第 5 章 故障假设测试

本章导读

故障假设是对程序员缺陷的反思！

成功的经验各有不同，失败的教训却总是相似。软件工程师通过研究故障模式来防止发生类似故障，对常见故障的经验总结也被用于软件设计方法和编程语言的改进。当然，并不是所有的故障都可以使用静态分析进行检测和预防，更多的故障必须通过动态的测试才能发现。边界是程序员最容易犯错的地方，也是计算时容易产生故障的区域所在。边界值分析可以更快速地找到软件缺陷，因为边界处的缺陷密度往往更大。边界故障可以分为三大类：输入边界故障、中间边界故障和输出边界故障。中间边界又分为静态的代码边界和动态的计算边界。5.1节主要介绍边界故障假设。其中，5.1.1节介绍输入边界值分析，主要分析输入数据类型的经验边界和理论局限性。5.1.2节介绍计算边界值分析，主要考虑浮点运算机制下的数值边界扰动分析技术。5.1.3节介绍输出边界值分析，并结合智能软件系统的输出边界问题展开讨论。

边界故障假设可以被看作最朴素的经验总结，而系统性工作当属变异故障假设。5.2节介绍变异故障假设，5.2.1节介绍变异分析的基本概念，通过极小语法改变产生变异来模拟程序员常犯的错误。阐述了故障建模的两个基本假设：熟练程序员假设和耦合效应假设。然而大量的变异算子带来极其昂贵的计算成本，制约了变异分析的工业应用。5.2.2节介绍变异测试优化技术，更多的常用变异分析优化方法可参阅其他文献。5.2.3节介绍变异分析理论框架，其中的变异微分器是重点，变异位置偏序格可以结合控制流和数据流分析一起理解。

在安全攸关软件系统分析与测试中，有一类特殊的变异分析尤其值得关注和深入研究。5.3节介绍变异分析在逻辑控制密集型的安全攸关软件中的应用。与前面的变异分析经验总结不同，本节更加侧重理论分析。5.3.1节首先将程序逻辑抽象成布尔范式进行故障建模，分析逻辑故障结构之间的理论联系。Kuhn 确定了布尔规范中 3 种类型故障之间的关系，为逻辑故障层次结构的第一次理论尝试。5.3.2节主要介绍笔者的博士后代表性工作，详细分析了 10 种逻辑故障，建立了正确且完备的故障层次结构，为后续章节的 MCDC 及其他逻辑测试覆盖准则提供理论基础。5.3.2节介绍逻辑可满足性及其传统逻辑可满足性问题扩展求解能力方法 SMT（Satisfiability Modulo Theories）。

5.1 边界故障假设

工程师研究故障以了解如何防止将来发生类似故障。对常见软件故障的经验分析也被用于软件设计方法和编程语言改进。例如，Java 中自动内存管理的主要目的不是让程序员免去释放未使用内存的麻烦，而是防止程序员犯空指针、冗余释放和内存泄露、内存管理错误。C 和 C++ 中的自动数组边界检查不能阻止程序员在数组边界之外使用索引表达式，但可以大大降低错误在测试中逃脱检测的可能性。当然，并不是所有的程序员错误都属于可以使用编程语言预防或静态检测的类别。有些故障必须通过动态运行的测试才能发现，而且也可以利用常见故障的知识来提高测试效率。基于故障的测试是选择能够将被测程序与包含假设故障替代程序区分开来的测试。故障注入可用于评估测试集的充分性，或用于选择测试以扩充测试集，也可以估计程序中的故障数量完成可靠性分析。本节介绍此类最常用的方法：边界故障假设。因为边界故障是程序员最容易犯的错误，也是计算机最容易产生故障的区域所在。边界值分析的优点是使用这种技术可以更轻松、更快速地找到缺陷。这是因为边界处的缺陷密度往往更大。

5.1.1 输入边界值分析

边界值指对于划分区域而言，稍高于其最高值或稍低于最低值的一些特定情况。边界值分析的步骤包括确定边界、选择测试两个步骤。根据大量的测试统计数据，很多错误出现在输入或输出范围的边界上，而不是出现在输入范围的中间区域。因此针对各种边界情况设计测试，可以查出更多的错误。边界值分析法是一种很实用的黑盒测试方法，且具有较强的故障检测缺陷能力。将输入域 D 划分为一组子域，并在此基础上生成测试输入。

等价类划分和边界值分析针对以下两大类故障：计算故障——在实现中对某些子域应用了错误的函数；域故障——实现中两个子域的边界是错误的。在等价类划分中，倾向于查找计算故障的测试输入。由于计算故障会导致在某些子域中应用错误的函数，因此在等价类划分中，从每个子域中仅选择几个测试输入是正常的。边界值分析倾向于通过使用靠近边界的测试输入来查找域故障。让假设相邻子域之间的边界被错误地实现，导致子域偏差。因此如果在实现中测试输入位于错误的子域中，则能够检测到此故障。因此，边界值分析方法旨在生成一组测试输入，使得如果存在域故障，那么在实现中至少有一个测试输入可能位于错误的子域中。

实践中，通常先使用位于等价类边界值的测试数据来分析应用程序的行为。然后通过使用位于边界的测试数据。下面考虑在等价类划分中使用的相同示例。一个应用程序接受一个数值在 10~100 的数字作为输入。在测试这样的应用程序时，不仅会用 10~100 的值来测试它，还会用其他值集来测试它，如小于 10、大于 100、特殊字符、字母、数字等。具有开放边界的应用程序不适合这种技术。在这种情况下，会使用其他黑盒技术，例如“域分析”。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，如果程序的规格说明中规定：“重量在 10~50kg 范围内的邮件，其邮费计算公式为……”。边界分析应取 10.00 及 50.00，还应取 10.01、49.99、

9.99及50.01等。

如果输入条件规定了值的个数,则用最大个数、最小个数、比最小个数少一、比最大个数多一的数作为测试数据。例如，一个输入文件应包括1~255条记录，则取1和255，还应取0及256等。根据规格说明的每个输出条件应用前面的原则。例如，某程序的规格说明要求计算出“每月保险金扣除额为0~1165.25元”，其测试可取0.00及1165.24，还可取−0.01及1165.26等。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和最后一个元素。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。分析规格说明，找出可能的边界条件。通常情况下，软件测试所包含的边界检验有几种类型：数字、字符、位置、质量、大小、速度、方位、尺寸、空间。相应地，以上类型的边界值应该为最大/最小、首位/末位、上/下、最快/最慢、最高/最低、最短/最长、空/满等。

定义5.1　边界值分析

输入域D划分为一组子域$D_1,D_2\cdots,D_n$，假设D_i存在最小值$\min_i$和最大值$\max_i$，+和−分别表示略大于或略小于最小值。则$\cup_i\{\min_i+,\max_i-\}$称为输入域$D$的正向边界值分析，$\cup_i\{\min_i-,\max_i+\}$称为输入域$D$的负向边界值分析，$\cup_i\{\min_i+,\min_i-,\min_i,\max_i,\max_i-,\max_i+\}$称为输入域$D$的边界值分析。 ♣

边界值分析法的基本原理是故障更可能出现在输入变量的极值附近。边界值分析法的基本思想是选取正好等于、刚刚大于或刚刚小于边界的值作为测试数据，而不是选取等价类中的典型值或任意值作为测试数据。将输入域D划分为一组子域$D_1,D_2\cdots,D_n$，并在此基础上生成测试输入。针对每个子域D_i的最小值min和最大值max，选取略大于最小值min+或者略小于最小值min−，和选取略大于最小值max+或者略小于最小值max−作为测试数据，是边界值分析的基本思路。让假设相邻子域D_i和D_j之间的边界被错误地实现，导致产生新的子域A_i和A_j。然后，如果x在实现中位于错误的子域中，则测试输入x将会应用错误的功能，因此能够检测到此故障。可能$x\in D_i$且$x\notin A_i$，或者$x\in S_j$且$x\notin A_j$。因此，针对边界值分析的方法旨在产生一组测试输入，以便在存在域故障的情况下，至少有一个测试输入可能会在实现中位于错误的子域中。

首先以三角形程序Triangle的边界值分析测试的生成过程。在三角形程序Triangle中，除了要求边长是整数外，没有给出其他的限制条件。在此，将三角形每条边边长的取范围值设置为[1, 100]的整数。因此边界值分析主要从以下几方面进行。

- min−：三条边中某一条边长度为0的数据，预期输出结果为“无效输入值”。
- min：三条边中某一条边长度为1的数据，预期输出结果还不能确定。
- min+：三条边中某一条边长度为2的数据，预期输出结果还不能确定。
- max−：三条边中某一条边长度为99的数据，预期输出结果还不能确定。
- max：三条边中某一条边长度为100的数据，预期输出结果还不能确定。
- max+：三条边中某一条边长度为101的数据，预期输出结果为“无效输入值”。
- 其他：三条边中某一条边长度为负数的数据，预期输出结果为“无效输入值”。

对于每种情况，测试人员应该输入合适的数据进行测试，并检查程序输出的结果是否符合预期输出。测试人员还应该检查程序对超出上下限的数据的处理方式，以避免程序意外崩溃或产生错误的输出。由于三角形类型由三条边组合才能确定，因此上述边界值分析还存在若干预期输出结果不能确定的情形。将边界值分析和组合测试相结合可以更全面地测试程序，并发现更多的错误。这样可以确保测试用例覆盖了所有可能的情况，并提高测试用例的效率和覆盖率。关于三角形的无效特性的边界值分析示例如表5.1所示，依然围绕参数a展开局部示例，其他留给读者思考。

表 5.1　无效特性的边界值分析示例

ID	a	b	c	预期输出
1	0	2	3	无效输入值
2	1	2	3	无效输入值
3	2	2	3	等腰三角形
4	99	2	3	无效输入值
5	100	2	3	无效输入值
6	101	2	3	无效输入值
7	−1	2	3	无效输入值
8	1	1	3	无效输入值
9	1	1	1	等边三角形
10	100	100	100	等边三角形

再看看日期程序NextDay的边界值分析示例，如表5.2所示。在NextDay中，隐含规定了参数month和参数day的取值范围为1⩽month⩽12和1⩽day⩽31，并设定参数year的取值范围为1900⩽year⩽2050。这里的显性边界是year的1900和2050，month的1和12，注意不同月份的天数区别，尤其是闰年和平年中2月的区别，day的边界要考虑1、28、29、30和31。总体来说，在日期程序NextDay的边界值分析中，大概需要考虑使用包含最大值、最小值、闰年、非闰年、30天和31天等因素。

表 5.2　边界值分析示例

ID	month	day	year	预　期　输　出
1	6	15	1812	6, 16, 1812
2	6	15	1813	6, 16, 1813
3	6	15	1912	6, 16, 1912
4	6	15	2011	6, 16, 2011
5	6	15	2012	6, 16, 2012
6	6	1	1912	6, 2, 1912
7	6	2	1912	6, 3, 1912
8	6	30	1912	7, 1, 1912
9	6	31	1912	无效
10	1	15	1912	1, 16, 1912

续表

ID	month	day	year	预 期 输 出
11	2	15	1912	2, 16, 1912
12	11	15	1912	11, 16, 1912
13	12	15	1912	12, 16, 1912

在针对均值方差程序MeanVar进行边界值分析和测试时，需要确定输入和输出的边界限制。该程序的输入是一组数字数据，输出是数据的均值和方差。该程序的一些可能的边界值包括以下几种。

- 只有一个数值的输入：此测试用例检查程序是否能处理最小的输入大小。
- 具有两个相同值的输入：此测试用例检查程序是否能处理最小的非零方差。
- 具有两个不同值的输入：此测试用例检查程序是否能处理非零方差。
- 具有最大允许值的输入：此测试用例检查程序是否能处理最大允许输入值。
- 具有最大允许值的输入：此测试用例检查程序是否能处理最大允许输入值。
- 具有超出允许范围的值的输入：此测试用例检查程序是否能处理无效的输入。

除边界值测试之外，还应执行其他测试，以确保程序可以处理典型和边缘情况。测试用例应涵盖不同类型的数据，例如整数和浮点值、正值和负值，以及大值和小值。针对均值方差程序MeanVar进行边界值分析，应该考虑一些特殊情况，以确保程序的正确性和稳健性。以下是一些可能需要考虑的情况。

- 边界情况：应该考虑输入数据的边界情况。例如，当输入数据包含最大值或最小值时，程序应当能够正确处理这些情况。
- 数据类型：应该考虑输入数据的数据类型。例如，当输入数据为浮点数时，程序应该使用浮点数计算，以避免舍入误差。
- 数据格式：应该考虑输入数据的格式。例如，当输入数据为时间序列时，程序应该考虑时间序列的特殊性，并使用适当的方法计算方差。
- 数据分布：应该考虑输入数据的分布情况。例如，当输入数据为正态分布时，程序可以使用均值和标准差计算方差。但是，当输入数据不是正态分布时，程序应该使用适当的方法计算方差。

为了确保程序的正确性和稳健性，可以使用一些测试用例来验证程序的输出结果。例如，可以使用包含最大值、最小值、浮点数、时间序列等特殊情况的测试用例，以验证程序的正确性。

边界值分析可以看作一种方法，它假定可以将实现的输入域划分为$A_1, A_2, \cdots, A_n$，使得对于所有$1 \leqslant i \leqslant n$，$A_i$类似于$D_i$，在每个$A_i$上实现的行为是一致的，并且由在$A_i$上的实现定义的函数$\bar{f}_i$符合$f_i$（如果$\bar{f}_i$不符合$f_i$，那么期望分区分析找到这个计算故障。因此，对于每个(D_i, f_i)，都有一个对应的$(A_i, \bar{f}_i)$。边界值分析中生成的测试输入主要针对这些假设所允许的故障类型——域错误，假设每个(S_i, f_i)都有一个$(A_i, \bar{f}_i)$。

将在边界周围生成成对的测试输入的基本思想扩展到边界值分析的其他方法。考虑规范中子域D_i和D_j之间的边界B，假设边界上的值在D_i中。为了检查边界B，生成(x, x')

形式的成对测试输入，使得x在边界上（因此在x'中），而D_i在D_j中并且接近x。如果x和x'在实现的正确子域A_i和A_j中，则实际边界必须在x和x'之间通过。如果两个子域都不包含边界，则将生成成对的测试输入，并且在边界的任一侧都有一个。通常会为边界生成多个这样的对，理想情况下会在边界上分布。

边界值分析可能会存在偶然正确的情况。假设在确定性规范中，存在两个相邻的子域S_i和S_j，系统功能应在这两个子域上分别为f_i和f_j。进一步假设子域S_i和S_j之间的边界是错误地实现导致在实现中出现子域A_i和A_j，并且使用了测试输入x，$x \in S_i$且$x \in A_j$。因此，x在实现中位于错误的子域中。但是，只有在观察到故障时，x才会检测到该域故障，并且仅当f_i和f_j在x上产生不同的输出时才会发生。因此，如果$f_i(x) = f_j(x)$，则测试输入x无法检测到其中x位于A_j而不是A_i的域故障。如果是这种情况，那么对于检查S_i和S_j之间的边界，x是一个较差的测试输入，且不管它是在边界上还是不在边界上。如果不在边界上，那么考虑是否接近边界。因此，假设规范和程序是确定性的，定义偶然正确性的概念。

定义5.2　偶然正确

对于边界值分析的输入x,如果$x \in D_i$且$x \in A_j$,对于某些$i \neq j$,但是$f_i(x) = \bar{f}_j(x)$,则输入x发生巧合正确性。假设没有计算故障，则条件的最后部分简化为$f_i(x) = f_j(x)$。 ♣

假设没有计算故障，条件的最后部分简化为$f_i(x) = f_j(x)$。从以上可以明显看出，边界值分析的测试输入选择不应仅基于几何参数，还应考虑不同子域中预期的功能。

在本节中，假定输入域D已被分区为形成子域$D_1, D_2, \cdots, D_n$，并且D_i成对不相交并且覆盖D。当使用边界值分析检查相邻子域D_i和D_j之间的边界时，产生成对的测试输入(x, x')。$x \in D_i$；$x' \in D_j$；x和x'是足够靠近的。如果可能，选择x和x_0之一在D_i和D_j之间的边界上。请注意，第三点要求至少要对输入值进行排序，最好是对度量进行排序；没有这个，就没有边界的概念，因此不能应用边界值分析。

数字或数字元组定义接近的概念相对简单。尽管有许多距离度量，但对于其他数据类型还不太清楚。即使有数字，也有接近的替代概念，但可能离实际应用还有偏差。在某些示例中，距离1会很近，而在另一些示例中，可能需要更小的距离，例如10^{-5}，因此定义可能依赖于测试人员的领域知识。在测试生成中，可以将靠近替换为足够靠近，并将其视为优化问题；需要一对适当的测试输入，并且它们之间的距离应最小。

下面通过一个示例来帮助读者理解边界值分析。一个简单系统确定在给定月份内向客户购买w单位水和e单位电所收取的费用，其中w和e为非负实数。每单位水的费用为c_1，每单位电的费用为c_2，因此，如果没有折扣，则向客户收取的总费用为$c_1w + c_2e$。但是，如果客户在一个月内购买了至少b单位的水（$w \geqslant b$），则可享受20%的电费折扣。因此，规范包含以下两种情况。

(1) 子域$D_1 = \{(w, e) \in \mathcal{R} \times \mathcal{R} | 0 \leqslant w < b \wedge e \geqslant 0\}$和相应的函数$f_1(w, e) = c_1w + c_2e$。

(2) 子域$D_2 = \{(w, e) \in \mathcal{R} \times \mathcal{R} | w \geqslant b \wedge e \geqslant 0\}$和相应的函数$f_2(w, e) = c_1w + 0.8c_2e$。

此示例表明，如果对测试输入进行了不适当的选择，那么巧合的正确性可能导致无法检测到任意大的边界偏移。此外，它表明在某些情况下，边界值分析的标准方法会导致测试输入具有预先知道的属性，即不会检测到这种边界偏移。找不到边界偏移是由于巧合的正确性。但是，这是可预测的巧合正确性的一个示例。为避免这种情况，以下定义可区分测试的属性。

定理5.1

假设有相邻的子域 D_i 和 D_j，当且仅当 $x \in S_i$ 和 $f_i(x) \neq f_j(x)$ 时，x 才是 (D_i, D_j) 的可区分测试输入。 ♡

这里重要的一点是，如果 $x \in D_i$ 不是 (D_i, D_j) 的判别测试，则它无法检测到 D_i 与 D_j 之间的边界的偏移。在这种情况下，不应该在边界值分析中使用 x 来检查 D_i 和 D_j 之间的边界。

定理5.2

假设有相邻的子域 D_i 和 D_j，测试输入 $x \in D_i$，测试输入 $x' \in D_j$。如果 x 是 (x, x') 的区分测试输入，而 x' 是 (D_i, D_j) 的区分测试输入，则 (x, x') 区分 (D_i, D_j)。 ♡

在边界值分析中，当生成测试输入以检查相邻子域 D_i 和 D_j 之间的边界时，应该生成形式为 (x, x') 的测试输入对，使得 (x, x') 对 (D_i, D_j) 进行区分；并且 x 和 x' 靠得很近。自然，要求紧密靠近有意义的条件等同于能够应用边界值分析所需的条件。对于假设的故障，可以确定输入的条件以检测故障。现在展示如何通过这些观察来驱动边界值分析的测试输入生成。

考虑上述示例的规范包含以下两种情况。

(1) 子域 $D_1 = \{(w, e) \in \mathcal{R} \times \mathcal{R} \mid 0 \leqslant w < b \wedge e \geqslant 0\}$ 和对应的函数 $f_1(w, e) = c_1 w + c_2 e$。

(2) 子域 $D_2 = \{(w, e) \in \mathcal{R} \times \mathcal{R} \mid w \geqslant b \wedge e \geqslant 0\}$ 和对应的函数 $f_2(w, e) = c_1 w + 0.8 c_2 e$。

可以生成一对 (x, x')，它通过以下方式进行区分。令 $x = (w, e)$ 和 $x' = (w', e')$，由于边界是 $w = b$，因此很自然地固定 $e = e'$。也可以坚持对于一些小的 $\epsilon > 0$，这两个点相距 ϵ，并且在不失一般性的情况下，假设 $w' > w$。因此，对于 $w < b$ 和 $w + \epsilon \geqslant b$ 的两个点是 $x = (w, e)$ 和 $x' = (w + \epsilon, e)$。现在想要 $f_1(x) \neq f_2(x)$ 和 $f_1(x') \neq f_2(x')$，这约简到 $c_1 w + c_2 e \neq c_1 w + 0.8 c_2 e$ 和 $c_1(w + \epsilon) + c_2 e \neq c_1(w + \epsilon) + 0.8 c_2 e$，有两个变量 w 和 e。这两个都简单地约简到 $e \neq 0$。在此示例中，通过假设 $e' = e$ 和 $w' = w + \epsilon$ 简化了测试数据生成。然而，如果没有这种简化，自动测试数据生成是可能的，因为在没有这些假设的情况下约束仍然是线性的（假设使用 x 和 x' 之间的距离为 $|w - w'| + |e - e'|$ 的度量）。因此自动测试数据生成问题是一个线性规划问题，可以使用标准算法来解决。如果约束边界不是线性的，则仍然可以使用更通用的搜索和约束求解算法自动生成测试数据。已经看到，在为边界值分析选择测试输入时，为了检查 D_i 和 D_j 之间的边界，应该使用区分 (D_i, D_j) 或区分 (D_j, D_i) 的测试输入，但往往是不够的。

5.1.2 计算边界值分析

计算稳定性是数值算法中普遍需要的特性，稳定性的精确定义取决于上下文。一种是数值线性代数，另一种是通过离散逼近求解微分方程的算法。在数值线性代数中，主要关注的是由于接近各种奇点而导致的不稳定性，此外，在微分方程的数值算法中，关注的是舍入误差的增长和/或初始数据的小波动，这可能导致最终答案与精确解的较大偏差。一些数值算法可能会抑制输入数据中的小波动；其他可能会放大此类波动。称不会放大近似误差的计算为数值稳定的。软件工程的一项常见任务是尝试选择稳健的数值算法，也就是说，输入数据的非常小的变化，不会产生截然不同的结果。通常，算法涉及一种近似方法，并且在某些情况下，可以证明该算法会在一定限度内接近正确的解决方案。当实际使用实数而不是浮点数时，在这种情况下，也不能保证它会收敛到正确的解，因为浮点舍入或截断误差可以被放大，而不是被阻尼和抑制，导致与精确解的偏差快速放大。

首先看看一个简单的数值程序：方差计算。$x_1, x_2, \cdots, x_n$ 的方差公式如下：

$$\sigma^2 = \frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1} = \frac{\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2 /n}{n-1} \tag{5.1}$$

代码5.1是式 (5.1) 的程序算法。这个算法里，因为SumSq和 (Sum∗Sum)/n 可能是非常相似的数字，所以会导致结果的精度远低于用于执行计算的浮点算法的固有精度。因此，该算法不应该在实践中使用，并且已经提出了几种替代的、数值稳定的算法。如果标准偏差相对于平均值很小，这尤其糟糕。方差具有平移不变性，即 $\mathrm{Var}(X-K) = \mathrm{Var}(X)$。可以根据这个特性来构造算法避免该公式中的灾难性计算，具体如下：

代码 5.1　方差计算的程序算法

```
Let n=0, Sum=0, SumSq=0
for each x:
    n=n+1
    Sum=Sum+x
    SumSq=SumSq +x*x
Var=(SumSq-(Sum*Sum ) /n)/(n-1)
```

$$\sigma^2 = \frac{\sum_{i=1}^{n}(x_i - K)^2 - \left(\sum_{i=1}^{n}(x_i - K)\right)^2 /n}{n-1} \tag{5.2}$$

这里，K 越接近真实平均值，计算结果越稳定，但只需选择样本范围内的值即可保证所需的稳定性。如果值 $(x_i - K)$ 小，则其平方和没有问题；相反，如果它们很大，则必然意味着方差也很大。在任何情况下，式 (5.2) 中的第二项总是小于第一项，因此不会发生对消，从而达到算法稳定性。

从航空航天和机器人技术到金融和物理学的现代基础设施，都依赖浮点代码。浮点运算代码的正确处理是计算机编程的一个挑战。例如，实数算术 $a + (b + c) = (a + b) + c$ 中的关联性规则在浮点算术中可能不成立。假定舍入模式是IEEE-754标准中定义的默认

最近舍入模式。该代码可能看起来正确，但是，如果将输入设置为0.999 999 999 999 999 9，将采用分支“if（x <1）”，但随后的“assert（x + 1 <2）”将失败（x + 1 = 2 在这种情况下）。现在，如果在不同的舍入模式下运行相同的代码（如舍入为零），则断言变为有效。为了对这种违反直觉的浮点行为进行推理，可能会认为需要进行正式的语义分析。实际上，一些数值计算程序可以确定浮点约束在舍入至最接近模式下是可满足的，而在其他舍入模式下则无法满足。但是，分析和语义之间的紧密联系，对于涉及浮点算术运算、非线性关系或超越函数的实际代码来说，可能会成为问题。

数值程序分析中经常使用前向、后向和混合稳定性的定义。将数值算法要解决的问题视为将数据 x 映射到解 y 的函数 f。算法的结果 y' 通常会偏离真实解 y。产生误差的主要原因是舍入误差和截断误差。算法的前向误差是结果与解的差值，在这种情况下，$\Delta y = y' - y$。后向误差是最小的 Δx 使得 $f(x + \Delta x) = y'$，换句话说，后向误差告诉算法实际解决了什么问题。如果所有输入 x 的后向误差都很小，则称该算法是后向稳定的。当然，“小”是一个相对术语，其定义取决于上下文。混合稳定性结合了前向误差和后向误差的概念。数值稳定性的通常定义使用一个更一般的概念，称为混合稳定性，它结合了前向误差和后向误差。

本书将数值不稳定性定义为以下问题：由于原始数据收集中的精度限制而导致截断引起的内部错误或外部输入错误发生重大变化。在不稳定的情况下，通常很难相信数据的处理结果。假如软件处理的问题的数学性质不稳定，那么无法通过改进软件开发来避免这种不稳定性，因此将其称为由数字引起的数值不稳定性。软件设计和实施不当也会导致不稳定，称其为由程序引起的不稳定性，因为可以通过改进开发来解决这种实践中的不稳定。除了检测软件中的数值不稳定性外，工具链还可以通过对不稳定性进行分类来自动诊断它们。诊断信息对于开发人员修复不稳定性很有用。如果不稳定性是由数字引起的，则开发人员应考虑在更高级别上改进其软件，这意味重新定义软件要求以缓解或解决不稳定性所带来的数学特性。否则，可以通过更好的数值设计和编程实践来改进软件本身。有限精度数值算法实现的系统存在一定的精度上的偏差，这种偏差会给病态系统扰动分析造成一定的噪声而导致误判。比对扰动算法避免了这种误判。

定义5.3　比对扰动

$S : I \to O$ 及其扰动 $P : I, S \to R$，令 $P_{\text{fix}} : I, S \to R_{\text{fix}}$ 为系统 S 的有限精度数值算法扰动，$P_{\text{inf}} : I, S \to R_{\text{inf}}$ 为系统 S 的精确实数算法扰动，则 $P = P_{\text{fix}} \oplus P_{\text{inf}}$ 为系统 S 的比对扰动，$\oplus : R = R_{\text{fix}} \circ R_{\text{inf}}$，其中 I 为系统 S 的输入域，O 为系统 S 的输出域，R 为扰动 P 的分析结果。 ♣

在二元操作符 $\circ$ 的计算定义中，以T表示系统 S 存在病态性；F表示系统 S 不存在病态性；U表示系统 S 病态性不能明确。当 R_{fix} 为T，R_{inf} 为T时，系统 S 在有限精度数值算法和精确实数算法扰动中都存在病态性，故系统 S 存在病态性，即 $R_{\text{fix}} \circ R_{\text{inf}}$ 为T，R_{fix} 为T，R_{inf} 为F时，系统 S 在有限精度数值算法扰动时存在病态性，系统 S 在精确实数算法扰动时不存在病态性，这是因为有限精度数值算法带来的精度问题造成了噪声，噪声干扰了病

态性的判断，在没有精度噪声的精确实数算法扰动中被证明，是没有病态性的。

比对扰动算法就是将待扰动的系统分别以有限精度数值算法和精确实数算法实现，在输入值一致的情况下，分别以有限精度数值算法和精确实数算法扰动，然后对比扰动结果以衡量该系统的病态性。对比扰动分析流程外部数据来源是需要分析病态性的系统程序片段和特定的扰动实验用例，输出数据是系统病态性分析结果。在整个流程中，最大的操作是对比扰动分析，内含很多子操作和数据。数值算法类型分析操作后和扰动点位置识别与选择后都需要数值算法类型的判断，分为精确实数算法和浮点数算法。在数值算法程序重写、数值算法条件扩展量级扰动生成中需要区别数值算法类型，其他流程操作中不需要区别。特别需要指出的是，在精确实数算法和浮点数算法扰动中，输入数据是相同的。

在对比扰动分析流程中，首先进行数值算法类型的判定，如果是浮点数程序，就用精确实数算法替代浮点数程序生成对应程序，如果是精确实数算法程序，就用浮点数替代精确实数算法生成对应程序。这些程序通过程序结构分析后识别并选择扰动点位置，根据浮点数扰动点进行浮点数条件扩展量级扰动的生成和添加，根据精确实数算法扰动点进行精确实数算法条件扩展量级扰动的生成和添加。然后对这些编辑以后程序结构重新生成对应的程序代码，以特定的扰动实验用例来编译并执行这些新的系统，最后分析这些结果，通过对比精确实数算法和浮点数算法扰动结果的比对来判定系统是否有病态性。

在对比扰动分析中，识别与选择出的扰动点将被插入相应的条件扩展量级扰动代码。扩展量级扰动是对特定的数值执行扰动操作以倍率改变原数值的大小，这种扰动操作并不关心数值的具体计算机实现结构，有扰动量级（Perturbation Magnitude）因子来控制扰动。对一组数值的扩展量级扰动可以由扰动概率（Perturbation Probability）控制其扰动，称为条件扩展量级扰动。扩展量级扰动算法是对单个目标数值进行扰动操作，扰动操作不会涉及计算机中数值算法的具体实现，被扰动的数值将在给定的倍数下进行随机改变。

对数值 $v = 3.1415 \times 10^{-2}$，进行以 $m = 1 \times 10^{-2}$ 的扰动，若 $\delta = -0.5$，则改变 $\delta \times m = -5 \times 10^{-3}$ 倍，即变为 $1 + \delta \times m = 9.95 \times 10^{-1}$ 倍，即 $v' = 3.125\,792\,5 \times 10^{-2}$；以 $m = 1 \times 2^{-1}$ 的扰动，若 $\delta = 0.25$，则改变 $\delta \times m = 1 \times 2^{-3}$ 倍，即 $v' \approx 3.534\,187\,5 \times 10^{-2}$；以 $m = 1 \times 3^{-1}$ 的扰动，若 $\delta = -0.33$，则改变 $\delta \times m \approx -1 \times 3^{-2}$ 倍，即 $v' \approx 2.7924 \times 10^{-2}$。注意到 $m = 1 \times 10^{-2}$ 是在十进制下的扰动，$m = 1 \times 2^{-1}$ 可以是在二进制下的扰动，$m = 1 \times 3^{-1}$ 是在三进制下的扰动。扰动改变值的量级 m 可以随着数值进制的改变而改变，从而使该扰动算法适合不同的数值，而不用考虑该数值在计算机中的具体实现结构。

设定扰动概率而不是设定具体数量有助于以随机性获得更好的覆盖，并胜任未知总共可以扰动的值数目时的大型系统的检测。通过扰动概率，就可以在条件扩展量级扰动中，对一组数值的实际被扩展量级扰动的数值个数进行界定，特别是当这组数值的个数非常大时，效果很好。如给定集合 $V = \{v_1, v_2, \cdots, v_{100\ 000}\}$，有 $\mathrm{card}(V) = 100\ 000$，在条件扩展量级扰动中，扰动概率 $p = 0.65$，以 θ 为一个随机数且 $\theta \in [0, 1]$ 进行。因为 θ 为一个随机数且 θ 在 $[0, 1]$ 上均匀分布，所以 $E(v_i' = \mathrm{perturb}(v_i, \delta, m)) = n \times p = 65\ 000$，即扰动后的集合 V' 中有 65 000 个 v_i' 是 v_i 的扩展量级扰动结果，35 000 个 v_i' 实际上未被执行扩展量级扰动。

δ 为一个随机数且 $\delta \in [1, 1]$，在扰动中，m 的量级决定了数值 v 扰动以后改变的大小，是扩展量级扰动的规模衡量指标。例如在数值 $v = 3.1415 \times 10^{-2}$ 的扰动中，θ 是一个随

机变量，改变倍率影响不大，最主要的影响取决于扰动量级m。分别称以$m=1\times10^{-2}$、$m=1\times2^{-1}$、$m=1\times3^{-1}$的扰动为扰动量级1×10^{-2}、扰动量级1×2^{-1}和扰动量级1×3^{-1}的扩展量级扰动。扩展量级扰动算法以扰动目标值v、扰动量级m为输入参数，以扰动后数值v'为返回值。扰动中，首先获得随机数δ且$\delta\in[-1,1]$，然后计算数值v扰动以后的改变倍率$\delta\times m$，进而得到扰动改变的大小$\delta\times m\times v$，最后得到扰动后的值$v+\delta\times m\times v$。扩展量级扰动针对的是单个数值，扰动因子是扰动量级，即可以对单个数值进行不同程度上的扰动。

对于一组数值，在扰动量级m不变的情况下挑选部分数值进行扰动，以p规定挑选的范围，则需要条件固定量级扰动。条件固定量级扰动针对一组数值，扰动因子是扰动概率p。即可以对一组数值进行不同比例的挑选来进行扰动。需要强调的是，在条件固定量级扰动中，所有被实施的扰动都有相同的扰动量级m值。两个单因子扰动算法可以分别研究单个因子对扰动的影响，但是不能研究扰动概率和扰动量级组合下对扰动的影响。扰动插桩算法中有4个重要步骤：将目标代码进行语法分析并生成抽象语法树，遍历抽象语法树并将找到的可扰动位置信息置入栈中，按栈中信息对抽象语法树进行相应的扰动函数插入工作，最后将修改以后的抽象语法树翻译成新的目标代码。其中抽象语法树的生成和抽象语法树的翻译是自带函数库可以完成的。主要的算法设计工作集中在从抽象语法树中定位可扰动点和向抽象语法树中插入对应扰动函数。接下来将依次描述这些算法。扰动定位标识算法中需要对参数的类型进行判断，区别浮点数和精确实数类型，然后分别进行浮点数或精确实数的扰动，浮点数或精确实数数值类型扰动的判断规则如表5.3所示。

表 5.3　数值类型扰动的判断规则

$p(e)$	规　　则
$p_{\text{fix}}(v)$	$e=v_{\text{fix}}$
$p_{\text{inf}}(v)$	$e=v_{\text{inf}}$
$p_{\text{fix}}(f(e))$	$f:F\to F\wedge e=v_{\text{fix}}$
$p_{\text{inf}}(f(e))$	$f:R\to R\wedge e=v_{\text{inf}}$
$-p_{\text{fix}}(e_1)$	$e=-e_1\wedge e_1=v_{\text{fix}}$
$-p_{\text{inf}}(e_1)$	$e=-e_1\wedge e_1=v_{\text{inf}}$
$p_{\text{fix}}(p_{\text{fix}}(e_1)\otimes p_{\text{fix}}(e_2))$	$e=e_1\otimes e_2\wedge e_1=v_{\text{fix}}\wedge e_2=v_{\text{fix}}$
$p_{\text{inf}}(p_{\text{inf}}(e_1)\otimes p_{\text{fix}}(e_2))$	$e=e_1\otimes e_2\wedge e_1=v_{\text{inf}}\wedge e_2=v_{\text{fix}}$
$p_{\text{inf}}(p_{\text{fix}}(e_1)\otimes p_{\text{inf}}(e_2))$	$e=e_1\otimes e_2\wedge e_1=v_{\text{fix}}\wedge e_2=v_{\text{inf}}$
$p_{\text{inf}}(p_{\text{inf}}(e_1)\otimes p_{\text{inf}}(e_2))$	$e=e_1\otimes e_2\wedge e_1=v_{\text{inf}}\wedge e_2=v_{\text{inf}}$

5.1.3　输出边界值分析

边界值分析可以从输入和输出的角度来检查软件的边界条件。从输出角度来看，边界值分析关注的是软件输出结果的边界情况。在进行边界值分析时，测试人员应该考虑输出数据的上下限，以及这些限制对程序的影响。因此，输出的边界值分析往往体现为某种功能性和可靠性的边界值分析。

输出边界值分析和输入边界值分析有时很容易混淆。前者更关注输出的某种边界反转，如日期程序NextDay关注年和月的反转边界，当然这里特别考虑闰年和不同月份天数带来的影响。对于均值方差程序MeanVar，重点关注均值的正负和零的边界问题，也关注方差零的边界问题。由这两个小程序的输出边界值分析很容易倒推出输入值要求。但对于大规模程序，从输出边界值分析派生有效输入值并不是一件简单的事情。

我们再看看三角形程序Triangle的边界值分析测试的生成过程。在三角形程序Triangle中，除要求边长是整数外，没有给出其他的限制条件。对于输入边界值分析，主要将三角形每条边边长的取范围值设置为[1, 100]。而对于输出边界值分析，则关注等腰三角形、等边三角形、无效三角形和普通三角形的隐形边界值分析。对于隐形边界等腰，需要取恰好是等腰和恰好不是等腰。对于隐形边界等边，需要取恰好是等边和恰好不是等边。对于隐形边界无效，需要取恰好是无效和恰好不是无效。当然输入边界值分析常常也与输出边界值分析结合起来使用。我们首先看以参数a为对象的等腰边界值分析示例，如表5.4所示。读者可以补充思考以参数b和c为对象的等腰边界值分析示例。关于三角形的无效特性的边界值分析如下，依然围绕参数a展开，如表5.5所示，其他留给读者思考。

表 5.4　等腰边界值分析示例

ID	a	b	c	预期输出
1	0	50	50	无效
2	1	50	50	等腰
3	2	50	50	等腰
4	101	50	50	无效
5	100	50	50	无效
6	99	50	50	等腰
7	50	50	50	等边

表 5.5　无效边界值分析示例

ID	a	b	c	预期输出
1	0	2	3	无效
2	1	2	3	无效
3	2	2	3	等腰
4	99	2	3	无效
5	100	2	3	无效
6	101	2	3	无效
7	4	2	3	普通
8	5	2	3	无效
9	6	2	3	无效

近年来研究人员发现，数据驱动的人工智能系统容易受到对抗样本攻击的影响，从而产生错误的识别结果。对抗样本攻击指有意制造出对人工智能系统产生误导的输入数据，

从而使它们产生错误的输出结果。在人脸识别系统中，对抗样本攻击可以通过修改或添加一些干扰信号来实现，从而使系统作出错误判断。以下是一些常见的具体示例。

- 添加噪声：攻击者可以在人脸周围添加噪声以干扰人脸识别系统的判断。
- 特性修改：攻击者可以通过修改人脸的颜色和形状来干扰人脸识别系统的判断。
- 伪装攻击：攻击者可以戴上眼镜或者口罩，以及遮挡自己的脸部特征，从而干扰人脸识别系统的判断。

对抗样本攻击不仅对人脸识别系统的可靠性产生影响，还会对人们的隐私和安全产生威胁。例如，在人脸识别系统中，黑客可以使用对抗样本攻击来欺骗安全检查系统，从而进入受限区域。这种攻击方法可以通过对人脸图像进行微小的修改来实现，例如在人脸图像中添加一些噪声或扭曲。这些修改对于人眼来说几乎不可察觉，却足以欺骗人脸识别系统，从而导致安全漏洞。此外，在金融服务和社交媒体等领域中，对抗样本攻击也可能被用来进行欺诈和虚假宣传等行为。例如，在金融服务领域中，黑客可以使用对抗样本攻击来欺骗风险评估系统，从而获得更高的贷款额度或更低的利率。在社交媒体领域中，对抗样本攻击可以被用来生成虚假的推文或评论，从而误导公众舆论。

本节介绍一种常见的对抗样本攻击生成方法——FGSM（快速梯度符号方法）。FGSM是一种用于生成对抗样本的方法，这些修改过的输入数据旨在欺骗机器学习模型。该方法通过向原始输入数据添加小扰动来运作，导致模型错误分类样本。这个扰动是使用模型的损失函数相对于输入数据的梯度来计算的。FGSM 的基本思想是沿着对损失函数相对于输入的梯度的符号方向扰动输入数据。这可以用数学方式表达为

$$x' = x + \varepsilon \cdot \text{sign}(\nabla_x J(x, y)) \tag{5.3}$$

其中，x是原始输入数据；ε是控制扰动量的小标量值；sign() 是返回其参数的符号函数；$\nabla_x J(x, y)$是模型的损失函数相对于输入数据的梯度。通过添加扰动从而得到新的输入数据x'，称为对抗样本。希望新的输入数据x'能够击穿模型的输出边界，即识别决策面。攻击成功常常就是模型分类错误。对模型而言，就是加了扰动的样本使得模型的损失增大。而所有基于梯度的攻击方法都是基于让损失增大这一点来做的。可以仔细回忆一下，在神经网络的反向传播当中，我们在训练过程时就是沿着梯度方向来更新w、b的值的。这样做可以使得网络往损失减小的方向收敛。

$$\begin{aligned} W_{ij}^{(l)} &= W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) \\ b_i^{(l)} &= b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b) \end{aligned} \tag{5.4}$$

那么现在既然是要使得损失增大，而模型的网络系数又固定不变，唯一可以改变的就是输入，因此就利用损失对输入求导从而“更新”这个输入。

假设有一个简单的图像分类模型，它将28×28像素的灰度图像作为输入并输出10个类别的概率分布。假设要为“3”数字的图像生成一个对抗样本，真实标签为“3”。可以使用反向传播计算损失函数的输入图像的梯度。假设损失函数相对于输入的梯度=[0.1, −0.2, 0.5, ⋯ , 0.3]，则可以根据式(5.4)计算对抗样本，例如设置ε的一个小值为0.1。添加扰动

后，得到一个与原始“3”数字图像略有不同的修改过的图像，但会导致模型高置信度地预测不同的类别。

为了防止对抗样本攻击，研究人员提出了各种防御方法。例如，对抗训练技术。这种方法是在训练模型时，向输入数据中添加一些干扰信号，从而使模型能够更好地抵御对抗样本攻击。例如，在训练过程中，可以向输入图像中添加小的扰动，这些扰动在人类视觉中不易被察觉，但可以使模型更容易地识别对抗性样本。由于对抗训练技术能够在训练过程中增加模型的稳健性，因此它是目前最常用的解决对抗样本攻击的方法之一。还有采用复杂的神经网络模型和算法。这种方法是使用更加复杂的神经网络模型和算法，以提高系统的稳健性。例如，可以使用深度神经网络模型来提高人脸识别系统的性能，同时使用卷积神经网络（CNN）来提高系统的稳健性。CNN 可以在输入图像中提取有用的特征，从而使系统更容易地识别对抗性样本。

5.2 变异故障假设

变异测试也称为变异分析，是一种对测试集的有效性和充分性进行评估的技术，能为研发人员在软件测试的各个阶段提供有效的帮助。变异测试用于评估测试集的适用性，有助于发现系统中的故障。在变异分析的指导下，测试人员可以评价测试的错误检测能力，并辅助构建错误检测能力更强的测试集。变异测试通常是将微小代码语法修改作为故障注入，以此检查定义的测试是否可以检测代码中的故障。变异是程序中的一个小变化。这些变化很小，它们不会影响系统的基本功能，在代码中代表了常见故障模式。

5.2.1 变异分析基本概念

变异的基本思想是构建缺陷并要求测试能够揭示这些缺陷。变异分析一定程度上揭示了形成的缺陷，即使这些特定类型的缺陷在分析的程序中不存在。故障假设形成的变异体代表了感兴趣的故障类型。当测试揭示简单的缺陷时，例如变异体类似于缺陷是简单句法改变的结果，它们通常足够强大，可以揭示更复杂的缺陷。后续从理论和实践两方面分析，揭示使用的简单缺陷类型的测试往往也揭示了更复杂的故障类型。因此，变异有助于揭示由所使用的简单和复杂类型的故障组成的更广泛的故障类别。当然，实践中让变异测试生成代表被测程序所有可能缺陷的变异体的策略并不可行，传统变异测试一般通过生成与原有程序差异极小的变异体来充分模拟被测软件的所有可能缺陷。下面首先介绍变异分析的两个重要基本假设，1978 年由变异分析奠基人 Richard DeMillo 提出。

假设 1（熟练程序员假设）：即假设熟练程序员因编程经验较为丰富，编写出的有缺陷代码与正确代码非常接近，仅需进行小幅度代码修改就可以完成缺陷的移除。基于该假设，变异测试仅需通过对被测程序进行小幅度代码修改就可以模拟熟练程序员的实际编程缺陷行为。

假设 2（耦合效应假设）：该假设关注软件缺陷类型，若测试可以检测出简单缺陷，则该测试也易于检测到更为复杂的缺陷。后续对简单缺陷和复杂缺陷进行了定义，即简单缺陷是仅在原有程序上执行单一语法修改形成的缺陷，而复杂缺陷是在原有程序上依次执行

多次单一语法修改形成的缺陷。

根据上述定义可以进一步将变异体细分为简单变异体和复杂变异体，同时在“假设2”的基础上提出变异耦合效应。复杂变异体与简单变异体间存在变异耦合效应是指若测试集可以检测出所有简单变异体，则该测试集也可以检测出绝大部分的复杂变异体。该假设为变异测试分析中仅考虑简单变异体提供了重要的理论依据。研究人员进一步通过实证研究对“假设2”的合理性进行了验证。测试人员会尽可能模拟各种潜在的故障场景，因而会产生大量的变异程序。同时，变异测试要求测试人员编写或工具自动生成大量新的测试，来满足对变异体中缺陷的检测。验证程序的运行结果也是一个代价高昂并且需要人工参与的过程，由此也影响了变异测试在生产实践中的应用。此外，由于等价变异程序的不可判定性，如何快速有效地检测和去除源程序的等价变异程序面临重要挑战。下面先介绍一些变异分析的基本概念。

定义5.4　变异体

变异体$\mathcal{P}'$是原程序$\mathcal{P}$进行符合语法规则的微小代码修改程序版本。 ♣

最常见的变异有值变异、语句变异和决策变异等。值变异：这些类型的变异会改变常数或参数的值。原程序为x= 5，变异程序为x= 20。语句变异，这些类型的变异通过删除它们、替换为其他语句或更改语句的顺序来更改代码。原程序为total= x − y，变异程序为 total= x + y。判定变异，这些类型的变异会改变程序中的逻辑或算术运算符。原程序为If(x>y)，变异程序为If(x<y)等。变异的分类通常由变异算子来刻画。

变异体的定义中强调微小代码修改是基于熟练程序员假设的。同时特别强调了符合语法规则是为了得到一个可编译可运行的变异程序。期待测试t对于变异体$\mathcal{P}'$是不通过的，此时称变异体$\mathcal{P}'$被t“杀死”。如果通过，则称变异体$\mathcal{P}'$对于t是存活的。如果存在$t \in T$“杀死”$\mathcal{P}'$，则称T“杀死”$\mathcal{P}'$。在某些情况下，可能无法找到可以“杀死”该变异体的测试。变异生成的程序在行为上与原始程序语义相同，尽管它们之间的语法存在微小差异。这样的变异体被称为等价变异体。

定义5.5　等价变异体

如果不存在$t \in T$能“杀死”$\mathcal{P}'$，则称对于T，$\mathcal{P}'$是等价变异体。 ♣

变异体也称为变异程序。若对于任意T，$\mathcal{P}'$都是等价变异体，则直接称$\mathcal{P}'$是等价变异体。显然，此时$\mathcal{P}$和$\mathcal{P}'$是语法不等价但语义等价的，也就是虽然两个程序语法有所不同，也就是对于任意输入t，$\mathcal{P}$和$\mathcal{P}'$均相同。变异分析还有助于分析测试集的质量，以编写和生成更有效的测试。T“杀死”的变异体越多，可直观认为测试质量越高。定义变异“杀死率”作为评估度量。

定义5.6　变异分数

给定原程序$\mathcal{P}$的一组变异体$\mathcal{P}'_1, \mathcal{P}'_2, \cdots, \mathcal{P}'_n$，测试集$T$的变异“杀死率”也称为变异

分数，定义为

$$\text{变异分数} = \frac{\text{“杀死”的}\mathcal{P}_i'\text{个数}}{n} \tag{5.5}$$ ♣

当设计“杀死”变异体的测试时，某种意义上是在生成强大的测试集。这是因为正在检查是否会在应用的每个位置或相关位置触发故障。在这种情况下，假设能够揭示变异体的测试也能够揭示其他类型的故障。变异体需要检查测试是否能够将感染的程序状态传播到可观察的程序输出。这里，通常要求变异体的失效状态很有可能成为可观察到的。

根据定义的程序行为，可以有不同的变异体“杀死”条件。通常，监控的是针对每个正在运行的测试的所有可观察程序输出：程序错误输出或由程序断言内容。如果变异体执行后的程序状态与原始程序对应的程序状态不同，则称变异体被弱“杀死”；如果原始程序和变异体在其输出中表现出一些可观察到的差异，则变异体将被强“杀死”。总体来说，对于弱变异和强变异，“杀死”一个变异体的条件是程序状态必须改变，而改变的状态不一定需要传播到输出（强变异所要求的）。因此，弱变异不如强变异有效。然而，由于错误传播失败，后续计算可能掩盖变异体引入的状态差异。弱变异测试要求仅满足PIE模型的第一个和第二个条件。而强变异测试要求满足所有三个条件。

基于耦合效应假设，给定原程序$\mathcal{P}$的一组变异体$\mathcal{P}_1', \mathcal{P}_2', \cdots, \mathcal{P}_n'$是变异版本，测试集$T$的“杀死率”越高，代表$T$的检错能力越强。同时也发现，对于不同的变异体序列$\mathcal{P}_1', \mathcal{P}_2', \cdots, \mathcal{P}_n'$，测试集$T$的“杀死率”不同。因此，如何定义合理的变异体就显得非常重要。为了合理定义变异体，引出变异算子的定义如下。

定义5.7　变异算子

在符合程序语法规则的前提下，变异算子定义了生成变异体的转换规则。 ♣

最初的变异分析用于早期程序语言，如FORTRAN。1987年，针对FORTRAN 77定义了22种变异算子，这22种变异算子的设定为随后其他编程语言变异算子的设定提供了重要的指导依据。并为其他程序语言，如Java、C等的变异算子设计提供了基础。

本节介绍一个简单易用的变异测试工具PITest[①]。PITest为Java提供了标准的测试覆盖率，并可快速扩展并与现代测试和构建工具集成。PITest目前提供了一些内置的变异算子，其中大多数是默认激活的。通过将所需运算符的名称传递给变异算子参数，可以覆盖默认集并选择不同的变异算子。PITest的主要功能包括以下几种。

- 变异代码：PITest会生成各种不同的变异版本的代码，比如删除一些语句、修改一些条件判断等。通过生成各种变异版本的代码，PITest可以模拟出各种可能的代码错误和缺陷，从而提高测试用例的覆盖率。
- 运行测试用例：PITest会自动运行现有的测试用例，以检测每个变异版本的代码是否能够通过测试。如果测试用例能够检测到某个变异版本的代码的错误，那么这个变异版本就被视为被“杀死”的。

① https://pitest.org/

- 生成测试报告：PITest 会生成详细的测试报告，包括测试用例的覆盖率、变异体的数量、被“杀死”的变异体的数量等。通过测试报告，可以了解代码的测试质量和覆盖率，并根据报告中的信息来改进测试和代码。

通过使用 PITest，可以提高现有测试用例的覆盖率，发现更多的缺陷，并提高代码质量。为了使配置更容易，PITest 的一些变异算子被归组。变异是在编译器生成的字节码上执行的，而不是在源文件上执行的。这种方法的优点是通常更快、更容易合并到构建中，但有时很难简单地描述变异运算符是如何映射到 Java 源文件进行等效更改的。关于变异算子和工具的使用后续详细介绍。

5.2.2 变异测试优化技术

虽然变异测试能够有效地评估测试集的质量，但它仍然存在若干问题。一个重要的困难是变异分析阐述大量的变异体，使得其执行成本极其昂贵。当然，变异分析还存在测试预言问题和等价变异等问题。测试预言问题指检查每个测试的原始程序输出。严格来说，这不是变异测试独有的问题。在所有形式的测试中都有检查输出的挑战。由于不确定性变异体等效，等效变异体的检测通常涉及额外的人工审查成本。虽然不可能彻底解决这些问题，随着变异测试的现有进展，变异测试的过程可以自动化，并且运行时可以允许合理的可扩展性。变异测试改进技术可以分为两大类：变异体选择优化，在不影响测试效力的前提下，尽可能少地执行变异体，包括选择变异、随机变异、变异体聚类等优化技术；变异体执行优化，通过优化变异体的执行时间来减少变异测试开销。包括变异体检测优化、变异体编译优化和并行执行优化等。本书重点介绍第一类方法。

变异体选择优化基本步骤如下。首先，为待测程序准备测试集，并为待测程序创建变异体。接着，运行测试，比较原程序和变异体的运行结果。若结果不同，则这个变异体将被该测试“杀死”；但如果结果相同，则该测试无法识别变异体中的缺陷。对测试集的其余测试重复上述步骤。其中变异体选择是关键的一个步骤。变异程序的句法修改由一组变异算子决定。变异体数量由待测程序和变异算子共同决定。实践中，变异算子选择代替变异体选择更为简单且可行。

变异测试的主要计算成本是在运行变异程序时产生的。分析为程序生成的变异体的数量，发现它大致与数据引用数量乘以数据对象数量的乘积成正比。通常，即使是很小的程序单元，这也是一个很大的数字。由于每个变异体必须针对至少一个甚至可能多个测试执行，因此变异测试需要大量计算。降低变异测试成本的一种方法是减少创建的变异程序的数量。

变异算子的创建目标之一：根据程序员通常犯的错误诱导简单的语法更改。早期主要针对 22 个变异算子进行了系统性研究。在 Offutt 的实验中，采用了以下变异算子选择策略：表达式/语句选择性变异仅选择具有表达式和语句运算符的变异；替换/语句选择性变异是仅选择替换和语句变异运算符的变异，替换/表达式选择性变异是仅选择替换和表达式变异运算符的变异。表5.6总结了常用变异算子。

变异体可以通过替代故障来评估测试的故障检测能力。大多数研究都暗示使用更多变异体来评估测试的故障检测能力将提高其结论的有效性。在选择了一个变异体子集后，也

可得到可以检测到这个子集中所有变异体的测试。然后在这些测试上执行所有非等价变异并计算变异检测率。变异检测率越高，变异体子集越好。实践中，没有令人信服的证据表明这种方法是否值得信赖。工业界研究了使用变异算子生成的故障与变异体之间的相似性，以及故障与变异体子集之间的相似性。实验结果表明，一部分变异体在评估测试的能力上更类似于实际故障。这一结果发现后，使用子集而不是所有变异体来评估测试的故障检测能力可能更合适。大多数实验都证实了 Offutt 的 5 个基本变异算子非常实用。大多数时候比所有变异体具有更好的评估测试的能力。

表 5.6　Offutt 的 5 个基本变异算子

简写	描　　述	算　子　示　例
ABS	绝对值插入	$\{(e,0),(e,\text{abs}(e)),(e,-\text{abs}(e))\}$
AOR	算术操作符替换	$\{((a\ \text{op}\ b),a),((a\ \text{op}\ b),\ b),(x,y)\mid x,y\in\{+,-,*,/,\%\}\wedge x\neq y\}$
LCR	逻辑连接符替换	$\{((a\ \text{op}\ b),a),((a\ \text{op}\ b),b),((a\ \text{op}\ b),\ \text{false}\ ),((a\ \text{op}\ b),\text{true}),(x,y)\mid x,y\in\{\&,\mid,\wedge,\&\&,\mid\text{I}\}\wedge x\neq y\}$
ROR	关系运算符替换	$\{((a\ \text{op}\ b),\ \text{false}\ ),((a\ \text{op}\ b),\ \text{true}\ ),(x,y)\mid x,y\in\{>,>=,<,<=,==,!=\}\wedge x\neq y\}$
UOI	一元操作符插入	$\{(\ \text{cond}\ ,!\ \text{cond}\ ),(v,-v),(v,\sim v),(v,--v),(v,v--),(v,++v),(v,v++)\}$

早期的实验使用了西门子程序集，当然也有实验表明西门子程序集中的故障比大多数变异体更难检测。子集中的变异体比大多数其他变异体更难被检测到。以 ABS 为例，检测 ABS 产生的变异体需要从与变异表达式相关的输入域的不同部分中选择测试。因此，只有少数测试可以检测到这些变异体。采用变异聚类分析，同一簇中的变异体可以被相似的测试检测出来。因此，如果选择的测试可以检测到一个变异体，那么从同一个簇中选择的变异体也可能被那些测试检测到。因此，可以检测到聚类子集中的大多数变异体。

下面介绍一个变异体选择的理论分析框架，以找到变异样本大小的下限 n。目标是近似变异分数 m，同时在置信区间为 $1-\delta$ 上确保绝对误差不超过 ϵ。也就是说，近似变异分数超出可接受误差范围的概率是 δ。在变异体上运行测试有两种可能：要么会被检测到，要么不会被检测到。因此，变异分数可以建模为随机变量的多次测试的平均值。许多研究发现存在冗余变异体，它们是彼此的副本，甚至等价变异体。假设很少有变异体是负相关的，也就是说，检测到一个变异体则不会检测到另一个变异体。

令随机变量 D_n 表示从样本 n（待确定）中检测到的变异体的数量。变异分数则为 $M_n=\dfrac{D_n}{n}$。随机变量 D_n 建模为代表变异体 X_i 的所有随机变量的总和。即 $D_n=\sum_i^n X_i$，其期望值 $E(M_n)$ 则由 $\dfrac{1}{n}E(D_n)$ 给出。随机变量和的平均值不依赖于它们的独立性，因此 $E(M_n)=m$。方差 $\text{Var}(M_n)$ 由 $\dfrac{1}{n^2}\text{Var}(D_n)$ 给出，可以写成分量随机变量 $X_1,X_2,\cdots,X_n$

为

$$\frac{1}{n^2}\mathrm{Var}(D_n) = \frac{1}{n^2}\sum_{i}^{n}\mathrm{Var}(X_i) + 2\sum_{i<j}^{n}\mathrm{Cov}(X_i, X_j) \tag{5.6}$$

使用前面变异体之间正相关性的简化假设，可得：

$$\sum_{i<j}^{n}\mathrm{Cov}(X_i, X_j) \geqslant 0 \tag{5.7}$$

这说明变异体 $\mathrm{Var}(M_n)$ 的方差大于或等于独立随机变量的方差。下面的不等式形式化了问题的约束条件，即绝对误差超过 ϵ 的概率低于 δ。

$$\Pr[|M_n - m| \geqslant \epsilon] \leqslant \delta \tag{5.8}$$

满足式 (5.8) 的变异样本的下限计算方法可由切比雪夫不等式给出:

$$\forall k : P(|x - \mu| \geqslant k) \leqslant \frac{\sigma^2}{k^2} \tag{5.9}$$

其中 μ 是均值，σ^2 是方差，$k > 0$。替换切比雪夫不等式中的变量，可得：

$$\Pr[|M_n - m| \geqslant \epsilon] \leqslant \frac{\mathrm{Var}(M_n)}{\epsilon^2} \tag{5.10}$$

要确保 $\Pr[|M_n - m| \geqslant \epsilon]$ 小于 δ，因此限制不等式的界限小于或等于 δ。

$$\frac{\mathrm{Var}(M_n)}{\epsilon^2} \leqslant \delta \tag{5.11}$$

通过将 M_n 替换为 $\frac{D_n}{n}$，可得：

$$\frac{\mathrm{Var}(D_n)}{n^2\epsilon^2} \leqslant \delta \tag{5.12}$$

为了寻找能够满足所需精度的最少样本数，需要高估这个最小数字（即 n），以此提高估计的准确性。因此，寻找满足不等式的 n 的保守估计。n 在分母中，因此 $\frac{\mathrm{Var}(D_n)}{n^2\epsilon^2}$ 的较低值对应于 n 的较高值。所以考虑 n 的解，如果 $\mathrm{Var}(D_n)$ 被低估，那么 n 的解将比正确的 $\mathrm{Var}(D_n)$ 对应的解大。变异检测的协方差大于或等于独立随机变量的协方差。因此，假设 D_n 独立，则 $\mathrm{Var}(D_n)$ 项将小于实际值，n 的值将大于实际值。因此，将在变异体是独立的假设下求解不等式，这将提供所需的最大样本量。由于每一个变异体 X_i 对于“杀死”情况是 0-1 分布，结合独立性假设，$D_n = \sum_{i}^{n} X_i$ 则为二项分布。

考虑正在采样的一组变异体，它们中的每一个都可以被给定的测试集“杀死”或者不“杀死”。也就是说，如果 k 是从 N 个变异体的完整种群中“杀死”的变异体子集，那么测试集有 $\frac{k}{N} = m$ 的概率“杀死”选择的变异体子集。这种直觉是基于假设一个随机测试集。不同的变异体被随机测试集“杀死”的概率不同。然而，一旦测试集是固定的，变异体“杀死”或不“杀死”的概率是固定的，以及该测试集“杀死”任何单个变异体的概率是“杀死”变异体与总体数的概率，即 m。D_n 是二项分布，用二项式 (n, m) 分布的方差替换 $\mathrm{Var}(D_n)$ 得到：

$$\mathrm{Var}(D_n) = n \times m(1-m) \Rightarrow n \geqslant \frac{m(1-m)}{\epsilon^2\delta} \tag{5.13}$$

我们知道当$m=\frac{1}{2}$时，样本量n将最大。所以在最坏的情况下，公式可重写为

$$n \geqslant \frac{1}{4\epsilon^2\delta} \tag{5.14}$$

不等式可用于找到下限。也就是说，给定一定的样本量n，以及置信区间$1-\delta$，可以计算公差ϵ。例如，对于$n=1000$，$\delta=0.05$，有$\epsilon=0.07$。请注意，这个界限是一个悲观的下限。变异体被“杀死”的概率受给定测试集的二项分布约束，假如允许近似分布，可依靠比切比雪夫不等式更强大的工具。对于足够大的n，二项分布近似于正态分布。正态分布ϵ由下式给出：

$$\epsilon=\frac{\sigma}{\sqrt{N}}\times z_{1-\frac{\delta}{2}} \tag{5.15}$$

其中，$z_{1-\frac{\delta}{2}}$是正态分布分位数，均值位于约束δ内的概率。也就是说，假设$\sigma^2 \leqslant m(1-m) \leqslant 0.25$，则

$$N \geqslant \left(\frac{z_{1-\frac{\delta}{2}}}{\epsilon}\right)^2 \times 0.25 \tag{5.16}$$

对于$\epsilon=0.01$和$\delta=0.05$，$N \geqslant 9604$，这比使用切比雪夫不等式50 000预测的样本量要小得多。这意味着对于变异数量大于9604的程序，可以保证9604的样本大小将在95%的样本中以99%的准确度接近真实的变异分数。

事实上，变异体彼此越相似，估计真实变异分数所需的样本数就越少。假设有10个互为耦合的变异体。在这种情况下，与一组10个不同的变异体相比，仅选择一个变异体的样本就足以证明是否可以检测到整组变异体。事实上，实验分析表明，比理论预测的样本量要小得多就足以高度准确地估计变异分数。由于变异体彼此相似，因此检测到的变异体的分布具有正的协方差。因此，所需的样本数量严格小于二项分布。请注意，在实际变异算子构造中，变异体往往是相似的。因此，二项式分布提供了所需样本量的上限。

5.2.3 变异分析理论框架

为了理论分析简单起见，假设测试程序的行为是确定性的并且独立于先前的行为。就测试中程序的故障而言，当P对t的行为与S不一致时，称t检测到P故障或缺陷，也就是导致预期行为（即S）与P的行为不一致。$P_0 \in \mathbb{P}$用来标记原始程序，$P_s \in \mathbb{P}$用来标记相对于规格S的正确程序，也就是完美程序。在实践中，S或P_s通常靠经验丰富的工程师审核进而充当测试预言角色。$M \in \mathbb{M} \subseteq \mathbb{P}$指的是从$P_0$生成的变异体，而变异体集合$\mathbb{M}$都是可运行的程序，因而它是$\mathbb{P}$的一个子集。请注意，$P_0$、$P_s$和$M$为抽象表示，暂时与编程语言或变异方法等细节隔离，以便于我们讨论变异分析和缺陷检测的理论结果。进一步阐述变异分析框架中形式化差异前，先引出测试差分器。

定义5.8　测试差分器

一个测试差分器$d: T \times \mathbb{P} \times \mathbb{P} \longrightarrow \{0,1\}$是一个函数

$$d(t,P,P')=\begin{cases}1, & t\text{对于}P\text{和}P'\text{行为不同}\\ 0, & \text{其他}\end{cases} \tag{5.17}$$

♣

测试差分器简明地表示了 P 和 P' 的行为对于 t 是否不同。请注意，这里还没有定义行为差异的具体定义。差异的具体定义只能在上下文中确定。例如，虽然 0.3333 与严格意义上的 1/3 不同，但在某些情况下 0.3333 将被视为与 1/3 相同。测试差分器 d 是分析变异和程序缺陷的一个重要手段，可以将许多重要概念形式化。首先将变异充分性形式化如下：

$$\forall M \in \mathbb{M}, \exists t \in T, d(t, P_0, M) \tag{5.18}$$

所谓变异充分性，就是所有变异体 M 都被至少一个测试 t “杀死”。通过变异充分性的形式化，表明变异充分性不仅由 P_0、M 和 t 决定，而且还由差分器 d 决定。例如，从强变异到弱变异的一系列方法往往就取决于使用的差分器 d。在强变异分析中，当 M 的输出与 t 的原始程序 P_0 的输出不同时，测试 t “杀死” 变异 M。在弱变异分析中，当 M 和 P_0 的内部状态对于 t 不同时，t 就称为 “杀死” M，尽管这种内部状态未必能传播出去构成软件失效。因此，应该仔细考虑 P_0、M、t 和 d 的整体视图，才能详细得到变异分析的相关结论。

一个测试预言 o 和一个差分器 d 在测试中的作用是相似的。o 表示测试程序的正确性，d 表示测试程序之间的差异。事实上，对于所有 $o \in O$，都有适当的 $d \in D$ 和 $P_s \in \mathbb{P}$，使得：

$$\forall t \in T, \forall P \in \mathbb{P}, o(t, P) \longrightarrow \neg d(t, P, P_s) \tag{5.19}$$

这说明，d 可以在 P_s 的辅助下扮演测试预言 o 的角色。例如，程序 P 对测试 t 的正确性不仅由 $o(t, P)$ 决定，而且由 $d(t, P, P_s)$ 编写。但是，o 一般不能扮演 d 的角色。这意味着 d 比 o 更通用，d 可以在没有 o 的情况下使用。也可以简单写为 $\forall M \in \mathbb{M}, \exists t \in T, \neg o(t, M)$。变异充分性基于 t 的 P_0 和 M 之间的差异，而 $d(t, P_0, M)$ 能够准确捕获这种行为差异。

为了形式化描述一组测试的程序之间的行为差异，引入一个测试向量。一个测试向量 $\boldsymbol{t} = \langle t_1, t_2, \cdots, t_n \rangle \in T^n$ 是一个向量，其中 $t_i \in T$。例如，两个测试 $t_1, t_2 \in T$ 可以形成一个测试向量 $\boldsymbol{t} = (t_1, t_2) \in T^2$。$d$ 和 $\boldsymbol{t}$ 下，定义 $\boldsymbol{d}$-向量来描述程序行为差异。

定义5.9　*d*-向量

一个 $\boldsymbol{d}$-向量 $\boldsymbol{d} : T^n \times \mathbb{P} \times \mathbb{P} \longrightarrow \{0, 1\}^n$ 是一个 n 维向量，使得：

$$\boldsymbol{d}(\boldsymbol{t}, P, P') = \langle d(t_1, P, P'), \cdots, d(t_n, P, P') \rangle \tag{5.20}$$

其中，$\boldsymbol{t} \in T^n$,$d \in D$ 和 $P, P' \in \mathbb{P}$。 ♣

一个 $\boldsymbol{d}$-向量 $\boldsymbol{d}(\boldsymbol{t}, P, P')$ 表示 P 和 P' 在 $\boldsymbol{t}$ 中的所有测试行为差异的向量形式。换句话说，$\boldsymbol{d}(\boldsymbol{t}, P, P')$ 有效地表明了测试 $t_i \in \boldsymbol{t}$ 在两个程序 P 和 P' 中表现出不同的行为。例如，如果 $d(t_i, P, P') = 1$ 对于某个测试 t_i，这意味着 P 和 P' 对于特定测试是不同的 t_i 包含在测试集 $\boldsymbol{t}$ 中。

表5.7展示了 P_s、P_0 和 M 关于 d 对 $\boldsymbol{t} =< t_1, t_2, t_3, t_4 >$ 的程序行为。用于测试的程序的每个行为都由一个大写字母抽象出来，d 通过字母差异来表示行为差异。右侧有代表测试程序之间行为差异的 $\boldsymbol{d}$-向量。$\boldsymbol{d}(\boldsymbol{t}, P_s, P_0)$ 等于 $\langle 0, 1, 1, 1 \rangle$,因为 $d(t_1, P_s, P_0) = 0, d(t_2, P_s, P_0) = 1, d(t_3, P_s, P_0) = 1$,$d(t_4, P_s, P_0) = 1$。请注意，$P_s$、$P_0$ 和 M 之间的所有行为差异都由 $\boldsymbol{d}$-向量表示。$\boldsymbol{d}$-向量通过采用向量的数学范数来提供数量上的差异。对于 $\boldsymbol{d}$-向量 $\boldsymbol{d}(\boldsymbol{t}, P_s, P_0) = \langle 0, 1, 1, 1 \rangle$，可以进一步计算定量差为 $0 + 1 + 1 + 1 = 3$，以 d 表示。这意味着 P_s 和 P_0 之间

的行为差异，就给定的$\boldsymbol{t}$和$\boldsymbol{d}$而言，在数量上是3，这写为$\|\boldsymbol{d}(\boldsymbol{t}, P_s, P_0)\| = 3$。

表 5.7　变异理论分析

$\boldsymbol{t}$	P_s	P_0	M	$\boldsymbol{d}(\boldsymbol{t}, P_s, P_0)$	$\boldsymbol{d}(\boldsymbol{t}, P_s, M)$	$\boldsymbol{d}(\boldsymbol{t}, P_0, M)$
t_1	A	A	A	$< 0, 1, 1, 1 >$	$< 0, 1, 1, 0 >$	$< 0, 0, 1, 1 >$
t_2	B	A	A	$< 0, 1, 1, 1 >$	$< 0, 1, 1, 0 >$	$< 0, 0, 1, 1 >$
t_3	C	A	D	$< 0, 1, 1, 1 >$	$< 0, 1, 1, 0 >$	$< 0, 0, 1, 1 >$
t_4	D	A	D	$< 0, 1, 1, 1 >$	$< 0, 1, 1, 0 >$	$< 0, 0, 1, 1 >$

一个向量可以看作一个点在多维空间中的位置。例如，考虑n维空间，向量$\boldsymbol{v} = \langle v_1, v_2, \cdots, v_n\rangle$表示其位置在第$i$维中的点是$v_i$，$i = 1, 2, \cdots, n$。这样，可以将$\boldsymbol{d}$-向量视为多维空间中位置的表示。

定义5.10　位置表示

在对应$\boldsymbol{t}$的多维空间中，程序P'相对于程序P的位置是：

$$\boldsymbol{d}_P^{\boldsymbol{t}}(P') = \boldsymbol{d}(\boldsymbol{t}, P, P') \tag{5.21}$$

其中，$\boldsymbol{d}(\boldsymbol{t}, P, P')$是$\boldsymbol{t}$的$P$和$P'$之间的$\boldsymbol{d}$-向量。 ♣

我们称这个多维空间是由$(\boldsymbol{t}, P, d)$诱导的程序空间，其中$\boldsymbol{t}$对应维度的集合，P对应原点程序，d对应位置差异。换句话说，对于所有$t \in \boldsymbol{t}$，P和P'对于t的行为差异由程序P'相对于原点P的位置标记在维度t。因为d返回0或1，所以每个维度中只有两个可能的位置：与原点相同的位置（即 0）和与原点不同的位置（即 1）。这意味着程序P'在$(\boldsymbol{t}, P, d)$空间中的语义由n位二进制向量$\boldsymbol{d}_P^{\boldsymbol{t}}(P')$表示其中$n = |\boldsymbol{t}|$。

程序空间的起源很重要，因为它决定了程序空间中位置的意义。换句话说，起源决定了程序空间的意义。例如，如果正确的程序P_s用于原点，则位置$\boldsymbol{d}_{P_s}^{\boldsymbol{t}}(P')$表示程序$P'$关于$t$的正确程度。此外，如果原程序$P_0$用作原点，而从$P_0$生成的变异$M$用作目标程序$P'$，则位置$\boldsymbol{d}_{P_0}^{\boldsymbol{t}}(M)$表示对$\boldsymbol{t}$的“杀死”$M$。从概念上讲，程序在$n$维程序空间中的位置将程序行为转换为$n$位二进制字符串。每个位代表程序与空间原点的另一个程序之间的行为差异。

例中，M相对于P_0的位置是$\boldsymbol{d}_{P_0}^{\boldsymbol{t}}(M) = \langle 0, 0, 1, 1\rangle$。这意味着变异体$M$在程序空间$(\boldsymbol{t}, P_0, d)$中由0011表示。$P'$相对于$P$的位置范数自然表示程序空间中$P$到$P'$的距离。例如，$M$相对于$P_0$的位置范数为$\|\boldsymbol{d}_{P_0}^{\boldsymbol{t}}(M)\| = 2$这意味着从$P_0$到$m$的距离是2。对于包含变异体的任意程序$P$，$\|\boldsymbol{d}_{P_s}^{\boldsymbol{t}}(p)\|$表示$P$相对于$\boldsymbol{t}$和$d$。对于任意变异体$m$ $\|\boldsymbol{d}_{P_0}^{\boldsymbol{t}}(M)\|$表示对于$\boldsymbol{t}$和$d$“杀死”$m$的容易程度。

在示例中，两个$\boldsymbol{d}$-向量$\boldsymbol{d}(\boldsymbol{t}, P_s, P_0)$和$\boldsymbol{d}(\boldsymbol{t}, P_s, m)$，表示两个位置$\boldsymbol{d}_{P_s}^{\boldsymbol{t}}(P_0)$和$\boldsymbol{d}_{P_s}^{\boldsymbol{t}}(M)$在同一个程序空间中。也就是说，有一个四维空间，原点是P_s，P_0和m这两个程序分别在$\langle 0, 1, 1, 1\rangle$和$\langle 0, 1, 1, 0\rangle$。$m$比$P_0$更接近原点$P_s$，这意味着$m$比$P_0$更正确。这表明变异可以产生部分正确性修改。这个发现后来被用于基于变异的故障定位和基于变异的程序修复中。

下面讨论两个程序的位置差异与其行为差异之间的关系。考虑在同一个程序空间中的两个任意位置。由于程序的位置表示程序相对于原点的行为差异，因此可以预期程序的位

置和行为之间存在关系。例如，如果两个程序在一维中处于不同的位置，那么意味着这两个程序对于相应的测试有不同的行为。

$$(\boldsymbol{d}_P^{\boldsymbol{t}}(P') \neq \boldsymbol{d}_P^{\boldsymbol{t}}(P'')) \Longrightarrow (\boldsymbol{d}(\boldsymbol{t}, P', P'') \neq 0) \tag{5.22}$$

其中，所有 $P, P', P'' \in \mathbb{P}$、$d \in D$ 和 $\boldsymbol{t} \in T^n$。在式 (5.22) 中，左边意味着 P' 和 P'' 在 $(\boldsymbol{t}, P, d)$ 的程序空间中的不同位置。右边意味着 P' 和 P'' 对于 $\boldsymbol{t}$ 有不同的行为。粗略地说，这个公式意味着程序位置的不同导致了程序行为的不同。通过这个公式可得出结论，对于空间中给定的一组测试，处于不同位置的程序具有不同的行为。请注意，这个公式的反向不成立。换言之，即使两个程序在程序空间中的位置相同，这也并不意味着这两个程序对于程序空间对应的测试具有相同的行为。这可以形式化表示如下：

$$(\boldsymbol{d}_P^{\boldsymbol{t}}(P') = \boldsymbol{d}_P^{\boldsymbol{t}}(P'')) \not\Longrightarrow (\boldsymbol{d}(\boldsymbol{t}, P', P'') = 0) \tag{5.23}$$

同样，这适用于所有 $P, P', P'' \in Pd \in D$ 和 $\boldsymbol{t} \in T^n$。在式 (5.23) 中，左边不暗示右边的原因是 P'、P'' 和 P 是彼此不同的情况。在前面的例子中，考虑 $\boldsymbol{t}' = \langle t_3 \rangle$，其中三个程序 P_s、P_0 和 M 有不同的行为。但是，P_0 相对于 P_s 的位置与 m 相对于 P_s 的位置相同，因为 $\boldsymbol{d}_{P_s}^{\boldsymbol{t}'}(P_0) = \boldsymbol{d}_{P_s}^{\boldsymbol{t}'}(M) = \langle 1 \rangle$。

总而言之，程序在程序空间中的位置表明了它相对于程序空间起源的行为差异。位置的含义取决于使用什么程序作为空间的原点。意味着两个位置之间的差异与这些职位中程序的行为差异有关。如果两个程序处于不同的位置，这就保证了两个程序的行为是不同的。但是，如果位置不同，这并不能保证两个程序的行为是相同的。因而位置上的形式关系构成了一个偏序关系。

定义5.11　偏序关系

由 $(\boldsymbol{t}, P, d)$ 定义的程序空间，称位置 $\boldsymbol{d}_P^{\boldsymbol{t}}(P')$ 与位置 $\boldsymbol{d}_P^{\boldsymbol{t}}(P'')$ 对于 $\boldsymbol{t}_d \subseteq \boldsymbol{t}$ 构成偏差关系，如果对于所有 $P', P'' \in \mathbb{P}$，以下条件成立：

(1) $\forall t \in \boldsymbol{t} - \boldsymbol{t_d}, d(t, P, P') = d(t, P, P'')$

(2) $\forall t \in \boldsymbol{t_d}, d(t, P, P') = 0$

(3) $\forall t \in \boldsymbol{t_d}, d(t, P, P'') = 1$

我们记为 $\boldsymbol{d}_P^{\boldsymbol{t}}(P') \xrightarrow{\boldsymbol{t_d}} \boldsymbol{d}_P^{\boldsymbol{t}}(P'')$ 或简单地记为 $P' \xrightarrow{\boldsymbol{t_d}} P''$。 ♣

换句话说，P' 的位置除了 $\boldsymbol{t_d}$ 之外的所有维度都与 P'' 的位置相同。P' 在 $\boldsymbol{t_d}$ 维度的位置中与原点相同。P'' 在 $\boldsymbol{t_d}$ 中的位置尺寸与原点相反。这里，$\boldsymbol{t_d}$ 表示每个测试 $t \in \boldsymbol{t}$，其中 $d(t, P', P'') = 1$。当三元组 $(\boldsymbol{t}, P, d)$ 不是主要关注点时，将程序位置 P 的符号缩短为位置向量 $\boldsymbol{p}$。偏序关系阐明测试如何影响测试过程中的位置。

一般来说，测试规模的增加能够更有效地检测故障。测试集的这种增长可由定义 5.11 描述，$\boldsymbol{t_d}$ 使得 P'' 偏离 P'。例如，若 P' 给出正确的程序 P_s，则 $\boldsymbol{t_d}$ 成为测试集检测 P'' 中每个程序的错误。此外，如果 P' 是原始程序，则 $\boldsymbol{t_d}$ 成为“杀死” P'' 中所有变异体的测试。

显然，这种偏差关系是不对称的。对于所有位置 P' 和 P'' 和测试 $\boldsymbol{t_d}$：

$$P' \xrightarrow{\boldsymbol{t_d}} P'' \Longrightarrow \neg(P'' \xrightarrow{\boldsymbol{t_d}} P') \tag{5.24}$$

有趣的是，它不仅是不对称的，而且是传递的。对于所有位置 P'、P''、P''' 和测试 $\boldsymbol{t}_x, \boldsymbol{t}_y$：

$$(P' \xrightarrow{\boldsymbol{t}_x} P'') \wedge (P'' \xrightarrow{\boldsymbol{t}_y} P''') \Longrightarrow (P' \xrightarrow{\boldsymbol{t}_x \text{ 或} \boldsymbol{t}_y} P''') \tag{5.25}$$

也就是说，所有偏离 P'' 的位置将更加偏离 P'。这种偏序位置关系的传递性与变异体的冗余密切相关。例如，可能是这样的情况：

$$P_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \xrightarrow{t_3} M_3 \longrightarrow \cdots \xrightarrow{t_n} M_n$$

对于原始程序 P_0，变异体 $M_1, M_2, \cdots, M_n$ 和测试 $t_1, t_2, \cdots, t_n$。满足上述情况，则根据偏序关系传递性，t_1 足以“杀死”所有变异体。此时 M_1，$M_2, \cdots, M_n$ 存在冗余。这样我们给出了明确的方法可以找到冗余变异体。

偏序关系自然地形成位置格。引入偏序位置格 PDL 表示位置及其偏差关系。

定义5.12　偏序位置格PDL

对于 $(\boldsymbol{t}, P, d)$ 给定的程序空间，偏序位置格由一个有向图 $G = (V, E)$ 组成：

(1) $V = \{P' \mid P' = \boldsymbol{d}_P^{\boldsymbol{t}}(P') \text{ 对于 } P' \in \mathbb{P}\}$

(2) $E = \{(P', P'') \mid P' \xrightarrow{t} P'' \text{ 对于单个 } t \in \boldsymbol{t}\}$

其中，$(P', P'') \in E$ 表明自 P' 到 P'' 的有向边。 ♣

上述定义说明：① 节点集合包括程序空间中对应于 $(\boldsymbol{t}, P, d)$ 的所有可能位置；② 边集合包括每个有向边与单个测试具有偏序关系的位置。PDL 通过位置和偏序关系表示 n 维程序空间的框架。请注意，n 维程序空间包含 2^n 个可能位置。

例如，图5.1呈现的偏序位置格 PDL 说明了包含 8 个位置的三维程序空间的框架。最左边矩形框中的箭头表示三个维度 $\boldsymbol{t} = \langle t_1, t_2, t_3 \rangle$ 的方向。主图说明了程序空间中所有可能的位置 $P_0, P_1, \cdots, P_7$ 及其偏序关系，最右侧的值表示位置距原点的距离。

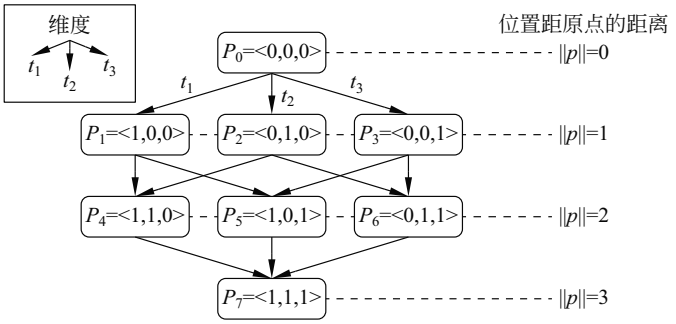

图 5.1　偏序位置格 PDL 示例（1）

下面进一步分析为给定的测试集引入变异集最小化方法，并应用偏序位置格 PDL 来提供更深入的理论理解。如果一个变异体 M 被一组测试 T 中的至少一个测试“杀死”，并且每当 M 被“杀死”时，另一个变异体 M' 总是被“杀死”，那么称为 M 蕴涵 M'，记为 $M \geqslant_T M'$。对于给定的 T，一个极小化的变异体集合 $M_{\min}$ 当且仅当在集合中不存在不同的变异体时互相蕴涵。这表示如果 M 蕴涵 M'，M' 对 M 是冗余的。

例如，考虑 4 个变异体 M_1、M_2、M_3 和 M_4，以及一组测试 $\boldsymbol{t} = \langle t_1, t_2, t_3 \rangle$。表5.8显示

了每个测试“杀死”的变异体。具体来说，表的(i,j)元素是$d(t_i,P_0,M_j)$的值：如果t_i“杀死”M_j则为 1 ，否则为 0 。根据变异体蕴涵的定义，M_1蕴涵M_3和M_4，M_2蕴涵M_4，M_3蕴涵M_4。通过删除冗余的变异体，$M_{\min}=\{M_1,M_2\}$成为极小化的变异体集，相对于测试集$T=\{t_1\ t_2\ t_3\}$。这为变异集最小化提供了坚实的理论基础。

表 5.8　变异集约简示例

$\boldsymbol{t}$	M_1	M_2	M_3	M_4
t_1	1	0	1	1
t_2	0	1	0	1
t_3	0	1	1	1

根据前面的理论，$\boldsymbol{d}^{\boldsymbol{t}}_{P_0}(M)=M$意味着$t\in\boldsymbol{t}$“杀死”了$M$。从表5.8看，$M_1=\langle1,0,0\rangle$，$M_2=\langle0,1,1\rangle,M_3=\langle1,0,1\rangle$和$M_4=\langle1,1,1\rangle$。这提供了一个偏序位置格，如图5.2所示。加粗方框指包含变异体的位置，很容易看出$M_1\longrightarrow M_3$,$M_1\longrightarrow M_4,M_2\longrightarrow M_4$和$M_3\longrightarrow M_4$。请注意，位置之间的偏序关系精确对应于变异体之间的蕴涵关系。对于从P_0、$\boldsymbol{t}$和d在程序空间$(\boldsymbol{t},P_0,d)$的所有$P_0\in P$，$M,M'\in M_{\min}$均成立：

$$\begin{aligned}&P_0\longrightarrow M\longrightarrow M'\\&\Leftrightarrow\exists\boldsymbol{t_d}\subseteq\boldsymbol{t}(\boldsymbol{d}^{\boldsymbol{t}}_{P_0}(M)\xrightarrow{\boldsymbol{t_d}}\boldsymbol{d}^{\boldsymbol{t}}_{P_0}(M'))\wedge(\boldsymbol{d}^{\boldsymbol{t}}_{P_0}(M)\neq0)\\&\Leftrightarrow\forall t\in\boldsymbol{t}:((d(t,P_0,M)=1)\Longrightarrow(d(t,P_0,M')=1))\wedge\exists t\in\boldsymbol{t}:d(t,P_0,M)=1\\&\Leftrightarrow M\longrightarrow M'\text{关于}\boldsymbol{t}\end{aligned}\tag{5.26}$$

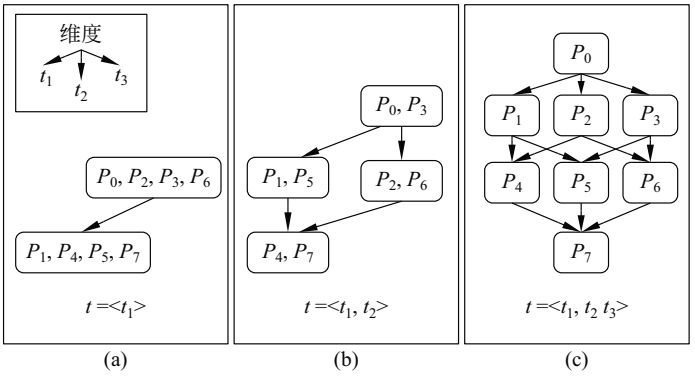

图 5.2　偏序位置格 PDL 示例（2）

这里介绍的变异分析理论框架能够帮助读者更好地理解基于变异的测试方法。通过定义一个测试差异化因素，以将测试范式从程序的正确性转变为程序之间的差异。测试差异化指标清晰简洁地表示测试中程序之间的行为差异。给定测试集后定义一个$\boldsymbol{d}$-向量，以向量形式表示两个程序之间的行为差异。向量表示多维空间中的一个点，从而定义了对应于$\boldsymbol{d}$-向量的程序空间。在程序空间中，程序在每个维度中相对于原点的位置表示程序与该维度对应的测试原点之间的行为差异。清楚地解决了不同位置和行为之间的关系。继续定义位置的推导关系，以表示测试如何影响测试过程中的位置。后续的偏序位置格被定义，用

于为位置及其推导关系提供帮助。这个理论框架是通用的，可以包含变异集最小化的所有理论基础。通过提供最小变异集的理论最大尺寸来改进变异集最小化理论。这个理论框架可以作为关于变异分析的测试经验和理论研究基础。

5.3 逻辑故障假设

软件逻辑故障指在程序代码中存在的逻辑错误或不当的设计，导致程序无法按照预期执行或出现异常行为。这种故障通常不会导致程序崩溃或停止运行，但会导致程序输出错误的结果或不符合预期的行为。逻辑测试通常非常昂贵，因为n变量的逻辑公式将需要2^n个测试进行穷举。逻辑测试的目标是生成小得多的测试集，但在检测故障方面仍然非常有效。回到变异分析，变异分析是一种基于故障的测试策略，它从要测试的程序开始，对其进行大量小的句法更改，从而创建一组变异程序。然后每个变异体在一个正在被评估的测试集上运行，以查看测试数据是否足够全面，可以将原始程序与每个不等价的变异体区分开来。直觉是每个这样的变异体都代表程序的错误版本。结合变异分析和逻辑测试准则来分析其检测条件，并进一步介绍常见逻辑故障的蕴涵结构关系及其应用。

5.3.1 逻辑测试基础

逻辑错误是程序源代码中的错误，它会导致意外的错误行为。逻辑错误被归类为一种运行时错误，可能导致程序产生不正确的输出。它还可能导致程序运行时崩溃。逻辑错误会导致程序无法正常运行。给定一个逻辑布尔公式，应该选择一个测试集，以便表示中的每个出现逻辑单元都表明其对结果的有意义的影响。只要存在不包含任何测试的测试集，这些测试证明了给定文字出现对公式值的有意义影响，则未否定的逻辑单元可以更改为否定的逻辑单元。从某种意义上说，这种策略是直接测试一种特定类型的故障，称为可变否定故障。其中最著名的是传统上用于检测组合电路中所谓的stuck-at-0和stuck-at-1故障的方法。根据经验确定明确设计以保证不存在可变否定错误的测试集是否也会以高概率检测其他类别的错误。

本节首先讨论逻辑表达的最简单形式：布尔逻辑。布尔表达式的计算结果为假(0)或真(1)。条件是布尔值不带布尔运算符的表达式。一个合乎逻辑的决定，或者只是一个决定，是一个布尔表达式组成条件和零个或多个布尔运算符。我们分别用“.”、“+”和“$^-$”来表示布尔运算符 AND、OR 和 NOT，“.”符号通常会被省略。例如，逻辑表达式a OR(b AND NOT c)可以简洁地表示为$a+b\bar{c}$。逻辑表达式中的条件由字母表示，例如a、b、c、$\cdots$，它可以表示基本的逻辑单元：布尔变量。例如“正在运行”表明汽车发动机是否正在运行。

考虑公式$a(b\bar{c}+\bar{d})$。在析取范式中，它可以表示为$ab\bar{c}+a\bar{d}$；在合取范式中，它可以表示为$(a)(b+\bar{d})(\bar{c}+\bar{d})$。这些表示形式不是唯一的，但存在对于给定布尔公式直至交换律而言唯一的规范表示形式。一种这样的表示形式称为规范析取范式。上述公式的规范析取范式表示如下：

$$ab\bar{c}d+ab\bar{c}\bar{d}+abc\bar{d}+a\bar{b}c\bar{d}+a\bar{b}\bar{c}\bar{d} \tag{5.27}$$

逻辑表达式中的变量a可以作为正文字a或负文字$\bar{a}$出现。n个变量的逻辑表达式S的

测试是一个向量$\boldsymbol{t}=(t_1,t_2,\cdots,t_n)$，其中$t_i$是分配给第$i$个变量的值和$t_i\in\mathbb{B}=\{0,1\}$。我们用$S(\boldsymbol{t})$表示$S$的变量分别赋值为$t_1,t_2,\cdots,t_n$时的结果。两个逻辑表达式$S_1$和$S_2$被称为等价的，表示为$S_1\equiv S_2$，如果对于所有测试$\boldsymbol{t}$均有$S_1(\boldsymbol{t})=S_2(\boldsymbol{t})$，则$S_1$和$S_2$被称为等价的，记为$S_1\equiv S_2$。如果$S_1(\boldsymbol{t})\neq S_2(\boldsymbol{t})$，则测试$\boldsymbol{t}$被认为可以区分两个逻辑表达式$S_1$和$S_2$。这种逻辑差分思想是后续故障理论分析的基础。

n个变量的逻辑表达式S可以等价变换成一种析取范式（DNF）作为乘积之和。也可以进一步简化为等价的不冗余析取范式（IDNF）。IDNF可以看作一个极简的DNF，其中没有一个可以省略的文字。然而，不幸的是，这种等价变换是NP-难的。因此仅仅用于理论分析，而非工程实践中。对于DNF逻辑表达式$S=S_1+\cdots+S_m$的测试$\boldsymbol{t}$，如果$S(\boldsymbol{t})=1$，则被称为真点；如果$S(\boldsymbol{t})=0$，则被称为假点。如果对于每个$j\neq i$，$S_i(\boldsymbol{t})=1$但$S_j(\boldsymbol{t})=0$，则$\boldsymbol{t}$被称为S关于第i项S_i的唯一真点（UTP）。

本节说明逻辑测试策略的基本操作，以IDNF形式对布尔规范进行形式化描述。给定布尔逻辑公式和测试集，若测试无法证明对给定文字出现的公式值有意义的影响，则缺陷难以检测到。从某种意义上说，这种策略是直接测试一种特定类型的故障，称为变量否定错误。这里著名的是传统上用于检测硬件组合电路中所谓的stuck-at-0和stuck-at-1故障的算法，来源于电路死焊点的常假（0）和常真（1）故障。

下面首先介绍一种基本的逻辑测试的思想，要求“每个变量值单独影响结果”。这种思想构建了基于故障假设的逻辑测试的一系列方法，并证明检测特定逻辑故障类别是有效的。首先介绍如何针对给定布尔规范自动生成测试集。如图5.3所示，考虑布尔公式$ab(cd+e)$，图中的每一行都描述了测试要满足的测试条件，以保证文字的出现对指定结果具有有意义的影响。例如，第1行指定a对结果1产生有意义测试应满足以下条件：$(a=1,b=1,cd+e=1)$。对于每个这样的条件，测试集必须至少包含一个满足该条件的测试。请注意，单个测试可能满足多个测试条件。例如，第5行和第7行中的测试条件是相同的。此外，可以选择满足两个或多个不同测试条件的测试。例如，测试$(a=1,b=1,c=0,d=1,e=1)$满足第1、3和9行的条件。因此，满足这些测试条件的测试集的基数可能小于测试条件本身的数量。它允许在满足指定条件的测试中进行选择。因此，在图5.3 的第1行中，对变量cd和e的任何赋值都会导致表达式$cd+e$计算为1是可以接受的，并且在5个这样的赋值中没有偏好给定。

行#	字符对输出的直接影响		测试条件				
	字符	输出	*a*	*b*	*c*	*d*	*e*
1	*a*	1	1	1	$cd+e=1$		
2	*a*	0	0	1	$cd+e=1$		
3	*b*	1	1	1	$cd+e=1$		
4	*b*	0	1	0	$cd+e=1$		
5	*c*	1	1	1	1	1	0
6	*c*	0	1	1	0	1	0
7	*d*	1	1	1	1	1	0
8	*d*	0	1	1	1	0	0
9	*e*	1	1	1	$cd=0$		1
10	*e*	0	1	1	$cd=0$		0

图 5.3　逻辑测试示例

可通过设置$c=0$和$d=0$来满足第9行中的$(cd=0)$来解决不确定性。因为c和d与e处于“或”关系。同样，为了在第1行强制执行$(cd+e=1)$，将$(c=1,d=1,e=1)$，因为它们与a处于“和”关系。总是使用这种方法来解决不确定性的问题是测试可能都正确评估，而满足条件的其他测试难以满足。例如，如果$ab(cd+e)$，将选择测试$(a=1,b=1,c=1,d=1,e=1)$来满足第1行。如果实现是$ab(cd+ce+\bar{d}e)$，那么这个测试将导致规范和实现都评估为1。但是，有5个不同的测试满足测试条件。对于$(a=1,b=1,c=0,d=1,e=1)$，规范评估为1，实现评估为0，从而暴露了缺陷。通过对满足给定条件的所有测试的集合进行随机抽样，就有可能检测到更大类别的故障。上述规则确定的某些测试可能不可行。例如，如果变量c表示条件（高度>5），变量e表示条件（高度<4），则c和e在同一个测试中不能都为1。因此，可能会遗漏一些满足有意义影响策略要求的测试，即使存在可行的测试，也有可能使用该规则选择不可行的测试。

基于上述分析，现在介绍算法用于自动生成测试，以实现旨在满足布尔公式规范的实现。其中4个策略通过增加从每组中选择的测试数量来增强影响，其中两个策略还对基本影响策略中未采样的集合进行采样。

在每种情况下，当从集合中选择点时，选择是使用均匀分布随机完成的。此外，一旦选择了一个点，它就会从所有其他集合中删除。前两个变体本质上实施了基本的有意义的影响策略。ONE 是该策略的直接实现，MIN 尝试优化 ONE 策略。

- MIN：从与每个项相关的唯一真点集合中选择一个点，并从公式的近假点集合中选择满足基本有意义的影响策略所需的最小点集合。
- ONE：从与每个项相关的唯一真点集合中选择一个点，从每组近假点$N_{i,j}$中选择一个点。

在有意义的影响策略的以下变体中，从给定的集合中选择多个点。在大多数情况下，所选点数由采样集的大小决定。

- MANY-A：对于与某个项关联的一组2^X唯一真实点，从该集合中选择$\lceil X\rceil$点，并从每组近邻中选择$\lceil X\rceil$点假点$N_{i,j}$，大小为2^X。如果对于某个集合$X=0$，则从该集合中选择一个点。
- MANY-B：对于与某个项关联的一组2^X唯一真实点，从该集合中选择$\lceil X\rceil$点，从每组近邻中选择$\lceil X\rceil$点近假点$N_{i,j}$，大小为2^X。另外，$\lceil X\rceil$点是从大小为2^X的重叠真点集合中选择的，$\lceil X\rceil$点是从大小为2^X的剩余近假点集合中选择的。如果对于某个集合$X=0$，则从该集合中选择一个点。
- MAX-A：选择与每个项相关的唯一真点集合中的每个点，并选择每组近假点$N_{i,j}$中的每个点。
- MAX-B：选择与每个项相关的唯一真点集合中的每个点，并选择每组近假点$N_{i,j}$中的每个点。另外，$\lceil X\rceil$点是从大小为2^X的重叠真点集合中选择的，$\lceil X\rceil$点是从大小为2^X的剩余近假点集合中选择的。如果对于某个集合$X=0$，则从该集合中选择一个点。

Weyuker等从航空防撞系统TCAS II规范中选择了一些较大的约束规范。它们的大小为从5~14个布尔变量不等。对于每个规范，检查变量之间是否存在任何依赖性。例如，如

果变量X表示飞机在某个范围内的高度，另一变量Y表示飞机处于某个不相交的高度范围内，那么两者不可能同时为真。因此，反映这一逻辑的子句$(\overline{XY})$将添加到公式中。每当存在此类变量依赖性时，就以两种方式表示规范：首先添加子句来反映这些依赖性，然后忽略依赖性。Weyuker等通过上述依赖分析，最终提供了一组20个布尔规范供早期的研究人员实验分析。20个布尔规范如下。

(1) $\overline{(ab)}(d\bar{e}\bar{f}+\bar{d}e\bar{f}+\bar{d}\bar{e}f)(ac(d+e)h+a(d+e)\bar{h}+b(e+f))$。

(2) $(a((c+d+e)g+af+c(f+g+h+i))+(a+b)(c+d+e)i)(ab)\overline{(cd)(ce)(de)(fg)(fh)(fi)(gh)(hi)}$。

(3) $(a(\bar{d}+\bar{e}+de\overline{(\bar{f}gh\bar{i}+\bar{g}hi)(\bar{f}glk+\bar{g}\bar{i}k)})+\overline{(\bar{f}gh\bar{i}+\bar{g}hi)(\bar{f}glk+\bar{g}\bar{i}k)}(b+c\bar{m}+f))(a\bar{b}\bar{c}+\bar{a}b\bar{c}+\bar{a}\bar{b}c)}$。

(4) $a(\bar{b}+\bar{c})d+e$。

(5) $a(\bar{b}+\bar{c}+bc\overline{(\bar{f}gh\bar{i}+\bar{g}hi)(\bar{f}glk+\bar{g}\bar{i}k)})+f$。

(6) $(\bar{a}b+a\bar{b})\overline{(cd)}(f\bar{g}\bar{h}+\bar{f}g\bar{h}+\bar{f}\bar{g}\bar{h})(jk)((ac+bd)e(f+(i(gj+hk))))$。

(7) $(\bar{a}b+a\bar{b})\overline{(cd)}\dfrac{(gh)}{(jk)}((ac+bd)e(\bar{i}+\bar{g}\bar{k}+\bar{j}(\bar{h}+\bar{k})))$。

(8) $(\bar{a}b+a\bar{b})(cd)\dfrac{(gh)}{((ac+bd)e(fg+\bar{f}h))}$。

(9) $\overline{(cd)}(\bar{e}f\bar{g}\bar{a}(bc+\bar{b}d))$。

(10) $a\bar{b}\bar{c}d\bar{e}f(g+\bar{g}(h+i))\overline{(jk+\bar{j}l+m)}$。

(11) $a\bar{b}\bar{c}\overline{((f(g+\bar{g}(h+i)))}+f(g+\bar{g}(h+i))d\bar{e})\overline{(jk+\bar{j}l\bar{m})}$。

(12) $a\bar{b}\bar{c}(f(g+\bar{g}(h+i))(\bar{e}\bar{n}+d)+\bar{n}(jk+\bar{j}l\bar{m})$。

(13) $a+b+c+\bar{c}\bar{d}ef\bar{g}\bar{h}+i(j+k)\bar{l}$。

(14) $ac(d+e)h+a(d+e)\bar{h}+b(e+f)$。

(15) $a((c+d+e)g+af+c(f+g+h+i))+(a+b)(c+d+e)i$。

(16) $a(\bar{d}+\bar{e}+de\overline{(\bar{f}gh\bar{i}+\bar{g}hi)(\bar{f}glk+\bar{g}\bar{i}k)})+\overline{(\bar{f}gh\bar{i}+\bar{g}hi)(\bar{f}glk+\bar{g}\bar{i}k)}(b+c\bar{m}+f)$。

(17) $(ac+bd)e(f+(i(gj+hk)))$。

(18) $(ac+bd)e(\bar{i}+\bar{g}\bar{k}+\bar{j}(\bar{h}+\bar{k}))$。

(19) $(ac+bd)e(fg+\bar{f}h)$。

(20) $\bar{e}f\bar{g}\bar{a}(bc+\bar{b}d)$。

5.3.2 逻辑故障结构

基于故障的测试方法首先假设程序员可能犯的某些类型的故障，然后针对这些故障设计测试。与其他测试方法相比，基于故障的测试可以证明不存在假设的故障。基于故障的测试和变异分析之间存在密切关系。变异算子通常用于对潜在故障进行建模。为便于基于故障的测试方法的发展，将同一变异算子诱发的故障归为同一类别，即故障类别或故障类型。一些实证研究观察到，某些故障类别通常比其他故障类别更难检测。这些发现促使研究人员对各种故障类别之间的关系进行分析调查。Kuhn确定了布尔规范中三种类型故障之间的关系，为布尔规范建立故障类别层次结构的第一次尝试。这种层次结构可用于确定应处理故障类别的顺序，以实现更具成本效益的测试。后续研究扩展了三个故障类别，以包括缺失条件的故障类别。本节讨论10类逻辑故障，分为操作符故障（Operator Fault）的前4种和操作数故障（Operand Fault）的后6种两大类。

(1) 操作符引用故障（Operator Reference Fault，ORF）指将逻辑连接词$\vee$替换为$\wedge$。例如原表达式$(x_1 \vee \neg x_2) \vee (x_3 \wedge x_4)$发生了ORF缺陷，则缺陷发生之后的表达式为$(x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$。

(2) 表达式取反故障（Expression Negation Fault，ENF）指子表达式被它的否定形式替换而形成的一个变异体。如原表达式$\neg(x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$发生ENF缺陷，缺陷发生之后的表达式为$(x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$。

(3) 变量取反故障（Variable Negation Fault, VNF）指将表达式的一个条件取反。例如表达式$(x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$发生了VNF缺陷，则缺陷发生之后的表达式为$(\neg x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$。

(4) 关联转移故障（Associative Shift Fault，ASF）指由于对操作符的理解错误而省略了公式中的括号，即部分运算的优先级被改变。例如表达式$(x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$发生了ASF缺陷，则缺陷发生之后的表达式为$x_1 \vee \neg x_2 \wedge (x_3 \wedge x_4)$。

(5) 变量丢失故障（Missing Variable Fault，MVF）指在表达式中条件被省略，需要注意的是MVF的条件可以由$\vee$或者$\wedge$连接。例如表达式$(x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$发生了MVF缺陷，那么缺陷发生之后的表达式为$(x_1 \vee \neg x_2) \wedge (x_3)$。

(6) 变量引用故障（Variable Reference Fault，VRF）指表达式中的条件被另一个可能的条件所替代，可能的条件则是指其变量已经在表达式中出现过。例如表达式$(x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$发生了VRF缺陷，那么缺陷发生之后的表达式为$(x_1 \vee \neg x_2) \wedge (\neg x_1 \wedge x_4)$。

(7) 子句合并故障（Clause Conjunction Fault，CCF）指表达式中的条件c被$c \wedge c'$所取代，其中c'是表达式中一种可能的条件。例如表达式$(x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$发生了CCF缺陷，那么缺陷发生之后的表达式为$(x_1 \vee \neg x_2) \wedge (x_1 \wedge x_3 \wedge x_4)$。

(8) 子句析取故障（Clause Disjunction Fault，CDF）指条件c被$c \wedge c'$替换，其中c'是表达式中一种可能的条件。例如表达式$(x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$发生了CDF缺陷，那么缺陷发生之后的表达式为$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_3 \wedge x_4)$。

(9) 固化0故障（Stuck-At-0 Fault，SA0）指表达式中的条件被0替换。例如表达式$(x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$发生了CDF缺陷，那么缺陷发生之后的表达式为$(x_1 \vee 0) \wedge (x_3 \wedge x_4)$。

(10) 固化1故障（Stuck-At-1 Fault，SA1）指表达中的条件被1替换。例如表达式$(x_1 \vee \neg x_2) \wedge (x_3 \wedge x_4)$发生了CDF缺陷，那么缺陷发生之后的表达式为$(x_1 \vee 1) \wedge (x_3 \wedge x_4)$。

谓词P的检测条件是对P的更改将影响谓词P的值的条件，当且仅当错误的谓词P'的计算结果与正确的谓词P不同时，即$\neg(P \leftrightarrow P')$，或$P \oplus P'$，其中$\oplus$是异或，也称为布尔导数或谓词差异。例如，为了确定检测到变量v的变量否定错误的条件，只需计算$P \oplus P_{\bar{v}}^{v}$，其中P_e^x是谓词P将所有自由出现的变量x替换为表达式e(P_e^x也可写成$P[x := e]$)。其他类型的故障可以用同样的方法分析，让P'成为插入故障的谓词P。给定针对特定规范假设的特定故障，可以计算故障将导致故障的条件，即故障将导致表达式评估为与故障未发生时不同的值的条件。假设$S = p \wedge \bar{q} \vee r$，可以通过计算布尔差值来计算变量$q$的变量否定故障将导致失败的条件：

$$dS_{\bar{q}}^{q} = (p \wedge \bar{q} \vee r) \oplus (p \wedge q \vee r) = p \wedge \bar{r} \tag{5.28}$$

Weyuker描述了一种计算测试条件以检测可变否定错误的算法，并提出了各种策略来为这些条件生成数据。尽管算法旨在检测可变否定错误，但表明的方法也可以检测到其他故障类型。根据检测到特定类型故障的条件开发故障类别的层次结构。然后表明，这种层次结构可用于解释基于故障的测试的经验结果。首先确定不同假设下各种故障类别的检测条件。令S成为析取范式的规范：$S = x1_1 \wedge x1_2 \wedge \cdots \vee x2_1 \wedge x2_2 \cdots \vee xn_1 \wedge xn_2 \cdots$。通常，$xi_j$变量可能不是不同的。例如，可以有$a \wedge b \vee a \wedge c$。然后，例如，将检测到变量$a$的变量否定错误的条件是$S \oplus S^a_{\bar{a}}$。变量取反错误(VNF)的检测条件如下：

$$d_{\text{VNF}} = S \oplus S^{xi_j}_{\neg x x_j} \tag{5.29}$$

变量引用错误(S_{VRF})的检测条件如下：

$$d_{\text{VRF}} = S \oplus S^{xi_j}_{xk_l} \tag{5.30}$$

其中，xk_l是替代xi_j的变量，$xk_l \neq xi_j$。表达式取反错误(S_{ENF})的检测条件如下：

$$d_{\text{ENF}} = S \oplus S^{X_i}_{\neg X_i} \tag{5.31}$$

其中，X_i是子句$xi_1 \wedge xi_2 \wedge \cdots \wedge xi_n$。容易证明$d_{\text{VRF}} \rightarrow d_{\text{VNF}} \rightarrow d_{\text{ENF}}$。

定理5.3

如果在VRF中替换的变量与在VNF中取反的变量相同，则$d_{\text{VRF}} \rightarrow d_{\text{VNF}}$。如果包含在VNF中被否定的变量的表达式在ENF中被否定，则$d_{\text{VNF}} \rightarrow d_{\text{ENF}}$。 ♡

再次回顾MCDC准则：程序中的每个进入点和退出点都至少被调用过一次，程序中逻辑表达式中的每个条件至少对所有可能的结果采取了一次，并且每个条件已被证明通过仅改变它来独立地影响逻辑表达式的结果条件，同时保持所有其他可能的条件。在配对表方法中，为感兴趣的布尔逻辑表达式定义了一个真值表。真值表中的行被编号，为每个条件添加一个附加列。通常，配对表中的许多条目都是空白的。当考虑短路操作时，会出现多个条目的可能性。MCDC 覆盖是通过在真值表中选择足够多的行来获得的，这样每个条件列都选择了一个对。也就是说，对于每一列，选择的行必须包括一对行，这样当相关条件发生变化时，布尔逻辑表达式的值也会发生变化。显式构造真值表有很大的缺点。

一种替代方法是使用布尔差异来为实现 MCDC 的情况制定规范。考虑某个布尔逻辑表达式P中的特定条件x。那么P关于x的布尔差，$dP^x_{\bar{x}} = P \oplus P^x_{\bar{x}}$，给出了$P$依赖的条件关于$x$的价值。因此，通过选择满足$dP^x_{\bar{x}}$的真值分配，然后选择$x$为真然后为假，生成两个满足 MCDC 的测试$x$。可采用配对表方法或布尔差分方法来产生相应的测试。仔细选择这些测试通常可以将测试的总数减少到$n+1$。

考虑一个例子（见表5.9），$A \wedge (B \vee C)$，分别使用配对表方法和布尔差分方法。配对表方法首先为变量A、B和C的所有可能值构建$A \wedge (B \vee C)$的真值表。标记为A、B和C的列显示哪些测试（第一列）可用于显示条件的独立性（第二列）。例如，A的独立性可以通过将测试 1 与测试5配对来显示。

布尔差分方法如下。首先，计算关于A、B和C的布尔差分。针对A、B、C三个变量的布尔差分公式分别如下，$dP^A_{\bar{A}} = A \wedge (B \vee C) \oplus \bar{A} \wedge (B \vee C) = B \vee C$；$dP^B_{\bar{B}} =$

$A\wedge(B\vee C)\oplus A\wedge(\bar{B}\vee C=A\wedge\bar{C})$；$dP_{\bar{C}}^{C}=A\wedge(B\vee C)\oplus A\wedge(B\vee\bar{C}=A\wedge\bar{B})$。测试集生成如下，$T_A$：从$dP_{\bar{A}}^{A}$中，选择$B\vee C$三种可能性和$A$真假，产生三个测试选择（符号表示真值分别分配给$A$、$B$和$C$），分别为$\{111,011\}$、$\{110,010\}$和$\{101,001\}$。$T_B$：从$dP_{\bar{B}}^{B}$中，选择$A\wedge\bar{C}$为真和$B$真假，得到$\{110,100\}$。$T_C$：从$dP_{\bar{C}}^{C}$中，选择$A\wedge\bar{B}$为真和$C$真假，得到$\{101,100\}$。结合上面生成的测试集，有三种可能的测试合并集合，$\{111,110,101,100,011\}(T_A(a),T_B,T_C)$、$\{110,101,100,010\}(T_A(b),T_B,T_C)$或者$\{110,101,100,001\}(T_A(c),T_B,T_C)$。第二种和第三种可能性更可取，因为它们使用最少数量$(n+1)$的测试。

表 5.9　$A\wedge(B\vee C)$ 分析示例

ID	*ABC*	结果	*A*	*B*	*C*
1	111	1	5		
2	110	1	6	4	
3	101	1	7		4
4	100	0		2	3
5	011	0	1		
6	010	0	2		
7	001	0	3		
8	000	0			

某些种类的基于规范的测试依赖于从软件规范中的谓词生成测试的方法。这些方法从逻辑表达式导出各种测试条件，目的是检测不同类型的故障。本节介绍一种计算测试集必须覆盖的条件的方法，以保证检测到特定的故障类别。结果表明，故障类的覆盖层次结构与基于故障的测试的实验结果一致，因此可以解释。该方法还被证明对计算MCDC充分测试有效。

下面，使用F来表示布尔表达式，F_δ表示故障类型δ的一个变异体，其故障关系如定义5.13所示。

定义5.13　故障关系$\geqslant_f$

对任何的布尔表达式F和两个故障类型δ_1和δ_2，如果任何测试检测到对F的δ_1任何可能的错误实现，能够保证该测试能够检测到对F的δ_2任何可能的错误实现，那么δ_1则被认为强于δ_2，表示为$\delta_1\geqslant_f\delta_2$。 ♣

Kapoor等定义如果存在一些引起错误的测试，那么该实现就被认为是错误的。换句话说，当且仅当F_δ是F的非等价变异体时，即$F_\delta\oplus F$是可满足的，F_δ才认为是F的错误实现。任何满足$F_\delta\oplus F$的赋值都是导致故障的测试。给定F，使用$\mathcal{K}(F,\delta)$来表示能够“杀死”F_δ所有可能的错误实现的测试集。那么当且仅当对于F_δ的每一个错误实现，都存在一个诱发故障的测试$t\in T$，能够“杀死”F_δ时，此时记为$t\in\mathcal{K}(F,\delta)$。上述定义可以得到以下关系：

$$\delta_1\geqslant_f\delta_2\Longleftrightarrow\forall F:\mathcal{K}(F,\delta_1)\subseteq\mathcal{K}(F,\delta_2) \tag{5.32}$$

Kapoor 等使用上述定义中的故障关系构建了十种故障类型的结构，认为VNF$\geqslant_f$ENF，MVF$\geqslant_f$VNF，VRF$\geqslant_f$VNF，CCF$\geqslant_f$VRF，CDF$\geqslant_f$VRF，CCF$\geqslant_f$ SA0 $\geqslant_f$VNF 和CDF$\geqslant_f$SA1 $\geqslant_f$VNF。$\delta_1 \geqslant_f \delta_2$ 表示为$\delta_1 \to \delta_2$。那么由定义$\geqslant_f$可以得出，这个结构意味着如果一个测试可以检测到ASF、ORF、MVF、CCF 和CDF 相关的所有可能的故障，那么这个测试也可以保证能够检测到其余5种故障类型的所有可能的故障。

然而，在证明中，Kapoor 和Bowen 忽略了被测布尔规范的变异体可能是等价的。故障关系有6种事实上是不正确的。这些不正确的关系分别是MVF$\geqslant_f$VNF，VNF$\geqslant_f$ENF，SA0$\geqslant_f$VNF，SA1$\geqslant_f$VNF，CCF$\geqslant_f$VRF 以及CDF$\geqslant_f$VRF。不难给出反例来证明这个结论，在此留给读者练习。

逻辑故障F_δ能够被检测当且仅当$F_\delta \oplus F$是可满足的，也只有此时F_δ被认为是一个故障。如果$F_\delta \oplus F$是满足的，那么任何满足$F_\delta \oplus F$的赋值被认为是F_δ的一个诱发故障的测试。因此$F_\delta \oplus F$也是F_δ的一个检测条件。对于任何布尔表达式F和任何故障F_{δ_2}，如果存在一个故障F_{δ_1}，使得$F \oplus F_{\delta_1} \to F \oplus F_{\delta_2}$，那么不难看出$\delta_1 \geqslant f\delta_2$，其中$F_1 \to F_2$是指任何满足$F_1$的赋值也满足$F_2$。

给定一个布尔表达式F和一个条件c，考虑F的以下6种故障类别VNF、SA0、SA1、VRF、CCF 和CDF。这6种故障类别的变异体F_δ分别是将c替换成$\neg c$、0、1、c'、$(c \wedge c')$和$(c \vee c')$，其中c'是与c不相同的条件。使用$F^{C,Z}$来表示各自的变异体。为了便于讨论，给定一个布尔表达式F，$dF^{c,z}$表示$F \oplus F^{c,z}$，其中Z是C的被变异的子表达式。同时dF^c表示$F^{c,0} \oplus F^{c,1}$。如果$F_1 \to F_2$且$F_2 \to F_1$，那么$F_1 \equiv F_2$，这里$\equiv$表示逻辑相等。

定理5.4　布尔差分模型定理

$$dF^{c,z} \equiv (c \oplus z) \wedge dF^c \tag{5.33}$$ ♡

基于上述布尔差分模型，VNF、SA0、SA1、VRF、CCF 和CDF 的检测条件可以仅由c、c'和dF^c表示，如表5.10所示。推导过程在此省略。

表 5.10　检测条件

故障类别	检　测　条　件				
VNF	$dF^{c,\neg c}$	$\equiv$	$(c \oplus \neg c) \wedge dF^c$	$\equiv$	dF^c
SA0	$dF^{c,0}$	$\equiv$	$(c \oplus 0) \wedge dF^c$	$\equiv$	cF^c
SA1	$dF^{c,1}$	$\equiv$	$(c \oplus 1) \wedge dF^c$	$\equiv$	$\neg cF^c$
VRF	$dF^{c,c'}$	$\equiv$			$(c \oplus c') \wedge dF^c$
CCF	$dF^{c,(c \wedge c')}$	$\equiv$	$(c \oplus (c \wedge c')) \wedge dF^c$	$\equiv$	$(c \wedge \neg c')F^c$
CDF	$dF^{c,(c \vee c')}$	$\equiv$	$(c \oplus (c \vee c')) \wedge dF^c$	$\equiv$	$(\neg c \wedge c')F^c$

进一步地，发现了一些现象，即两个故障类型总体上比其他单个故障类型更强或者联合更强。联合蕴涵故障关系定义如下。

在联合增强定义的基础上，进一步确定了一些更有趣的关系，如定理5.5所示。证明过程略。

定义5.14　联合蕴涵故障关系

对任何的布尔表达式F和故障类型δ_1、δ_2和δ_3，如果任何测试检测到对F的δ_1和δ_2任何可能的错误实现，能够保证该测试检测到对F的δ_3任何可能的错误实现，那么δ_1和δ_2则被认为联合强于$(\delta_1 \cup \delta_2) \geqslant_f \delta_3$。 ♣

定理5.5　联合蕴涵定理

(1) (CCF∪CDF) $\geqslant_f$VRF

(2) (SA0∪SA1) $\geqslant_f$VRF

(3) (SA0∪VNF) $\geqslant_f$VRF ♡

综上，一个完善的增强逻辑故障结构如图5.4所示。它表明存在5个核心的故障类别，分别是ASF、ORF、CCF、CDF和ENF。仅适用这5个核心故障类型的测试集，便足以检测到三种故障类型的所有错误。换句话说，非核心故障类型中的测试集是冗余的，因此该层次机构可以帮助识别冗余测试，进而减少测试开销。进一步也可以用于测试优先级排序。同时在这5个核心故障类型中，CCF和CDF应当优先于ASF、ORF和ENF，这样的排序可以更早地检测到错误，因为可以检测到CCF和CDF的测试集同时可以检测到剩余的故障类型的错误，而ASF、ORF和ENF则无法做到。回顾SA0和SA1的定义，以及MCDC和CACC中单一变量改变逻辑表达式的确定性概念，不难发现，SA0+SA1的逻辑故障假设测试等价于MCDC和CACC逻辑覆盖准则。再看SA0和SA1两个故障类型，它们处于故障结构中层，这也能说明为何MCDC在逻辑测试中性价比高并被广泛使用。

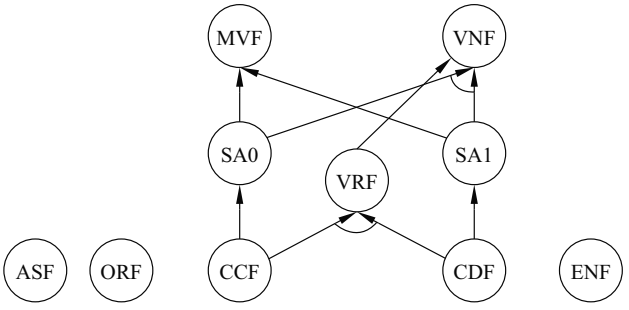

图 5.4　增强逻辑故障结构

5.3.3　逻辑约束求解

SAT（可满足）求解器对软件验证、程序分析、约束求解、人工智能、电子设计自动化和运筹学等领域产生了重大影响。功能强大的求解器已经作为免费和开源软件提供，并且内置于某些编程语言中。首先看布尔逻辑的SAT求解器。输入布尔变量的公式，SAT求解器输出该公式是否可满足。自20世纪60年代引入SAT算法以来，现代SAT求解器已发展成为复杂的系统软件。布尔可满足性通常是一个NP-完全问题。尽管如此，高效且可扩展的SAT算法在21世纪初被开发出来，这极大地提高了解决涉及数万个变量和数百万个

约束的问题实例的能力。

SAT 求解器通常首先将公式转换为合取范式（特别注意，转换成析取范式很难）。Tseitin 变换将任意逻辑公式转换为合取范式。该方法是：① 分配新变量与原公式的子部分进行等价结合；② 使用逻辑蕴涵重写公式使得新变量结合起来；③ 操纵等价关系变换并约简获取最终合取范式。SAT 求解器通常基于 DPLL 等核心算法，但包含许多扩展和功能。通常，SAT 求解器不仅可以提供答案，还可以提供进一步的信息，包括在公式可满足的情况下的变量赋值信息，或者在公式不可满足的情况下提供不可满足的子句的最小集合。

SMT（Satisfiability Modulo Theories）是一种自动化推理技术，它可以用于程序分析和验证。SMT 技术的基本思路是将程序分析问题转换为逻辑公式的判定问题，然后使用 SMT 求解器来判断逻辑公式的可满足性。如果逻辑公式是可满足的，那么就存在一组输入数据，使得程序在这组输入数据下能够执行到目标状态。如果逻辑公式是不可满足的，那么程序就不可能在任何输入数据下执行到目标状态。SMT 求解器是旨在解决实际输入子集的 SMT 问题的工具。Z3 和 cvc5 等 SMT 求解器已被用作计算机科学领域广泛应用的构建块，包括自动定理证明、程序分析、程序验证和软件测试。

SMT 技术将程序分析问题表示为一个逻辑公式的集合，这个集合包含了程序的约束条件和目标状态的约束条件。SMT 求解器会尝试找到这个逻辑公式的一个可满足解，这个解表示了程序可以达到目标状态的一组输入数据。在实际应用中，SMT 技术通常与静态分析技术和动态分析技术相结合，以提高程序分析的精度和效率。例如，可以使用静态分析技术来分析程序的结构和控制流，并将这些信息转换为逻辑公式的约束条件。然后，使用 SMT 技术来求解这个逻辑公式集合，以判断程序是否能够达到目标状态。如果 SMT 求解器无法找到可满足解，那么可以使用动态分析技术来生成测试用例，以进一步探索程序的执行行为。

从形式上来说，SMT 实例是一阶逻辑中的公式，其中一些函数和谓词符号有附加的解释。SMT 就是确定这样的公式是否可满足。换句话说，想象一个布尔可满足性问题 (SAT) 的实例，其中一些二元变量被一组合适的非二元变量上的谓词替换。谓词是非二元变量的二元值函数。示例谓词包括线性不等式 (例如，$3x + 2y - z \geqslant 4$) 或涉及未解释术语和函数符号的等式。这些谓词根据指定的每个相应理论进行分类。例如，实变量的线性不等式使用线性实数算术理论的规则进行评估，而涉及未解释的项和函数符号的谓词使用未解释的等式函数理论的规则进行评估。

大多数常见的 SMT 方法都支持可判定的理论。然而，许多现实世界的系统，只能通过涉及超越函数的实数的非线性算术来建模。这一事实促使 SMT 问题扩展到非线性，例如确定以下方程是否可满足，$b \in \mathbb{B}, x, y \in \mathbb{R}$:

$$(\sin(x)^3 = \cos(\log(y) \cdot x) \vee b \vee -x^2 \geqslant 2.3y) \wedge (\neg b \vee y < -34.4 \vee \exp(x) > \frac{y}{x})$$

然而，这些问题一般来说是无法判定的。此外，实数闭域理论，以及实数的完整一阶理论，可以使用量词消除来判定，这是由 Alfred Tarski 提出的。加法自然数的一阶理论（但不是乘法），称为 Presburger 算术，也是可判定的。由于与常数的乘法可以实现为嵌套加法，因此许多计算机程序中的算术可以使用 Presburger 算术来表达，从而产生可判定的

公式。

SMT 允许在比SAT 更广泛的意义上推理可满足性。也就是说，对于在不同领域理论中的表达式的可满足性，而不仅仅是简单的布尔表达式。本质上，定义了我们可以使用的变量和操作以及组合它们的规则。一些例子是位向量、整数、实数或浮点数。整数理论是抽象数学整数的模型，而不是固定大小的32位或64位的编程整数。同样，实数理论处理的是抽象实数，而不是任何类型的任意精度浮点数。

在很大程度上，程序分析依赖于位向量理论。这里，变量是固定大小的位向量，即变量被视为n位的向量：n位值。该理论还包括常见的算术和逻辑运算，以及连接（例如，将两个32位值连接成一个64位值）和提取（例如，从32位变量中检索最低8位）操作。了解如何利用此类技术进行程序分析的最佳方法是使用实际的SMT 求解器查看一些具体示例。

我们将使用Z3 Theorem Prover，它是 Microsoft 的开源SMT 求解器。选择Z3主要是因为它提供了一个很棒的Python API，即使没有SMT 求解器或SMT-LIB 标准语法的经验，也易于安装、使用和阅读。下面举两个小例子。

例5.1 求解x∗x∗y +(3+x)∗(7+y)== 1337的约束。

我们需要做的第一件事是在 python shell/脚本中导入，并按照Z3的格式进行输入，如代码5.2所示。

代码 5.2 SMT 示例（1）

```
x, y = BitVecs('x y', 32)

solver = Solver()

solver.add(x*x*y +(3+x)*(7+y)== 1337)

if solver.check()== sat:
    m = solver.model()
    print(f"x = {m[x]}\ny = {m[y]}")
```

上述约束求解器得到一组解：x=204374014；y=334463242。这里特别注意，约束求解器得出的解不是唯一的。实际上，如果你多次运行这段代码，很可能会得到不同的解。请记住，SMT 结果唯一保证的是针对一组约束存在某种解决方案，并且它将找到一个具体的解。再看一个例子，见代码5.3。

代码 5.3 SMT 示例（2）

```
bool verify_license(char *code)
{
    size_t sz = strlen(code);
    if(sz != 6)return false;

    char flag = 'A';
    for(int i = 0; i < sz; i++)
    {
        if((code[i] >= '0' && code[i] <= '9')||
            (code[i] >= 'A' && code[i] <= 'Z'))
```

代码 5.3 (续)

```
        {
            flag = flag ^ code[i];
        }
        else
        {
            return false;
        }
    }

    return flag == 'Z';
}
```

调用Z3并按Z3格式输入进行求解，如代码5.4所示。

代码 5.4 SMT示例（3）

```
code = [BitVec(f"c{i}", 8)for i in range(6)]

flag = BitVecVal(ord('A'), 8)

solver = Solver()

for c in code:
    solver.add(Or(
        And(c >= ord('0'), c <= ord('9')),
        And(c >= ord('A'), c <= ord('Z'))
    ))

    flag = flag ^ c

solver.add(flag == ord('Z'))

if solver.check()== sat:
    m = solver.model()
    s = ''.join(chr(m[c].as_long())for c in code)
    print(f"code = {s}")
```

Z3得到一个解：code=E2JJ4X。可以发现这种SMT求解器方法不需要反转任何内容。只需将完全相同的验证算法编码为一组要满足的约束，然后让SMT求解器生成有效密钥。

5.4 本章练习