Table 8.3. Sample test cases for the program solving the equation $ax^2 + bx + c = 0$

test case	condition $d = b^2 - 4ac$	input a	b	c
1	$d > 0$	1	2	-1
2	$d = 0$	1	2	1
3	$d < 0$	1	2	3

a partition into three subsets defined by the three conditions on d. Such partitioned subsets of input domains, or the set of possible input values, are called input sub-domains.

In this case, three test cases can be used, as illustrated in Table 8.3.. Notice that the choice of specific test cases for each sub-domain that satisfy specific conditions is not unique. However, if simple sub-domain coverage is the goal, then it doesn't matter which one we choose, because every point in a given sub-domain receives the same kind of treatment, or belongs to the same equivalent class. In fact, this testing strategy directly corresponds to the checklist-based testing, with the domain partition as the checklist, and each sub-domain corresponding to a single element in the checklist.

This idea can be generalized to support partition-based testing for the general case: We can sample one test case from inside each subset in the partition, thus achieving complete coverage of all the subsets of corresponding equivalence classes. As illustrated in this example, the key to partition-based testing is to define the partitions first and then try to sensitize the test cases to achieve partition coverage.

8.2.2 Partition: Concepts and definitions

Partition is a basic mathematical concept, generally associated with set theory. Formally, a partition of a set S is a division of a set into a collection of subsets $G_1,\ G_2,\ \ldots,\ G_n$ that satisfies the following properties:

- The subsets are *mutually exclusive*, i.e., no two subsets share any common members, or formally,
$$\forall i, j,\ i \neq j \Rightarrow G_i \cap G_j = \emptyset$$
That is, the intersection of any such two subsets must necessarily be empty.

- The subsets are *collectively exhaustive*, i.e., they collectively *cover* or include all the elements in the original set, or formally,
$$\bigcup_{i=1}^{n} G_i = S$$
That is, the union of all subsets (G_i's) is the set S.

Each of these subsets in a partition is called an *equivalent class* (or equivalence class), where each member is the same with respect to some specific property or relation used to define the subset. Formally, the specific relation for the members in an equivalent class is symmetric, transitive, and reflexive, where,

- A *symmetric* relation is one that still holds if the order is changed. For a binary relation R defined on two elements a and b, R is symmetric if $R(a, b) \Rightarrow R(b, a)$. For example, "=" is symmetric; but "$>$" is not.

- A *transitive* relation is one that holds in a relation chain. A transitive binary relation R is one that $R(a, b) \wedge R(b, c) \Rightarrow R(a, c)$. For example, "$>$" is transitive; but "is-mother-of" is not.

- A *reflexive* relation is one that holds on every member by itself. A reflexive binary relation R is one that $R(a, a)$. For example, "=" is reflexive; but "$>$" is not.

Similarly, if a relation is not symmetric, not transitive, or not reflexive, we call it asymmetric, intransitive, or irreflexive, respectively.

With these formal definitions and descriptions of partition in mind, we next examine various uses of partitions and related ideas for software testing (White and Cohen, 1980; Clarke et al., 1982; Weyuker and Jeng, 1991; Beizer, 1990).

8.2.3 Testing decisions and predicates for partition coverage

Since partitions are a special subclass of checklists, the types of partitions can closely resemble the type of checklists we described in Section 8.1. However, we group them in a different way according to the specific decisions associated use in defining different types of partitions, as follows:

- Partitions of some product entities, such as external functions (black-box view), system components (white-box view), etc. The definition of such partitions are typically a simple "member" relation in sets, i.e., $x \in S$ for x as a member of the set S. As a concrete example, whether a component belongs to a sub-system or not is easy to decide. The key is to ensure the partitioned subsets truly form a partition of the original set of all entities. That is, they are mutually exclusive and collectively exhaustive.

- Partitions based on certain properties, relations, and related logical conditions. These partitions can be further divided into two sub-groups:

  - Direct use of logical predicates, through logical variables (those take T/F or True/False values) and connected through logical operators AND ($\wedge$), OR ($\vee$), or NOT ($\neg$).
  - Comparison of numerical variables using some comparison operators, such as "$<$", "$=$", "$\leq$", "$\geq$", "$>$", and "$\neq$". For example, all possible values (a property) of a variable x can be partitions into two subsets $S_1, \text{ and } S_2$ defined by $S_1 \equiv \{x : x < 0\}$ and $S_2 \equiv \{x : x \geq 0\}$.

- Combinations of the above basic types are also commonly used in decision making. For example, the sub-domain of non-negative integers less than 21 can be specified as $(x \in I) \wedge (x \geq 0) \wedge (x < 21)$, where I denotes the set of integers. The values range is also conveniently represented as [0, 21), as we see in mathematical literature.

For the first type of partitions, the testing would be essentially the same as for checklist-based testing: we simply select one item for testing at a time from the subset as a representative of the corresponding equivalent class until we have exhausted all the subsets.

For the other types of partitions above, testing would involve some sensitization in determining the input variables and their values in consultation with the specific conditions or logical predicates used to define each partitioned subset. For example, to satisfy both conditions $(x \geq 0)$ and $(x < 21)$ for the subset [0, 21) above, we might as well select $x = 10$. Notice that in each subset, there might be many elements. Partition-based testing selects one from each subset as the representative for the subset based on the equivalence relations defined.

In addition, the conditions for partitions, or the logical predicate values, are often closely related to either the product specifications (black-box view) or actual implementations (white-box view). For example, we might specify that a function works for certain data types, such as the distinction between integer arithmetic operations and floating point ones in most numerical computing systems. Similarly, many conditional statements in programs may be related to some partitioning conditions, such as those used to guard unexpected input. For example, most arithmetic software packages would not accept divide by 0 as legitimate input. In general, such domain partitions can be represented by simple decisions in product specification or multiple decisions in program code. When multiple decisions are involved, decision trees or decision tables can be used. Therefore, we can consider decision testing and related predicate testing as part of the general partition-based testing.

The above combinations of partition definitions can often be mapped to a series of decisions made along the way of program execution. These decisions in a sequence can be represented by a decision tree, starting from the initial decision, progressing through intermediate decisions, until we finish with a final decision and program termination. Sometimes, the decision can be organized into stages, with the same question asked at each stage regardless of the previous decisions. This would result in a uniform decision tree. Alternatively, some of the later decisions and questions asked may be dependent on the earlier ones, resulting in a non-uniform decision tree.

In either case, we can use the unique path property, from the initial decision node to the last decision outcome to enumerate all the series of decisions, and treat each unique path as a sub-domain and derive test cases to cover each sub-domain. In this way, a decision tree is equivalent to a hierarchical checklist, but with the items of each checklist at each decision point form a partition. For example, in the hierarchical checklist using test scenario classes, scenario numbers, and variation numbers in Section 8.1 can be interpreted as a three-level decision tree, and test cases can be selected to cover these individual variations from specific scenarios in different scenario classes.

The key to deriving coverage based testing for such decisions is to select values to make the test execution follow specific decision paths. For example, if we have decisions using logical variables P and Q in two stages, then we can realize the four combinations:

- $P \wedge Q$ or TT, i.e., P = True and Q = True.
- $P \wedge \neg Q$ or TF, i.e., P = True and Q = False.
- $\neg P \wedge Q$ or FT, i.e., P = False and Q = True.
- $\neg P \wedge \neg Q$ or FF, i.e., P = False and Q = False.

For a specific combination, if other numerical variables are involved, we need to select the numerical variable values to make them satisfy the conditions. For example, to select a test for $(x \geq 0) \wedge (x < 21)$, we can choose $x = 10$, that satisfies both $(x \geq 0)$ and $(x < 21)$. In more complicated situations, we might want to generate a list of candidate test cases based on one condition, and then use other conditions to eliminate certain elements

from this initial candidate list. In this example, we might start with a list like $\{1, 10, 100, \ldots\}$, and the second condition would reduce it to $\{1, 10\}$, and we can finally select 10 or $x = 10$ as our test case to cover this specific decision combination.

8.3 USAGE-BASED STATISTICAL TESTING WITH MUSA'S OPERATIONAL PROFILES (OPS)

One important testing technique, the usage-based statistical testing with Musa's operational profiles (OPs) (Musa, 1993; Musa, 1998), shares the basic model with partition testing techniques, and enhances it to include probabilistic usage information. We next describe Musa OPs and their usage in testing.

8.3.1 The cases for usage-based statistical testing

The many sub-domains for large software systems may include many different operations for each sub-domain. In such situations, the equivalence relation as represented by partition testing described earlier in this chapter represents a uniform sampling of one test point from each sub-domain. However, if operations associated with one particular sub-domain are used more often than others, each underlying defect related to this sub-domain is also more likely to cause more problems to users than problems associated with other sub-domains.

This likelihood for problems to customers, or related system failures defined accordingly, is captured in software product *reliability*. As already introduced in Chapter 2, reliability is defined to be the probability of failure-free operations for a specific time period or a specific input set (Musa et al., 1987; Lyu, 1995a; Tian, 1998). The best way to assess and ensure product reliability during testing is to test the software product as if it is used by customers through the following steps:

- The information related to usage scenarios, patterns, and related usage frequency by target customers and users needs to be collected.
- The information collected above needs to be analyzed and organized into some models — what we call operational profiles (OPs) — for use in testing.
- Testing needs to be performed in accordance with the OPs.
- Testing results can be analyzed to assess product reliability and provide feedback to the testing and the general software development process.

Most of the common testing related activities were described in Chapter 7, and reliability analysis is described in Chapter 22. Therefore, we concentrated on the information collection, OP construction, and it usage in testing in the rest of this chapter.

Like most test activities, the actually testing is typically performed late in the overall product development process, and the model construction could be and should be started much earlier. Usage-based statistical testing actually pushes both these activities to the extremes at both ends as compared with most other testing techniques. On the one hand, the operational profiles (OPs) need to be constructed with customer and user input. It makes more sense to start them right at the requirement analysis phase, or even earlier, in the product planning and market assessment phase. On the other hand, testing according to customer usage scenarios and frequencies captured in OPs can not be performed until most of the product components have been implemented. Therefore, such OP-based testing

Table 8.4. Usage frequencies (hits) and probabilities (% of total) for different file types for SMU/SEAS

File type	Hits	% of total
.gif	438536	57.47%
.html	128869	16.89%
directory	87067	11.41%
.jpg	65876	8.63%
.pdf	10784	1.41%
.class	10055	1.32%
.ps	2737	0.36%
.ppt	2510	0.33%
.css	2008	0.26%
.txt	1597	0.21%
.doc	1567	0.21%
.c	1254	0.16%
.ico	849	0.11%
Cumulative	753709	98.78%
Total	763021	100%

could only be performed in the very late sub-phases of testing, such as in the integration, system, or acceptance testing sub-phases.

8.3.2 Musa OP: Basic ideas

According to Musa (Musa, 1993; Musa, 1998), an operational profile, or an OP for short, is a list of disjoint set of operations and their associated probabilities of occurrence. It is a quantitative characterization of the way a software system is or will be used. As a simple example, consider the usage of `www.seas.smu.edu`, the official web site for the School of Engineering (which used to be called School of Engineering and Applied Science, or SEAS) of Southern Methodist University (SMU/SEAS). Table 8.4. gives the OP for this site, or the number of requests for different types of files by web users over 26 days and the related probabilities.

The "operations" represented in the operational profiles are usually associated with multiple possible test cases or multiple runs. Therefore, we typically assume that each "operation" in an OP can be tested through multiple runs without repeating the exact execution under exactly the same environment. In a sense, each operation corresponds to an individual sub-domain in domain partitions, thus representing a whole equivalence class. In this example, each item in the table, or each operation, represents a type of file requested by a web user, instead of individual web pages. Of course, we could represent each web page as an operation, but it would be at a much finer granularity. When the granularity is too fine, the statistical testing ideas may not be as applicable, because repeated testing may end up repeating a lot of the same test runs, which adds little to test effectiveness. In addition, such fine-granularity OPs would be too large to be practical. For example, the number of individual web pages on an average web site would be more than tens of thousands, while the number of file types is usually limited to a hundred or so, including many variations

definition of ϵ distance, we can detect boundary shift problems in the weak $N \times 1$ input domain testing strategy (Cohen, 1978; White and Cohen, 1980) summarized below:

- For each sub-domain boundary in a n-dimensional input space, n linearly independent boundary points are selected as the ON points.
- The OFF point will be "on the open side of boundary" (White and Cohen, 1980), i.e., it will always receive different processing than that for the ON points. Therefore, we have two situations:
  - If the boundary is a closed boundary with respect to the sub-domain under consideration, the OFF point will be outside the sub-domain or be an exterior point.
  - If the boundary is an open boundary with respect to the sub-domain under consideration, the OFF point will be inside the sub-domain or be an interior point.

  In either of the above cases, the OFF point will be ϵ distance away from the boundary.
- In general, an interior point is also sampled as the representative of the equivalence class representing all the points in the sub-domain under consideration, resulting in $(n + 1) \times b + 1$ test points for each domain with b boundaries.

Weak $N \times 1$ strategy: Other detectable problems

In addition to the boundary shift problem, other problems can be detected as well, which we describe in general terms here. However, the readers might want to refer to concrete examples given later when examining general descriptions below:

- *Closure problems* can be easily detected because such problems will be manifested as ON and OFF points receiving the same processing instead of the expected different processing. For an open boundary, the ON points should receive exterior processing while the OFF point should receive interior processing. A closure problem would cause ON points to receive interior processing. For a closed boundary, the ON points should receive interior processing while the OFF point should receive exterior processing. A closure problem would cause ON points to receive exterior processing.
- *Boundary tilt* and other boundary changes can be easily detected by the ON and OFF points because any such change would result in some or all the ON points not on the boundary anymore. For each of these ON points falling off the boundary, the part of boundary associated with it is either pushed inward or outward, which can be detected the same way as the boundary shift problem we described above.
- *Missing boundary* would be detected by the same processing received by the ON and OFF points as opposed to the different processing expected.
- *Extra boundary* would likely be detected by the different processing associated with some of the ON or OFF points for different boundaries. For each boundary, there will be an OFF point or n ON points which receive interior processing. Let's call these ON or OFF points that receive interior processing "IN" points. All these IN points as well as the selected interior test point should received the same processing.

- Weak 1×1 uses few test points and can detect most of the boundary problems most of the time. Therefore, it should be a primary candidate for boundary testing.
- When high quality requirements need to be met or specific types of problems that weak 1×1 can not address are suspected, weak $N \times 1$ or other testing strategies can be selectively used.
- If inconsistencies exist in some boundaries, strong testing strategies can be used to select one set of test points for each boundary segment.
- When non-linear boundaries are involved, some approximate testing strategies can be used, where one set of test points is used for each segment in the linear approximations of non-linear boundaries.

In addition to its original applications in testing input domain partitions, the basic idea of boundary testing can be applied to other situations where a logical boundary exist between different information processing needs. One such concrete example is the queuing testing we described in Section 9.4, where the upper and lower bounds of the queue buffer can be tested. Additional examples of this nature will be included in Chapter 11 when we apply boundary testing ideas to testing loops.

There are some practical problems with various boundary testing strategies, particularly in the choice of OFF points and related ϵ-limits. OFF point selection for closed domain might extend into undefined territory to cause system crash if the system is not robust enough to guard against unexpected input. In addition, coincidental correctness is common. For example, when different processing gives same results, much of the basis for our problem detection is taken away. These testing strategies are also limited by their simple processing models for more complex interactions. We examine alternative testing strategies based on more complicated models in subsequent chapters.

Problems

9.1 Define the terms: input, input space, input vector, test point, input domain, domain partition, sub-domain, boundary, boundary point, interior point, exterior point, vertex point, under-defined point, over-defined point.

9.2 Give some concrete examples in drawing for the boundary problems in a 2-dimensional space.

9.3 Of the different boundary problems, which ones are observed most often at your work?

9.4 If we have three sub-domains defined by $f(x_1, x_2, \ldots, x_n) < K$, $f(x_1, x_2, \ldots, x_n) = K$, and $f(x_1, x_2, \ldots, x_n) > K$ respectively. Define the boundaries, and discuss how boundary problems would be different in this case.

9.5 So far, we have assumed that each sub-domain is connected. A disconnected sub-domain consists of several disconnected parts or regions. What would be the effect of disconnected sub-domains on boundary problems, and how would you perform boundary testing for them?

9.6 For some of the programs/projects your are working on, find some domain/sub-domain or boundary problems, apply the different boundary testing strategies described in this chapter, and discuss the result.