CHAPTER 1

INTRODUCTION

Software Testing is the process of executing programs with the intention of finding out errors. An important issue in the management of software testing is to ensure that before any testing, the objectives of that testing are known and agreed upon and those objectives are set in terms that can be measured. Such objectives should be quantified, reasonable and achievable. Effective software testing will contribute to the delivery of higher quality software products, more satisfied users, lower maintenance costs, more accurate and reliable results. Software testing for large-scale products is a very expensive process and consumes one-third of the total cost involved in a typical project development. In commercial systems, it is often difficult to estimate the cost of errors. For example, in a banking system, the potential cost of even a minor software error could be enormous. Exhaustive testing is not possible and hence the idea is now shifted towards automatic test generation. Automated testing methods promise to save a great deal of time and money throughout the software industry.

1.1 SPECIFICATION

Specifications provide valuable information for testing programs. Most software testing techniques, however, rely on the implementation of information from which test data is selected. This implementation-based testing technique focuses on the actual behavior of the implementation, but ignores intended behavior. On the other hand, considering information from formal specifications enable testing of intended behavior as well as actual functionality (Debra et al 1989). Specification-based testing techniques may direct attention to the aspects of the problem that have been
implemented incorrectly or completely neglected, while implementation-based techniques reveal such aspects only by chance. Currently, the focus is on utilizing specifications in selecting test cases. Specification-based test case selection traditionally consists of user-selected test cases based on a required specification. When the specification is informal, as is too often the case, this is effectively all that can be done. Existing specification-based testing systems manage user-selected test cases (Ostrand et al 1986), but automated test case selection is not feasible.

Formal specification is a repository of knowledge about a system. They are increasingly used in modeling software systems. Property oriented formal functional specifications specify software functions by a set of properties that the software should possess. The specification of a software product can be used as a guide for designing functional tests for the product. The specification precisely defines fundamental aspects of the software.

Formal specifications provides a significant opportunity for testing safety critical system software because they precisely describe what functions the software is supposed to provide in a form that can easily be manipulated by automated means. Specification based test data generation has several advantages over code based generation.

i. The specification can be used as a basis for output checking, significantly reducing one of the major costs of testing.

ii. Generating tests from a specification can proceed independently of program development and the created test can be applied to all implementations of the specification (Paul et al 2000).

Designing a specification is the process of describing a system and its desired properties. These kinds of system properties might include functional
behaviour, timing behaviour, performance characteristics or internal structure. It is through specification process that developers uncover design flaws, inconsistencies, ambiguities and incompleteness. A specification acts as a communication device between the customer and the designer, the designer and the implementer, and the implementer and the tester (Edmund et al 1996).

A formal software specification is a potential preamble for software testing. If the specifications are not properly designed, any kind of sophisticated testing will culminate in failures. The specification serves as a companion document to the system's source code, but at a higher level of description.

Specification-based testing is guided by a formal specification, whereby the testing activity is directly related to what a component under test is supposed to do, rather than what it actually does. Specification-based testing is a significant advance in testing, because it is often more straightforward to accomplish, and it can reveal failures that are often missed by traditional code-based testing techniques. Specification-fault-based testing attempts to detect faults in the implementation that are derived from misinterpreting the specification or the faults in the specification.

Formal specifications are increasingly used in verifying that a design meets critical requirements such as safety or security. In addition to design verification, formal models are useful as analytical tools to answer questions about how the system will behave in various circumstances. A model should also be useful to investigate the effect of changes in design or requirements. For example, let design P and requirement specification S be stated formally, then through formal proof it can be shown P → S (That is, design meets requirements specification).
1.2 BOOLEAN SPECIFICATION

A given boolean formula $F$ can be represented in various standard formats including sum-of-products form or product-of-sums form. These are also known respectively, as Disjunctive Normal Form (DNF) and Conjunctive Normal Form (CNF). For example, consider the formula $a(b\overline{c} + \overline{d})$. In DNF, this can be represented as $\overline{abc} + ad$. In conjunctive normal form it can be represented as $(a)(b + \overline{d})(\overline{c} + \overline{d})$.

Each occurrence of a variable or its negation in a boolean formula is called a literal.

A test case is an assignment of values to variables in a formula. A formula in DNF is said to be irreducible, provided that none of its literals or terms can be deleted without altering the value of the formula in some test case. A literal occurrence in a boolean formula is said to have meaningful impact on the value of the formula of a given test case if, everything else being the same, a different truth value assigned to that literal would have resulted in the formula evaluating to a different value.

To test an implementation of a given formula, test cases have to be picked from n-dimensional boolean space where n is the number of distinct variables in the boolean formula $F$ (Weyuker et al 1994). The true points are those that cause the formula to evaluate to 1. False points are those that cause the formula to evaluate to 0.

Let $P_i$ denote the $i$th product-term in the DNF representation of the boolean formula $F$, the points of the input space that cause $P_i$ to evaluate to 1 constitute the true points associated with $P_i$. $R_i$ denotes the true points associated with $P_i$. Since the sum-of-products form is assumed to be irreducible, the true points associated with a given term are not a subset of the true points associated with any other term.

Let a boolean formula $F = (a(b + \overline{c})d + e)$ be taken for illustration and this formula can be represented in irreducible DNF as $\overline{a}bd + a\overline{c}d + e$. Note that $\overline{a}bd$
represents four min-terms: \((a \land \bar{b} \land c \land d \land e), (a \land \bar{b} \land c \land d \land \bar{e}), (a \land \bar{b} \land \bar{c} \land d \land e)\) and \((a \land \bar{b} \land \bar{c} \land d \land \bar{e})\). Each of these min-terms can be represented by a five-digit binary number, with a 0 representing a negated variable and a 1 representing a non-negated variable. Then these four min-terms can be represented as 10111, 10110, 10011, 10010 respectively, or, using decimal notation 23, 22, 19, and 18. These four points are the true points for \(a\bar{b}d\). Similarly, the true points for \(a\bar{c}d\) can be represented by 18, 19, 26, and 27, and the true points for \(e\) can be represented by 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, and 31. Note that the union of the sets of all the true points of the product-terms of \(F\) is simply the set of true points of \(F\). The true points for the boolean formula \(F\) are therefore represented by a set. This is just the min-term representation of the chosen boolean formula \(F\).

\[F = \{1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 22, 23, 25, 26, 27, 29, 31\}.\]

### 1.2.1 Unique True Points (UTP)

There are two types of true points (Weyuker et al 1994) for a given term \(P_i\) of the boolean formulae \(F\). The Unique True Points (UTP) for the term \(P_i\) are those points that are in \(R_i\) but do not belong to any \(R_j\). \(U_i\) denotes these points. These points demonstrate the meaningful impact of each literal of a term on the evaluation of the formula to true. There is one unique true point for \(a\bar{b}d\) namely, 22, because 18 and 19 are also true points for the term \(a\bar{c}d\) and 23 is a true point for \(e\). There is just one unique true point for \(a\bar{c}d\): 26. For \(e\), the unique true points are 1, 3, 5, 7, 9, 11, 13, 15, 17, 21, 25, 29, 31.

### 1.2.2 Overlapping True Points (OTP)

The other type of true point is known as Overlapping True Point (OTP). \(O\) denotes these points. \(O\) consists of those points that are true points for at least two
terms. Thus the set of true points of a formula can be viewed as consisting of disjoint subsets $U_i, \ldots, U_m, O$. For the given formula $F$, $O$ consists of points 18, 19, 23 and 27.

1.2.3 Near False Points (NFP)

There are different types of false points (Weyuker et al 1994). The false points of $F$ can be represented by the set \{0, 2, 4, 6, 8, 10, 12, 14, 16, 20, 24, 28, 30\}. Let $P_i$ denote the product-term obtained by complementing the $j^{th}$ literal of the product-term $P_i$, for example if the product term is $\overline{abcd}$, then $P_{i3}$ is $\overline{abcd}$. Let $k_i$ be the number of distinct variables in the formula $F$ and the true points for $P_{ij}$ be denoted by $D_{ij}$. The points in $D_{ij}$ are true points for $P_{ij}$ but may be either true points or false points of the formula $F$. Now select those points in $D_{ij}$ that are false points for $F$ and denoted by $N_{ij}$. For a given product term, $P_i$, the union of all the $N_{ij}$ is denoted by $N_i$ and will be called Near False Points (NFP) for $P_i$.

1.2.4 Remaining False Points (RFP)

The Remaining False Points (RFP) of $F$ are those false points that are not in $N$ and this set is denoted by $M$. For the given formula $F$, $M$ is empty.

1.3 FORMAL METHODS

Hardware and software systems will inevitably grow in state and functionality. Because of this increase in complexity, the likelihood of subtle errors is much greater. Moreover some of the errors may cause catastrophic loss of money, time, or even human life. Formal methods can successfully enhance the quality of software and therefore increase the confidence in the system behavior. The major goal of software engineering is to enable developers to construct systems that operate reliably despite this complexity. One way of achieving this goal is by using formal methods for specifying and verifying such systems. Use of formal methods does not
guarantee correctness (Edmund et al 1996). However they can greatly increase our understanding of a system by revealing inconsistencies, ambiguities, and completeness that might otherwise go undetected.

Predicate difference can be an effective analytical tool for evaluating the effects of changes in formal specifications. They may also be useful in re-verifying specifications after modifications, determining whether a change will cause a previously secure system to become non-secure (Kuhn 1992).

1.4 FAULT CLASSES

During the process of software creation many types of errors may occur. An error is a mental mistake by a programmer or designer. It may result in textual problems with the code called a fault (Morell 1990). A failure occurs when a program computes an incorrect output for an input in the domain of specification. For example, a programmer may make an error in analyzing an algorithm. This may result in writing a code fault; say, a reference to an incorrect variable. When the program is executed, it may fail for some inputs.

The set of possible fault classes is much larger. Because the difference between an implementation and its specification is the result of human error, some types of faults may be virtually impossible to predict in advance (Kuhn 1999). The faults taken for analysis are related to the operands in the boolean expression. The boolean expression is assumed to be in DNF. There are many types of faults, such as Expression Negation Fault (\(ENF\)), Term Negation Fault (\(TNF\)), Literal Negation Fault (\(LNF\)), Term Omission Fault (\(TOF\)), Literal Omission Fault (\(LOF\)), Literal Insertion Fault (\(LIF\)), Literal Reference Fault (\(LRF\)) and Term Insertion Fault (\(TIF\)).

One of the realistic models of faults that occur in the software is Missing Condition Fault (\(MCF\)). The most common implementation error is the failure to
validate input data or check preconditions. In this fault, the programmer does not implement one or more conditions. Variable Reference Fault (VRF) can now be divided into two. One in which the variable substitution results in a Missing Condition Fault and the other in which one condition is replaced by another. The second type of fault could be described as an Incorrect Condition Fault, since the boolean variables typically represent some relation or condition in a specification. Missing Condition Faults are extremely common (Marick 1990). Approximately half of the faults posted on Usenet bug reports are faults of omission that are necessarily Missing Condition Faults. Tests based on this fault model do appear to be effective in finding errors.

1.5 PARTITION TESTING

The term “Partition testing”, in its broadest sense, refers to a very general family of testing strategies. Its primary characteristic is that the program’s input domain is divided into subsets, with the tester selecting one or more elements from each sub-domain (Jeng et al 1989). The term “Partition” does not restrict to the mathematical meaning of a division into disjoint subsets, but it will be overlapping subsets. The goal of such a partitioning is to make the division in such a way that when the tester selects test cases based on the subsets, the resulting test set is a good representation of the entire domain.

Ideally, the partition divides the domain into disjoint sub-domains with the property that within each sub-domain, either the program produces the correct answer for every element, or the program produces an incorrect answer for every element. Such a sub-domain is called revealing.

If a partition’s sub-domains are revealing, one need to randomly select an element from each subset and run the program on that test case in order to determine faults. In practice it is common to divide the input domain into non-disjoint subsets and it is extremely unusual for the sub-domain to be truly revealing (Jeng et al 1989).
Test selection is an activity that attempts to partition the input and output domains of the program into a finite number of sub domains that are approximations of equivalent classes. Test conditions are used to characterize each sub domain. A test condition evaluates to true only for test data that are members of the sub domain associated with that test condition.

1.5.1 Call-State Test Conditions

These are derived from input conditions of the function and constrain values of input parameters only. They characterize sub domains that are partitions of a function in input domain (Juei Chang et al 1996). It can be used to measure comprehensiveness of test data, even if an implementation is not available. (Black-box test suits during design and not after implementation).

1.5.2 Return-State Test conditions

These are derived from both input and output conditions of the function and constraint values of both input and output parameters. Return state test conditions provide more thorough test coverage, but cannot be evaluated without an implementation.

1.6 DEFECT DETECTION

The way to measure how well the objective of software testing has been achieved is to inject some artificial faults into the predicate and check if they are detected by the test (Hong Zhu et al 1997). A predicate with the planted fault is called the mutant of the original predicate. If a mutant and the original predicate produce different outputs on at least one test case, then the fault is detected. Mutation analysis is proposed as a procedure for evaluating the degree to which a program is tested. Mutation analysis allows a great degree of automation. Mutation-based testing system
provides an interactive test environment that allows the tester to locate and remove errors. Mutation analysis introduces small changes or mutants into the specifications producing a mutant specification. Better test sets are those that reveal the mutants.

The original program and the mutated program or 'mutant' are run with the same test suite. A mutant is said to be killed if it produces an output different from the original program (Woodward 1993). The ratio of killed mutants to the total number of mutants is determined and the test data set considered is mutation-adequate if it has killed a sufficient portion of the mutants. The more the mutants killed, the more is the mutation-adequate and better quality of the test data set.

The basic idea is that a mutation adequate test set, which detects enough of the seeded faults, is likely to detect naturally occurring defects.

Weak mutation testing (Hong Zhu et al 1997) is a variant of strong mutation testing. Mutation testing requires the output between the original program and the mutated program to be different, whereas weak mutation testing requires the intermediate values computed by the mutated code fragment to be different (typically at the statement level). Weak mutation testing is weaker than strong mutation testing in the sense that even though the mutated code fragment may compute different values from the original fragment, the mutated program may still compute the same output as the original program. Thus weak mutation coverage of a test suite is always higher than its mutation coverage.

Weak mutation testing (Howden 1982) has the advantage, however, that it is more easily automated through the use of source code instrumentation. It also requires less computation time, since the instrumented program is only run once for each test case, whereas mutation testing requires every mutated program and the original program to be run for each test case.
In weak mutation, there are two types of coverage. One is operator coverage and the other is operand coverage. In weak mutation, two types of coverage are used (Juei Chang et al 1999). In operator coverage, every possible mutant that replaces an operator with another must be killed. The basic idea is to detect where the programmer used the wrong operator. In operand coverage, every mutant that replaces an operand with another operand of compatible type must be killed. The basic idea is to detect where the programmer used the wrong operand.

Mutation testing is an example of bounded fault based arena and is therefore finitely distinguishable (Morell 1990). Since the mutation-testing arena is bounded, all alternate programs are unfortunately large and indivisible program-equivalence problem remains. Another problem with mutation testing relates to the phenomenal number of mutants that can be generated for even a single program.

Mutation testing researchers' justification for predicting defect detection and their assumptions are given here. The "competent programmer hypothesis" states that a programmer only makes small syntactic errors and not complex logical, design, and control errors (Acree et al 1979) and the code is relatively close to being correct. The "coupling effect" states that complex faults are coupled to simple faults and test data detecting simple faults are sensitive enough to detect more complex faults (Offutt 1989). The coupling effect justifies the utility of mutation testing, since the capability to detect simple mutations is, therefore, indicative of the capability to detect complex faults.

1.7 FAULT-BASED TESTING

Testing is a method of program verification that deduces from execution information that a program possesses required properties. The role of testing theory is to study this deductive process and to investigate the conditions under which it can be accomplished. Testing may be viewed in two stages. First, a testing strategy is
developed which describes how testing is to be performed. That is, how the input data is to be selected, what information is to be collected and how to collect and analyze this information. Second, the strategy is applied to a given program, resulting in a test of that program.

A testing theory must provide a means of judging the quality of either stage of testing. Intuitively, a good test provides convincing evidence that a program is correct. A testing strategy could therefore be considered good if it produces good tests when applied to a program. This approach has been taken by Howden (1976) who defines a reliable test to be one whose success implies program correctness.

The means taken for assessing the quality of tests is to compare them on the basis of how many faults, each demonstrates, are not in the program. That is, how many faults each test eliminates. The highest quality test by this approach is one, which eliminates all faults, implying the program is correct. Similarly the quality of a strategy is determined solely by what faults it is guaranteed to eliminate.

Testing is fault based when its motive is to demonstrate the absence of pre-specified faults. This approach defines the mechanism whereby the faults can be defined, and once defined, how they are eliminated. The difference between this approach and that of conventional testing can be understood by reflecting on the "information content" of a test. In the traditional point of view, a test that fails to reveal an error bears no useful information. For example, Myers (1979) states, "since a test case that does not find an error is largely a waste of time and money", the descriptor 'successful' seems inappropriate (Myers 1979). Thus, the information content of a test is measured by the failure it causes.

It is argued that every correct program execution contains information that proves the program could not have contained particular faults. Fault-based approach treats successful tests as indicative of the absence of prespecified faults.
Fault based testing is adopted from mutation testing (Morell 1990). Expressions are supposed incorrect in a limited manner as defined by a set of alternatives, which contains legal replacements for the expression. The mutation model is extended here in a significant way; alternative sets are allowed to be infinite. This entails developing a fault-based testing that does not rely on the ability to generate and execute all alternatives, as is done in mutation testing. Fault-based testing therefore must prove the absence of infinitely many faults based on finitely many executions.

The goal of fault-based testing is to detect faults in the specification by revealing specification failures or to detect coding faults that are due to misunderstanding the specification by revealing implementation failures. Fault based testing techniques postulate that faults exist in the implementation and select test cases to detect those faults if they exist. A test case consists of an input criterion and an acceptance criterion. The input criterion is a condition describing data that satisfies their test case, it may be as specific as actual test data or as general as a condition on the input domain or output range. The acceptance criterion is a condition describing whether execution of this test case is acceptable. A good test case is one that has a high probability of finding an undiscovered error. Since the software reliability is intimately connected to the absence of faults, techniques that demonstrate the absence of faults should impact the assessment reliability (Morell 1990). Fault based testing is a realistic approach to the inherent limitation on testing. The development of cost-effective tests for larger systems may well require test stimuli targeting actual faults.

1.8 TRAFFIC ALERT AND COLLISION AVOIDANCE SUBSYSTEM (TCAS II) – AN INTRODUCTION

TCAS is an on-board aircraft conflict detection and resolution system used by all US commercial aircrafts. TCAS has been studied by both the industry and academia (Richard Anderson et al 1998, Mats Heimdahl 1996,
Nancy Leveson et al (1994) and can be considered as the benchmark for safety critical applications. TCAS continuously monitors the RADAR information to check whether there is a neighbour aircraft that could represent a potential threat by getting too close. It is said that in such a case an intruder aircraft is entering the protected zone. Whenever an intruder aircraft enters the protected zone, TCAS issues a Traffic Advisory (TA) and estimates the time remaining until the two aircrafts reach the closest point of approach. Such estimate is used to calculate the vertical separation between the two aircrafts assuming that the controlled aircraft either maintains its current trajectory or performs immediately an upward (downward) maneuver. Depending on the results obtained, TCAS may issue a Resolution Advisory (RA) suggesting the pilot either to climb or to descend.

1.9 MOTIVATIONS AND OBJECTIVES OF THE RESEARCH

The motivations for the work are given below:

- For large formulae such as the 14 literal specification given by Weyuker et al (1994), the generalized fault condition given by Kuhn (1999) is very complex and time consuming.
- More efficient test set generation methods can be proposed, compared to the other methods.
- The availability of formal specifications of TCAS II – Traffic Alert and Collision Avoidance Subsystem has further motivated to analyze the condition using these specifications for the proposed study.
Hence, the objectives of this thesis are as follows:

- To apply the fault condition proposed by Kuhn (1999) and Tsuchiya et al (2002) to the fault classes proposed by Lau et al (2001), study the results and simplify it.
- To propose some new fault classes, which have not been considered earlier and to extend the hierarchy of fault classes with the new fault classes.
- To find variations to the conclusion on fault detecting capability available in the literature.
- Lau et al (2001) provided theoretical results to show the hierarchy of fault classes. To provide empirical analysis and to check the results of Lau et al (2001).
- Study the characteristics of various fault classes and propose a new fault condition.

1.10 ORGANIZATION OF THE THESIS

The objectives stated above have been carried out and organized for presentation in this thesis in seven chapters. While this first chapter gives an introduction to the research work, chapter 2 provides the state of the art in specification-based testing. The chapter mainly deals with the past work done by various researchers in this area.

Chapter 3 provides a detailed description of the various fault classes considered for analysis throughout the thesis. Also, a new fault class is proposed. The process of analysis is explained. Further, the fault condition proposed by Kuhn (1999) is analyzed and applied to the various fault classes, including the proposed fault and results are derived.
Chapter 4 proposes the novel idea of generating test sets for various faults from a single fault namely, Literal Reference Fault. The hierarchy of fault classes proposed by Lau et al (2001) is studied in detail and analyzed. Empirical results are provided.

Chapter 5 indicates some exceptions in hierarchy of fault classes. The Null Fault is defined and discussed in detail. A method to detect the Null Fault is also proposed. An alternate strategy is proposed to generate test sets for various faults to overcome the exceptions in the hierarchy of fault classes. For this, a test set generation from Literal Negation Fault is considered.

Chapter 6 provides a detailed discussion about the characteristics of various fault classes. This study is used to propose a new fault condition. A comparison of the fault condition proposed by Kuhn (1999) and Tsuchiya et al (2002) with the new fault condition is also provided. In all the chapters, the results are applied to real life Traffic Alert and Collision Avoidance Subsystem and verified.

The summary of the work reported, conclusions and future directions are given in Chapter 7.