作文件上传操作。在测试过程中，应该注意输入文件的扩展名。从测试角度来看，要求开发人员必须对扩展名进行输入限制，并且在适当的地方输入格式提示。当输入是空值等不合法的输入时，系统给出必要的判断提示信息。另外，对于上传的文件大小应该做限制，不宜太大
	输入字符长度验证：输入字符的长度是否超过实际系统接收字符长度的能力。当输入超出长度时，系统给出必要的判断提示信息
	必填项验证：输入不允许为空的时候，系统需要有提示用户输入信息功能
	格式、规则输入验证：当输入需要一定的格式时，系统需要有提示用户输入信息功能。比如身份证号码可以输入 18 位或者 15 位、部分身份证最后一位为字母、身份证上生日与身份证号码有一定规则
	系统错误定位的输入验证：当输入存在问题时，被系统捕获到，此时页面上的光标能够定位到发生错误的输入框
	单选框、多选框的输入验证：单选框需要依次验证单选框的值是否都有效；多选框需要依次验证多选框的值是否都有效
	验证码验证：做验证码输入验证时，先结合“字符输入验证”进行测试。注意的地方是，当利用 IE 回退或者刷新时，显示的验证码应该和实际系统验证码一致。如果验证码以图片形式显示，但图片由于其他原因（如网络）不能看到或者显示不完整，系统应该允许进行重新获取，最好不要做整个页面刷新
2. 操作验证	
该用例库主要针对页面操作	页面链接检查：每一个链接是否都有对应的页面，并且页面之间切换正确
	相关性检查：删除/增加一项会不会对其他项产生影响，如果产生影响，这些影响是否都正确
	检查按钮的功能是否正确：如增加、删除、修改、查询等功能是否正确
	重复提交表单：一条已经成功提交的记录，用 IE 回退后再提交，看看系统是否做了处理
	多次 IE 回退：检查多次使用 IE 回退的情况，在有回退的地方，回退，回到原来页面，再回退，重复多次，看是否出错

续表

部分功能测试用例库	
该用例库主要针对页面操作	快捷键检查：是否支持常用快捷键，如Ctrl+C、Ctrl+V、Backspace等，对一些不允许输入信息的字段，如选人、选日期对快捷方式是否也做了限制
	Enter键检查：在输入结束后直接按Enter键，看系统如何处理，能否报错
	上传、下载文件检查：上传、下载文件的功能是否实现、上传文件是否能打开、对上传文件的格式有何规定、系统是否有解释信息，并检查系统是否能做到
	其他验证：在页面上图片不宜太大，需要第三方软件支持时，应该给出必要的信息，比如需要JRE的支持，但用户机器还没有安装JRE，那么此时在页面上应该有显著的标志来提醒用户进行安装
3. 登录模块测试用例	
该用例库主要针对登录模块。需要结合“访问控制验证”用例库	登录名输入：进行输入验证。需要注意登录名是否区分大小写和是否有空格
	密码输入：进行输入验证
	提交操作：结合访问空值验证。当输入正确的登录名和密码后，该用户能够进入指定的正确页面。当输入的登录名和密码有误时，系统限制其登录，并且给出适当的提示信息。当遇到错误时，应该进行“错误页面测试”
	重设操作：当进行重设操作时，当前页面上所有输入项被清空
4. 增加操作测试用例	
该用例库主要针对增加操作	添加输入内容，进行输入验证
	应该限制重复增加。具体操作：利用网络传输以及服务器的延迟，多次单击“增加”按钮，经常在数据库中发现重复提交的数据
	当增加成功或者失败后，应该有必要的信息提示
	文件数据的增加：有些增加包含了数据库数据的增加和一些文件的增加，此时的数据会保存在两个地方，所以测试时需要对相关的数据做全面的验证
	文件数据验证：进行输入验证和文件输入框输入验证。注意：当上传的文件名为中文时，上传到服务器后可能会出现乱码现象。现在一般的做法是将原文件名替换成字母和数字的组合，以克服汉字文件名的弊端，另外可以增加文件的安全性
5. 删除操作测试用例	
该用例库主要针对删除操作	选择需要删除的数据字段。有时系统会根据ID来删除，有时系统会根据名称来删除，测试的时候应该多注意。一般要求按照ID来删除，因为根据名称来删除，名称可能会存在重名问题
	应该限制重复删除。具体操作：利用网络传输以及服务器的延迟，多次单击“删除”按钮，经常在数据库中发现重复提交的数据
	当删除的数据还有文件时，需要去验证存在数据库中的数据，以及硬盘下的文件是否都被同时删除
	当数据被删除成功或者失败后，要有相应的信息提示
	进行操作验证
6. 修改操作测试用例	
该用例库主要针对修改操作	打开需要修改的数据页面，注意与增加页面相比，只能修改部分数值，例如关键字等是不能被修改的，并且二者数据应该是一致的
	增加页面上的输入限制与修改页面的输入限制应该一致
	修改成功或者失败后，应该有相应的信息提示
7. 查询操作测试用例	
该用例库主要针对查询操作	条件输入查询，先进行条件输入框的输入验证
	条件组合查询，将多个条件进行组合查询，结果可以通过数据库验证。需要注意的是，整个数据查询和条件查询数据结果的条数要一致。另外，如果遇到某天的查询时间段，有的数据库认为一天不包括零点零分，有的数据库认为包括零点零分

续表

部分功能测试用例库	
该用例库主要针对页面操作	所有查询结果，必须进行一定顺序的排列，可以按照 ID 或名称来排列
	当查询成功或者失败后，系统应给出必要的信息提示
8. 翻页操作测试用例	
该用例库主要针对翻页操作	当数据量很大的时候，需要进行分页显示，每页显示的行数最好不要超过 20 行，每页列表上最好有序号标识，行与行之间的颜色要有一定区分，这样有利于用户的查找
	翻页按钮应该包括首页、前一页、后一页、尾页、当前 X 页、共 X 页，这些常用按钮都能正常显示，并且按钮都能正常翻页
	翻页按钮的每页显示的数据要准确，确保没有查不出来的数据，最好的做法就是和数据库结合起来验证
	页面太多，翻页数据不能全部显示时，系统应该有完善的应对机制，比如只显示当前页的前三页和该页的后三页的页数码
	当翻到某页时，系统应该有明显的标识，标出该页面所处的页码
9. 错误页面测试	
错误页面是在遇到系统异常的情况时产生的友好界面	当系统遇到致命错误时，不能让服务器的调试信息出现在页面上，因为这样做会带来不安全，应该给出一个合适的提示信息
	由于系统繁忙，无法及时给出正确信息时，系统可以给出友好的错误页面，如“请用户稍后再试”等提示信息

3.1.4 因果图法

1. 因果图法的起源

在前面阐述的边界值分析法和等价类划分法中，我们着重考虑输入条件，但未考虑输入条件之间的联系、相互组合等，但是，如果考虑输入条件之间的相互组合，会由于组合情况数目相当大，需要设计大量的测试用例。因此，必须考虑描述多种条件的组合，这些组合相应地会产生多个行动，可以考虑根据这些组合和行动来设计测试用例，这就需要利用因果图。在软件工程中，有些程序的功能可以用判定表的形式来表示，并根据输入条件的组合情况规定相应的操作。这样，可以依据判定表中的每一列设计一个测试用例，以保证被测试程序在输入条件的某种组合下，操作是正确的。

2. 因果图法的特点

- 考虑输入条件间的组合关系；
- 考虑输出条件对输入条件的信赖关系，即因果关系；
- 测试用例发现错误或缺陷的效率高；
- 能检查出功能说明书(规约)中的某些不一致或遗漏；
- 因果图方法最终生产的就是判定表，它适合于检查程序输入条件和各种组合情况。

3. 因果图法的基本步骤

(1) 分割功能说明书。对于规模比较大的程序来说，由于输入条件的组合数太大，所以很难整体上使用一个因果图。可以把它划分为若干部分，然后分别对每个部分使用因果图。例如，可以把一个系统的功能分解成不同子系统的功能，把子系统的功能分解成不同模块的功能，针对每个模块分析因果图。

(2) 识别出原因和结果，并加以编号。所谓原因，是指输入条件或输入条件的等价类；而结果则是指输出条件或输出条件的等价类。每个原因或结果都对应于因果图中的一个结点。当原因或结果成立(或出现)时，相应的结点取值为 1，否则为 0。

(3) 根据功能说明书中规定的原因和结果之间的关系画出因果图的基本符号,如图 3-4 所示。

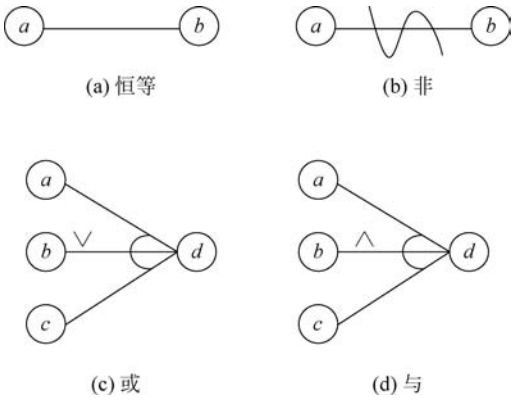


图 3-4 因果图的基本符号

图 3-4 中左边的结点表示原因,右边的结点表示结果。恒等、非、或、与的含义为:

- 恒等:若 $a=1$,则 $b=1$;若 $a=0$,则 $b=0$。即若原因出现,则结果出现;若原因不出现,则结果也不出现。
- 非(∽):若 $a=1$,则 $b=0$;若 $a=0$,则 $b=1$。即若原因出现,则结果不出现;若原因不出现,则结果出现。
- 或(∨):若 $a=1$ 或 $b=1$ 或 $c=1$,则 $d=1$;若 $a=b=c=0$,则 $d=0$。即若几个原因中有 1 个出现,则结果出现;若几个原因都不出现,则结果不出现。
- 与(∧):若 $a=b=c=1$,则 $d=1$;若 $a=0$ 或 $b=0$ 或 $c=0$,则 $d=0$。即若几个原因都出现,结果才出现;若其中有一个原因不出现,则结果不出现。

画因果图时,原因在左,结果在右,由上而下排列,并根据功能说明书中规定的原因和结果之间的关系,用上述基本符号连接起来。在因果图中还可以引入一些中间结点。

(4) 根据功能说明在因果图中加上约束条件。由于语法或环境限制,有些原因与原因之间、原因与结果之间的组合情况不可能出现。为表明这些特殊情况,在因果图上用一些记号表明约束或限制条件。因果图的约束符号如图 3-5 所示。

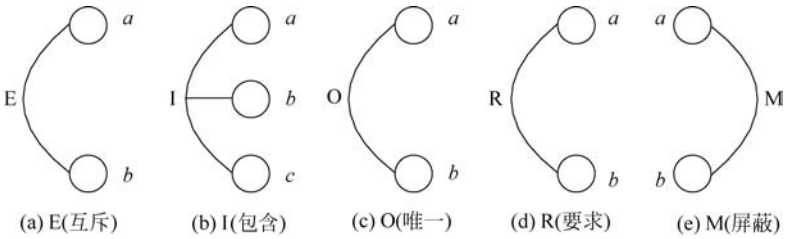


图 3-5 因果图的约束符号

- E(互斥):表示 a、b 两个原因不会同时成立,两个中最多有一个可能成立。即表示不同时为 1,也即 a、b 中至多只有一个为 1。
- I(包含):表示 a、b、c 这三个原因中至少有一个必须成立。即表示至少有一个为 1,也即 a、b、c 中不同时为 0。
- O(唯一):表示 a 和 b 当中必须有一个,且仅有一个成立。即表示 a、b 中有且仅有一个 1。
- R(要求):表示当 a 出现时,b 必须也出现;a 出现时不可能 b 不出现。即表示若 $a=1$,则 b 必须为 1。即不可能 $a=1$ 且 $b=0$。
- M(屏蔽):表示当 a 是 1 时,b 必须是 0;而当 a 为 0 时,b 的值不定。即表示若 $a=1$,则 b 必须为 0。

(5) 根据因果图画出判定表。画判定表的方法一般比较简单,可以把所有原因作为输入条件,每一项原因(输入条件)安排为一行,而所有的输入条件的组合一一列出(真值为 1,假值为 0),对于每一种条件组合安排为一列,并把各个条件的取值情况分别添入判定表中对应的每一个单元格中。例如,如果因果图中的原因有 4 项,那么,判定表中的输入条件则共有 4 行,

而列数则为 $2^4=16$。确定好输入条件的取值之后，便可以很容易地根据判定表推算出各种结果的组合，即输出，其中也包括中间结点的状态取值。上述方法考虑了所有条件的所有组合情况，在输入条件比较多的情况下，可能会产生过多的条件组合，从而导致判定表的行数太多，过于复杂。然而在实际情况中，由于这些条件之间可能会存在约束条件，因此很多条件的组合是无效的，也就是说，它们在判定表中也完全是多余的。因此，根据因果图画出判定表时，可以有意识地排除掉这些无效的条件组合，从而使判定表的列数大幅度减少。

（6）为判定表的每一列设计一个测试用例，即为从因果图中导出的判定表中的每一列设计一个测试用例。因果图生成的测试用例包括了所有输入数据取 True 与取 False 的情况，且测试用例数目随输入数据数目的增加而增加。事实上，对于测试较为复杂的软件，因果图方法常常是十分有效的，它能有力地帮助人们设计有效的测试用例。当然，如果被测软件在设计阶段就采用了判定表，也就不必再画因果图了，而是可以直接利用判定表设计测试用例。

4. 因果图测试实例

某公司产假规定如下：

- 女员工产假为 90 天，符合晚婚、晚育（男 25 周岁，女 23 周岁）的，可增加产假 30 天，共计 120 天。
- 难产凭医院证明，产假增加 15 天。
- 怀孕不满 7 个月小产，产假不超过 30 天，由医生检查酌情确定。
- 男员工符合晚婚、晚育的，可享受陪产假 7 天。

分析因果关系，绘制的实例因果图如图 3-6 所示。

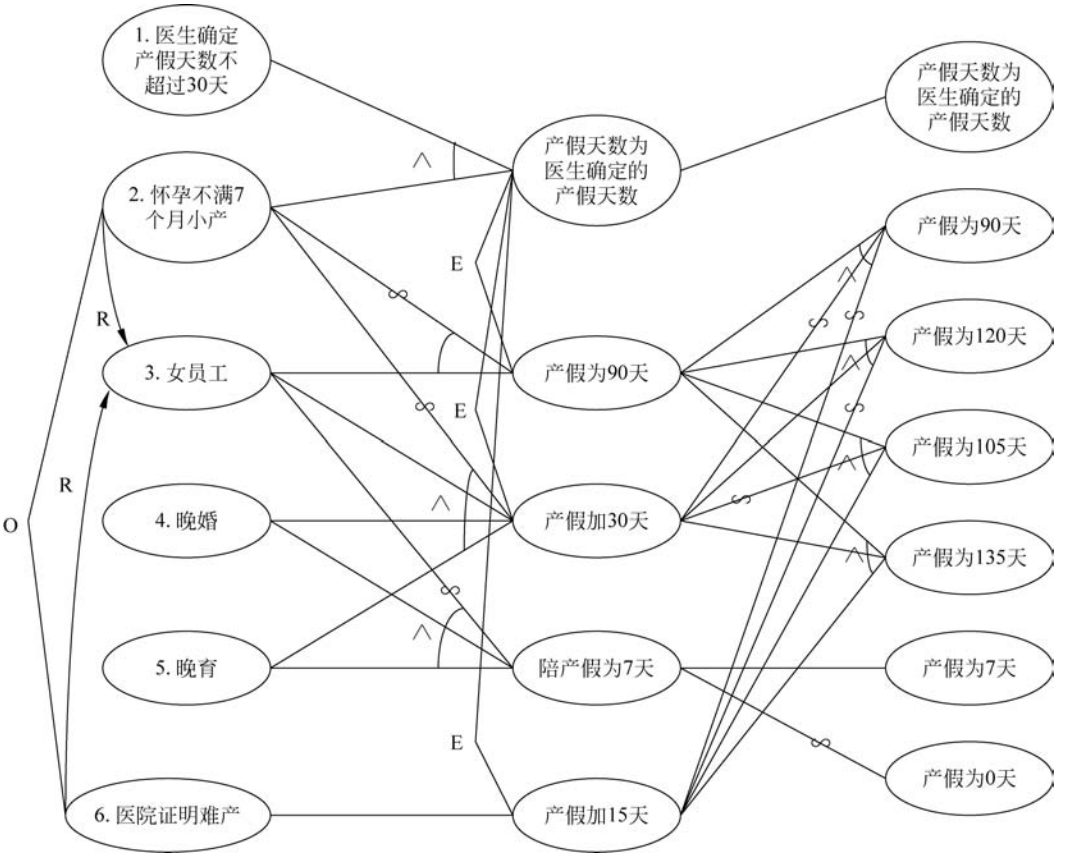


图 3-6　实例因果图

从图 3-6 所示因果图绘制的过程发现,此规定有些条目未予以明确,那么有些情况出现时,就找不到相应的依据了。比如,第二胎的情况如何处理?怀孕不满 7 个月小产时,如果医生认为的产假天数超过了 30 天怎么处理?等等。发现需求、设计的不完善也是科学运用测试方法理清思路进行测试设计一个有益的方面。这些情况如果是在软件开发过程中,无论是开发人员还是测试人员都应当找到制度规定者,并请其明确,否则就会使系统的容错性、健壮性降低,甚至会丢失需求。此例因果图对应的判定表如表 3-7 所示,判定表的每个数据列对应一个测试用例。

表 3-7　实例判定表

条件	女员工	1	1	1	1	1	1	1	1	1	0	0	0	0
	晚婚	/	1	1	0	0	1	1	0	0	1	1	0	0
	晚育	/	1	0	1	0	1	0	1	0	1	0	1	0
	怀孕不满 7 个月小产	1	0	0	0	0	0	0	0	0	/	/	/	/
	医院证明难产	0	1	1	1	1	0	0	0	0	/	/	/	/
	医生明确小产后的产假天数≤30 天	1	/	/	/	/	0	0	0	0	/	/	/	/
中间结果	普通产假天数(90 天)	0	1	1	1	1	1	1	1	1	0	0	0	0
	小产产假天数(医生确定天数)	1	0	0	0	0	0	0	0	0	0	0	0	0
	难产产假天数(15 天)	0	1	1	1	1	0	0	0	0	0	0	0	0
	晚婚晚育产假天数(30 天)	0	1	0	0	0	1	0	0	0	0	0	0	0
	陪产假天数(7 天)	0	0	0	0	0	0	0	0	0	1	0	0	0
结果	假期/天	90	135	105	105	105	120	90	90	90	7	0	0	0

采用因果图分析的一个好处是可以清晰地归纳出输入条件之间的限制关系,直接将某些条件的组合忽略掉,比如男员工的产假、难产、小产情况等。而这种异常情况并非不需要测试,虽然有些输入组合是不可能出现的,但为了检验软件的容错性,还应针对因果图中的各个约束条件,灵活采用等价类划分法和边界值法等测试方法设计测试用例进行有针对性的测试作为补充。

3.1.5　决策表测试法

在所有功能性测试方法中,基于决策表的测试方法是最严格的,因为决策表具有逻辑严格性。在实际的测试中,因果图法和决策表法是两种密切关联的方法,与其他的黑盒测试方法相比,这两种方法的测试用例设计过程比较麻烦。

自从 20 世纪 60 年代初以来,决策表一直被用来表示和分析复杂逻辑关系。决策表很适合描述不同条件集合下采取行动的各种组合的情况。图 3-7 所示为决策表组成示意图。决策表的组成描述如下。

- 条件桩(Condition Stub):列出了问题的所有条件。通常认为列出条件的先后次

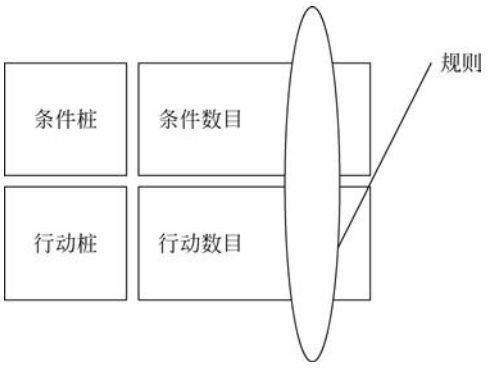


图 3-7　决策表组成示意

序无关紧要。

- 行动桩(Action Stub)：列出了所有可能采取的操作。这些操作之间的排列先后顺序没有约束。
- 条件条目(Condition Item)：列出针对各条件桩的所有可能取值，这些值可能为真假值或其他取值。
- 行动条目(Action Item)：列出在条件条目下的各种取值情况应该采取的动作或操作。
- 规则：任何一个条件组合的特定取值及其相应要执行的操作。

在决策表中贯穿条件条目和行动条目的一列就是一条规则。显然，决策表中列出多少组条件取值，也就有多少条规则，条件条目和行动条目就有多少列，每列对应一个测试用例。表 3-8 给出了决策表的一个例子，共有 6 个测试用例，其中的 X 表示行动桩对应的操作有效，即如果 c_1、c_2 和 c_3 都为真，则采取行动 a_1 和 a_2。如果 c_1 和 c_2 都为真而 c_3 为假，则采取行动 a_1 和 a_3。在 c_1 为真、c_2 为假的条件下，规则中的 c_3 条目叫作“不关心条目”。不关心条目有两种主要解释：条件无关、条件不适用或此条件不需要考虑。通常用“n/a”来表示。

表 3-8　决策表例子

桩	规则 1	规则 2	规则 3、4	规则 5	规则 6	规则 7、8
c_1	T	T	T	F	F	F
c_2	T	T	F	T	T	F
c_3	T	F	—	T	F	—
a_1	X	X		X		
a_2	X				X	
a_3		X		X		
a_4			X			X

在决策表中，如果所有条件都是二叉条件的决策表，即每个条件只能取两个值：“真”和“假”，这样的决策表叫作“有限条目决策表”，如条件“ATM 中现金是否够?”，只能取“真”表示够和“假”表示不足。如果决策表的每个条件可以有多个值，甚至每个取值对应于变量的等价类，则对应的决策表叫作“扩展条目决策表”，如条件“ATM 交易类型?”，可以取“存款”“查询”和“取款”三种值。

在使用决策表设计测试用例时，把条件解释为输入，把行动解释为输出。有时条件也可以为输入的等价类，行动是被测软件的主要功能处理部分。这时规则就解释为测试用例。由于决策表是机械地强制为完备的，因此决策表具有测试用例的完整性。

下面用三角形程序的例子分析在决策表的条件条目中出现“不关心条目”和“不可能条目”的情况，该决策表是有限条目的决策表。

在表 3-9 所示的决策表中，给出了“不关心条目”和“不可能条目”使用的例子。本例中有 4 个条件 c_1、c_2、c_3 和 c_4，根据数学的排列组合原理，4 个条件中的每个条件均可以取“真”和“假”两个值，一共有 2^4 种组合。而当 c_1 条件为“假”时(这里“真”用 Y 表示，“假”用 N 表示)，c_2、c_3 和 c_4 均为不关心条目，因为条件 c_1 为“假”，其取值为“不能构成三角形”，再考虑 c_2、c_3 和 c_4 取什么值就没有实际意义了，所以这时候 c_2、c_3 和 c_4 均是不关心条目，用“—”来表示。在表 3-9 中条件条目为“Y，Y，Y，N”“Y，Y，N，Y”和“Y，N，Y，Y”时均是“不可能”条目，因为这

些条件组合和三角形的实际业务逻辑结合理解是不可能存在的。之所以有这些不可能组合的存在是因为这些组合是根据数学排列组合的原理得到的,这些包含有不关心条目和不可能的规则也可能对应测试用例,以测试例外的情况。

表 3-9　三角形问题决策表

桩	规划 1	规划 2	规划 3	规划 4	规划 5	规划 6	规划 7	规划 8	规划 9
c_1：a、b、c 构成三角形?	N	Y	Y	Y	Y	Y	Y	Y	Y
c_2：$a=b$?	—	Y	Y	Y	Y	N	N	N	N
c_3：$a=c$?	—	Y	Y	N	N	Y	Y	N	N
c_4：$b=c$?	—	Y	N	Y	N	Y	N	Y	N
a_1：非三角形	X								
a_2：不等边三角形									X
a_3：等腰三角形					X		X	X	
a_4：等边三角形		X							
a_5：不可能			X	X		X			

在决策表的设计中,由于条件的选取不同得到的决策表也不同,当然,根据决策表设计的测试用例也不同。所以在设计决策表时,条件的选取至关重要。表 3-10 所示的决策表给出了关于三角形问题决策表的条件的另一种考虑,读者可以进一步分析表 3-10 中的不可能条目和不关心条目。另外,决策表的条件越多,将会大大地扩展决策表的规模。这里将条件(c_1：a、b、c 构成三角形?)扩展为三角形特性的三个不等式的详细表示。如果有一个不等式不成立,则三个整数就不能构成三角形。还可以进一步扩展,因为不等式不成立有两种方式：一条边等于另外两条边之和,或严格大于另外两条边之和。

表 3-10　不同条件的三角形问题决策表

桩	规划 1	规划 2	规划 3	规划 4	规划 5	规划 6	规划 7	规划 8	规划 9	规划 10	规划 11
c_1：$a<b+c$?	F	T	T	T	T	T	T	T	T	T	T
c_2：$b<a+c$?	—	F	T	T	T	T	T	T	T	T	T
c_3：$c<a+b$?	—	—	F	T	T	T	T	T	T	T	T
c_4：$a=b$?	—	—	—	T	T	T	T	F	F	F	F
c_5：$a=c$?	—	—	—	T	T	F	F	T	T	F	F
c_6：$b=c$?	—	—	—	T	F	T	F	T	F	T	F
a_1：非三角形	X	X	X								
a_2：不等边三角形											X
a_3：等腰三角形							X		X	X	
a_4：等边三角形				X							
a_5：不可能					X	X		X			

另外,在设计决策表时应该考虑规则之间可能出现的冗余及不一致的情况,在表 3-11 中,规则 9 的行动条目与规则 1～4 的行动条目相同,这种情况只要冗余规则中的行动与决策表相应的部分相同,决策表的设计就不会有大问题。如果条件条目相同而行动条目不同,例如表 3-11 所示的情况,则决策表的设计会有问题,也就是说决策表的设计存在错误。

表 3-11　一个冗余的决策表

桩	规划 1～4	规划 5	规划 6	规划 7	规划 8	规划 9
c_1	T	F	F	F	F	T
c_2	—	T	T	F	F	F
c_3	—	T	F	T	F	F
a_1	X	X	X	—	—	X
a_2	—	X	X	X	—	—
a_3	X	—	X	X	X	X

如果表 3-12 所示的决策表是和具体的业务逻辑相对应，其中 c_1 是真，c_2 和 c_3 都是假，那么规则 1～4 和规则 9 都适用，即规则 1～4 隐含包含规则 9。这样可以观察到以下情况：

- 规则 1～4 和规则 9 是不一致的。
- 决策表是非确定的。

规则 1～4 和规则 9 是不一致的，是因为行动的组合不同，所以整个决策表是不确定的。测试人员的基本原则是在决策表中小心使用不关心条目。

表 3-12　一个不一致的决策表

桩	规划 1～4	规划 5	规划 6	规划 7	规划 8	规划 9
c_1	T	F	F	F	F	T
c_2	—	T	T	F	F	F
c_3	—	T	F	T	F	F
a_1	X	X	X	—	—	—
a_2	—	X	X	X	—	X
a_3	X	—	X	X	X	—

针对三角形例子，使用表 3-10 给出的决策表，可得到 11 个功能性测试用例：3 个不可能测试用例；3 个测试用例违反三角形性质；1 个测试用例可得到等边三角形；1 个测试用例可得到不等边三角形；3 个测试用例可得到等腰三角形，如表 3-13 所示。如果将表 3-10 中的条件“两边之和是否大于第三边”扩展为“一条边等于另外两条边之和”和“一条边严格大于另外两条边之和”，则决策表所对应的测试用例会增加，如会增加满足“一条边正好等于另外两条边的和”情况的测试用例。

表 3-13　根据决策表 3-10 得到的测试用例

测试用例 ID	*a*	*b*	*c*	预 期 输 出
TTC001	4	1	2	非三角形
TTC002	1	4	2	非三角形
TTC003	1	2	4	非三角形
TTC004	5	5	5	等边三角形
TTC005	?	?	?	不可能
TTC006	?	?	?	不可能
TTC007	2	2	3	等腰三角形
TTC008	?	?	?	不可能
TTC009	2	3	2	等腰三角形
TTC010	3	2	2	等腰三角形
TTC011	3	4	5	不等边三角形

以上主要讨论涉及有限条目的决策表。对于扩展条目决策表的测试用例设计,重要的是要分析条件的划分和每个条件的取值,因为扩展条目的决策表的条件可以取多个值。下面以一个具体的例子来讨论扩展条目决策表的测试用例设计。

被测软件的需求描述为:假设有一个银行信誉卡系统,当月刷卡销费额及本年度未按时还款的次数和当月赠送礼品折合金额占总刷卡额的比例关系如表3-14所示,当未按时还款次数大于或等于其对应的刷卡消费额所允许的未按时还款次数时,则免于赠送礼品,用决策表设计测试用例。为了使问题简单化,在此仅考虑本年度的情况,不考虑跨年的累计。

表3-14 刷卡消费额与未按时还款次数及奖励额关系

刷卡消费额/元	本年度容许未按时还款最大次数	赠送礼品折合金额占总刷卡额的比例/%
$2000 < N \leqslant 4000$	0	0.3
$4000 < N \leqslant 7000$	1	0.4
$7000 < N \leqslant 10\,000$	2	0.5
$10\,000 < N \leqslant 15\,000$	3	0.7
$N > 15\,000$	5	0.8

首先,必须确定条件变量的个数,根据需求进行分析,有"刷卡消费额"和"本年度未按时还款次数"两个条件变量,分别用 N 和 M 来表示。其中变量 N 可以划分成如下等价类,每个等价类分别是条件变量 N 的取值范围:

$N_1 = \{2000 < N \leqslant 4000\}$

$N_2 = \{4000 < N \leqslant 7000\}$

$N_3 = \{7000 < N \leqslant 10\,000\}$

$N_4 = \{10\,000 < N \leqslant 15\,000\}$

$N_5 = \{N > 15\,000\}$

对于条件变量 M,由于其最多的本年度未按时还款次数为11次(隐含需求),所以合理的等价类划分如下,且每个等价类分别是条件变量 M 的取值:

$M_1 = \{0\}$

$M_2 = \{1\}$

$M_3 = \{2\}$

$M_4 = \{3\}$

$M_5 = \{4,5\}$

$M_6 = \{6,7,8,9,10,11\}$

以上分析条件变量 M 和 N 的取值分别是等价类,由此依据扩展条目决策表的设计原则设计的决策表如表3-15所示。每个规则对应一个测试用例,共10个测试用例。

表3-15 扩展决策表用例设计实例

标号	1	2	3	4	5	6	7	8	9	10
$C_1:N$	N_1		N_2		N_3		N_4		N_5	
$C_2:M$	M_1	$M_2 \sim M_6$	M_1, M_2	$M_3 \sim M_6$	M_1, M_2, M_3	$M_4 \sim M_6$	M_1, M_2, M_3, M_4	M_5, M_6	$M_1 \sim M_5$	M_6
A_1:有刷卡奖励	X		X		X		X		X	

续表

标号	1	2	3	4	5	6	7	8	9	10
A_2：无刷卡奖励		X		X				X		X
奖励额度	$N\times$ 0.3%	0	$N\times$ 0.4%	0	$N\times$ 0.5%	0	$N\times$ 0.7%	0	$N\times$ 0.8%	0

决策表的设计建立在软件规格说明的基础上，其设计基本步骤如下：

(1) 根据规约分析条件个数和条件的取值，决定用有限条目的决策表还是用扩展条目的决策表。

(2) 分析理论规则的个数。假如用有限条目的决策表设计，每个条件取“真”和“假”两个值，那么对于 n 个条件的决策表规则数为 2^n 个；假如用扩展条目的决策表设计，规则的个数可以根据不同变量划分的等价类个数的积及结合其他方法来确定。

(3) 列出所有可能的行动桩。

(4) 列出所有的条件条目和行动条目，并考虑“不可能条目”和“不关心条目”。

(5) 根据规则完成测试用例的设计。

(6) 评审决策表和测试用例集。

3.1.6 场景法

现在的软件几乎都是用事件触发来控制流程的，像 GUI 软件、游戏软件等事件触发时的情景便形成了场景(Use Case)，而同一事件不同的触发顺序和处理结果就形成事件流。这种在软件设计方面的思想也可引入软件测试中，可以比较生动地描绘出事件触发时的情景，有利于设计者设计测试用例，同时使测试用例更容易理解和执行。

提出这种测试思想的是 IBM Rational 公司，并在 RUP2000 中文版中有详尽的解释和应用。在使用场景法测试一个软件时，测试流程按照一定的事件流正确地实现某个软件的功能时，这个流称为该软件功能的基本流；而凡是出现故障或缺陷或例外的流程，就称为备选流。分别将基本流和备选流加以标注，这样的话，备选流就可以是源于基本流，或是由备选流中引出的。

用例场景用来描述流经用例的路径，从用例开始到结束遍历这条路径上所有基本流和备选流，也就是说，场景是由基本流和备选流组成。下面分析基本流和备选流的概念。

1. 理解规约确定基本流和备选流

如图 3-8 所示，直黑线表示基本流，是测试用例对应的最简单的路径。备选流用带有弧线的箭头线表示，一个备选流可能从基本流开始，在某个特定条件下执行，然后重新加入基本流中(如图 3-8 中的备选流 1 和 3)；也可能起源于另一个备选流(如备选流 2 就是起源于备选流 1 的备选流)，或者终止用例而不再重新加入某个流(如备选流 2 和 4)。经过用例的每条路径都由基本流和备选流来表示。

总之，基本流就是正常的业务流；备选流是非正常的业务流，即被中断的或是意外的业务流。按照图 3-8 中所示的基本流和备选流，可以确定以下不同的用例场景。

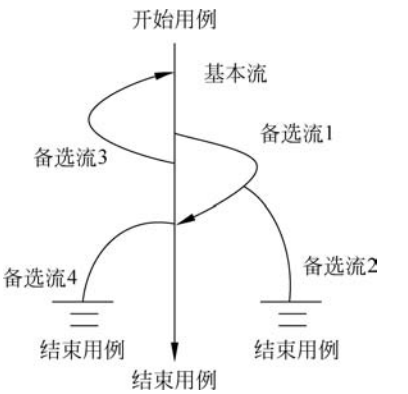


图 3-8 基本流和备选流

场景 1：基本流。

场景 2：基本流、备选流 1；基本流。

场景 3：基本流、备选流 1、备选流 2。

场景 4：基本流、备选流 3；基本流。

场景 5：基本流、备选流 3、备选流 1；基本流。

场景 6：基本流、备选流 3、备选流 1、备选流 2。

场景 7：基本流、备选流 4。

场景 8：基本流、备选流 3、备选流 4。

场景 9：基本流、备选流 3；基本流、场景 3。

在以上场景中，场景 2、4 和 7 只是经过了一种备选流，而场景 3、5、6、8、9 经过了一种以上的备选流。实际上备选流在一个场景中可以出现多次甚至是无数次，当然，无数次的情况实际上是不可能的，因为这样测试用例的执行就不会结束了。需要说明的是，为了能清晰地说明场景，这里所举的例子都是非常简单的，在实际应用中往往比较复杂。下面再举一个例子。

需求描述：在一个学生成绩管理系统中，教师登录系统后，具有增加学生成绩、删除学生成绩、修改学生成绩和打印学生成绩的功能。

其中，教师修改学生成绩功能的基本流可以理解为教师修改学生成绩成功这个业务流。对于备选流，可以理解为：

- 修改学生成绩时学生信息不存在；
- 修改学生成绩时学生信息存在但成绩不存在；
- 修改学生成绩失败。

可以根据以上分析的基本流和备选流设计不同的场景。至于其他功能，基本流和备选流的设计方法相同。

2. 场景法设计步骤

(1) 根据说明书或规约，分析出系统或程序功能的基本流及所有可能的备选流。

(2) 根据基本流和各项备选流设计不同的场景。

(3) 对每个场景生成相应的逻辑测试用例。

(4) 根据逻辑测试用例设计实际(物理)测试用例。

(5) 对生成的测试用例集进行评审，基本的覆盖指标是基本流和所有的备选流在所设计的场景中至少覆盖一次。

3. 实例分析

这里以一个网上书店为例说明场景法设计测试用例的过程。网上书店的订购过程为：用户登录网站后，进行书籍的选择，当选好自己心仪的书籍后进行订购，这时把所需图书放进购物车，等进行结账时，用户需要登录自己注册的用户，登录成功后，进行结账并生成订单，整个购物过程结束。

(1) 分析基本流和所有可能的备选流，如表 3-16 所示。

表 3-16　基本流和备选流

分类	说　明
基本流	用户登录网站，选择书籍，进行订购，把所需图书放进购物车，结账时登录自己的用户，登录成功后生成订单，选择支付方式(银行卡在线支付)，订单确认
备选流 1	用户不存在
备选流 2	用户密码错误

续表

分类	说　明
备选流 3	银行账号不存在
备选流 4	银行账号资金不足
备选流 5	银行账号密码错误
备选流 6	银行卡达到每日最大消费金额(假设每天的最大消费金额为 2000 元)
备选流 7	无选购书籍
备选流 x	退出系统

(2) 根据基本流和备选流来设计场景,如表 3-17 所示。

表 3-17　场景设计列表

分　类	说　明	
场景 1——购物成功	基本流	
场景 2——用户不存在	基本流	备选流 1
场景 3——用户密码错误	基本流	备选流 2
场景 4——银行账号不存在	基本流	备选流 3
场景 5——银行账号资金不足	基本流	备选流 4
场景 6——银行账号密码错误	基本流	备选流 5
场景 7——银行卡达到每日最大消费金额	基本流	备选流 6
场景 8——无选购书籍	基本流	备选流 7
场景 9——退出系统	基本流	备选流 x
场景 10	基本流	备选流 1,备选流 3,备选流 5,备选流 x
场景 11	基本流	备选流 2,备选流 3,备选流 6

表 3-17 中的场景 1～9 覆盖了所有的基本流和备选流。场景 10 和场景 11 是所列举的另外两种情况的场景,这样的场景有很多甚至无穷,因为在一个场景中任何一个备选流可能出现多次。在设计场景时在满足基本流和所有的备选流在所设计的场景中至少覆盖一次的情况下,其他的场景根据业务流的具体情况去设计,如根据业务的使用概率来设计等。

(3) 逻辑测试用例的设计。

对于每一个场景都需要设计测试用例。可以采用矩阵或决策表来设计和管理测试用例。下面显示了一种通用格式,其中各行代表各个逻辑测试用例,而各列则代表测试用例的信息。本例中,对于每个测试用例,存在一个测试用例 ID、对应的场景(条件)、测试用例中涉及的所有数据元素(作为输入或已经存在于数据库中)以及预期结果。

通过从确定执行用例场景所需的数据元素入手构建矩阵。然后,对于每个场景,至少要确定包含执行场景所需的适当条件的测试用例。例如,在下面的矩阵中,V(有效)用于表明这个条件必须是 Valid(有效的)才可执行基本流,而 I(无效)用于表明这种条件下将激活所需备选流。表 3-18 中使用的"n/a"(不可获得或不考虑)表明这个条件不适用于测试用例。表 3-18 为场景对应的逻辑测试用例,在该表中场景 9、场景 10、场景 11 对应的逻辑测试用例有其特殊性。在场景 9 中,在购书的任何时刻或购书过程中的任何阶段均可以退出系统,不能保证购书过程结束,该情况有多种,表 3-18 列出的只是其中的一种逻辑测试用例。场景 10 也是类似的情况,不过该场景必须覆盖备选流 1、备选流 3 和备选流 5 之后才能退出系统,不能保证购书过程结束。场景 11 要求该场景必须覆盖备选流 2、备选流 3、备选流 6。

表 3-18　逻辑测试用例列表

测试用例编号	场景(条件)	用户	用户密码	银行账号	账号密码	消费的金额	账面金额	没超过每天最大金额	选购书籍	退出系统	预期结果
1	场景 1	V	V	V	V	V	V	V	V	V	成功购书
2	场景 2	I	n/a	V	V	n/a	V	n/a	V	n/a	用户不存在
3	场景 3	V	I	V	V	n/a	V	n/a	V	n/a	用户密码错误
4	场景 4	V	V	I	n/a	n/a	n/a	n/a	V	n/a	银行账号不存在
5	场景 5	V	V	V	V	V	I	V	V	n/a	银行账号资金不足
6	场景 6	V	V	V	I	V	V	V	V	n/a	银行账号密码错误
7	场景 7	V	V	V	V	V	V	I	V	n/a	银行卡达到每日最大消费金额
8	场景 8	n/a	n/a	n/a	n/a	n/a	n/a	n/a	I	n/a	无选购书籍
9	场景 9	V	V	V	V	V	V	V	V	I	购书成功退出或不成功退出
10	场景 10	I	V	I	I	n/a	n/a	n/a	n/a	I	不成功退出
11	场景 11	V	I	I	V	V	V	I	V	V	购书成功退出或不成功退出

(4) 设计实际(物理)测试用例。

实际(物理)测试用例就是前面描述的逻辑测试用的实例化,也就是给逻辑测试用例填上相应的数据。例如,上例中的用户名、密码、银行账号、账号密码、消费金额、账目金额、所购书籍等给出具体的值。具体测试用例如表 3-19 所示。

表 3-19　实际(物理)测试用例

测试用例编号	场景(条件)	用户	用户密码	银行账号	账号密码	消费的金额/元	账面金额/元	没超过每天最大金额/元	选购书籍	退出系统	预期结果
1	场景 1	Duolc	654321	123-54321	123456	580	5000	有效	《软件测试》	正常退出	成功购书
2	场景 2	Duolc	n/a	123-54321	123456	n/a	5000	n/a	《软件质量控制》	n/a	用户不存在
3	场景 3	Duolc	543210	123-54321	123456	n/a	5000	n/a	《软件工程》	n/a	用户密码错误
4	场景 4	Duolc	654321	012-54321	n/a	n/a	n/a	n/a	《一地鸡毛》	n/a	银行账号不存在
5	场景 5	Duolc	654321	123-54321	123456	1008	1000	有效	《印象》	n/a	银行账号资金不足
6	场景 6	Duolc	654321	123-54321	654321	105	1000	有效	《易经》	n/a	银行账号密码错误
7	场景 7	Duolc	654321	123-54321	123456	2005	3000	无效	《易经解读》	n/a	银行卡达到每日最大消费金额

续表

测试用例编号	场景（条件）	用户	用户密码	银行账号	账号密码	消费的金额/元	账面金额/元	没超过每天最大金额/元	选购书籍	退出系统	预期结果
8	场景 8	n/a	n/a	n/a	n/a	n/a	n/a	n/a	*Nature Science*	n/a	无选购书籍
9	场景 9	Duolc	654321	123-54321	123456	200	3000	有效	《史记》	正常或非正常退出	购书成功退出或不成功退出
10	场景 10	duolc	654321	113-54321	123456	n/a	n/a	n/a	n/a	非正常退出	不成功退出
11	场景 11	Duolc	123456	112-54321	123456	2500	3000	无效	《二十四史》	正常或非正常退出	购书成功退出或不成功退出

（5）用例评审及完善。

评审对象是表 3-18 和表 3-19 中的测试用例，主要关注测试用例本身的合理性，如预期输出是否正确等；另外需要关注的是测试用例是否覆盖了所有的备选流等。对测试用例不合理的部分进行完善修改。

3.1.7 正交实验法

1. 正交实验法的由来

“拉丁方”名称的由来：古希腊是一个多民族的国家，国王在检阅臣民时要求每个方队中每行有一个民族代表，每列也要有一个民族代表。数学家在设计方阵时，以每一个拉丁字母表示一个民族，所以设计的方阵称为“拉丁方”。

什么是 n 阶拉丁方？用 n 个不同的拉丁字母排成一个 $n(n<26)$ 阶方阵，如果每行的 n 个字母均不相同，每列的 n 个字母均不相同，则称这种方阵为 $n*n$ 拉丁方或 n 阶拉丁方，即每个字母在任一行、任一列中只出现一次。什么是正交拉丁方？设有两个 n 阶的拉丁方，如果将它们叠合在一起，恰好出现 n^2 个不同的有序数对，则称为这两个拉丁方为互相正交的拉丁方，简称正交拉丁方。例如，3 阶拉丁方如下：

```
A B C        A B C
B C A   和    C A B
C A B        B C A
```

用数字替代拉丁字母：

```
1 2 3        1 2 3                (1,2) (2,2) (3,3)
2 3 1   和    3 1 2   -------->    (2,3) (3,1) (1,2)
3 1 2        2 3 1                (3,2) (1,3) (2,1)
```

2. 正交实验法介绍

根据正交拉丁方的由来，可得知正交实验设计（Orthogonal Experimental Design）是研究

多因素(也称因子)、多水平的又一种设计方法,它是根据正交性从全面实验中挑选出部分有代表性的点进行实验,这些有代表性的点具备了“均匀分散,齐整可比”的特点。正交实验设计是分式析因设计(析因设计是一种多因素的交叉分组设计)的主要方法,是一种高效率、快速、经济的实验设计方法。

日本著名的统计学家田口玄一将正交实验选择的水平组合列成表格,称为正交表。例如作一个三因素三水平的实验,按全面实验要求,需进行 $3^3=27$ 种组合的实验,且尚未考虑每一组合的重复数。若按 $L_9(3^3)$ 正交表安排实验,只需做 9 次,按 $L_{18}(3^7)$ 正交表进行 18 次实验,显然大大减少了工作量。因而正交实验设计在很多领域的研究中已经得到广泛应用。

在利用决策表或因果图来设计测试用例时,作为输入条件的原因与输出结果之间的因果关系,有时很难从软件需求规格说明中得到。也往往由于因果关系非常庞大,导致利用决策表或因果图而得到的测试用例数目多得惊人,给软件测试带来沉重的负担。为了有效、合理地减少测试的工时与费用,可利用正交实验法进行测试用例的设计。

正交实验法是依据 Galois 理论,从大量的(实验)数据(测试用例)中挑选适量的、有代表性的点(用例),从而合理地安排实验(测试)的一种科学实验设计方法。类似的方法有聚类分析方法、因子方法等。下面通过一个例子来说明正交实验法。

需求描述:为提高某化工产品的转化率,选择了三个有关因子进行条件实验,为反应温度 A、反应时间 B、用碱量 C,并确定了它们的实验范围如下。

A:80～90℃;

B:90～150min;

C:5%～7%。

实验目的是搞清楚因子 A、B、C 的取值对转化率有什么影响,哪些是主要的,哪些是次要的,从而确定最适的生产条件,即反应温度、反应时间及用碱量各为多少才能使转化率最高。这里对因子 A、B 和 C 在实验范围内都选了三个水平,如下所示。

A:$A_1=80$℃,$A_2=85$℃,$A_3=90$℃;

B:$B_1=90$min,$B_2=120$min,$B_3=150$min;

C:$C_1=5\%$,$C_2=6\%$,$C_3=7\%$。

当然,在正交实验设计中,因子可以是定量的,也可以是定性的。而定量因子各水平间的距离可以相等,也可以不相等。这个三因子三水平的条件实验,通常有两种实验方法:取三因子所有水平之间的组合,共有 $C_3^1\ C_3^1\ C_3^1=27$ 次实验,即 $A_1B_1C_1$,$A_1B_1C_2$,$A_1B_2C_1$,……,$A_3B_3C_3$。用图 3-9 表示立方体的 27 个结点。这种实验法叫作全面实验法。

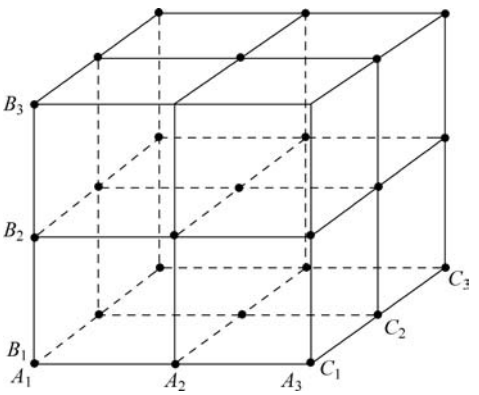


图 3-9　全面实验法图例

全面实验法对各因子及各因子的不同取值间的组合关系剖析得比较清楚,但实验次数太多。特别是当因子数目多,且每个因子的水平数目也很多时,实验量非常大。如选 6 个因子,每个因子取 5 个水平时,如要做全面实验,则需 $5^6=15\ 625$ 次实验,这实际上是不可能实现的,如果应用正交实验法,只做 25 次实验就够了。而且在某种意义上讲,这 25 次实验代表了 15 625 次实验。

下面用简单对比法分析，即变化一个因子而固定其他因子，如首先固定 B、C 于 B_1、C_1，使 A 变化：

$$B_1C_1 \begin{cases} \rightarrow A_1 \\ \rightarrow A_2 \\ \rightarrow A_3(\text{好结果}) \end{cases}$$

若得出结果以 A_3 为最好，则固定 A 于 A_3，C 还是 C_1，使 B 变化：

$$A_3C_1 \begin{cases} \rightarrow B_1 \\ \rightarrow B_2(\text{好结果}) \\ \rightarrow B_3 \end{cases}$$

若得出结果以 B_2 为最好，则固定 B 于 B_2，A 于 A_3，使 C 变化：

$$A_3B_2 \begin{cases} \rightarrow C_1 \\ \rightarrow C_2(\text{好结果}) \\ \rightarrow C_3 \end{cases}$$

实验结果以 C_2 为最好。于是就认为最好的工艺条件是 $A_3B_2C_2$。

这种方法一般也有一定的效果，但缺点很多。首先，这种方法的选点代表性很差，如按上述方法进行实验，实验点完全分布在立体的一个角上，而在其他大的范围内没有选点，因此这种实验方法不全面，所选的工艺条件 $A_3B_2C_2$ 不一定是 27 个组合中最好的。其次，用这种方法比较条件好坏时，是把单个的实验数据拿来进行数值上的简单比较，而实验数据中必然包含着误差成分，所以单个数据的简单比较不能剔除误差，必然造成结论的不稳定。另外，对于 A、B 和 C 先固定哪两个变量的取值或两个相同变量取值的不同也决定最后的实验结果。

考虑兼顾这两种实验方法的优点，从全面实验的点中选择具有典型性、代表性的点，使实验点在实验范围内分布得很均匀，能反映全面情况。但我们又希望实验点尽量地少，为此还要具体考虑一些问题。如上例，对应于 A 有 A_1、A_2、A_3 共 3 个平面，对应于 B、C 也各有 3 个平面，共 9 个平面。则这 9 个平面上的实验点都应当一样多，即对每个因子的每个水平都要同等看待。具体来说，每个平面上都有 3 行 3 列，要求在每行每列上的点一样多，这符合正交拉丁方的思想。如图 3-10 所示的设计，实验点用⊙表示。

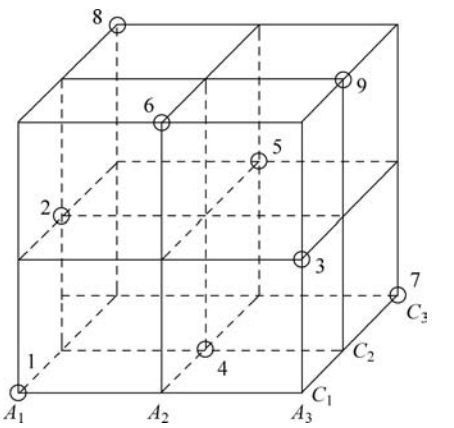


图 3-10　正交实验法图例

可以看到，在 9 个平面中每个平面上都恰好有 3 个点，而每个平面的每行每列都有 1 个点，而且只有 1 个点，总共 9 个点。这样的实验方案，实验点的分布很均匀，实验次数也不多。

当因子数和水平数都不太大时，尚可通过作图的办法来选择分布很均匀的实验点。但是因子数和水平数多了，作图的方法就不行了。实验工作者在长期的工作中总结出一套办法，创造出所谓的正交表。按照正交表来安排实验，既能使实验点分布得很均匀，又能减少实验次数，而且计算分析简单，能够清晰地阐明实验条件与指标之间的关系。用正交表来安排实验及分析实验结果，这种方法叫正交实验设计法，也称正交实验法。

3. 利用正交实验法测试用例的步骤

(1) 提取功能说明,构造因子(因素)——状态(水平)表。

把影响实验指标的条件称为因子,而影响实验因子的条件叫因子的状态。利用正交实验法来设计测试用例时,首先要根据被测试软件的规格说明书找出影响其功能实现的操作对象和外部因素,把它们当作因子;而把各个因子的取值当作状态(水平)。对软件需求规格说明中的功能要求进行划分,把整体的、概要性的功能要求进行层层分解与展开,分解成具体的有相对独立性的、基本的功能要求。这样就可以把被测试软件中所有的因子都确定下来,并为确定每个因子的权值提供参考的依据。确定因子与状态(水平)是设计测试用例的关键。因此要求尽可能全面、正确地确定取值,以确保测试用例的设计做到完整与有效。

(2) 加权筛选,生成因素分析表。

对因子与状态的选择可按其重要程度分别加权。可根据各个因子及状态的作用大小、出现频率的大小以及测试的需要确定权值的大小。

(3) 利用正交表构造测试数据集。

利用正交实验法设计测试用例与使用等价类划分、边界值分析、因果图等方法相比,具有以下优点:节省测试工作工时;可控制生成的测试用例数量;测试用例具有一定的覆盖率。

在使用正交实验法时,要考虑到被测系统中要准备测试的功能点,而这些功能点就是要获取的因子或因素。但每个功能点要输入的数据按等价类划分有多个,也就是每个因素的输入条件,即状态或水平值。

用例设计的简洁实用步骤为:

(1) 确定因素(变量);

(2) 确定每个因素有几个状态(水平)(变量的取值);

(3) 选择一个合适的正交表;

(4) 把变量的值映射到表中;

(5) 把每一行的各因素水平的组合作为一个测试用例;

(6) 添加认为可疑且没有在正交表中出现的组;

(7) 评审测试用例集。

4. 正交表的构成分析

行数(Runs):正交表中行的个数,即实验的次数,也是通过正交实验法设计的测试用例的个数。

因素数(Factors):正交表中列的个数,即要测试的功能点。

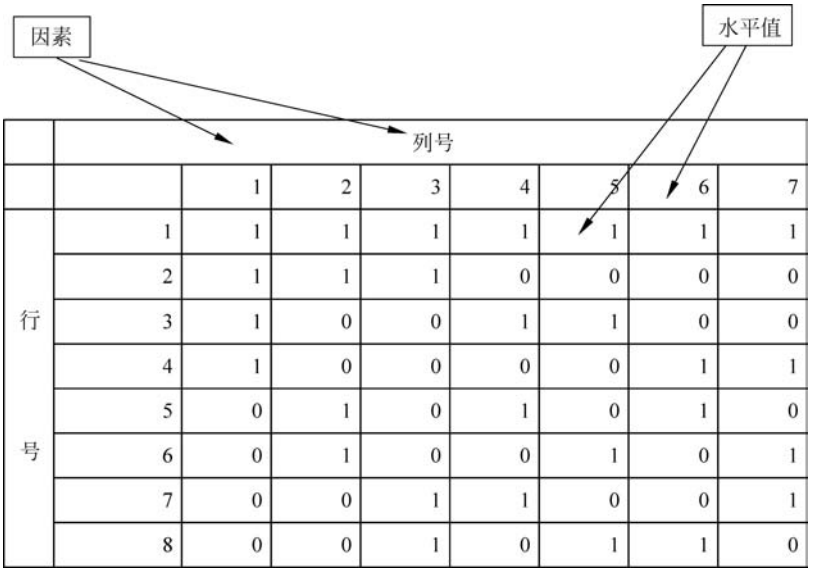
状态或水平数(Levels):任何单个因素能够取得的值的最大个数。正交表中的包含的值为从0到"水平数-1"或从1到"水平数",即要测试功能点的输入条件。

正交表的形式:$L_{行数}(水平数^{因素数})$。

如$L_8(2^7)$,其中7为此表列的数目(最多可安排的因子数);2为因子的水平数;8为此表行的数目(实验次数),如图3-11所示。

又例如$L_{18}(2\times3^7)$,有7列是3水平的,有1列是2水平的,$L_{18}(2\times3^7)$的数字告诉我们,用它来安排实验,做18个实验最多可以考查1个2水平因子和7个3水平因子。在行数为mn型的正交表中(m,n是正整数),实验次数(行数)$=\sum$(每列水平数-1)$+1$,如$L_8(2^7)$,$8=7\times(2-1)+1$。利用上述关系式可以从所要考查的因子水平数来决定最低的实验次数,进而选择合适的正交表。比如要考查5个3水平因子及一个2水平因子,则起码的实验次数为

$5\times(3-1)+1\times(2-1)+1=12$(次)，这就是说，要在行数不小于 12，既有 2 水平列又有 3 水平列的正交表中选择，$L_{18}(2\times3^7)$适合。



行号	列号							
		1	2	3	4	5	6	7
	1	1	1	1	1	1	1	1
	2	1	1	1	0	0	0	0
	3	1	0	0	1	1	0	0
	4	1	0	0	0	0	1	1
	5	0	1	0	1	0	1	0
	6	0	1	0	0	1	0	1
	7	0	0	1	1	0	0	1
	8	0	0	1	0	1	1	0

图 3-11　正交表 $L_8(2^7)$

正交表具有两条性质：每一列中各数字出现的次数都一样多；任何两列所构成的各有序数对出现的次数都一样多。所以称之为正交表。例如在 $L_9(3^4)$中(如表 3-20 所示)，各列中的 1、2、3 都各自出现 3 次；任何两列，例如第 3、4 列，所构成的有序数对从上到下共有 9 种，既没有重复也没有遗漏。其他任何两列所构成的有序数对也是这 9 种各出现一次，这反映了实验点分布的均匀性。

表 3-20　$L_9(3^4)$正交表

行号	列号			
	1	2	3	4
	水平			
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

实验方案应该如何设计呢？安排实验时，只要把所考查的每个因子任意地对应于正交表的一列(一个因子对应一列，不能让两个因子对应同一列)，然后把每列的数字“翻译”成所对应因子的水平。这样，每一行的各水平组合就构成了一个实验条件(不考虑没安排因子的列)。对于上面例子，因子 A、B、C 都是 3 水平的，实验次数要不少于 $3\times(3-1)+1=7$(次)，可考虑选用 $L_9(3^4)$。因子 A、B、C 可任意地对应于 $L_9(3^4)$的某 3 列，例如 A、B、C 分别放在 1、2、3 列，然后

实验按行进行,顺序不限,每一行中各因素的水平组合就是每一次的实验条件,从上到下就是这个正交实验的方案,如表 3-21 所示。这个实验方案的几何解释正好是正交实验设计图例。

表 3-21　产品转化率的实验方案

列号 / 行号	A	B	C	
	1	2	3	4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

实验号	水平组合	实验条件		
		温度/℃	时间/min	加碱量/%
1	$A_1B_1C_1$	80	90	5
2	$A_1B_2C_2$	80	120	6
3	$A_1B_3C_3$	80	150	7
4	$A_2B_1C_2$	85	90	6
5	$A_2B_2C_3$	85	120	7
6	$A_2B_3C_1$	85	150	5
7	$A_3B_1C_3$	90	90	7
8	$A_3B_2C_1$	90	120	5
9	$A_3B_3C_2$	90	150	6

3 个 3 水平的因子做全面实验需要 3^3 次试验,现用 $L_9(3^4)$来设计实验方案,只要做 9 次,工作量减少了 2/3,而在一定意义上代表了 27 次实验。

5. 正交实验法举例

设计测试用例时的 3 种情况:

- 因素数(变量)、水平数(变量值)符合正交表。
- 因素数不相同。
- 水平数不相同。

(1) 因素数与水平数刚好符合正交表。

下面举一个包括 3 个控件的界面的例子,如图 3-12 所示。

图 3-12　一个包括 3 个控件的界面

这是个人信息查询系统中的一个窗口。可以看到要测试的控件有 3 个:姓名、身份证号码、手机号码,也就是要考虑的因素有 3 个。而每个因素里的状态(水平)有两个:填与不填。

选择正交表时需要分析:

- 表中的因素数大于或等于 3;
- 表中至少有 3 个因素数的水平数大于或等于 2;
- 行数取最少的一个,即行数最少为 3×(2−1)+1=4。

从正交表中开始查找,结果为 $L_4(2^3)$。

变量映射结果如下:

		列号		
		1	2	3
行号	1	0	0	0
	2	0	1	1
	3	1	0	1
	4	1	1	0

		列号		
		姓名	身份证号码	手机号码
行号	1	填	填	填
	2	填	不填	不填
	3	不填	填	不填
	4	不填	不填	填

0—填　　1—不填

对应的测试用例如下：

- 填写姓名、填写身份证号码、填写手机号码。
- 填写姓名、不填身份证号码、不填手机号码。
- 不填姓名、填写身份证号码、不填手机号码。
- 不填姓名、不填身份证号码、填写手机号码。

根据其他测试方法增补以下测试用例作为补充：

不填姓名、不填身份证号码、不填手机号码。

从测试用例可以看出，如果按每个因素两个水平数来考虑，则需要 8 个测试用例，而通过正交实验法进行的测试用例只有 5 个，减少了测试用例数。用最小的测试用例集合去获取最大的测试覆盖率。

(2) 因素数不相同。

如果因素数不同，可以采用包含的方法，在正交表公式中找到包含该情况的正交表，如果有 N 个符合条件的正交表，那么选取行数最少的正交表。

(3) 水平数不相同。

采用包含和组合的方法选取合适的正交表，下面以例子来说明。

上面就正交实验法进行了讲解，现在再拿 PowerPoint 软件打印功能作为例子，希望能让大家更好地理解该方法的具体应用。

假设功能描述如下：

打印范围分为全部、当前幻灯片、给定范围 3 种情况；

打印内容分为幻灯片、讲义、备注页、大纲视图 4 种方式；

打印颜色/灰度分为颜色、灰度、黑白共 3 种设置；

打印效果分为幻灯片加框和幻灯片不加框两种方式。

因素状态(水平)如表 3-22 所示。

表 3-22　因素状态(水平)

状态(水平)/因素	A(打印范围)	B(打印内容)	C(打印颜色/灰度)	D(打印效果)
0	全部	幻灯片	颜色	幻灯片加框
1	当前幻灯片	讲义	灰度	幻灯片不加框
2	给定范围	备注页	黑白	
3		大纲视图		

先将中文字转换成字母，这样便于设计。得到新的因素状态表 3-23 所示。

表 3-23　转换后的因素状态表

状态/因素	A	B	C	D
0	A_1	B_1	C_1	D_1
1	A_2	B_2	C_2	D_2
2	A_3	B_3	C_3	
3		$B4$		

进一步分析：被测项目中一共有 4 个被测对象，每个被测对象的状态(水平)都不一样。

选择正交表：

- 表中的因素数大于或等于 4。
- 表中至少有 4 个因素的水平数大于或等于 2。
- 行数取最少的一个,即满足($3^2\times4^1\times2^1$)的最少行数:$2\times(3-1)+1\times(4-1)+1\times(2-1)+1=9$。

最后选中正交表公式 $L_{16}(4^5)$,对应的正交矩阵如表 3-24 所示。

表 3-24 $L_{16}(4^5)$正交表

	1	2	3	4	5
1	0	0	0	0	0
2	0	1	1	1	1
3	0	2	2	2	2
4	0	3	3	3	3
5	1	0	1	2	3
6	1	1	0	3	2
7	1	2	3	0	1
8	1	3	2	1	0
9	2	0	2	3	1
10	2	1	3	2	0
11	2	2	0	1	3
12	2	3	1	0	2
13	3	0	3	1	2
14	3	1	2	0	3
15	3	2	1	3	0
16	3	3	0	2	1

用字母替代正交矩阵如表 3-25 所示。

表 3-25 替代后的正交表

	1	2	3	4	5
1	A_1	B_1	C_1	D_1	0
2	A_1	B_2	C_2	D_2	1
3	A_1	B_3	C_3	2	2
4	A_1	B_4	3	3	3
5	A_2	B_1	C_2	2	3
6	A_2	B_2	C_1	3	2
7	A_2	B_3	3	D_1	1
8	A_2	B_4	C_3	D_2	0
9	A_3	B_1	C_3	3	1
10	A_3	B_2	3	2	0
11	A_3	B_3	C_1	D_2	3
12	A_3	B_4	C_2	D_1	2
13	3	B_1	3	D_2	2
14	3	B_2	C_3	D_1	3
15	3	B_3	C_2	3	0
16	3	B_4	C_1	2	1

从表 3-25 中了解到，第一列水平值为 3，第 3 列水平值为 3，第 4 列水平值 3、2 都需要由各自的字母替代。注意，应该按照顺序进行替代，如果不够替代，则继续从头开始替代，如第一列的 4 个“3”分别用 A_1、A_2、A_3 替代，由于没有 A_4，所以，最后一个“3”用 A_1 替代。最后得到表 3-26。

表 3-26　各自替代后的正交表

	1	2	3	4	5
1	A_1	B_1	C_1	D_1	0
2	A_1	B_2	C_2	D_2	1
3	A_1	B_3	C_3	D_1	2
4	A_1	B_4	C_1	D_2	3
5	A_2	B_1	C_2	D_1	3
6	A_2	B_2	C_1	D_2	2
7	A_2	B_3	C_2	D_1	1
8	A_2	B_4	C_3	D_2	0
9	A_3	B_1	C_3	D_2	1
10	A_3	B_2	C_3	D_1	0
11	A_3	B_3	C_1	D_2	3
12	A_3	B_4	C_2	D_1	2
13	A_1	B_1	C_1	D_2	2
14	A_2	B_2	C_3	D_1	3
15	A_3	B_3	C_2	D_2	0
16	A_1	B_4	C_1	D_1	1

第 5 列去掉，因为没有意义。这样就可以设计 16 个测试用例，具体用例(1～12)如表 3-27～表 3-38 所示。

表 3-27　测试用例 1

测试用例编号	PPT_ST_FUNCTION_PRINT_001
测试项目	测试 PowerPoint 打印功能
测试标题	打印 PowerPoint 文件 A 全部的幻灯片，有颜色，加框
重要级别	高
预置条件	PowerPoint 文件 A 已被打开，计算机主机已连接有效打印机
输入	文件 A：D:\系统测试. ppt
操作步骤	1. 打开打印界面； 2. 打印范围选择“全部”； 3. 打印内容选择“幻灯片”； 4. 颜色/灰度选择“颜色”； 5. 选中“幻灯片加框”复选框； 6. 单击“确定”按钮
预期输出	打印出全部幻灯片，有颜色且已加框

表 3-28　测试用例 2

测试用例编号	PPT_ST_ FUNCTION_PRINT_002
测试项目	测试 PowerPoint 打印功能
测试标题	打印 PowerPoint 文件 A 全部的幻灯片为讲义，灰度，不加框

续表

重要级别	中
预置条件	PowerPoint 文件 A 已被打开,计算机主机已连接有效打印机
输入	文件 A:D:\系统测试.ppt
操作步骤	1. 打开打印界面; 2. 打印范围选择“全部”; 3. 打印内容选择“讲义”; 4. 颜色/灰度选择“灰度”; 5. 单击“确定”按钮
预期输出	打印出全部幻灯片为讲义,灰度且不加框

表 3-29 测试用例 3

测试用例编号	PPT_ST_ FUNCTION_PRINT_003
测试项目	测试 PowerPoint 打印功能
测试标题	打印 PowerPoint 文件 A 全部的备注页,黑白,加框
重要级别	中
预置条件	PowerPoint 文件 A 已被打开,计算机主机已连接有效打印机
输入	文件 A:D:\系统测试.ppt
操作步骤	1. 打开打印界面; 2. 打印范围选择“全部”; 3. 打印内容选择“备注页”; 4. 颜色/灰度选择“黑白”; 5. 选中“幻灯片加框”复选框; 6. 单击“确定”按钮
预期输出	打印出全部备注页,黑白且已加框

表 3-30 测试用例 4

测试用例编号	PPT_ST_ FUNCTION_PRINT_004
测试项目	测试 PowerPoint 打印功能
测试标题	打印 PowerPoint 文件 A 全部的大纲视图,黑白
重要级别	中
预置条件	PowerPoint 文件 A 已被打开,计算机主机已连接有效打印机
输入	文件 A:D:\系统测试.ppt
操作步骤	1. 打开打印界面; 2. 打印范围选择“全部”; 3. 打印内容选择“大纲视图”; 4. 颜色/灰度选择“黑白”; 5. 单击“确定”按钮
预期输出	打印出全部大纲视图,黑白

表 3-31 测试用例 5

测试用例编号	PPT_ST_ FUNCTION_PRINT_005
测试项目	测试 PowerPoint 打印功能
测试标题	打印 PowerPoint 文件 A 当前幻灯片,灰度,加框

续表

重要级别	中
预置条件	PowerPoint 文件 A 已被打开，计算机主机已连接有效打印机
输入	文件 A：D:\系统测试. ppt
操作步骤	1. 打开打印界面； 2. 打印范围选择“当前幻灯片”； 3. 打印内容选择“幻灯片”； 4. 颜色/灰度选择“灰度”； 5. 选中“幻灯片加框”复选框； 6. 单击“确定”按钮
预期输出	打印出当前幻灯片，灰度且已加框

表 3-32　测试用例 6

测试用例编号	PPT_ST_ FUNCTION_PRINT_006
测试项目	测试 PowerPoint 打印功能
测试标题	打印 PowerPoint 文件 A 当前幻灯片为讲义，黑白，加框
重要级别	中
预置条件	PowerPoint 文件 A 已被打开，计算机主机已连接有效打印机
输入	文件 A：D:\系统测试. ppt
操作步骤	1. 打开打印界面； 2. 打印范围选择“当前幻灯片”； 3. 打印内容选择“讲义”； 4. 颜色/灰度选择“黑白”； 5. 选中“幻灯片加框”复选框； 6. 单击“确定”按钮
预期输出	打印出当前幻灯片为讲义，黑白且已加框

表 3-33　测试用例 7

测试用例编号	PPT_ST_ FUNCTION_PRINT_007
测试项目	测试 PowerPoint 打印功能
测试标题	打印 PowerPoint 文件 A 当前幻灯片的备注页，有颜色，不加框
重要级别	中
预置条件	PowerPoint 文件 A 已被打开，计算机主机已连接有效打印机
输入	文件 A：D:\系统测试. ppt
操作步骤	1. 打开打印界面； 2. 打印范围选择“当前幻灯片”； 3. 打印内容选择“备注页”； 4. 颜色/灰度选择“颜色”； 5. 单击“确定”按钮
预期输出	打印出当前幻灯片的备注页，有颜色且不加框

表 3-34　测试用例 8

测试用例编号	PPT_ST_ FUNCTION_PRINT_008
测试项目	测试 PowerPoint 打印功能
测试标题	打印 PowerPoint 文件 A 当前幻灯片的大纲视图，有颜色

续表

重要级别	中
预置条件	PowerPoint 文件 A 已被打开,计算机主机已连接有效打印机
输入	文件 A:D:\系统测试.ppt
操作步骤	1. 打开打印界面; 2. 打印范围选择“当前幻灯片”; 3. 打印内容选择“大纲视图”; 4. 颜色/灰度选择“颜色”; 5. 单击“确定”按钮
预期输出	打印出当前幻灯片为讲义,黑白且已加框

表 3-35 测试用例 9

测试用例编号	PPT_ST_ FUNCTION_PRINT_009
测试项目	测试 PowerPoint 打印功能
测试标题	打印 PowerPoint 文件 A 给定范围的幻灯片,黑白,不加框
重要级别	中
预置条件	PowerPoint 文件 A 已被打开,计算机主机已连接有效打印机
输入	文件 A:D:\系统测试.ppt
操作步骤	1. 打开打印界面; 2. 打印范围选择“幻灯片”; 3. 打印内容选择“幻灯片”; 4. 颜色/灰度选择“黑白”; 5. 单击“确定”按钮
预期输出	打印出给定范围的幻灯片,黑白且不加框

表 3-36 测试用例 10

测试用例编号	PPT_ST_ FUNCTION_PRINT_0010
测试项目	测试 PowerPoint 打印功能
测试标题	打印 PowerPoint 文件 A 给定范围的幻灯片为讲义,有颜色,加框
重要级别	中
预置条件	PowerPoint 文件 A 已被打开,计算机主机已连接有效打印机
输入	文件 A:D:\系统测试.ppt
操作步骤	1. 打开打印界面; 2. 打印范围选择“幻灯片”; 3. 打印内容选择“幻灯片”; 4. 颜色/灰度选择“颜色”; 5. 单击“确定”按钮
预期输出	打印出给定范围的幻灯片为讲义,有颜色且加框

表 3-37 测试用例 11

测试用例编号	PPT_ST_ FUNCTION_PRINT_0011
测试项目	测试 PowerPoint 打印功能
测试标题	打印 PowerPoint 文件 A 给定范围的幻灯片的备注页,灰度,加框
重要级别	中

续表

预置条件	PowerPoint 文件 A 已被打开，计算机主机已连接有效打印机
输入	文件 A：D:\系统测试. ppt
操作步骤	1. 打开打印界面； 2. 打印范围选择“幻灯片”； 3. 打印内容选择“备注页”； 4. 颜色/灰度选择“灰度”； 5. 选中“幻灯片加框”复选框； 6. 单击“确定”按钮
预期输出	打印出给定范围的幻灯片的备注页，灰度且加框

表 3-38　测试用例 12

测试用例编号	PPT_ST_ FUNCTION_PRINT_0012
测试项目	测试 PowerPoint 打印功能
测试标题	打印 PowerPoint 文件 A 给定范围的幻灯片的大纲视图，灰度
重要级别	中
预置条件	PowerPoint 文件 A 已被打开，计算机主机已连接有效打印机
输入	文件 A：D:\系统测试. ppt
操作步骤	1. 打开打印界面； 2. 打印范围选择“幻灯片”； 3. 打印内容选择“大纲视图”； 4. 颜色/灰度选择“灰度”； 5. 单击“确定”按钮
预期输出	打印出给定范围的幻灯片的大纲视图，灰度

总而言之，正交实验法在软件测试中是一种有效的方法。例如，在平台参数配置方面，要选择哪种组合方式是最好的，每个参数可能就是一个因素，参数的不同取值就是不同的水平，这样我们可以采用正交实验法设计出最少的测试组合，达到有效的测试目的。又如，图形界面中有多个控件，每个控件均有多种取值情况的测试可以考虑用正交实验法进行测试，这在以上的例子中也有体现。

3.1.8　黑盒测试方法选择的策略

本章中讲到的黑盒测试用例设计方法包括等价类划分法、边界值分析法、错误推测法、因果图法、决策表法、正交实验设计法、场景法等。这些测试用例的设计方法不是单独存在的，具体到每个测试项目里可能会用到多种方法。不同类型的软件有各自的特点，每种测试用例设计的方法也有各自的特点，针对不同软件如何利用这些黑盒方法是非常重要的，在实际测试中，往往是综合使用各种方法才能有效地提高测试效率和测试覆盖度，这就需要认真掌握这些方法的原理，积累更多的测试经验，以有效地提高测试水平。

下面是各种测试方法选择的综合策略，可供读者在实际应用的测试过程中参考。

(1) 首先考虑等价类划分，包括输入条件和输出条件的等价类划分，将无限测试变成有限测试，这是减少工作量和提高测试效率最有效的方法。可以充分利用不同的等价类方法，最好既考虑有效的等价类，也考虑无效的等价类。

(2) 在任何情况下都必须使用边界值分析法。经验表明,用这种方法设计出的测试用例发现软件缺陷的能力最强。但是,边界值分析法没有考虑变量之间的依赖关系,所以,如果被测软件的变量有比较严密的逻辑关系,最好在使用边界值分析法的同时考虑使用决策表和因果图之类的方法。

(3) 可以用错误推测法追加一些测试用例作为补充,这需要依靠测试工程师的智慧和经验。

(4) 如果软件的功能说明中含有输入条件的组合情况,也就是输入变量之间有很强的依赖关系,则一开始就可选用因果图法或决策表法。但是不要忘记用边界值分析法或其他方法设计测试用例作为补充。

(5) 如果被测软件的业务逻辑清晰,同时又是系统级别的测试,那么可以考虑用场景法来设计测试用例。涉及系统级别的测试时,在考虑使用场景法的同时,理解需求规约尤为重要。在分析测试是否达到了所有的功能点覆盖时,功能点的划分要根据具体情况划分得越细越好,但要考虑每个功能的高内聚性和低耦合性。同时,综合考虑使用其他测试方法。

(6) 对于参数配置类的软件,选用正交试验法可以达到测试用例数量少且分布均匀的目的。

3.2 白盒测试技术

3.2.1 白盒测试的概念

在第1章中简述了白盒测试是一种用于检查代码是否按照预期工作的验证技术,又称结构测试(Structural Testing)、逻辑驱动测试(Logic-driven Testing)或基于程序的测试(Program-based Testing)。"白盒"是指可视的,"盒子"是指被测试的软件。所以说白盒测试是一种可视的测试软件的方法,即它把测试对象看作一个透明的盒子,测试人员要了解程序结构和处理过程,按照程序内部逻辑测试程序,检查程序中的每条通路是否按照预定要求正确工作。

白盒测试的主要特点是它主要针对被测程序的源代码,测试者可以完全不考虑程序的功能,所以,如果需求规约中的功能没有实现,那么白盒测试很难发现。白盒测试能解决程序中的逻辑错误和不正确假设。当我们的代码实现了不合理或不正确的条件和控制时,这些错误往往功能性测试很难发现,只有通过白盒测试方法找到问题所在。人为地把一个详细设计或伪代码翻译为某种程序设计语言后,有可能产生某些人为的错误,虽然语法检查机制能够发现很多错误。但是,还有一些错误只有通过动态白盒测试才会被发现。而人为错误在每个逻辑路径上的概率是一样的。

另外一个原因就在于功能测试本身的局限性。简单地说,如果程序实现了需求规约里没有要求的功能,功能测试是无法发现的(病毒就是这样一个例子),这将会给软件带来隐患,而白盒测试能够发现这样的缺陷。正如 Beizer 所说的:"错误潜伏在角落里,聚集在边界上。"相对而言,白盒测试更容易发现它。

白盒测试的测试方法有代码检查法、静态结构分析法、静态质量度量法、逻辑覆盖法、基本路径测试法、域测试、符号测试、Z 路径覆盖、程序变异以及程序控制流分析、数据流分析等。其中多数方法比较成熟,也都有较高的实用价值,个别的方法仍有些问题没有得到圆满的解决。例如,符号测试和基本路径测试的分析方法都是很重要的,但在程序分支过多及程序路径过多时,已有的方法将会显示出它们的局限性。本书主要介绍程序结构分析、逻辑覆盖和程序

插装等技术，而代码检查法、静态结构分析法、静态质量度量法放在静态测试方法中讲解，也可称这些方法为静态白盒测试，在第 2 章已经分析过。其中的域测试、符号测试、Z 路径覆盖、程序变异将在本章中做简单介绍。本节后续提及的白盒测试方法均是指动态白盒测试方法。

白盒测试主要用于单元测试、集成测试及其回归测试，但实际上白盒测试的思想也可以用于系统级别的测试。单元测试就是对一组相关组件或单元的独立测试。对单元进行白盒测试是用来检查单元编码是否正确。而大多数对单元进行的白盒测试是由代码人员自己进行的，也称为自测试(self-testing)，软件公司常常不对其测试过程中所发现的缺陷进行跟踪，也就是不公开单元测试的缺陷。因此，是代码人员自己先找出错误或缺陷，在还没有提交给测试人员之前先修复它。但有些时候，除了代码人员进行的自测试之外，还可能进行专门的单元测试，这主要视项目的实际情况而定。集成测试是对集成到一起的软件组件和硬件组件进行的测试，用于评估这些组件之间能否进行正确的交互。集成测试的主要目的就是检查各种组件之间的接口，测试员可以通过白盒测试来检查各个单元接口，也就是将各个组件通过接口连在一起。由于回归测试可以用于不同的测试层次，是一种具有选择性的对系统或组件的重复测试，用来验证对软件所做的修改是否带来不良的影响、系统或组件是否仍然符合特定的需求，因此在应用白盒测试方法进行单元测试和集成测试时，回归测试一样适用。在实际的测试过程中，对测试用例文档也必须做配置管理，即对白盒的单元测试和集成测试用例进行版本控制，为后续的回归测试带来方便，因为回归测试往往只执行以前测试的部分测试用例。

在白盒测试过程中，程序员通常要开发桩模块和驱动模块来配合白盒测试的进行。驱动模块就是用于触发被测模块的一个软件模块，一般要提供测试输入、控制和监测并报告测试结果。最简单的形式就是使用一行能够调用一个方法或模块的代码。例如，如果想移动游戏中的一个人物，驱动代码就可能如下：

```
movePerson(Person,diceRoll);
```

这个驱动代码就可能被主方法调用。而白盒测试用例将要执行这行驱动代码并且检查人物的位置(如使用 person.getPosition()方法)，以确定人物现在所处的位置。桩模块就是能够代替被测模块所调用的软件模块的程序，可以模拟实际组件行为的组件或对象。例如，若 movePerson()方法还没有完成，那么就可以暂时使用下面所示的代码，把人物移动到标识为 1 的位置。

```
Public void movePerson(Person Person,int diceValue){
Person.setPosition(1);}
```

以上驱动方法 movePerson()是被主方法调用的，但是在实际的测试中将驱动方法 movePerson()写成主方法也是可行的，直接运行这个主方法调用被测的方法。当然，最后要由正确的程序逻辑来代替这个方法。但是，开发桩模块的程序员可以调用正在开发的代码中的方法，甚至是一个还没有规定预期行为的方法。桩模块和驱动模块通常被看作是随时可以抛弃的代码，但是这些代码往往要被充分使用，因此最好对桩代码和驱动代码进行配置管理。

3.2.2 程序结构分析

程序的结构形式是白盒测试的主要依据。下面将从控制流分析、数据流分析和信息流分析的不同方面讨论几种机械性的方法并分析程序结构。我们的目的总是要找到程序中隐藏的各种错误或缺陷。

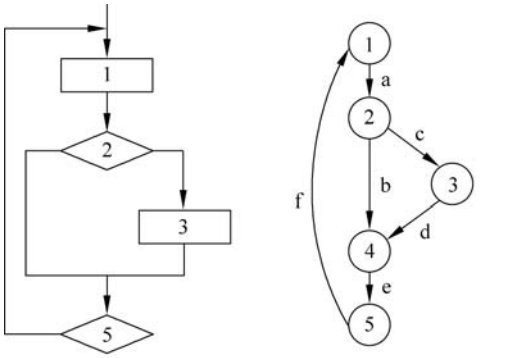
1. 控制流分析

由于非结构化程序会给测试、排错和程序的维护带来许多不必要的困难,人们有理由要求写出的程序是结构良好的。自 20 世纪 70 年代以来,结构化程序的概念逐渐被人们普遍接受。体现这一要求对于若干语言,如 Pascal、C 等并不困难,因为它们都具有反映基本控制结构的相应控制语句。但对于早期开发的语言来说,做到这一点,程序编写人员需要特别注意,不应忽视程序结构化的要求。使用汇编语言编写程序,要注意这个问题的道理就更为明显了。正是由于这个原因,系统地检查程序的控制结构成为十分有意义的工作。

(1) 控制流图(程序图)。

程序流程图(Flow Chart)又称框图,也许是人们最熟悉、最容易接受的一种程序控制结构的图形表示。在这种图的框内常常标明了处理要求或条件,这些在做路径分析时是不重要的。为了更加突出控制流的结构,需要对程序流程图做些简化。在图 3-13 中给出了简化的例子。其中图 3-13(a)是一个含有两出口判断和循环的程序流程图,把它简化成图 3-13(b)的形式,称这种简化了的流程图为控制流图(Control-flow Graph)或程序图。在控制流图中只有两种图形符号,它们是:

- 结点:以标有编号的圆圈表示。它代表了程序流程图中矩形框所表示的处理、两个或多出口判断以及两条或多条流线相交的汇合点。
- 控制流线或弧:以箭头表示。它与程序流程图中的流线是一致的,表明了控制的顺序。为讨论方便,控制流线通常标有名字,如图 3-13 中所标的 a、b、c 等。为便于在机器上表示和处理控制流图,可以把它表示成矩阵的形式,称为控制流图矩阵(Control-flow Graph Matrix)。图 3-14 表示了图 3-13 的控制流图矩阵。这个矩阵有 5 行 5 列,是由该控制图中含有的 5 个结点决定的。矩阵中 6 个元素 a、b、c、d、e 和 f 的位置决定于它们所连接结点的号码。例如,弧 d 在矩阵中处于第 3 行第 4 列,那是因为它在控制流图中连接了结点 3 至结点 4。这里必须注意方向。图 3-13(b)中结点 4 至结点 3 是没有弧的,因此矩阵中第 4 行第 3 列也就没有元素。



(a) 程序流程图　　(b) 控制流图(程序图)

图 3-13　程序流程图和控制流图

	a			
		c	b	
			d	
				e
f				

图 3-14　控制流图矩阵

路径测试法的基本依据就是程序图,程序图是有向图。程序图由结点(其中程序图的开始结点称为源结点,结束结点称为汇结点)和边组成,结点表示语句或语句片段,边表示控制流,如 if 语句对应的程序图中,结点表示语句片段,即多个语句片段构成一个完整的语句。再如,赋值语句是一个完整的语句构成一个程序图结点。对边的理解是:如果 i 和 j 是程序图中的

结点，从结点 i 到结点 j 存在一条边，当且仅当对应结点 j 的语句片段或语句可以在对应结点 i 的语句片段或语句之后立即执行。

程序图中基本的控制结构对应的图形符号如图 3-15 所示。

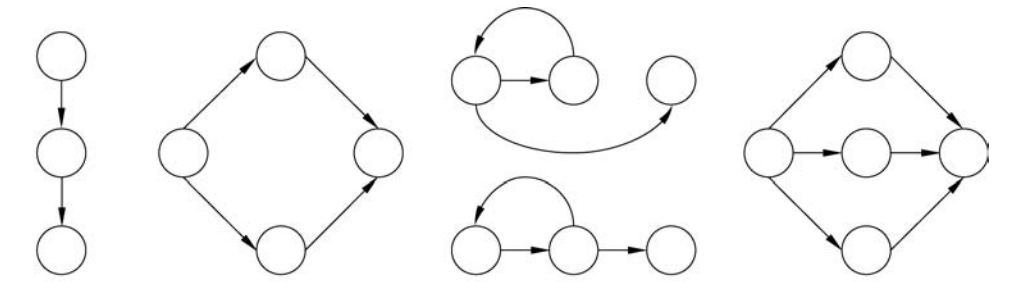


图 3-15　程序图的图形符号

如果判断中的条件表达式是复合条件，即条件表达式是由一个或多个逻辑运算符（or、and 和 not）连接的逻辑表达式，则需要改变复合条件的判断为一系列只有单个条件的嵌套的判断。例如，对应图 3-16 所示的复合逻辑下的图 3-16(a)所示程序逻辑的复合条件的判定，应该画成图 3-16(b)所示的程序图。条件语句 if a and b 中条件 a 和条件 b 各有一个只有单个条件的判断结点。

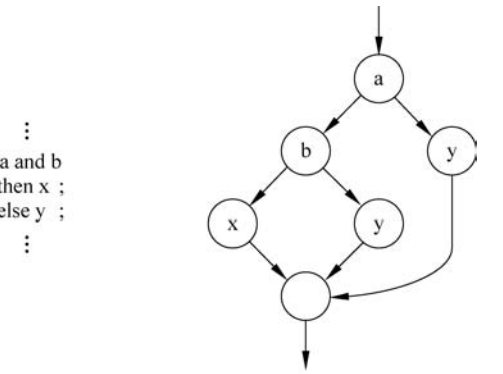


图 3-16　复合逻辑的程序图

下面是一个判断三角形类型的伪代码程序，以此为例来构造程序图。

```
1. Program triangle2 'Structured programming version of simpler specification
2. Dim a,b,c As Integer
3. Dim IsATriangle As Boolean
       'Step 1: Get Input
4. Output("Enter 3 integers which are sides of a triangle")
5. Input(a,b,c)
6. Output("Side A is ",a)
7. Output("Side B is ",b)
8. Output("Side C is ",c)
       'Step 2: Is A Triangle?
9. if (a < b + c) and (b < a + c) and (c < a + b)
10.     then IsATriangle = True
11.     else IsATriangle = False
12 endif
       'Step 3: Determine Triangle Type
13. if IsATriangle
14.     then    if (a = b) and (b = c)
15.                 then Output ("Equilateral")
16.                 else    if (a ≠ b) and (a ≠ c) and (b ≠ c)
17.                             then    Output ("Scalene")
18.                             else    Output ("Isosceles")
19.                         endif
20.             endif
21.     else    Output("Not a Triangle")
22. endif
23. end triangle2
```

对应的程序图如图 3-17 所示。在程序图中为了表示方便,对应的判断条件简化表示如下:a < b+c→a1; b < a+c→b1; c < a+b→c1; a=b→d1; b=c→e1; a≠b→f1; a≠c→g1; b≠c→h1。

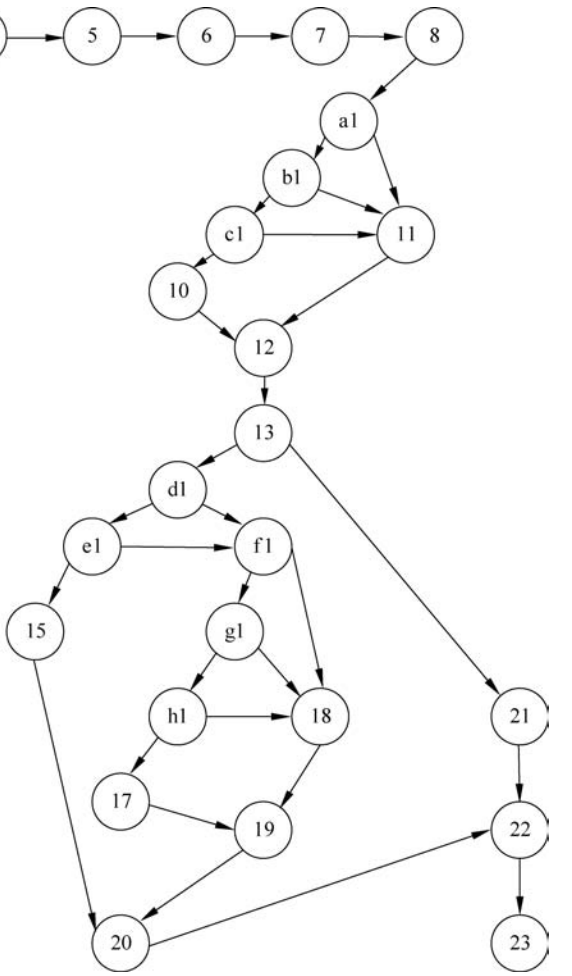


图 3-17　判断三角形类型的程序图

(2) 程序结构的基本要求。

对于程序结构提出以下 4 点基本要求,这些要求是写出的程序不应包含:

- 转向并不存在的标号;
- 有用的语句标号;
- 从程序入口进入后无法达到的语句;
- 不能达到程序出口语句的语句。

显然,提出这些要求是合理的。在编写程序时稍加注意,能做到这几点也是很容易的。这里我们更为关心的是如何进行检测,把以上 4 种问题从程序中找出来。目前对这 4 种情况的检测主要通过编译器和程序分析工具来实现。

2. 数据流分析

数据流分析最初是随着编译系统要生成有效的目标码而出现的,这类方法主要用于代码优化。近年来数据流分析方法在软件测试中也得到成功的运用,用以查找如引用未定义变量

等程序错误，也可用来查找对以前未曾使用的变量再次赋值等数据流异常的情况。找出这些错误是很重要的，因为这常常是常见程序错误的表现形式，如错拼名字、名字混淆或者丢失了语句。这里将首先说明数据流分析的原理，然后指明它可揭示的程序错误。

(1) 数据流问题。

如果程序中某一语句执行时能改变某程序变量 V 的值，则称 V 是被该语句定义的。如果某一条语句的执行引用了内存中变量 V 的值，则说该语句引用变量 V。例如，语句：

```
X: = Y + Z
```

定义了 X，引用了 Y 和 Z，而语句：

```
if Y > Z then goto exit
```

只引用了 Y 和 Z。输入语句：

```
READ X
```

定义了 X。输出语句：

```
WRITE X
```

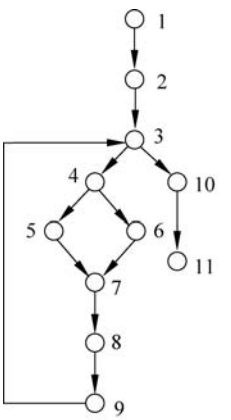
引用了 X。执行某个语句也可能使变量失去定义，成为无意义的量。例如，循环语句的控制变量在经循环的正常出口离开循环时，就变成无意义的了。

图 3-18 给出了一个小程序的控制流图，图中结点号就是语句编号，同时指明了每一个语句定义和引用的变量。可以看出，第一个语句定义了 3 个变量 X、Y 和 Z，这表明它们的值是程序外赋给的。例如，该程序以这 3 个变量为输入参数的过程或子程序。同样，出口语句引用 Z 表明，Z 的值被送给外部环境。该程序中含有两个错误：

- 语句 2 使用了变量 W，而在此之前并未对其定义。
- 语句 5、6 使用变量 V，这在第一次执行循环时也未对其定义过。

此外，该程序还包含两个异常：

- 语句 6 对 Z 的定义从未使用过。
- 语句 8 对 W 的定义也从未使用过。



节点	被定义变量	被引用变量
1	X, Y, Z	
2	X	W, X
3		X, Y
4		Y, Z
5	Y	V, Y
6	Z	V, Z
7	V	X
8	W	Y
9	Z	V
10	Z	Z
11		Z

图 3-18　控制流图及其定义和引用的变量

当然,程序中包含有些异常,如程序中含有错误;也许可以把程序写得更容易理解,从而能够简化验证工作以及随后的维护工作(去掉那些多余的语句一般会缩短执行时间,不过在此并不关心这些)。目前通过编译器或程序分析工具,通过数据流分析可以查找出对未定义变量的使用和未曾使用的定义。

(2) 数据流分析应用的其他方面。

在优化的编译系统中,数据流分析除了用于以前已说明的以外,还用于多种目的。一个常数传播的例子是:如果变量 V 的所有定义(该定义达到引用 V 的一个特定语句)都把同一已知常数赋给该变更,对 V 的引用便可用这一常数所代替。这里是一个普通的例子。程序段:

```
a: = 4
b: = a +
…
c: = 3 * (a + b)
```

可以用下列程序段代替:

```
a: = 4
b: = 5
…
c: = 27
```

常数传播除了能节省执行时间外,还能提高程序的清晰度,确认系统可以表明进行这种修改的可能性。另一个例子是找出循环内的不变定义。这种定义并不引用其值在执行循环时可改变的任何变量。在优化的编译系统中查找不变定义是很重要的,因为它可使得将这一定义从循环中移出,放在循环前面或者放在循环后面,从而减少它的执行次数。在程序确认中,我们也对不变定义感兴趣,因为它会提醒用户注意粗心的程序设计。

3. 信息流分析

直到目前,信息流分析主要用在验证程序变量间信息的传输遵循的保密要求。然而,近来发现可以导出程序的信息流关系,这就为软件开发和确认提供了十分有益的工具。为了说明信息流分析的性质,下面以整除算法作为例子。图 3-19 是这一算法的程序。图 3-20 是 3 个关系的表。其中第 1 个关系(如图 3-20(a)所示)给出每一条语句执行时所用到的其输入值的变量。例如,从图 3-19 的算法很明显地看出,M 的输入值在语句 2 中得到直接使用,由于这一条语句将 M 的值传送给 R,M 的初始值也间接地用于语句 3 和语句 5。而且,语句 3 中表达式 R >=N 的值决定了语句 4 的重复执行次数,即对 Q 多少次重复赋值,就是说 M 的值也间接地用于语句 4。

```
语句号      begin
  1           Q: =0;
  2           R: =M;
  3           while R>=N do
                begin
  4               Q: = Q+1;
  5               R: =R-N
                end
            end
```

图 3-19　整除算法

	M	N
1		
2	√	
3	√	√
4	√	√
5	√	√

(a)第1个关系

	Q	R
1	√	
2	√	√
3	√	√
4	√	
5	√	√

(b)第2个关系

	Q	R
M	√	√
N	√	√

(c)第3个关系

图 3-20　整除算法中输入值、语句与输出值的关系

第 2 个关系(如图 3-20(b)所示)给出了其执行可能直接或间接影响输出变量终值的一些语句。可以看出,所有语句都可能影响到商 Q 的值。而语句 1 和语句 4 并未关系到余数 R 的值。最后的关系表明了哪个输入值可能直接或间接地影响到输出值。针对结构良好的程序快速算法(只需多项式时间)已经开发出来,可用以建立这些关系,这在程序的测试中是非常有用的。例如,第 1 个关系能够表明对未定义变量的所有可能的引用。第 2 个关系在查找错误中也是有用的,比如假定某个变量的计算值在使用以前被错误地改写了,这可能是因为有并不影响任何输出值的语句被发现。在程序的任何指定点查出其执行可能影响某一变量值的语句,这在程序排错和程序验证中都是很有用的。第 3 个输入输出关系还提供一种检查,看看每个输出值是否由相关的输入值而不是其他值导出。

3.2.3 逻辑覆盖测试法

结构测试是依据被测程序的逻辑结构设计测试用例,驱动被测程序运行完成测试。结构测试中的一个重要问题是测试进行到什么地步就达到要求,可以结束测试了。这就是说需要给出结构测试的覆盖准则。下面给出的几种逻辑覆盖测试方法都是从各自不同的方面出发,为设计测试用例提出依据的。为方便讨论,将结合一个小程序段加以说明:

```
if (( A > 1) and ( B = 0 ))then
      X = X/A
if (( A = 2) or ( X > 1 )) then
      X = X + 1
```

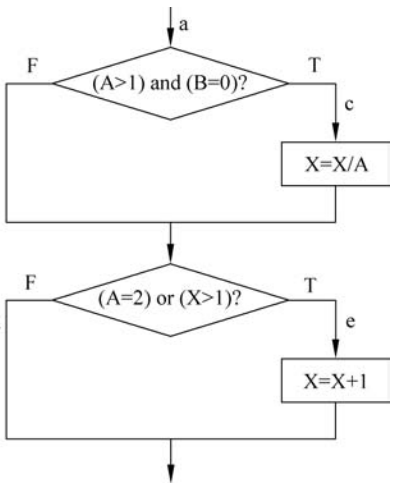


图 3-21 被测程序段流程图

其中,and 和 or 是两个逻辑运算符。图 3-21 给出了它的流程图,a、b、c、d 和 e 是控制流上的若干程序点。

1. 语句覆盖

语句覆盖的含义是在测试时,首先设计若干个测试用例,然后运行被测程序,使程序中的每个可执行语句至少执行一次。这里所谓"若干个",自然是越少越好。在上述程序段中,如果选用的测试用例是:

```
A = 2
B = 0
X = 3  ……case1
```

则程序按路径 ace 执行。这样,该程序段的 4 个语句均得到执行,从而做到了语句覆盖。但如果选用的测试用例是:

```
A = 2
B = 1
X = 3  ……case2
```

程序按路径 abe 执行,便未能达到语句覆盖。

从程序中每个语句都得到执行这一点来看,语句覆盖的方法似乎能够比较全面地检验每一个语句。但它也绝不是完美无缺的。假如这一程序段中两个判断的逻辑运算都有问题,例如,第一个判断的运算符 and 错成运算符 or 或是第二个判断中的运算符 or 错成了运算符

and。这时仍使用上述前一个测试用例 case1,程序仍将按路径 ace 执行。这说明虽然也做到了语句覆盖,却发现不了判断中逻辑运算的错误。此外,还可以很容易地找出已经满足了语句覆盖,却仍然存在错误的例子。如有一程序段:

```
if(I≥ 0)
then I = J
```

如果错写成:

```
if(I > 0)
then I = J
```

假定给出的测试数据执行该程序段时,I 的值大于 0,则 I 被赋予 J 的值,这样虽然做到了语句覆盖,但是却掩盖了其中的错误。

实际上,和后面介绍的其他几种逻辑覆盖比较起来,语句覆盖是比较弱的覆盖原则。做到了语句覆盖可能给人们一种心理的满足,以为每个语句都经历过,似乎可以放心了。其实这仍然是不十分可靠的。语句覆盖在测试被测程序中,除去对检查不可执行语句有一定作用外,并没有排除被测程序包含错误的风险。必须看到,被测程序并非语句的无序堆积,语句之间的确存在着许多有机的联系。

2. 判定(判断)覆盖

按判定覆盖准则进行测试是指设计若干测试用例,运行被测程序,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,即判断的真假值均被满足。判定覆盖又称为分支覆盖或判断覆盖。仍以上述程序段为例,若选用的两组测试用例是:

```
A = 2
B = 0
X = 3 ……case1

A = 1
B = 0
X = 1 ……case3
```

则可分别执行路径 ace 和 abd,从而使两个判断的 4 个分支 c、e 和 b、d 分别得到覆盖。当然,也可以选用另外两组测试用例:

```
A = 3
B = 0
X = 3 ……case4

A = 2
B = 1
X = 1 ……case5
```

则可分别执行路径 acd 及 abe,同样也可覆盖 4 个分支。上述两组测试用例不仅满足了判定覆盖,同时还做到语句覆盖。从这一点看,似乎判定覆盖比语句覆盖更强一些。假如此程序段中的第二个判断条件 X > 1 错写成 X < 1,使用上述测试用例 case5,照样能按原路径(abe)执行,而不影响结果。这个事实说明,只做到判定覆盖仍无法确定判断内部条件的错误。因此,需要有更强的逻辑覆盖准则去检验判断内的条件。以上仅考虑了两出口的判断,还应把判定覆盖

准则扩充到多出口判断(如 case 语句)的情况。

3. 条件覆盖

条件覆盖是指设计若干测试用例,执行被测程序以后,要使每个判断中每个条件的可能取值都至少满足一次。在上述程序段中,第一个判断应考虑到:

A>1,取真值,记为 T1;

A>1,取假值,即 A≤1,记为 F1;

B=0,取真值,记为 T2;

B=0,取假值,即 B≠0,记为 F2。

第二个判断应考虑到:

A=2,取真值,记为 T3;

A=2,取假值,即 A≠2,记为 F3;

X>1,取真值,记为 T4;

X>1,取假值,即 X≤1,记为 F4。

给出 3 个测试用例:case6、case7 和 case8,执行该程序段所走路径及覆盖条件如表 3-39 所示。

表 3-39 case6～case8 条件覆盖结果

测试用例	A B X	所走路径	覆盖条件
case6	2 0 3	a c e	T1,T2,T3,T4
case7	1 0 1	a b d	F1,T2,F3,F4
case8	2 1 1	a b e	T1,F2,T3,F4

从表 3-39 中可以看到,3 个测试用例把 4 个条件的 8 种情况均做了覆盖。进一步分析表 3-39,覆盖了 4 个条件的 8 种情况的同时,把两个判断的 4 个分支 b、c、d 和 e 似乎也覆盖了。这样是否可以说做到了条件覆盖,也就必然实现了判定覆盖呢?来分析另一种情况,假定选用两个测试用例是 case8 和 case9,执行该程序段的覆盖情况如表 3-40 所示。

表 3-40 case8 和 case9 条件覆盖结果

测试用例	A B X	所走路径	覆盖分支	覆盖条件
case9	1 0 3	a b e	b e	F1,T2,F3,T4
case8	2 1 1	a b e	b e	T1,F2,T3,F4

这一覆盖情况表明,覆盖了条件的测试用例不一定覆盖了分支。事实上,它只覆盖了 4 个分支中的两个。为解决这一矛盾,需要对条件和分支兼顾。

4. 判定-条件覆盖

判定-条件覆盖要求设计足够的测试用例,使得判断中每个条件的所有可能至少满足一次,并且每个判断本身的判定结果也至少出现一次。根据判定-条件覆盖的定义,只需设计下面两个测试用例便可覆盖例子的 8 个条件取值以及 4 个判断分支,执行程序段的覆盖情况如表 3-41 所示。

表 3-41 case8 和 case10 的判定-条件覆盖结果

测试用例	A B X	所走路径	覆盖分支	覆盖条件
case10	2 0 4	a c e	c e	T1,T2,T3,T4
case8	2 1 1	a b e	b e	F1,F2,F3,F4

判断-条件覆盖的不足之处：表面上看来，判断-条件覆盖测试了所有条件的取值，但实际上并非如此，而是某些条件掩盖了另一些条件(由于多重条件判定)。例如，对条件表达式(A＞1)and(B＝0)来说，若(A＞1)的测试结果为 false，可以立即确定表达式的结果为 false，这时往往就不再测试(B＝0)的取值了，因此，条件(B＝0)就没有被检查。同样，对条件表达式(A＝2)or(x＞1)来说，若(A＝2)的测试结果为 true，就立即确定表达式的结果为 true，这时，条件(x＞1)就没有被检查。因此，采用判断-条件覆盖测试，逻辑表达式中的错误不一定能够查得出来。

5. 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都至少执行一次。在本例子中两个判断各包含两个条件，这 4 个条件在两个判断中可能有 8 种组合，它们是：

(1) A＞1，B＝0，记为 T1，T2；

(2) A＞1，B≠0，记为 T1，F2；

(3) A≤1，B＝0，记为 F1，T2；

(4) A≤1，B≠0，记为 F1，F2；

(5) A＝2，X＞1，记为 T3，T4；

(6) A＝2，X≤1，记为 T3，F4；

(7) A≠2，X＞1，记为 F3，T4；

(8) A≠2，X≤1，记为 F3，F4。

这里设计了 4 个测试用例，用以覆盖上述 8 种条件组合，具体如表 3-42 所示。

表 3-42　条件组合覆盖结果

测试用例	A B X	覆盖组合号	所走路径	覆盖条件
case1	2 0 3	①⑤	a c e	T1，T2，T3，T4
case8	2 1 1	②⑥	a b c	T1，F2，T3，F4
case9	1 0 3	③⑦	a b e	F1，T2，F3，T4
case11	1 1 1	④⑧	a b d	F1，F2，F3，F4

注意，这一程序段共有 4 条路径。以上 4 个测试用例固然覆盖了条件组合，同时也覆盖了 4 个分支，但仅覆盖了 3 条路径，却漏掉了路径 acd。前面讨论的多种覆盖准则，有的虽提到了所走路径问题，但尚未涉及路径的覆盖，而路径能否全面覆盖在软件测试中是一个重要问题，因为程序要取得正确的结果，就必须消除遇到的各种障碍，沿着特定的路径顺利执行。如果程序中的每一条路径都得到考验，才能说程序受到了全面检验。

6. 路径覆盖

按路径覆盖要求进行测试是指设计足够多的测试用例，要求覆盖程序中所有可能的路径。针对例子中的 4 条可能路径，有：

ace，记为 L1；

abd，记为 L2；

abe，记为 L3；

acd，记为 L4。

给出 4 个测试用例：case1、case7、case8 和 case12，使其分别覆盖这 4 条路径，如表 3-43 所示。

表 3-43　路基覆盖结果

测试用例	A B X	覆盖路径
case1	2 0 3	a c e (L1)
case7	1 0 1	a b d (L2)
case8	2 1 1	a b e (L3)
case12	3 0 1	a c d (L4)

这里所讨论的程序段非常简短，也只有 4 条路径。但在实际问题中，一个不太复杂的程序，其路径数都是一个庞大的数字，要在测试中覆盖这样多的路径是无法实现的。为解决这一难题，只得把覆盖的路径数压缩到一定限度内，例如，程序中的循环体只执行有限次。其实，即使对于路径数很有限的程序已经做到了路径覆盖，仍然不能保证被测程序的正确性。例如，在上述语句覆盖的描述中最后给出的程序段中出现的错误也不是路径覆盖可以发现的。由此看出，各种结构测试方法都不能保证程序的正确性。这一残酷的事实对热心测试的程序人员似乎是一个沉重的打击。但要记住，测试的目的并非要证明程序的正确性，而是要尽可能找出程序中的错误或缺陷。确实并不存在一种十全十美的测试方法，能够发现所有的错误或缺陷。想要撒下几网把湖中的鱼全都捕上来是做不到的，软件测试是有局限性的。

下面重点讨论当路径数庞大或路径数接近无穷的情况下如何有效、合理地压缩路径数量，以达到基路径覆盖。

(1) 基路径算法讨论。

重新分析图 1-2 所示的程序图，根据我们的分析，对于这样一个带有循环的程序图如果要穷举测试程序，遍历所有可执行路径需要设计 $5^{20}+5^{19}+5^{18}+\cdots+5^{1}+5^{0}$ 个测试用例，这是一个非常庞大的数值，在实际的测试过程中不可能做到。为了解决这个难题，需要把测试用例覆盖的路径数压缩到一定限度内，并且被覆盖的这些路径应该具有代表性。

先回顾向量空间的概念，向量空间的概念是解决这个问题的理论基础。假设有一个向量空间 $V_n=\{x_1,x_2,x_3,\cdots,x_i,\cdots\}$，其中向量空间中的向量 x_i 为 n 维。假设向量空间 V_n 有一个基：$y_1,y_2,\cdots,y_i,\cdots,y_p$（$p$ 为基中向量的个数，向量 y_i 对应于向量空间 V_n 中的某个向量），向量空间 V_n 的基必须满足：$y_1,y_2,\cdots,y_i,\cdots,y_p$ 之间是互相独立的；$y_1,y_2,\cdots,y_i,\cdots,y_p$ 之间不能互相线性表示；V_n 中的任何一个向量均能由这个基中的向量来线性表示，例如 $x_3=y_1+0*y_2+0*y_3+\cdots+5y_2-y_p$ 就是一种线性表示。向量空间的基不唯一。

从以上分析中能得出如下结论：假设把程序图中的每个可执行路径都看成向量空间的一个向量，程序图中所有的可执行路径将构成一个向量空间，向量空间中的向量可以是无穷多。如果能从由所有可执行路径构成的向量空间中找到该向量空间的一个基，那么程序图的所有可执行路径将能由这个基中的向量来线性表示。因此，设计测试用例时只要将这个向量空间的基中的每个向量所对应的路径转换成测试用例就可以了，因为所有其他可执行路径均能由向量空间的这个基中的向量对应的路径来线性表示。完成这样的测试设计称为基路径测试，其结果达到了基路径覆盖，也叫 McCabe 覆盖。

现在要解决的问题是：

- 基路径中的独立路径个数，也就是向量空间的基中的向量个数。
- 基路径中的路径如何得到，也就是如何得到向量空间的一个基。

解决第一个问题的办法是根据程序图的环路复杂性（也称为程序图的独立路径数）公式来计算：

$$V_G = e - n + 2$$

其中,e 为程序图的边数,n 为程序图的结点数。V_G 就是基路径中的独立的路径个数。

解决第二个问题的办法是执行寻找基路径的算法,算法描述如下:

① 根据程序图,利用公式 $V_G = e - n + 2$ 得到独立路径数。

② 根据程序图找到一条基线路径,这条基线路径不唯一。基线路径应该满足:从源结点开始到汇结点结束的路径;尽量长;尽量多经过出度大于或等于 2 的结点;基线路径对应的业务最好是执行正常的业务流。

③ 以这条基线路径为基础,从这条基线路径的第一个结点开始,从头往后搜索以找到第一个出度大于或等于 2 的结点(包括第一个结点本身),以这个结点为轴进行旋转,将这个结点在基线路径中的儿子结点和不在基线路径中的其他儿子结点互换,如果不在基线路径中的其他儿子结点数大于一个,则任意选择一个即可,同时,尽量多地保留基线路径中的其他结点不变(称为回溯),这样就得到了另外一条路径。

④ 一般情况下仍然是以基线路径为基础,从刚才找到的出度大于或等于的结点开始往后继续寻找,执行与③同样的流程得到其他路径。

⑤ 如果基线路径中的出度大于或等于 2 的结点全部找到,并根据③的流程得到了其他所有可能的路径,这时路径总数仍然没有达到 V_G 的值,则可以把以基线路径旋转得到的路径看成另外一个基线路径,同样执行③一直到得到的路径数量等于 V_G 为止。

通过以上算法得到的这组路径就是基路径。由于基线路径的选择不同、旋转结点的选择不同均可以影响到基路径,因此基路径不唯一。

值得注意的是,通过以上算法得到的基路径是以程序图为基础的,并没有将程序图和其对应的业务逻辑进行关联。实际上很多时候得到了基路径之后会发现基路径中有些路径不符合业务逻辑,也就是说将这些路径和业务逻辑进行关联的时候,这些路径是不可能执行的。在这种情况下,有两种不同的处理方法:

- 找到和业务逻辑进行关联不可行的路径,在这个路径中找到出度大于或等于 2 的结点按照以上③进行旋转得到另外的路径,直到得到的这个路径符合业务逻辑为止。
- 找到和业务逻辑进行关联不可行的路径,通过人工干预得到一条符合业务逻辑的路径。

(2) 基路径测试的实际应用。

首先总结基路径测试法的步骤:

① 以详细设计或源代码作为基础,导出程序图。

② 计算得到程序图的独立路径数或环路复杂性 V_G。

③ 根据程序图分析出一条合理的基线路径。

④ 依据算法得到除基线路径以外的其他独立路径,并使总路径数满足 V_G,即得到一组基路径。

⑤ 依据这组基路径设计对应的测试用例,并评审。

例子一:图 3-22 所示是一个程序图,根据基路径算法得出一组基路径。

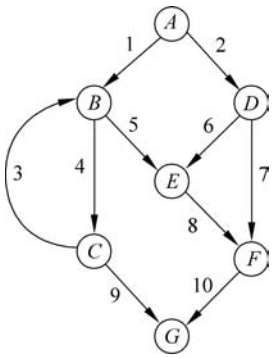


图 3-22 例子程序图

首先,依据 $V_G = e - n + 2V_G = 10 - 7 + 2 = 5$,基路径中应该包含 5 条独立的路径。根据必须确定一条基线路径的原则,依据基路径的算法选择基线路径 $A \rightarrow B \rightarrow C \rightarrow B \rightarrow E \rightarrow F \rightarrow G$。这条基线路

径中的第 1 个结点 A 就是出度等于 2 的结点。以结点 A 为轴进行旋转，以 D 来替换 A，基线路径中的其他结点尽量多保留，这样得到了第 2 条路径 $A\rightarrow D\rightarrow E\rightarrow F\rightarrow G$。再以基线路径中的结点 B 为轴旋转，用结点 E 来替换结点 C，同样，基线路径中的其他结点尽量多保留，这样得到第 3 条路径 $A\rightarrow B\rightarrow E\rightarrow F\rightarrow G$。再搜索基线路径发现第 3 个结点 C 也是出度等于 2 的结点，以 C 结点为轴旋转，用 G 来替换 B，得到第 4 条路径 $A\rightarrow B\rightarrow C\rightarrow G$。再往后搜索基线路径发现结点 B 已经被旋转过，继续往后搜索找不到没有旋转过的出度大于或等于 2 的结点了，这时，可以在刚才旋转得到的其他路径中找出度大于或等于 2 的结点，这里可以在第 2 条路径 $A\rightarrow D\rightarrow E\rightarrow F\rightarrow G$ 中搜索，发现结点 D 没被旋转过，同时该结点的出度为 2，以 D 为轴进行旋转，用结点 F 替换结点 E，得到路径 $A\rightarrow D\rightarrow F\rightarrow G$。这样就得到了 5 条路径，这 5 条路径是独立的，构成了图 3-22 所示程序图的一组基路径：

- $A\rightarrow B\rightarrow C\rightarrow B\rightarrow E\rightarrow F\rightarrow G$；
- $A\rightarrow D\rightarrow E\rightarrow F\rightarrow G$；
- $A\rightarrow B\rightarrow E\rightarrow F\rightarrow G$；
- $A\rightarrow B\rightarrow C\rightarrow G$；
- $A\rightarrow D\rightarrow E\rightarrow F\rightarrow G$。

最后，结合程序图所对应的业务逻辑，根据这 5 条路径设计出测试用例。由于图 3-22 所示的程序图没有和具体的业务绑定，这里不再考虑。

例子二：以一个求平均值的过程 averagy 为例，说明基路径法设计测试用例的过程。用伪代码语言描述的 averagy 过程如下所示。

```
PROCEDURE averagy;
* This procedure computes the average of 100 or fewer numbers that lie bounding values; it also
  computes the total input and the total valid.
  INTERFACE RETURNS averagy, total.input, total.valid;
  INTERFACE ACCEPTS value, minimum, maximum;
  TYPE value[1:100] IS SCALAR ARRAY;
  TYPE averagy, total.input, total.valid, minimum, maximum, sum IS SCALAR;
  TYPE i IS INTEGER;
  i = 1;
  total.input = total.valid = 0;
  sum = 0;
  do while value[i] <> -999 and total.input < 100
     increment total.input by 1;
        if value[i] >= minimum and value[i] <= maximum
           then increment total.valid by 1;
                sum = sum + value[i];
           else skip;
        endif;
     increment i by 1;
  enddo
  if total.valid > 0
     then averagy = sum/total.valid;
     else averagy = -999;
  endif
end avergy
```

以详细设计或源代码作为基础，导出程序的程序图。

依据前面程序图的分析描述标出 1～13 号结点，如图 3-23 所示，并结合本程序画出程序图，如图 3-24 所示。

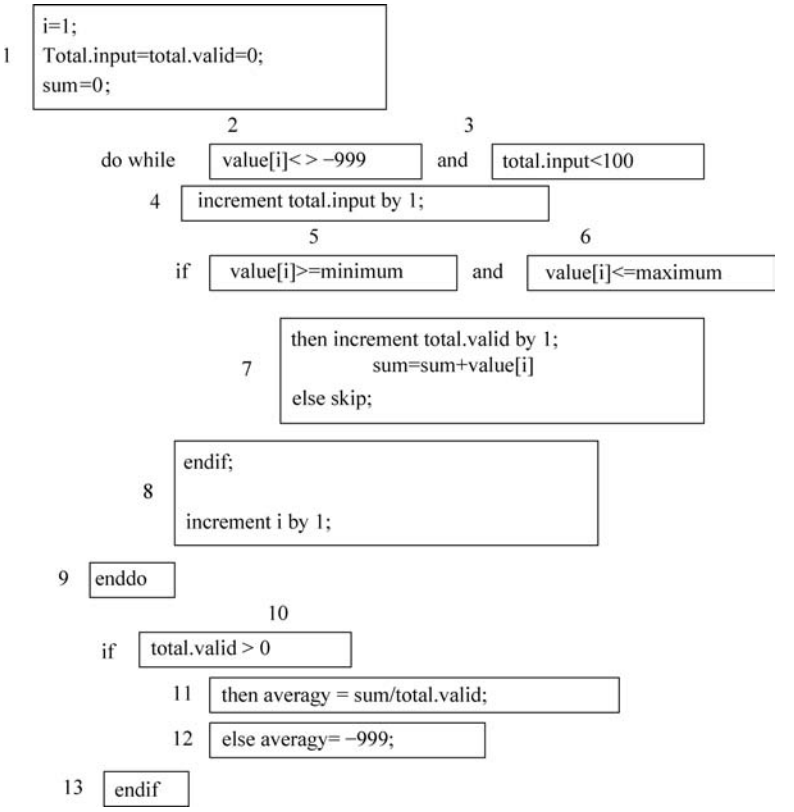


图 3-23　程序 averagy 的结点定义

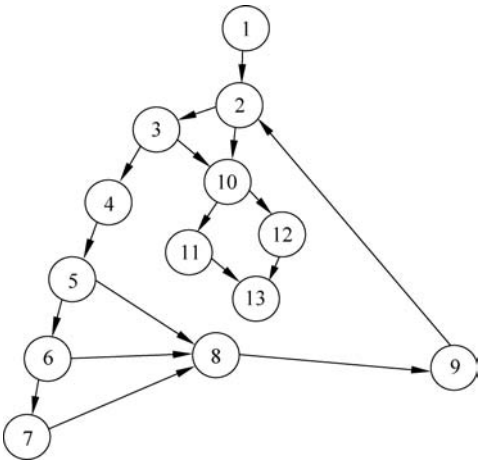


图 3-24　averagy 过程的程序图

计算程序图的独立路径数或环路复杂性 V_G，利用前面给出的计算程序图的独立路径数或环路复杂性的方法，可以算出 $V_G=17-13+2=6$。再根据算法确定一组独立的基路径，基路径共有 6 条独立的路径组成。针对图 3-24 所示的 averagy 过程的程序图选择一条基线路径：1→2→3→4→5→6→7→8→9→2→10→12→13。再根据算法得到如下 5 条路径：

- 1→2→10→12→13；
- 1→2→3→10→12→13；

- 1→2→3→4→5→8→9→2→10→12→13；
- 1→2→3→4→5→6→8→9→2→10→12→13；
- 1→2→3→4→5→6→7→8→9→2→10→11→13。

这 6 条路径构成了程序图 3-24 的一组基路径。因为基路径不唯一，所以也可以得到如下可能的一组基路径：

- 1→2→10→11→13；
- 1→2→10→12→13；
- 1→2→3→10→11→13；
- 1→2→3→4→5→8→9→2→…；
- 1→2→3→4→5→6→8→9→2→…；
- 1→2→3→4→5→6→7→8→9→2→…。

上面的最后 3 条路径后面的省略号表示在程序图中以后剩下的路径是可选择的。程序图中的结点 2、3、5、6 和 10 都是判断结点，是出度大于或等于 2 的结点，这些结点通常是旋转结点。

最后再生成测试用例，确保基路径中每条路径的执行。满足上述第二组基本路径集的测试用例如下所示。

路径 1：

输入数据：value[k]＝有效输入，限于 $k<i$。i 定义如下：

value[i]=－999，当 $2\leqslant i\leqslant 100$ 时.

预期结果：n 个值的正确的平均值、正确的总计数。

注意：不能孤立地进行测试，应当作为路径 4、5、6 测试的一部分来测试。

路径 2：

输入数据：value[1]＝－999。

预期结果：平均值＝－999，总计数取初始值。

路径 3：

输入数据：试图处理 101 个或更多的值，而前 100 个应当是有效的值。

预期结果：与测试用例 1 相同。

路径 4：

输入数据：value[i]＝有效输入，且 $i<100$；value[k]<最小值，当 $k<i$ 时。

预期结果：n 个值的正确的平均值、正确的总计数。

路径 5：

输入数据：value[i]＝有效输入，且 $i<100$；value[k]>最大值，当 $k\leqslant i$ 时。

预期结果：n 个值正确的平均值、正确的总计数。

路径 6：

输入数据：value[i]＝有效输入，且 $i<100$。

预期结果：n 个值的正确的平均值、正确的总计数。

测试用例执行之后，与预期结果进行比较。如果所有测试用例都执行完毕，则可以确信程序中所有的可执行语句至少都被执行了一次，达到了基路径覆盖，即达到 McCabe 覆盖。

7. 最少测试用例数计算

为实现测试的逻辑覆盖，必须设计足够多的测试用例，并使用这些测试用例执行被测程

序,实施测试。我们关心的是,对某个具体程序来说,至少应设计多少测试用例。这里提供一种估算最少测试用例数的方法。结构化程序由如下3种基本控制结构组成:

- 顺序型:构成串行操作。
- 选择型:构成分支操作。
- 重复型:构成循环操作。

为了把问题简化,避免出现测试用例极多的组合爆炸,把构成循环操作的重复型结构用选择结构代替。也就是说,并不指望测试循环体所有的重复执行,而只对循环体检验一次。这样,任一循环便改造成进入循环体或不进入循环体的分支操作了。图3-25给出了类似于流程图的N-S图表示的基本控制结构(图中A、B、C、D、S均表示要执行的操作,P是可取真假值的谓词,Y表示真值,N表示假值)。其中图3-25(c)和图3-25(d)两种重复型结构代表了两种循环。在做了如上简化循环的假设以后,对于一般的程序控制流,只考虑选择型结构。事实上它已能体现顺序型和重复型结构了。

例如,图3-26表达了两个顺序执行的分支结构。两个分支谓词P1和P2取不同值时,将分别执行a或b及c或d操作。显然,要测试这个小程序,需要至少提供4个测试用例才能做到逻辑覆盖。使得ac、ad、bc及bd操作均得到检验。其实,这里的4个用例是图中第一个分支谓词引出的两个操作和第二个分支谓词引出的两个操作组合起来而得到的,即$2\times2=4$。这里的2是由于两个并列的操作$1+1=2$而得到的。

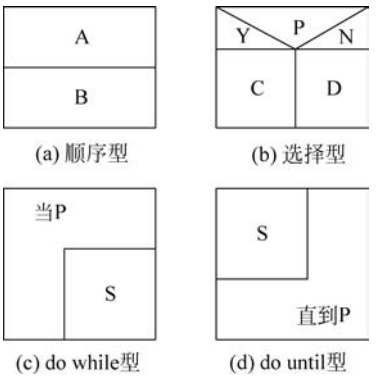


图3-25 N-S图表示的基本控制结构

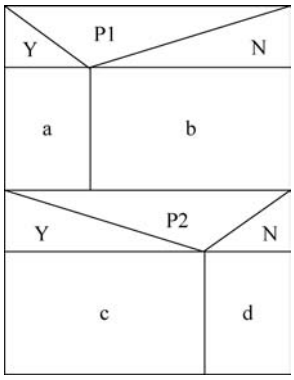


图3-26 两个串行的分支结构的N-S图

对于一般的、更为复杂的问题,估算最少测试用例数的原则也是同样的。现以图3-27表示的程序为例。该程序中共有9个分支谓词,尽管这些分支结构交错起来似乎十分复杂,很难一眼看出应至少需要多少个测试用例,但如果仍用上面的方法,也是很容易解决的。注意到图3-27可分上下两层:分支谓词1的操作域是上层,分支谓词8的操作域是下层。这两层正像前面图3-26中的P1和P2的关系一样。只要分别得到两层的测试用例个数,再将其相乘即得到总的测试用例数。这里需要首先考虑较为复杂的上层结构。谓词1不满足时要做的操作又可进一步分解为两层,这就是图3-28(a)和图3-28(b)部分。它们所需测试用例个数分别为$1+1+1+1+1=5$及$1+1+1=3$。因而两层组合,得到$5\times3=15$。于是整个程序结构上层所需测试用例数为$1+15=16$,而下层所需测试用例数显然为3。故最后得到整个程序所需测试用例数至少为$16\times3=48$。

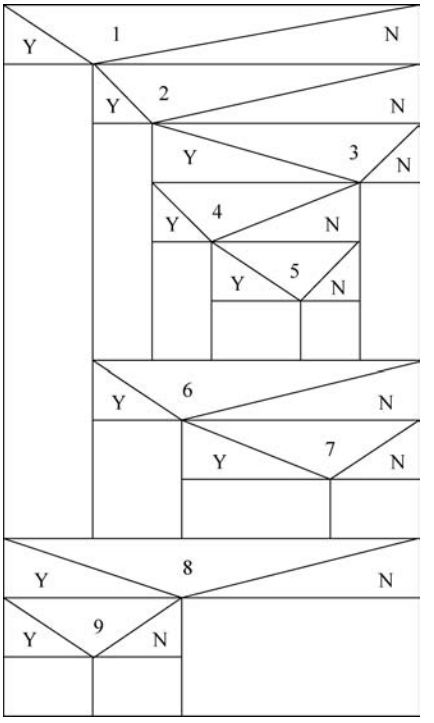


图 3-27　计算最少测试用例数实例

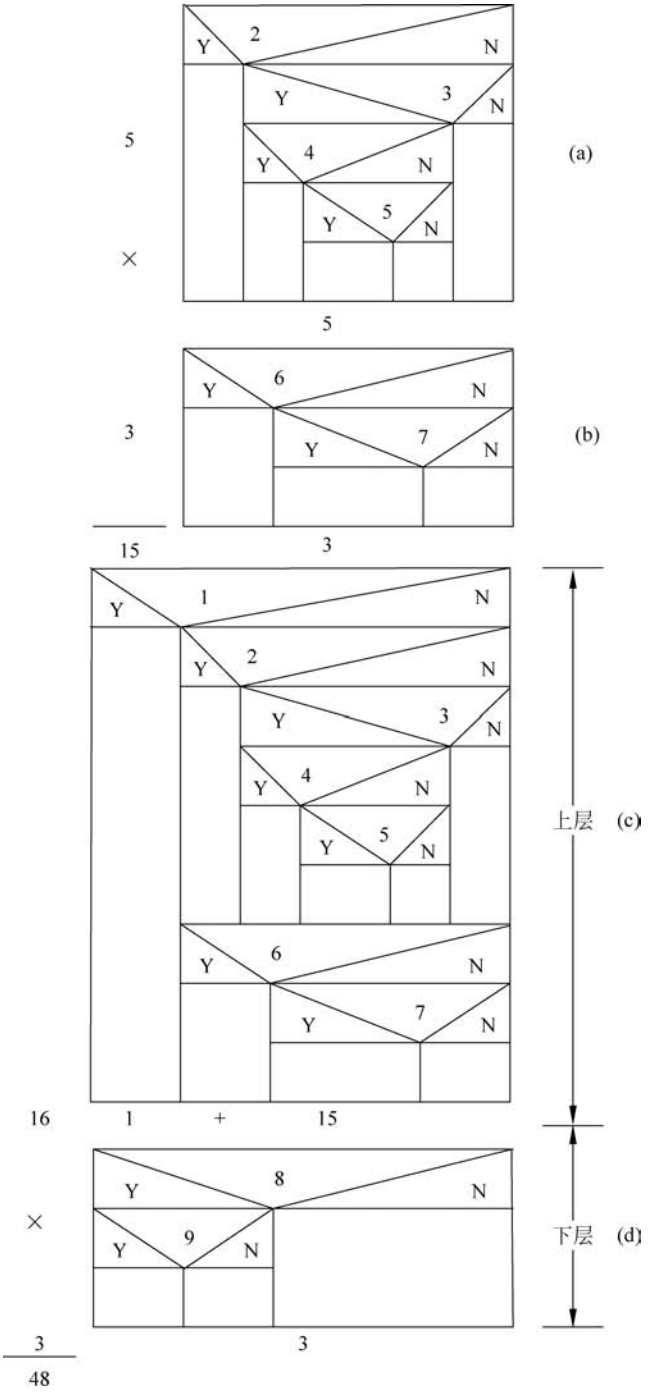


图 3-28　最少测试用例数计算(标号(a)、(b)、(c)、(d)为计算过程标识)

8. 测试覆盖准则

(1) Foster 的 ESTCA 覆盖准则。

前面介绍的逻辑覆盖其出发点似乎是合理的。所谓“覆盖”就是想要做到测试全面,而无遗漏。但事实表明,它并不能真的做到无遗漏。甚至像前面提到的将程序段:

```
if(I≥0)
then I = J
```

错写成：


```
if(I > 0)
then i = J
```

这样的小问题都无能为力。

分析出现这种情况的原因在于错误区域仅仅在 I=0 这个点上，即仅当 I 取 0 时，测试才能发现错误。它的确是在我们力图全面覆盖来查找错误的测试“网上”钻了空子，并且恰恰在容易发生问题的条件判断那里未被发现。面对这类情况应该从中吸取的教训是测试工作要有重点，要多针对容易发生问题的地方设计测试用例。K. A. Foster 从测试工作实践的教训出发，吸收了计算机硬件的测试原理，提出了一种经验型的测试覆盖准则，较好地解决了上述问题。

Foster 的经验型覆盖准则是从硬件的早期测试方法中得到启发的。我们知道，硬件测试中，对每个门电路的输入、输出测试都是有额定标准的。通常，电路中一个门的错误常常是“输出总是 0”，或是“输出总是 1”。与硬件测试中的这一情况类似，常常要重视程序中谓词的取值，但实际上它可能比硬件测试更加复杂。Foster 通过大量的实验确定了程序中谓词最容易出错的部分，得出了一套错误敏感测试用例分析(Error Sensitive Test Cases Analysis，ESTCA)规则。事实上，规则十分简单。

规则 1：对于 A rel B(rel 可以是<、=和>)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 rel1 C(rel1 可以是>或<，A 是变量，C 是常量)型的分支谓词，当 rel1 为< 时，应适当地选择 A 的值，使 A=C−M(M 是距 C 最小的容器容许正数，若 A 和 C 均为整型时，M=1)。同样，当 rel1 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。

显然，上述规则 1 是为了检测 rel 的错误，规则 2 是为了检测“差一”之类的错误(如本应是“if A > 1”而错成“if A > 0”)，而规则 3 则是为了检测程序语句中的错误(应引用一变量而错成引用一常量)。上述 3 个规则并不是完备的，但在普通程序的测试中却是有效的。原因在于规则本身针对程序编写人员容易发生的错误，或是围绕着发生错误的频繁区域，从而提高了发现错误的概率。

根据这里提供的规则来检验上述小程序段错误。应用规则 1，对它测试时，应选择 I 的值为 0，使 I=0 的情况出现一次。这样一来就立即找出了隐藏的错误。当然，ESTCA 规则也有很多缺陷。一方面是有时不容易找到输入数据，使得规则所指的变量值满足要求。另一方面是仍有很多缺陷发现不了。对于查找错误的广度问题在变异测试中得到较好的解决。

(2) Woodward 等人的层次 LCSAJ 覆盖准则。

Woodward 等人曾经指出结构覆盖的一些准则，如分支覆盖或路径覆盖，都不足以保证测试数据的有效性。为此，他们提出了一种层次 LCSAJ 覆盖准则。LCSAJ(Linear Code

Sequence And Jump)的意思是线性代码序列与跳转。一个 LCSAJ 是一组顺序执行的代码，以控制流跳转为其结束点。它不同于判断-判断路径。判断-判断路径是根据程序有向图决定的。一个判断-判断路径是指两个判断之间的路径，但其中不再有判断。程序的入口、出口和分支结点都可以是判断点。而 LCSAJ 的起点是根据程序本身决定的。它的起点是程序第一行或转移语句的入口点，或是控制流可以到达的点。几个首尾相接，且第一个 LCSAJ 起点为程序起点、最后一个 LCSAJ 终点为程序终点的 LCSAJ 串就组成了程序的一条路径。一条程序路径可能是由两个、三个或多个 LCSAJ 组成的。基于 LCSAJ 与路径的这一关系，Woodward 提出了 LCSAJ 覆盖准则。这是一个分层的覆盖准则：

第一层：语句覆盖。

第二层：分支覆盖。

第三层：LCSAJ 覆盖。即程序中的每一个 LCSAJ 都至少在测试中经历过一次。

第四层：两两 LCSAJ 覆盖。即程序中每两个首尾相连的 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。

它们说明了越是高层的覆盖准则越难满足。在实施测试时，要实现上述的 Woodward 层次 LCSAJ 覆盖，需要产生被测程序的所有 LCSAJ。

3.2.4 程序插装

程序插装(Program Instrumentation)是一种基本的测试手段，在软件测试中有广泛的应用。

1. 方法概述

程序插装方法简单地说是借助往被测程序中插入操作来实现测试目的的方法。程序插装的基本原理是在不破坏被测试程序原有逻辑完整性的前提下，在程序的相应位置上插入一些探针。这些探针本质上就是进行信息采集的代码段，可以是赋值语句或采集覆盖信息的函数调用。通过探针的执行并输出程序的运行特征数据。基于对这些特征数据的分析，揭示程序的内部行为和特征。例如，在调试程序时，常常要在程序中插入一些打印语句。其目的在于希望执行程序时打印出我们最为关心的信息。进一步通过这些信息了解执行过程中程序的一些动态特性。比如，程序的实际执行路径，或是特定变量在特定时刻的取值。从这一思想发展出的程序插装技术能够按用户的要求获取程序的各种信息，成为测试工作的有效手段。

如果想要了解一个程序在某次运行中所有可执行语句被覆盖(或称被遍历)的情况，或是每个语句的实际执行次数，最好的办法是利用插装技术。这里仅以计算整数 X 和整数 Y 的最大公约数程序为例，说明插装方法的要点。图 3-29 给出了这一程序的流程图。图中虚线框并不是原来程序的内容，而是为了记录语句执行次数而插入的。这些虚线框要完成的操作都是计数语句，其形式为：

$C(i)=C(i)+1 \quad i=1,2,\cdots,6$

程序从入口开始执行，到出口结束。凡经历的计数语句都能记录下该程序点的执行次数。如果在程序的入口处还插入了对计数器 C(i)初始化的语句，在出口处插入了打印这些计数器的语句，就构成了完整的插装程序。它便能记录并输出在各程序点上语句的实际执行次数。

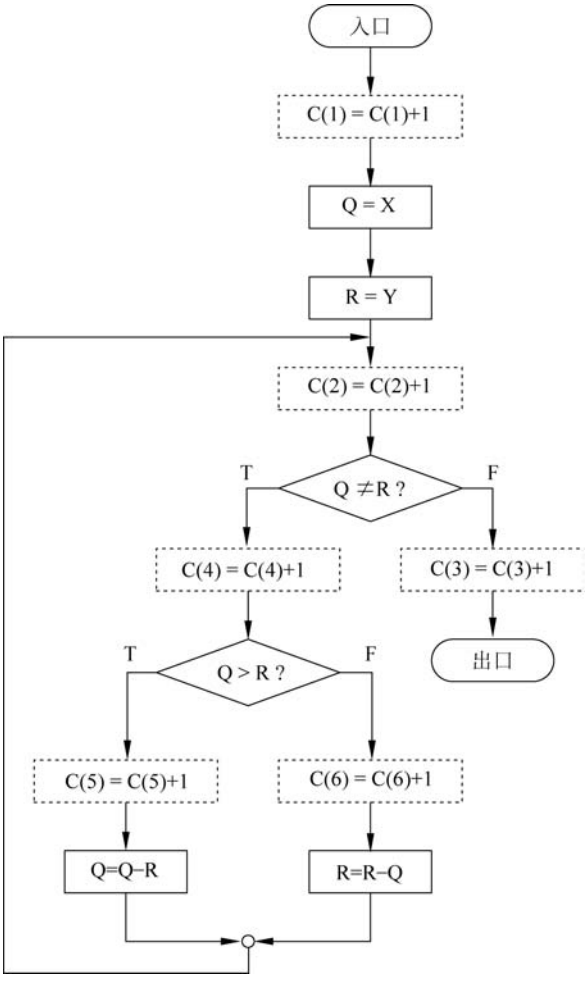


图 3-29　插装后的求最大公约数程序流程图

图 3-30 所示为插装后的语句,图中箭头所指均为插入的语句(原程序的语句已略去)。通过插入的语句获取程序执行中的动态信息,这一做法正如在刚研制成的机器特定部位安装记录仪表。安装好以后开动机器试运行,除了可以从机器加工的成品检验得知机器的运行特性外,还可以通过记录仪表了解其动态特性。这就相当于在运行程序以后,一方面可检验测试的结果数据,另一方面还可借助插入语句给出的信息了解程序的执行特性。正是这个原因,有时把插入的语句称为“探测器”,借以实现“探查”或“监控”的功能。

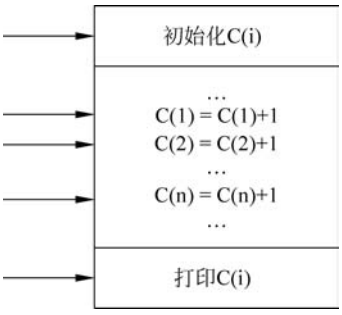


图 3-30　插装程序中插入的语句

在程序的特定部位插入记录动态特性的语句,最终是为了把程序执行过程中发生的一些重要历史事件记录下来。例如,记录在程序执行过程中某些变量值的变化情况、变化的范围等。又如本章中所讨论的程序逻辑覆盖情况,也只有通过程序的插装才能取得覆盖信息。实践表明,程序插装方法是应用很广的技术,特别是在完成程序的测试和调试时非常有效。

设计程序插装程序时需要考虑的问题包括:

(1) 探测哪些信息。

(2) 在程序的什么部位设置探测点。

(3) 设置多少个探测点。

其中前两个问题需要结合具体情况解决,不能给出笼统的回答。至于第三个问题,需要考虑如何设置最少探测点的方案。例如,图 3-29 中程序入口处,若要记录语句 Q=X 和 R=Y 的执行次数,只需插入 C(1)=C(1)+1 这样一个计数语句就够了,没有必要在每个语句之后都插入一个计数语句。一般情况下,可以认为在没有分支的程序段中只需一个计数语句。但程序中由于出现多种控制结构,使得整个结构十分复杂。为了在程序中设计最少的计数语句,需要针对程序的控制结构进行具体的分析。这里列举应在哪些部位设置计数语句:

- 程序块的第一个可执行语句之前;
- do、do while、do until 及 do 终端语句之后;
- if、else if、else 语句之后;
- 输入/输出语句之后;
- for 语句的开始前和结束之后。

2. 断言语句

断言(Assertion)是指变量应满足的条件,例如 I < 10、A(6) > O 等。在所测试的源程序中,在指定位置按一定格式,用注释语句写出的断言叫作断言语句。在程序执行时,对照断言语句检查事先指定的断言是否成立,有助于复杂系统的检验、调试和维护。

断言分为局部性断言和全局性断言两类。局部性断言是指在程序的某一位置上,例如重要的循环或过程的入口和出口处,或者在一些可能引起异常的关键算法之前设置的断言语句。例如,在赋值语句 A-B/Z 之前,设置局部性断言语句:

```
C ASSERT L()CAL(Z<>0)
```

全局性断言是指在程序运行过程中自始至终都适用的断言。例如,变量 I、J、K 只能取 0～100 的值,变量 M、N 只能取 2、4、6、8 这 4 个值等。全局性断言写在程序的说明部分。描述格式为:

```
C ASSERT VALUES(I,J,K)(O: 100)
C ASSERT VALUES(M,N)(2,4,6,8)
```

程序员在每个变量、数组的说明之后,都可写上反映其全局特性的断言。动态断言处理程序的工作过程如下:

(1) 动态断言处理程序对语言源程序做预处理,为注释语句中的每个断言都插入一段相应的检验程序。

(2) 运行经过预处理的程序,检验程序将检查程序的实际运行结果与断言所规定的逻辑状态是否一致。对于局部性断言,每当程序执行到这个位置时,相应的检验程序就要工作;对于全局性断言,在每次变量被赋值后,相应的检验程序就进行工作。动态断言处理程序还要统计检验的结果(即断言成立或不成立的次数),在发现断言不成立的时候,还要记录当时的现场信息,如有关变量的状态等。处理程序还可按测试人员的要求,在某个断言不成立的次数已达指定值时中止程序的运行,并输出统计报告。

(3) 一组测试结束后,程序输出统计结果、现场信息,供测试人员分析。

下面介绍 JUnit 中提供的一些断言。JUnit 为我们提供了一些辅助函数,用来帮助我们确

定被测试的方法是否按照预期的效果正常工作,通常把这些辅助函数称为断言。下面介绍一下 JUnit 的断言。

1) assertEquals

函数原型 1: assertEquals([String message],expected,actual)。

参数说明: message 是一个可选的消息,如果提供,将会在发生错误时报告这个消息。expected 是期望值,通常都是用户指定的内容。actual 是被测试的代码返回的实际值。

函数原型 2: assertEquals([String message],expected,actual,tolerance)。

参数说明: message 是一个可选的消息,如果提供,将会在发生错误时报告这个消息。expected 是期望值,通常都是用户指定的内容。actual 是被测试的代码返回的实际值。tolerance 是误差参数,参加比较的两个浮点数在这个误差之内则会被认为是相等的。

2) assertTrue

函数原型: assertTrue([String message],Boolean condition)。

参数说明: message 是一个可选的消息,如果提供,将会在发生错误时报告这个消息。condition 是待验证的布尔型值。该断言用来验证给定的布尔型值是否为真,如果结果为假,则验证失败。当然,还有验证为假的测试条件,函数原型: assertFalse([String message],Boolean condition)。该断言用来验证给定的布尔型值是否为假,如果结果为真,则验证失败。

3) assertNull

函数原型: assertNull([String message],Object object)。

参数说明: message 是一个可选的消息,如果提供,将会在发生错误时报告这个消息。Object 是待验证的对象。该断言用来验证给定的对象是否为 null,如果不为 null,则验证失败。相应地,还存在可以验证非 null 的断言,函数原型: assertNotNull([String message],Object object)。该断言用来验证给定的对象是否为非 null,如果为 null,则验证失败。

4) assertSame

函数原型: assertSame([String message],expected,actual)。

参数说明: message 是一个可选的消息,如果提供,将会在发生错误时报告这个消息。expected 是期望值。actual 是被测试的代码返回的实际值。该断言用来验证 expected 参数和 actual 参数所引用的是否是同一个对象,如果不是,则验证失败。相应地,也存在验证不是同一个对象的断言,函数原型: assertNotSame ([String message],expected,actual)。该断言用来验证 expected 参数和 actual 参数所引用的是否是不同对象,如果所引用的对象相同,则验证失败。

5) Fail

函数原型: Fail([String message])。

参数说明: message 是一个可选的消息,如果提供,将会在发生错误时报告这个消息。该断言会使测试立即失败,通常用在测试不能达到的分支上(如异常)。

从 JUnit 4.4 开始引入了 Hamcrest 框架,Hamcrest 提供了一套匹配符 Matcher,这些匹配符更接近自然语言,可读性强,更加灵活。也使用全新的断言语法: assertThat,结合 Hamcrest 提供的匹配符,只用这一个方法,就可以实现所有的测试。assertThat 语法如下:

```
assertThat(T actual, Matcher<T> matcher);
assertThat(String reason, T actual, Matcher<T> matcher);
```

其中，actual 为需要测试的变量，matcher 为使用 Hamcrest 的匹配符来表达变量 actual 期望值的声明。应注意的是，必须导入 JUnit4.4 之后的版本才能使用 assertThat 方法。不需要继承 TestCase 类，但是需要测试方法前必须加@Test。以下是一个例子。

TTest.java:

```
01. package cn.edu.ahau.mgc.junit4.test;
02.
03. import Java.util.List;
04. import Java.util.Map;
05.
06. import org.junit.Test;
07. import static org.junit.Assert.*;
08. import cn.edu.ahau.mgc.junit4.T;
09. import static org.hamcrest.Matchers.*;
10.
11. public class TTest {
12.
13.         @Test
14.         public void testAdd() {
15.
16.                 //一般匹配符
17.                 int s = new T().add(1,1);
18.                 //allOf: 所有条件必须都成立,测试才通过
19.                 assertThat(s,allOf(greaterThan(1),lessThan(3)));
20.                 //anyOf: 只要有一个条件成立,测试就通过
21.                 assertThat(s,anyOf(greaterThan(1),lessThan(1)));
22.                 //anything: 无论什么条件,测试都通过
23.                 assertThat(s,anything());
24.                 //is: 变量的值等于指定值时,测试通过
25.                 assertThat(s,is(2));
26.                 //not: 和 is 相反,变量的值不等于指定值时,测试通过
27.                 assertThat(s,not(1));
28.
29.                 //数值匹配符
30.                 double d = new T().div(10,3);
31.                 //closeTo: 浮点型变量的值在 3.0±0.5 范围内,测试通过
32.                 assertThat(d,closeTo(3.0,0.5));
33.                 //greaterThan: 变量的值大于指定值时,测试通过
34.                 assertThat(d,greaterThan(3.0));
35.                 //lessThan: 变量的值小于指定值时,测试通过
36.                 assertThat(d,lessThan(3.5));
37.                 //greaterThanOrEuqalTo: 变量的值大于或等于指定值时,测试通过
38.                 assertThat(d,greaterThanOrEqualTo(3.3));
39.                 //lessThanOrEqualTo: 变量的值小于或等于指定值时,测试通过
40.                 assertThat(d,lessThanOrEqualTo(3.4));
41.
42.                 //字符串匹配符
43.                 String n = new T().getName( "Magci" );
44.                 //containsString: 字符串变量中包含指定字符串时,测试通过
45.                 assertThat(n,containsString( "ci" ));
46.                 //startsWith: 字符串变量以指定字符串开头时,测试通过
```

```
47.                 assertThat(n,startsWith( "Ma" ));
48.                 //endsWith: 字符串变量以指定字符串结尾时,测试通过
49.                 assertThat(n,endsWith( "i" ));
50.                 //euqalTo: 字符串变量等于指定字符串时,测试通过
51.                 assertThat(n,equalTo( "Magci" ));
52.                 //equalToIgnoringCase: 字符串变量在忽略大小写的情况下等于指定字符串时,
                    //测试通过
53.                 assertThat(n,equalToIgnoringCase( "magci" ));
54.                 //equalToIgnoringWhiteSpace: 字符串变量在忽略头尾任意空格的情况下等于指
                    //定字符串时,测试通过
55.                 assertThat(n,equalToIgnoringWhiteSpace( " Magci " ));
56.
57.                 //集合匹配符
58.                 List<String> l = new T().getList( "Magci" );
59.                 //hasItem: Iterable 变量中含有指定元素时,测试通过
60.                 assertThat(l,hasItem( "Magci" ));
61.
62.                 Map<String,String> m = new T().getMap( "mgc" ,"Magci" );
63.                 //hasEntry: Map 变量中含有指定键值对时,测试通过
64.                 assertThat(m,hasEntry( "mgc","Magci" ));
65.                 //hasKey: Map 变量中含有指定键时,测试通过
66.                 assertThat(m,hasKey( "mgc" ));
67.                 //hasValue: Map 变量中含有指定值时,测试通过
68.                 assertThat(m,hasValue( "Magci" ));
69.         }
70.
71. }
```

3.2.5 其他白盒测试方法简介

1. 域测试

域测试(Domain Testing)是一种基于程序结构的测试方法。Howden 曾对程序中出现的错误进行分类,他把程序错误分为域错误、计算型错误和丢失路径错误三种。这是相对于执行程序的路径来说的。我们知道,每条执行路径都对应于输入域的一类情况,是程序的一个子计算。如果程序的控制流有错误,对于某一特定的输入可能执行的是一条错误路径,这种错误称为路径错误,也叫作域错误。如果对于特定输入执行的是正确路径,但由于赋值语句的错误致使输出结果不正确,则称此为计算型错误。另一类错误是丢失路径错误,它是由于程序中某处少了一个判定谓词而引起的。域测试是指主要针对域错误进行的程序测试。域测试的“域”是指程序的输入空间。域测试方法基于对输入空间的分析。当然,任何一个被测程序都有一个输入空间。测试的理想结果就是检验输入空间中的每个输入元素是否都产生正确的结果。而输入空间又可分为不同的子空间,每一子空间对应一种不同的计算。在查看被测试程序的结构以后就会发现,子空间的划分是由程序中分支语句中的谓词决定的。输入空间的一个元素,经过程序中某些特定语句的执行而结束(当然也可能出现无限循环而无出口),这都是满足了这些特定语句被执行所要求的条件的。域测试正是在分析输入域的基础上,选择适当的测试点以后进行测试的。域测试有两个致命的弱点:一是为进行域测试对程序提出的限制过多;二是当程序存在很多路径时,所需的测试点也就很多。

2. 符号测试

符号测试的基本思想是允许程序的输入不仅仅是具体的数值数据,而且包括符号值,这一

方法也是因此而得名。这里所说的符号值可以是基本符号变量值，也可以是这些符号变量值的一个表达式。这样，在执行程序过程中以符号的计算代替了普通测试执行中对测试用例的数值计算。所得到的结果自然是符号公式或符号谓词。更明确地说，普通测试执行的是算术运算，符号测试则执行的是代数运算。因此符号测试可以认为是普通测试的一个自然的扩充。符号测试可以看作是程序测试和程序验证的一个折中方法。一方面，它沿用了传统的程序测试方法，通过运行被测程序来验证它的可靠性。另一方面，由于一次符号测试的结果代表了一大类普通测试的运行结果，实际上是证明了程序接受此类输入，所得输出是正确的还是错误的。最为理想的情况是，程序中仅有有限的几条执行路径。如果对这有限的几条路径都完成了符号测试，就能较有把握地确认程序的正确性了。从符号测试方法的使用来看，问题的关键在于开发出比传统的编译器功能更强、能够处理符号运算的编译器和解释器。目前符号测试存在一些未得到圆满解决的问题，分别是：

1）分支问题

当采用符号执行方法进行到某一分支点处，分支谓词是符号表达式，这种情况下通常无法决定谓词的取值，也就不能决定分支的走向，需要测试人员进行人工干预，或是执行树的方法进行下去。如果程序中有循环，而循环次数又取决于输入变量，那就无法确定循环的次数。

2）二义性问题

数据项的符号值可能是有二义性的。这种情况通常出现在带有数组的程序中。我们来看以下程序段：

```
X( I ) = 2 + A
      X( J ) = 3
      C = X( I )
```

如果 I=J，则 C=3，否则 C=2+A。但由于使用符号值运算，这时无法知道 I 是否等于 J。

3）大程序问题

符号测试中总要处理符号表达式。随着符号执行的继续，一些变量的符号表达式会越来越庞大。特别是当符号执行树很大、分支点很多时，路径条件本身变成一个非常长的合取式。如果能够有办法将其化简，自然会带来很大好处。但如果找不到化简的办法，那将给符号测试的时间带来大幅度的增长，甚至使整个问题的解决遇到难以克服的困难。

3. Z 路径覆盖

分析程序中的路径是指检验程序从入口开始，执行过程中经历的各个语句，直到出口。这是白盒测试最为典型的问题，前面已经做了分析。着眼于路径分析的测试可称为路径测试。完成路径测试的理想情况是做到路径覆盖。对于比较简单的小程序实现路径覆盖是可以做到的。但是如果程序中出现多个判断和多个循环，可能的路径数目将会急剧增长，达到天文数字，以致实现路径覆盖不可能做到。为了解决这一问题，前面讨论了基路径测试方法。这里将简单讨论另外一种解决该问题的方法，思路是必须舍掉一些次要因素，对循环机制进行简化，从而极大地减少路径的数量，使得覆盖这些有限的路径成为可能。我们称简化循环意义下的路径覆盖为 Z 路径覆盖。这里所说的对循环化简是指限制循环的次数。无论循环的形式和实际执行循环体的次数多少，只考虑循环一次和零次两种情况，即只考虑执行时进入循环体一次和跳过循环体这两种情况。图 3-31(a)和图 3-31(b)表示了两种最典型的循环控制结构。前者先进行判断，循环体 B 可能执行(假定只执行一次)，也可能不执行，这就如同图 3-31(c)所表示的条件选择结构。后者先执行循环体 B(假定也执行一次)，再经判断转出，其效果也与

图 3-31(c)中给出的条件选择结构只执行右支的效果一样。

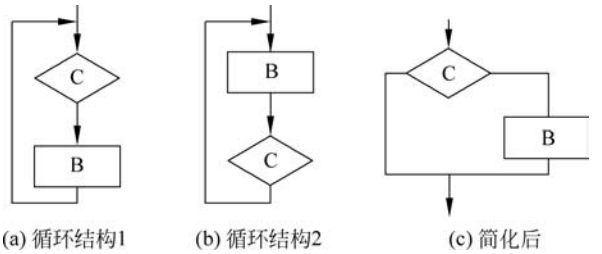


图 3-31　循环结构简化成选择结构

对于程序中的所有路径可以用路径树来表示,具体表示方法本书略。当得到某一程序的路径树后,从其根结点开始,一次遍历,再回到根结点时,把所经历的叶结点名排列起来,就得到一个路径。如果设法遍历了所有的叶结点,那就得到了所有的路径。当得到所有的路径后,生成每个路径的测试用例,就可以做到 Z 路径覆盖测试。

4. 程序变异

程序变异方法与前面提到的结构测试和功能测试都不一样,它是一种错误驱动测试。所谓错误驱动测试方法,是指该方法是针对某类特定程序错误的。经过多年的测试理论研究和软件测试的实践,人们逐渐发现要想找出程序中所有的错误几乎是不可能的。比较现实的解决办法是将错误的搜索范围尽可能地缩小,以利于专门测试某类错误是否存在。这样做的好处在于,便于将目标集中于对软件危害最大的错误,而暂时忽略对软件危害较小的可能错误。这样可以取得较高的测试效率,并降低测试的成本。错误驱动测试主要有两种,即程序强变异和程序弱变异。为便于测试人员使用变异方法,一些变异测试工具被开发出来。关于程序变异测试方法,请参见其他资料。

3.2.6　白盒测试方法选择的策略

在白盒测试中,使用各种测试方法的综合策略参考如下。

- 在测试中,应尽量先用人工或工具进行静态结构分析。
- 测试中可采取先静态后动态的组合方式:先进行静态结构分析、代码检查并进行静态质量度量,再进行覆盖率测试。
- 利用静态分析的结果作为引导,通过代码检查和动态测试的方式对静态分析结果进行进一步的确认,使测试工作更为有效。
- 覆盖率测试是白盒测试的重点,一般可使用基路径测试法达到语句覆盖标准;对于软件的重点模块,应使用多种覆盖率标准衡量代码的覆盖率。
- 在不同的测试阶段,测试的侧重点不同:在单元测试阶段,以检查代码、逻辑覆盖为主;在集成测试阶段,需要增加静态结构分析、静态质量度量;在系统测试阶段,应根据黑盒测试的结果,采取相应的白盒测试。

练　　习

1. 在万年历程序中年份(Y)的范围是 $2000 \leqslant Y \leqslant 2500$,分别用边界值分析法、等价类法、因果图法及决策表法设计测试用例。

2. 表 3-44 为个人所得税税率的计算方法列表：

表 3-44　习题 2 表

金额/元	税率及说明
0～500	0%
500～2000	超出 500 元部分按照 5%计税
2000～5000	10%
5000～20 000	15%
20 000 以上	20%

理解表中的数据，分别用边界值分析法、等价类法设计测试用例。

3. 理解一个宾馆在线订购服务系统，用场景法设计测试用例。

4. 理解下列程序结构，分别用逻辑覆盖法和基路径覆盖法设计测试用例。

```
#include <stdio.h>
int partition(int *data,int low,int high)
{    int t = 0;
    t = data[low];
    while(low < high)
    {    while(low < high && data[high] >= t)
            high--;
        data[low] = data[high];
        while(low < high && data[low] <= t)
            low++;
        data[high] = data[low];
    }
    data[low] = t;
    return low;
}
```

5. 举一个例子应用正交试验法设计测试用例。