Chapter 1

Getting Started

1.1 What is a test case

In its most basic form, a *test case* is a simple pair of < input, expected outcome >. If a program under test is expected to compute the square-root of non-negative numbers, then four examples of test cases are shown in Figure 1.1.

TB_1 : $< 0, 0 >$,
TB_2 : $< 25, 5 >$,
TB_3 : $< 40, 6.3245553 >$, and
TB_4 : $< 100.5, 10.024968 >$.

Figure 1.1: Examples of basic test cases.

In state-less systems, where the outcome depends solely on the current input, test cases are very simple in structure as shown above. A program to compute the square-root of non-negative numbers is an example of a state-less system. A compiler for the C programming language is another example of a state-less system.

In state-oriented systems, where the program outcome depends both on the current state of the system and the current input, a test case may consist of a sequence of < input, expected outcome > pairs. A telephone switching system and an automated teller machine (ATM) are examples of state-oriented systems. For an ATM machine, a test case for testing the *withdraw* function is shown in Figure 1.1. Here, we assume that the user has already entered inputs like the cash card and the PIN, and those information have been validated by the ATM.

TS_1 : < check balance, \$500.00 >, < withdraw, "amount?" >,
< \$200.00, "\$200.00" >, < check balance, \$300.00 >.

Figure 1.2: An example of a test case with a sequence of < input, expected outcome >.

In the test case TS_1, "check balance" and "withdraw" in the first, second and fourth tuples represent the pressing of the appropriate keys on the ATM machine. It is assumed that the user account has \$500.00 on it, and the user wants to withdraw an amount of \$200.00. The expected outcome "\$200.00" in the third tuple represents the cash dispensed by the ATM. After the withdrawal operation, the user makes sure that the remaining balance is \$300.00.

For state-oriented systems, most of the test cases include some form of decision and timing in providing input to the system. Thus, a test case may include loops and timers, which we do not show at this moment.

1.2 Expected outcome

The *outcome* of program execution is a complex entity that may include

- values produced by the program
  - outputs for local observation (integer, text, audio, image)
  - outputs (messages) for remote storage, manipulation, or observation
- state change
  - state change of the program
  - state change of the database (due to add/delete/update operations)
- a sequence or set of values which must be interpreted together for the outcome to make sense.

Ideally, the expected outcome should be computed while designing the test case, that is before the program is executed with the selected input. The idea here is that one should be able to compute the expected outcome from an understanding of the program's requirements. Precomputation of the expected outcome will eliminate any implementtaion bias in case the test case is designed by the developer.

In exceptional cases, where it is extremely difficult, impossible, or even undesirable to compute a single expected outcome, one may do the following:

1. execute the program with the selected input
2. observe the actual outcome of program execution
3. verify that the acutal outcome is the expected outcome
4. use the verified acutal outcome as the expected outcome in subsequent runs of the test case.

1.3 Failure, Error and Fault

In the literature on software testing, one may find references to the terms *failure*, *error*, *bug*, and *fault*. In the following, we present these terms as they are understood in the fault-tolerant community.

- *Failure:* A failure is said to occur whenever the external behavior of a system does not conform to that prescribed in the system specification.
- *Error:* An error is a *state* of the system which, in the absence of any corrective action by the system, could lead to a failure which would not be attributed to any event subsequent to the error.
- *Fault:* A fault is the adjudged cause of an error.

Sometimes test engineers and software developers interchangeably use the terms errors, faults and bugs. In this book, we too will use them interchageably. The above definition of failure assumes that the given specification is acceptable to the customer. However, if the specification does not meet the expectations of a customer, then, of course, even a fault-free implementation fails to satisfy the customer. Because of the "human factor" involved in the overall acceptance of a system, it is a difficult task to give a precise definition of fault, error, or bug. In spite of this difficulty, we will stick to the given definition above.

1.4 The Objectives of Testing

The people having a stake in the test process are the programmers, QA engineers, project managers, and the customers. These people view the test process from different perspectives, which can be categorized as four viepoints as explained below.

It does work: After implementing a module as a function, procedure, or method, a programmer may want to test whether or not the module works to in normal circumstances. The same idea applies to an entire system as well–once a system has been integrated, the developers may want to test whether or not the system performs the basic functions. Here, for psychological reason, the test's objective is to show that the system works, rather than it does not work.

It does not work: Once the programmer or the development team is satisfied that a module or the system works to a certain degree, more tests are conducted with the objective of finding faults with the module or the system. Here, the idea is to try to make the module of the system fail.

Reduce the risk of failure: Most of the complex software systems contain faults, which cause the system to fail from time to time. This concept of failing from time to time gives rise to the notion of *failure rate* of the system. As faults are discovered and fixed while performing more and more tests, the failure rate of a system generally decreases. Thus, a higher-level objective of performing tests is to bring down the risk of failing to an acceptable level. Failure rates are mathematically modeled using the idea of software reliability, which will be explained in Chapter **??**.

Reduce the cost of testing: The different kinds of costs associated with the test process include

- the cost of designing, maintaining, and executing the test cases,
- the cost of analyzing the result of executing each and every test case,
- the cost of documenting the test cases, and
- the cost of actually executing the system.

Therefore, the less the number of test cases designed, the less will be the associated cost of testing. However, producing a small number of arbitrary test cases is not a good way of saving cost. The highest level of objective of performing tests is to produce low-risk software with fewer number of test cases. This idea leads us to the concept of *effectiveness of test cases*. Test engineers must judiciously select fewer, effective test cases.

1.5 What is *Complete* Testing

It is not unusual to find people making claims such as "I have exhaustively tested the program." Complete or exhaustive testing means *there are no undiscovered faults at the end of the test phase.* Thus, all problems must be known at the end of complete testing. For most practical systems, complete testing is near impossible, because of the following reasons:

- The domain of possible inputs of a program is too large to be used in the test process. There are both valid inputs and invalid inputs. The program may have a large number of states. There may be timing constraints on the inputs, i.e., an input may be valid at a certain time and invalid at other times. An input value, which is valid but is not properly timed, is called an *inopportune* input. Therefore, the input domain of a system can be very large to be applied to the program.
- The design issues may be too complex to completely test. The design might have included implicit design decisions and assumptions.
- It may not be possible to create all possible execution environments of the system. This becomes more significant when the behavior of the software system depends on the real, outside world, such as weather, temperature, altitude, pressure, geographic topology, and so on.

1.6 The Central Issue in Testing

We must realize that though the outcome of complete testing–that is discovering all faults–is highly desirable, it is a near impossible task and it must not be attempted. The next best thing is to select a subset of the input domain to test a program. Referring to Figure 1.3, let D be the input domain of a program P. Suppose that we select a subset $D1$ of D, i.e. $D1 \subset D$, to test program P. It is possible that $D1$ exercises only a part $P1$, i.e. $P1 \subset P$, of the *execution behavior* of P, in which case faults with the other part $P2$ will go undetected.

By selecting a subset of the input domain $D1$, the test engineer attempts to deduce properties of an entire program P by observing the behavior of a part $P1$ of the entire behavior of P on selected inputs $D1$. Therefore, the *selection* of the subset of the input domain must be done in a systematic and careful manner so that the deduction is as accurate

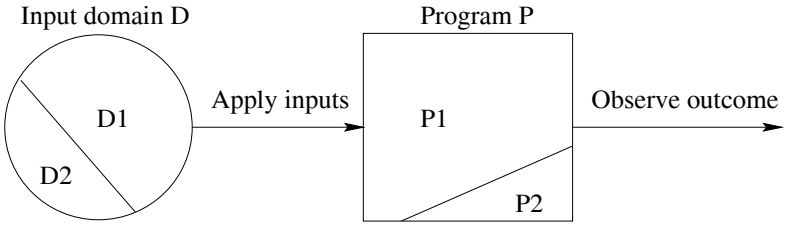

Figure 1.3: A subset of the input domain exercising a subset of the program behavior.

and complete as possible. Thus, lack of complete testing gives rise to the possibility of residual faults in a system.

1.7 Test Activities

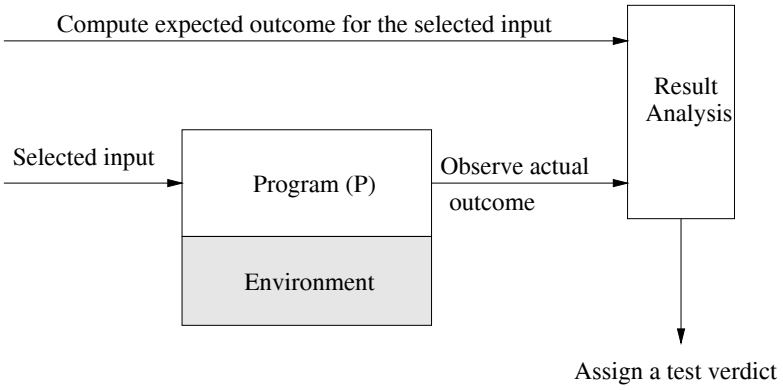

Figure 1.4: Different activities in program testing.

In order to test a program, a test engineer must perform a sequence of activities called test activities. Most of these activities have been shown in Figure 1.4, and explained in the following. These explanations focus on a single test case.

Identify a feature to be tested: The first activity is to identify the *feature* to be tested. The feature defines the intention or *purpose* of designing one or more test cases to ensure that the program supports the feature. A purpose must be associated with every test case.

Suppose that a module under test is meant to sort a given array of integers in the ascending order. Then, some examples of test features are as follows.

- Sort an array of two elements which take on different values and the initial array is unsorted.
- Sort an array of 20 elements where half of the elements have the same value.

Select inputs: The second activity is to select test inputs. Selection of test inputs can be based on the requirements specification, the source code, or our expectations. Test inputs are selected by keeping the test feature in mind. For example, for the first test feature given above, one may select test inputs as follows: size of input array = 2, input array = [20, 10].

Compute expected outcome: The third activity is to compute the expected outcome of the program or the module with the selected inputs. In most of the cases, this can be done from a high-level understanding of the test feature and the specification of the module under test. We must realize that in some cases, it may be extremely difficult, near impossible, or undesirable to compute a single, specific expected outcome. Soon we will explain what to do in such a case.

Setup the execution environment of the progam: The fourth step is to prepare the right execution environment of the program. In this step all the external assumptions behind program execution must be satisfied. A few examples are as follows.

- Initialize the local system external to the program;
- Initialize any remote, external system (e.g. remote partner)
- Sometimes the test engineer may have to wait until the physical environment is set up. For example, wait for the right weather to happen, wait for the time of peak telephone traffic, and so on.

Execute the program: In the fifth step, the test engineer executes the program with the selected inputs and observes the actual outcome of the program. Test execution may not be a simple task like compiling a program or editing a file. To execute a test case, inputs may be supplied to the program at different physical locations at different times. In a later part of the book, we will discuss the issue of *test coordination*.

Analyze the test result: The final test activity is to analyze the result of test execution. Here, the main task is to compare the actual outcome of program execution with the expected outcome. The complexity of comparison depends on the complexity of the data observed. The observed data can be as simple of an integer or a string of characters, or as complex as an image, a video clip or audio. At the end of the analysis step, a test verdict is assigned to the program. There are three kinds of test verdicts *Pass*, *Fail* and *Inconclusive* as explained below.

- If the program produces the expected outcome and the purpose of the test case is satisfied, then a *Pass* verdict is assigned.
- If the program does not produce the expected outcome, then a *Fail* verdict is assigned.
- If the program produes the expected outcome, but the purpose of the test case is not satisfied, then an *Inconclusive* verdict is assigned. Additional tests must be conducted to refine an *Inconclusive* verdict into a *Pass* or a *Fail*.

A *test report* must be written after analyzing the test result. The motivation for writing a test report is to get the fault fixed, if the test revealed a fault. Therefore, the report must be *informative*. A test report may contain the following information:

- Explain how to reproduce the failure.
- Analyze the failure to be able to describe it.
- A pointer to the actual outcome and the test case complete with the input, the expected outcome, and the execution environment.

Chapter 2

Theory of Program Testing

In this chapter, we discuss a few theories put forward in the past two to three decades.

2.1 Goodenough and Gerhart's Theory

In the year 1975, Goodenough and Gerhart published a seminal paper on test data selection in the IEEE Transactions on Software Engineering. In that paper, they gave a fundamental testing concept, identified a few types of program erros, and gave a theory for selecting test data from a program's input domain. Though this theory is not without critiques, it is widely quoted and appreciated in the research community of software testing.

2.1.1 Fundamental Concepts

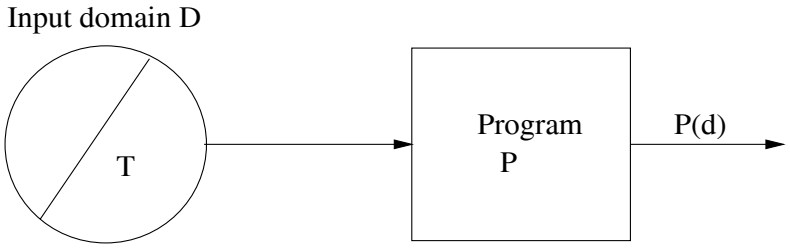

Figure 2.1: Executing a program with a subset of the input domain.

Referring to Figure 2.1, let D be the input domain of a program P. Let $T \subseteq D$. The result of executing P with input $d \in D$ is denoted by $P(d)$.

OK(d): Define a predicate $OK(d)$ which expresses the acceptability of result $P(d)$. Thus, $OK(d) = true$ if and only if $P(d)$ is an acceptable outcome.

SUCCESSFUL(T): For a given $T \subseteq D$, T is a *successful* test, denoted by $SUCCESSFUL(T)$, if and only if $\forall t \in T$, $OK(t)$. Thus, $SUCCESSFUL(T) = true$, if and only if $\forall t \in T$, $OK(t)$.

Ideal Test: Let $T \subseteq D$. T constitutes an *ideal test* if

$$OK(t), \forall t \in T \Rightarrow OK(d), \forall d \in D.$$

An ideal test is interpreted as follows. If from the successful execution of a sample of the input domain we can conclude that the program contains no errors, then the sample constitutes an ideal test. Practitioners may loosely interpret "no error" as "not many errors of severe consequences." The validity of the above definition of an ideal test depends on how "thoroughly" T exercises P. Some people equate *thorough test* with exhaustive or complete test in which case $T = D$.

COMPLETE(T, C): A *thorough test*, T, is defined to be one satisfying $COMPLETE(T, C)$, where $COMPLETE$ is a predicate that defines how some test *selection criteria*, C, is used in selecting a particular set of test data T from D. $COMPLETE(T, C)$ will be defined in a later part of this section. Essentially, C defines what properties of a program must be exercised to constitute a thorough test.

Reliable Criterion: A selection criterion C is *reliable* if and only if either every test selected by C is successful, or no test selected is successful. Thus, reliability refers to consistency.

Valid Criterion: A selection criterion C is *valid* if and only if whenever P is incorrect, C selects at least one test set T which is not successful for P. Thus, validity refers to the ability to produce meaningful results.

Theorem: If C is a reliable and valid criterion, then any test selected by C is an ideal test.

One may be tempted to find a reliable and valid criterion, if it exists, so that all faults can be detected with a small set of test cases. However, there are several difficulties in applying the above theory, as explained in the following.

- Since faults in a program are unknown, it is impossible to prove the reliability and validity of a criterion. A criterion is guaranteed to be both reliable and valid if it selects the entire input domain D. However, this is undesirable and impractical.
- Neither reliability nor validity is preserved during the debugging process, where faults keep disappearing.
- If the program P is correct, then any test will be successful and every selection criterion is reliable and valid.
- If P is not correct, there is in general no way of knowing whether a criterion is ideal without knowing the errors in P.

2.1.2 The Theory of Testing

Let D be the input domain of a program P. Let C denote a set of test predicates. If $d \in D$ satisfies test predicate $c \in C$, then $c(d)$ is said to be true. Selecting data to satisfy a test predicate means selecting data to exercise the condition combination in the course of executing P.

With the above idea in mind, $COMPLETE(T, C)$, where $T \subseteq D$, is defined as follows:

$$\begin{aligned} COMPLETE(T,C) \equiv \quad & (\forall c \in C)(\exists t \in T)c(t) \\ & \wedge \\ & (\forall t \in T)(\exists c \in C)c(t). \end{aligned}$$

What the above theory essentially means is this. For every test predicate, we select a test such that the test predicate is satisfied. Also, for every test selected, there exists a test predicate which is satisfied by the selected test.

The definitions of an ideal test and thoroughness of a test do not reveal any relationship between them. However, we can establish a relationship between the two in the following way.

Let B be the set of faults (or bugs) in a program P, which are revealed by an ideal test T_I. Let a test engineer identify a set of test predicates C_1, and design a set of test cases T_1, such that $COMPLETE(T_1, C_1)$ is satisfied. Let B_1 represent the set of faults revealed by T_1. There is no guarantee that T_1 reveals all the faults. Later, the test engineer identifies a larger set of test prediactes C_2, such that $C_2 \supset C_1$, and designs a new set of test cases T_2, such that $T_2 \supset T_1$ and $COMPLETE(T_2, C_2)$

is satisfied. Let B_2 be the set of faults revealed by T_2. Assuming that the additional test cases selected reveal more faults, we have $B_2 \supset B_1$. If the test engineer repeats this process, he may ultimately identify a set of test predicates C_I and design a set of test cases T_I, such that $COMPLETE(T_I, C_I)$ is satisfied and T_I reveals the entire set of faults B. In this case, T_I is a thorough test satisfying $COMPLETE(T_I, C_I)$ and represents an ideal test set.

2.1.3 Program Errors

Any approach to testing is based on assumptions about how program faults occur. Faults are due to two main reasons as explained below.

- Faults occur due to our inadequate understanding of all conditions that a program must deal with.
- Faults occur due to our failure to realize that certain combinations of conditions require special treatments.

Goodeough and Gerhart classify program faults as follows:

Logic fault: This class of faults mean a program produces incorrect results independent of resource required. That is, the program fails because of the faults present in the program and not because of a lack of resources. Logic faults can be further split into three categories as follows:

Requirements fault: This means our failure to capture the real requirements of the customer.

Design fault: This represents our failure to satisfy an understood requirement.

Construction fault: This represents our failure to satisfy a design.Suppose that a design step says "Sort array A." To sort the array with N elements, one may choose one of the several sorting algorithms. Let

```
for (i = 0; i < N; i++) {
            :
}
```

be the desired `for` loop construct to sort the array. If a programmer writes the `for` loop in the following form

```
for (i = 0; i <= N; i++){
            :
}
```

then there is a construction error in the implementation.

Performance fault: This class of faults lead to a failure of the program to produce expected results within specified or desired resource limitations.

Thorough testa must be able to detect faults arising from any of the above reasons. Test data selection criteria must reflect information derived from each stage of software development. Since, untimately, each type of fault is manifested as an improper effect produced by an implementation, it is useful to categorize the sources of faults in implementation terms. In the following, we categorize faults in implementation terms.

Missing control-flow paths: Intuitively, a control-flow path, or simply a path, is a feasible sequence of instructions in a program. A path may be missing from a program if we fail to identify a condition and specify a path to handle that condition. An example of a missing path is our failure to test for a zero divisor before executing a division. If we fail to recognize that a divisor can take up a zero value, then we will not include a piece of code to handle the special case. Thus, a certain desirable computation will be missing from the program.

Inappropriate path selection: A program executes an inappropriate path if a condition is expressed incorrectly. In Figure 2.1.3, we have shown a desired behavior and an implemented behavior. Both the behaviors are identical except in the condition part of the `if` statement. The `if` part of the implemented behavior contains an additional condition `B`. It is easy to see that both the desired part and the implemented part behave in the same way for all combinations of values of A and B except when $A = 1$ and $B = 0$.

Inappropriate or missing action: There are three instances of this class of fault.

- One may calculate a value using a method that does not necessarily give the correct result. For example, a desired expression is `x = x * w`, whereas it is wrongly written as

Desired behavior	**Implemented behavior**
`if (A) proc1();`	`if (A && B) proc1();`
`else proc2();`	`else proc2();`

Figure 2.2: An example of inappropriate path selection.

`x = x + w`. These two expressions produce identical results for several combinations of `x` and `w`, such as `x = 1.5`, `w = 3`, for example.

- Failing to assign a value to a variable is a kind of missing action.
- Calling a function with the wrong argument list is a kind of inappropriate action.

The main danger due to an inappropriate or missing action is that the action is incorrect only under certain combinations of conditions.

Therefore, to find test data that reliably reveal errors, one must do the following:

- Identify all the conditions relevant to the correct operation of a program.
- Select test data to exercise all possible combinations of these conditions.

The above idea of selecting test data leads us to define the following terms

Test data: Test data are actual values from a program's input domain that collectively satisfy some test selection criterion.

Test predicate: A test predicate is a description of conditions and combinations of conditions relevant to the program's correct operation.

- Test predicates describe *what aspects* of a program are to be tested. Test data cause these aspects to be tested.
- Test predicates are the motivating force for test data selection.

- Components of test predicates arise first and primarily from the specifications for a program.
- As implementations are considered, further conditions and predicates may be added.

2.1.4 Conditions for Reliability

A set of test predicates must at least satisfy the following conditions to have any chance of being reliable. These conditions are key to meaningful testing.

- Every individual branching condition in a program must be represented by an equivalent condition in C.
- Every potential termination condition in the program, for example, an overflow, must be represented by a condition in C.
- Every condition relevant to the correct operation of the program that is implied by the specification and knowledge of program's data structure, must be represented as a condition in C.
- Test predicates must be independent.

2.2 Weyuker and Ostrand's Theory

Goodenough and Gerhart's theory of an ideal test suffers from a fault, namely its dependence on the program being tested. In practice, as program failures are observed, the progarm is debugged to locate faults, and the faults are generally fixed as soon as they are found. During this debugging phase, as the program changes, so does the idealness of a test set. This is because a fault which was revealed before debugging is no more revealed after debugging and fault fixing.

[Complete this part]

2.3 Gourlay's Theory

2.3.1 Definitions

To motivate a theoretical discussion of testing, we begin with an ideal process for software development, which consists of the following steps.

- First, a customer and a development team specify what is needed.
- Second, the development team takes the specification and attempts to write a program to meet the specification.
- A test engineer takes both the specification and the program, and selects a set of test cases. The test cases are based on the specification and the program.
- The program is executed with the selected test data, and the test outcome is compared with the expected outcome.
- If the program fails some of the test cases, the program contains faults.
- If the program passes all the test cases, we call the program to be ready for use.

We focus on the selection of test cases and the interpretation of their results. We assume that the specification is correct, and the specification is the sole arbiter of the correctness of the program. The program is said to be correct if and only if it satisfies the specification. Gourlay's testing theory establishes a relationship between three sets of entities: specifications, programs, and tests. The set of all programs are denoted by $\mathcal{P}$, the set of all specifications by $\mathcal{S}$, and the set of all tests by $\mathcal{T}$. Members of $\mathcal{P}$ will be denoted by p and q, members of $\mathcal{S}$ will be denoted by r and s, and members of $\mathcal{T}$ will be denoted by t and u.

Upper case letters will denote subsets of $\mathcal{P}$, $\mathcal{S}$, and $\mathcal{T}$. For examples, $p \in P \subseteq \mathcal{P}$ and $t \in T \subseteq \mathcal{T}$, where t denotes a single test case. The correctness of a program p with respect to a specification s will be denoted by p corr s. Given s, p, and t, the predicate "$p\ ok(t)\ s$" means that the result of testing p under t is judged successful by specification s. The reader may recall that T denotes a set of test cases, and $p\ ok(T)\ s$ is true if and only if $p\ ok(t)\ s$, $\forall t \in T$.

We must realize that if a program is correct, then it will never produce any unexpected outcome with respect to the specification. Thus, p corr $s \Rightarrow p\ ok(t)\ s$, $\forall t$.

Definition: A *testing system* is a collection $< \mathcal{P}, \mathcal{S}, \mathcal{T},$ corr, ok $>$, where $\mathcal{P}$, $\mathcal{S}$, and $\mathcal{T}$ are arbitrary sets, corr $\subseteq \mathcal{P} \times \mathcal{S}$, sets, $ok \subseteq \mathcal{T} \times \mathcal{P} \times \mathcal{S}$, and $\forall p \forall s \forall t (p \text{ corr } s \Rightarrow p\ ok(t)\ s)$.

Definition: Given a testing system $< \mathcal{P}, \mathcal{S}, \mathcal{T},$ corr, ok $>$ a new system $< \mathcal{P}, \mathcal{S}, \mathcal{T}',$ corr, ok$' >$ is called a *set construction*, where $\mathcal{T}'$ is the set of all subsets of $\mathcal{T}$, and where $p\ ok'(T)\ s \Leftrightarrow \forall t (t \in T \Rightarrow p\ ok(t)\ s)$. (The reader may recall that T is a member of $\mathcal{T}'$, because $T \subseteq \mathcal{T}$.)

Theorem: $< \mathcal{P}, \mathcal{S}, \mathcal{T}',$ corr, ok$' >$, a set construction on a testing system $< \mathcal{P}, \mathcal{S}, \mathcal{T},$ corr, ok $>$, is itself a testing system.

[Proof]

The set construction is interpreted as follows. A test consist of a number of trials of some sort, and success of the test as a whole depends on success of all the trials. In fact, this is the rule in testing practice, where a test engineer must run a program again and again on a variety of test data. Failure of any one run is enough to invalidate the program.

Definition: Given a testing system $< \mathcal{P}, \mathcal{S}, \mathcal{T},$ corr, ok $>$ a new system $< \mathcal{P}, \mathcal{S}, \mathcal{T}',$ corr, ok$' >$ is called a *choice construction*, where $\mathcal{T}'$ is the set of subsets of $\mathcal{T}$, and where $p\ ok'(T)\ s \Leftrightarrow \exists t (t \in T \wedge p\ ok(t)\ s)$. (The reader may recall that T is a member of $\mathcal{T}'$, because $T \subseteq \mathcal{T}$.)

Theorem: $< \mathcal{P}, \mathcal{S}, \mathcal{T}',$ corr, ok$' >$, a choice construction on a testing system $< \mathcal{P}, \mathcal{S}, \mathcal{T},$ corr, ok $>$, is itself a testing system.

[Proof]

The choice construction models the situation in which a test engineer is given a number of alternative ways of testing the program, all of which are assumed to be equivalent.

Definition: A test method is a function $M : \mathcal{P} \times \mathcal{S} \rightarrow \mathcal{T}$.

That is, in the general case, a test method takes the specification S and an implementation program P, and produces test cases. In practice, test methods are predominantly program dependent, specification dependent, or totally dependent on the expectations of customers, as explained below.

Program dependent: In this case, $T = M(P)$, that is, test cases are derived solely based on the source code of a system. This is called *while-box* testing. Here, a test method has complete knowledge of the internal details of a program. However, from the viewpoint of practical testing, a white-box method is generally not applied to an entire program. Rather, one applies such a method to small *units* of a given large system. A *unit* refers to a function, procedure, method, and so on. A white-box method allows a test engineer to use the details of a program unit. Effective use of a program unit

requires a thorough understanding of the unit. Therefore, white-box test methods are used by programmers to test their own code.

Specification dependent: In this case, $T = M(S)$, that is, test cases are derived solely based on the specification of a system. This is called *black-box* testing. Here, a test method does not have access to the internal details of a program. Rather, such a method uses information provided in the specification of a system. Because specifications are much smaller in size than their corresponding implementations, it is not unusal to use an entire specification in the generation of test cases. Black-box methods are generally used by the develepment team and an independent quality-assurance (QA) group.

Expectation dependent: In practice, customers may generate test cases based on their *expectations* from the product at the time of taking delivery of the system. These test cases may include *continuous operation* tests, *usability* tests, and so on.

2.3.2 Power of Test Methods

A test engineering is concerned with what method to use to produce test cases, and how to compare test methods so that he can find out what test methods are better than others. Let M and N be two test methods. For M to be *at least as good as* N, we must have the situation that whenever N finds an error, so does M. In other words, whenever a program fails under a test case produced by method N, it will also fail under a test case produced by method M, with respect to the same specification. Therefore, $F_N \subseteq F_M$, where F_N and F_M are the sets of faults discovered by test sets produced by methods N and M, respectively.

Let T_M and T_N be the set of test cases produced by methods M and N, respectively. Then, we need to follow two ways to compare their fault-detection power.

Case 1: $T_M \supseteq T_N$. In this case, it is clear that method M is at least as good as method N. This is because, method M produces test cases which reveal all the faults revealed by test cases produced by method N. This case has been depicted in Figure 2.3(a).

Case 2: T_M and T_N overlap, but $T_M \not\supseteq T_N$. This case suggests that T_M does not totally contain T_N. To be able to compare their fault detection ability, we execute the program P under both the sets

of test cases, namely T_M and T_N. Let F_M and F_N be the sets of faults detected by test sets T_M and T_N, respectively. If $F_M \supseteq F_N$, then we say that method M is at least as good as method N. This situation has been explained in Figure 2.3(b).

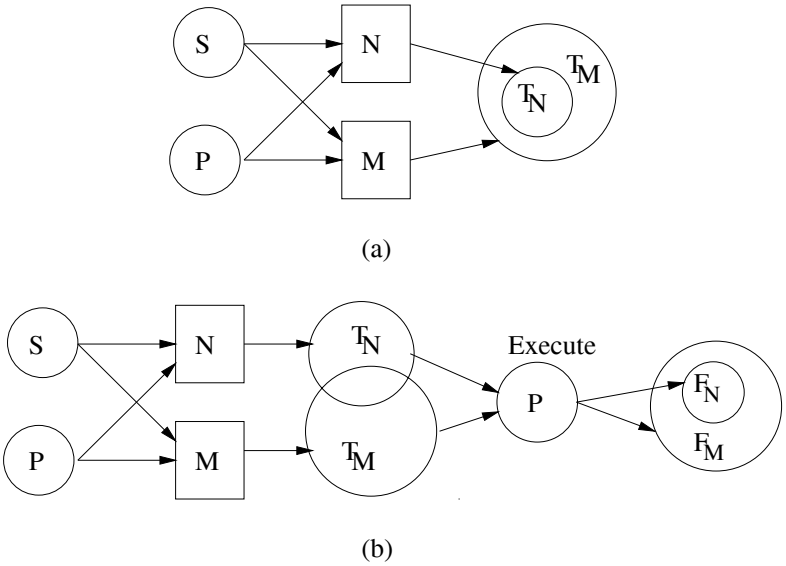

Figure 2.3: Different ways of comparing power of test methods. When one method produces all test cases produced by another method (a), and when the test sets intersect (b).

Chapter 3

Test Generation Methods

3.1 Sources of Information for Test Case Selection

Test cases can be designed by considering all sources of information related to a progarm. These sources are listed and explained below.

Requirements and functional specifications: The process of software development begins by capturing user needs. Depending on the particular life-cycle model chosen, the amount of user needs identified will vary. For example, in the Waterfall model, a requirements engineer tries to capture most of the requirements. On the other hand, in the spiral model, only a few requirements are identified in the beginning. Whatever may be the life-cycle model chosen, to test a program (or, system), a test engineer considers whatever requirements the program is expected to satisfy. The requirements might have been specified in an informal manner, such as a combination of plain text supplemented with equations, graphics, some flow-charts, and so on. Though this form of a requirements specification may be ambiguous, it is well-understood by customers. For example, the Bluetooth specification consists of about 1100 pages of documentation that describes how various subsystems of a Bluetooth interface is expected to work. The specification is written in plain text form supplemented with mathematical equations, state diagrams, tables, and figures. For some systems, requirements might have been captured in the form of **use cases**, **entity-relationship diagrams**, **and class diagrams**. Sometimes the requirements of a system might have been specified in a formal language or notation, such as SDL, Estelle, finite-state machine, and so on. All of these informal and formal specifications

are prime sources of test cases.

Source code: Whereas a requirements specification describes the *intended behavior* of a system, the source code describes the *actual behavior* of the system. High-level assumptions and constraints take concrete form in an implementation. Though a software designer may produce a very detailed design, programmers may introduce additional details into the system. For example, a step in the detailed design can be "sort array A." To sort an array, there are many sorting algorithms with different characteristics, such as iteration, recursion, and temporarily using another array. Therefore, test cases must be designed based on what actually a program does and how it does so by considering the source code.

Input and Output Domains: Some values in the input domain of a program have special meanings, and hence must be treated separately. To illustrate this point, let us consider the `factorial` function. The factorial of a non-negative integer `n` is computed as follows:

```
factorial(0) = 1;
factorial(1) = 1;
factorial(n) = n * factorial(n-1);
```

A programmer may wrongly implement the factorial function as

```
factorial(n) = 1 * 2 * ... * n;
```

without considering the special case of `n = 0`. The above wrong implementation will produce the correct result for all positive values of `n`, but will fail for `n = 0`.

Sometimes even some output values have special meanings, and a program must be tested to ensure that it produces the special values for all possible causes. In the above example, the output value `1` has special significance: (i) it is the minimum value computed by the `factorial` function, and (ii) it is the only value produced for two different inputs.

In the integer domain, the values `0` and `1` exhibit special characteristics if arithmetic opeartions are performed. These characteristics are: `0 * x = 0` and `1 * x = x` for all values of `x`.

The summary of the above discussion is that all the special values in the input and output domains of a program must be considered while testing the program.

3.2 Structural and Functional Testing

It is clear that a program must be tested with test cases designed from several sources, such as the specification, source code, and knowledge of the special properties of the program's input and output domains. A test selection technique is called a structural technique or a functional technique based on the information it uses to produce test cases.

In structural testing, one primarily examines the source code with focus on control flow and data flow. Control flow refers to sequencing of instructions by using different control methods. Control passes from one instruction to another instruction in a number of ways, such as one instruction after another, function call, message passing, and interrupts. Conditional statements control the flow of instruction executions in a progarm. Data flow refers to the propagation of values from one variable or constant to another variable. Definitions and uses of variables determine the data flow aspect in a program.

[Data base?]

In functional testing, one assumes no knowledge of the internal details of a program. A program is treated like a black box. A test engineer concerns himself only with what is visible outside the program, that is, just the input and the externally visible outcome. A test engineer applies input to a program, observes the examines the externally visible outcome of the program, and determines whether or not the program outcome is the expected outcome. Inputs are selected from the program's requirements specification and properties of the program's input and outout domains. A test engineer concerns himself only with the functionality and features of a program as found in the program's specification.

At this point it is necessary to make a distinction between the scopes of structural testing and functional testing. One applies structural testing techniques to individual units of a large program, whereas functional testing techniques can be applied to both an entire system as well as to individual program units. Let us consider a program unit U which is a part of a larger program P. A program unit is just a piece of source code with a well-defined objective and well-defined input and output domains. Now, if a programmer derives test cases for testing U from a knowledge of the internal details of U, then the programmer is said to be doing

structural testing. On the other hand, if the programmer designs test cases from the stated objective of the unit U and from his knowledge of the special properties of the input and output domains of U, then he is said to be doing functional testing.

The ideas of strcutural and functional testing do not give programmers and test engineers a choice of whether to design test cases from the source code or from the requirements specification of a program. Rather, these techniques are used by different groups of people at different times in a software life-cycle model. For example, individual programmers use both the structural and functional testing techniques to test their own code, whereas independent test engineers use functional testing techniques.

Neither structural testing nor functional testing is by itself good enough to detect all bugs. Even if one selects all possible inputs, a structural testing technique cannot detect all faults, if there are *missing paths* in a program. Similarly, without a knowledge of the structural details of a program, many faults will go undetected.

[Give an example. Give two sorting algorithms: one iterative and one recursive.]

Therefore, a combination of both structural and functional testing techniques must be used in program testing.