# An Evaluation Scheme of Software Testing Techniques

Huey–Der Chu

Centre for Software Reliability, Dept. of Computing Science
Bedson Building, University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, U. K.

Tel: +(44) 191 222 8972
Fax:+(44) 191 222 8788
Email: huey–der.chu@newcastle.ac.uk
URL: http://www.ncl.ac.uk/~n4521677

# An Evaluation Scheme of Software Testing Techniques

Huey–Der Chu

Centre for Software Reliability, Department of Computing Science

UNiversity of Newcastle upon Tyne, NE1 7RU, UK

**ABSTRACT**

This paper briefly surveys the software testing techniques based on the works of classification and evaluation. In addressing the two major software testing issues, that is when should testing stop and how good the technique is after testing, I present a scheme by a data flow diagram (DFD) for evaluating software testing techniques. Following this diagram step by step, all the activities involved and the relative techniques were described. A strategy proposal for software testing in the development of applications is advocated later.

**Keywords:** Deterministic Testing, Statistical Testing, Test Data Adequacy Criterion

## 1 INTRODUCTION

The history of software testing is as long as the history of software development itself. It is an integral part of the software life–cycle and must be structured according to the type of product, environment and language used. In the absence of feasible and cost–effective theoretical methods for verifying the correctness of software designs and implementations, software testing plays a vital role in validating both. The goal of software testing is [1, 2]: Firstly, to reveal that hidden number of defects which are created during the specification, design and coding stages of development, secondly, to provide confidence that failures do not occur and thirdly, to reduce the cost of software failure over the life of a product.

Software testing has progressed through five major paradigms [3]: the debugging, demonstration, destruction, evaluation and prevention periods, as outlined by a number of authors. During its development, software testing has focused on two separate issues, verification (static testing) and validation (dynamic testing).

Verification, as defined by IEEE/ANSI [4], is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. It is the process of evaluating, reviewing, inspecting, and doing desk checks of work products such as requirement specifications, design specifications, and code. For code it means the static analysis of the code – a code review – not the dynamic execution of the code. Verification thus tries to answer the question: Have we built the system right?

Validation, as defined by IEEE/ANSI [4], is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. It normally involves the execution of actual software on a computer and usually exposes any defects. Validation then boils down to the question: Have we built the right system?

Verification and validation are complementary [5]. The effectiveness of defect detection suffers if one or the other is not done. Each of them provides filters that are designed to expose different kinds of problems in the product. Historically software testing has been, and continues to be, largely validation–orientated. It is not that we should stop doing validation, but we want to be much cleverer about how we do it and how we do it in combination with verification. We must also ensure that we do each of them at the right time on the right work products.

In practice, the software development methodologies typically employ a combination of several software testing techniques. That there is no "silver bullet" testing approach and that no single technique alone is satisfactory has been pointed out by many leading researchers such as [6 – 8]. The need to combine testing techniques is further visible when we consider the primary characteristics of each approach and find that each testing strategy addresses only a narrow set of concerns.

From this viewpoint, a framework is presented in this paper to the classification of software testing techniques, to the evaluation of software testing techniques and to the proposal of testing strategy. A framework for surveying software testing techniques is given in Section 2. According to this framework, an evaluation scheme of software testing techniques is presented in section 3. The proposal of software testing strategy will be discussed in section 4. Concluding remarks are made in Section 5.

## 2 A FRAMEWORK FOR SURVEYING SOFTWARE TESTING TECHNIQUES

### 2.1 The Classification Of Software Testing Techniques

An agreed standard for testing technique classification would allow testers to compare and evaluate testing techniques more easily when attempting to choose the testing strategy for the software development. The software testing techniques can be classified according to the following viewpoints:

(1) Does the technique require us to execute the software? If so, the technique is dynamic testing; if not, the technique is static testing.

(2) Does the technique require examining the source code in dynamic testing? If so, the technique is white–box testing; if not, the technique is black–box testing.

(3) Does the technique require examining the syntax of the source in static testing? If so, the technique is syntactic testing; if not, the technique is semantic testing.

(4) How does the technique select the test data? Test data is selected depending on whether the technique refers to the function or the structure of the software, leading respectively to functional testing and structural testing, where as test data is selected according to the way in which software is operated with respect to random testing.

(5) What type of test data does the technique generate? In deterministic testing, test data are predetermined by a selective choice according to the adopted criteria. In random testing, test data are generated according to a defined probability distributed on the input domain.

### 2.2 The Evaluation Of Software Testing Techniques

With reference to this classification, the work on the evaluation of software testing techniques can be done in correspondence with the two major testing issues as shown:

(1) When should testing stop? The exit criterion can be based on a reliability measure when the test data have been selected by random testing, whereas a test data adequacy criterion for determining whether or not a test set is sufficient for deterministic testing.

(2) How good is the technique after testing? The definition of software reliability measure with failure rate can be applicable to test software with discrete or continuous test data [9]. Test data adequacy criteria are measures of the quality of testing. From this viewpoint, the classification of test adequate criteria can be divided into fault–based testing and error–based testing [10]. However, as this is the most important aspect of test quality, there are many experimental works to measure it by metrics, mutation analysis and the expected number of failures detected *et al*.

A framework for surveying software testing techniques based on the works of classification and evaluation is shown in Figure 1.

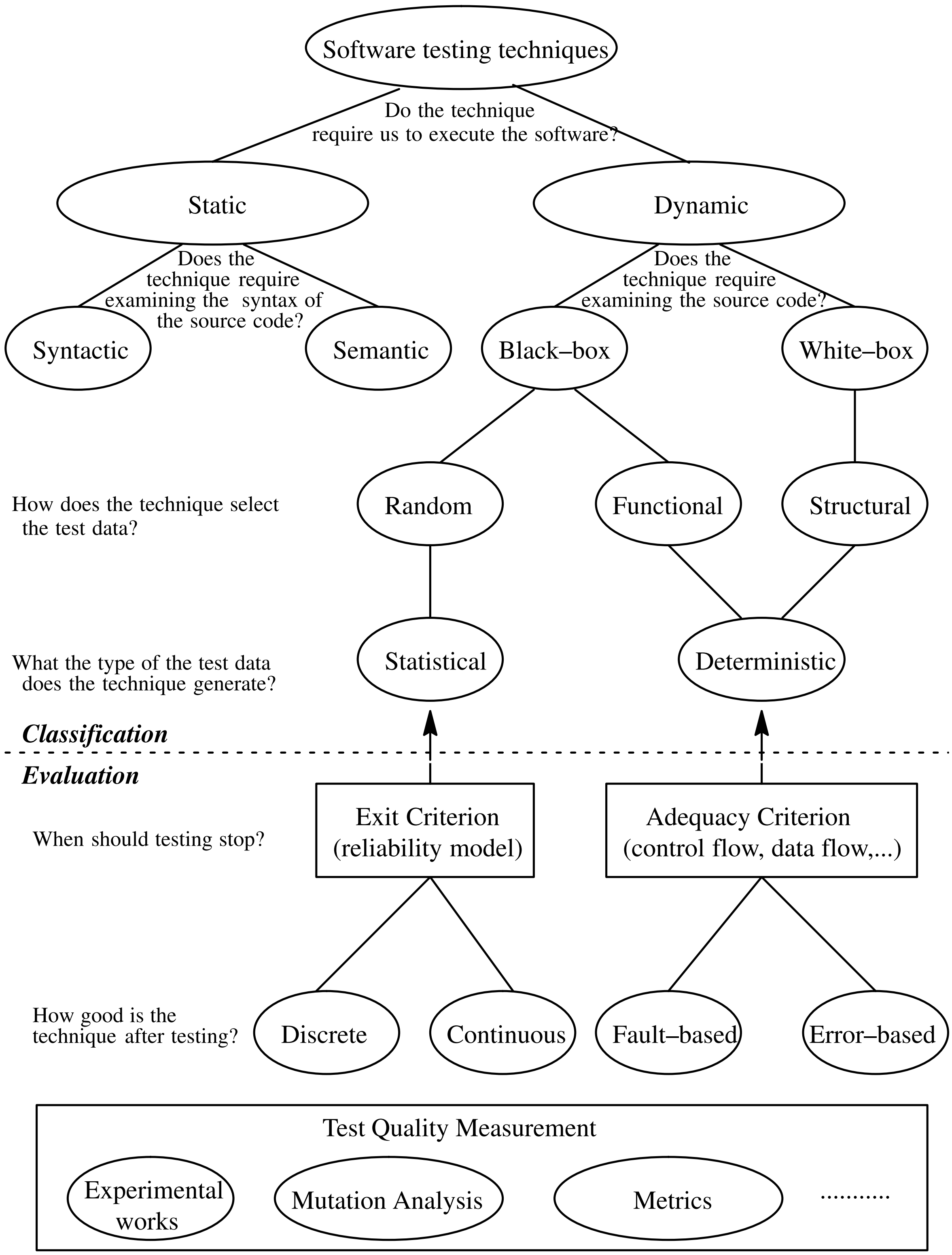


Figure 1: A framework for surveying software testing techniques

## 3 AN EVALUATION SCHEME OF SOFTWARE TESTING TECHNIQUES

According to the framework for surveying software testing techniques, we present an evaluation scheme of software testing techniques. The purpose of this evaluation scheme is to allow us to identify the strengths and weakness of current software testing techniques. This will provide the information for selecting the testing strategy in the development of applications. In addressing the two major testing issues, that is when should testing stop and how good the technique (or the software) is after testing, a Data Flow Diagram (DFD) depicting the evaluation scheme is shown in Figure 2; the circles in the diagram correspond to the tasks that will be identified in the following sub–sections.

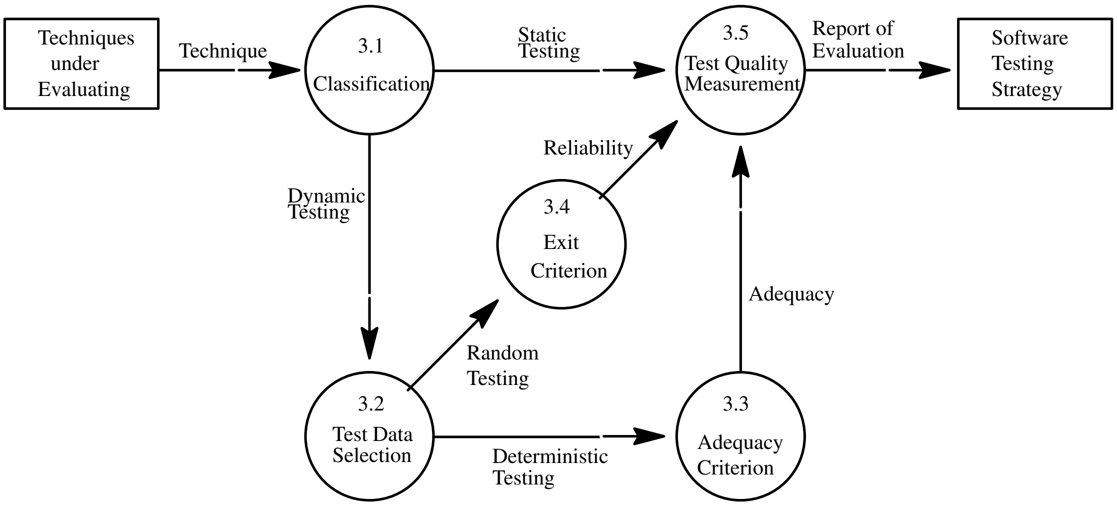


Figure 2: An evaluation scheme of software testing techniques

### 3.1 Classification

Historically there have emerged different classifications of testing techniques. Existing software testing techniques are divided into two categories [11, 12]: static and dynamic testing. Static testing techniques were those that examined the software without executing it and encompassed activities such as inspection, symbolic execution and verification. Dynamic testing techniques are those that examined the software with a view to generating test data for execution by the software.

#### 3.1.1 Static Testing

Static testing techniques are concerned with the analysis and checking of system representations such as the requirements documents, design diagrams and the program source code, either manually or automatically, without actually executing the code [13]. During static testing, specifications are compared with each other to verify that errors have not been introduced during the process. In comparison to dynamic testing, static testing does not require input distributions, since they do not require that the software be executed. This is convenient when the input distributions are not known. However, since input distributions are not known to the techniques, static testing techniques cannot take advantage of this knowledge when it is available. Static testing techniques can be classed according to whether or not the technique requires examining the syntax of the source code. If so, the technique is syntactic testing; if not, the technique is semantic testing.

Syntactic testing may include the reviews and walk–throughs [2] held by a design team to check that the refinements of accepted requirements are proceeding as desired through each transformation stage. However, the informal nature of such reviews and walk–throughs leaves some doubts about their overall effectiveness and their repeatability [14].

Unlike the informal reviews and walk–throughs, a software inspection is a formal evaluation of the work items of a software product. The technique was originally devised by Fagan at IBM [15] and has proved to be an effective technique for the design, code and test phases. A software inspection is led by an independent moderator with the intended purposes of effectively and efficiently finding defects early in the development process, recording these defects as a basis for analysis and history and initiating re–work to correct such defects. Re–worked items are subsequently re–inspected to ensure their quality. Software developers can literally remove a part from the development line, re–work it at the most appropriate time in the process and replace it in the development line. Therefore, inspections ensure that a high level of quality is delivered to the testers and ultimately to the users of a software product.

Semantic testing includes formal methods such as proof of correctness. Proof of correctness [2, 16] is a mathematical method of verifying the logic or function of a program or program segment. In order to be able to do so, the specification must be expressed formally as well. We achieve this by expressing the specification in terms of two assertions which come before and after the program's execution, respectively. Next, we prove that the software transforms one assertion (the pre–condition) into the other (the post–condition). As it is the most labour–intensive a validation method, it offers a consistent and reputable approach. Refined specification or design can always be proven correct and probably defect–free against higher level specification or design.

### 3.1.2 Dynamic Testing

Dynamic testing techniques are generally divided into the two categories – black–box and white–box testing [13, 17], which correspond with two different starting points for software testing: the internal structure of the software and the requirements specification. They involve the execution of a piece of software with test data and a comparison of the results with the expected output which must satisfy the users' requirements. The process of dynamic testing is shown in Figure 3.

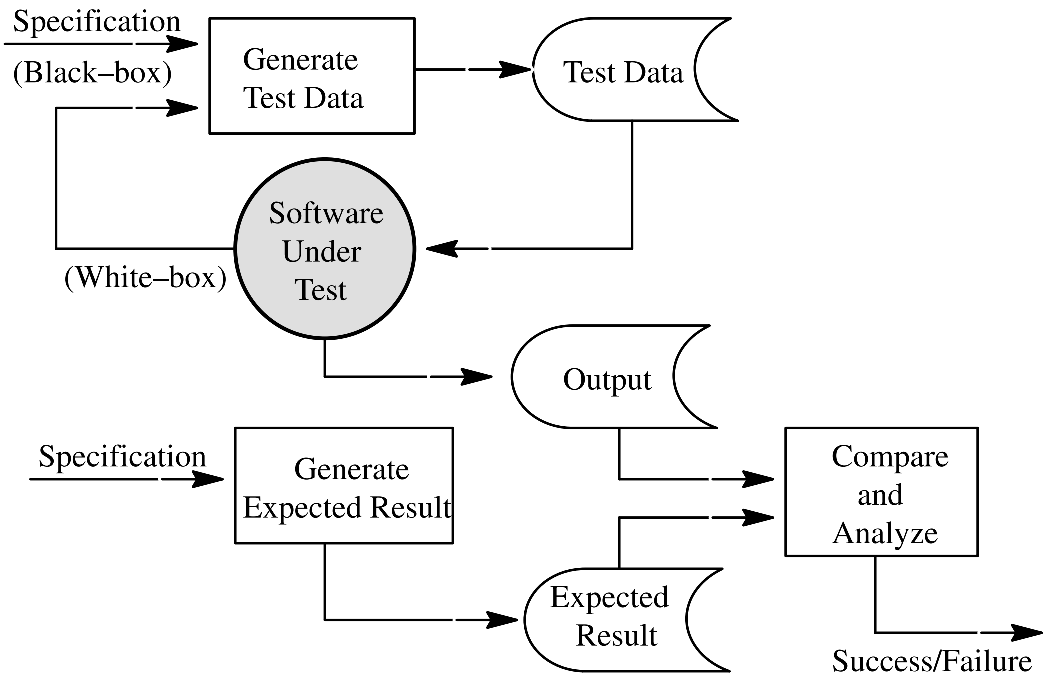


Figure 3: The process of dynamic testing

Black–box testing uses a 'toaster mentality': You plug it in, it is supposed to work. Created input data is designed to generate variations of outputs without regard to how the logic actually functions. The results are predicted and compared to the actual results to determine the success of the test. In contrast to this, white–box testing opens up the 'box' and looks at the specific logic of the application to verify how it works. Tests use logic specifications to generate variations of processing and to predict the resulting outputs. Intermediate and final output results can be predicted and validated using white–box testing. Other terms have been introduced over the years and now the black–box testing method is sometimes called "functional" or "specification–based" while white–box testing method may be referred to as "structural" or "code–based" or even "glass–box" [12].

It is important to examine why this distinction appeared. Black–box testing is frequently a vague formalization of good testing practice. Its drawback is that without examining the code in some way you do not know how much of it is being tested. Black–box testing is typically used to check if the product conforms to its specification. But what if there is something in the product that does not meet the specification? What if the software performs some undesirable task that the black–box inputs have not detected? This is where the white–box techniques come in. They allow you to examine the code in detail and be confident that at least you have achieved a level of test coverage, for example, the execution of every statement. White–box testing is in itself insufficient since the software under examination may not perform one of its desired tasks–the function to do this may even be missing–and the examination of the code is unlikely to reveal this. The objective perspective of black–box testing is needed to be able to spot such missing functionality.

### 3.2 Test Data Selection

Dynamic testing involves selecting input test data, executing the software on that data and comparing the results to some test oracle, which determines whether the results are correct or erroneous. To be sure of the certainty of the validity of software through dynamic testing, ideally we should try the software on the entire input domain. In fact, due to the intrinsically discontinuous nature of software, given an observation on any point of input domain, we can not infer any property for other points in the input domain. However, excluding trivial cases, the input domain is usually too large for exhaustive testing to be practical. Instead, the usual procedure is to select a relatively small subset termed "subdomain", which is in some sense representative of the entire input domain. From this limited number of test runs, we then infer the behaviour of software for the entire input domain. Therefore, dynamic testing corresponds to sampling a certain number of executions of a software from amongst all its possible executions, by sampling a number of input data within the input domain. Ideally, the test data should be chosen so that executing the software on this subdomain will uncover all errors, thus guaranteeing that any software which produces correct results for the test data will produce correct results for any data in the input domain. However, discovering such an ideal set of test data is in general an impossible task [18, 19]. The identification of a suitable sampling strategy is known as the test data selection problem. The model of test data selection is shown in Figure 4.

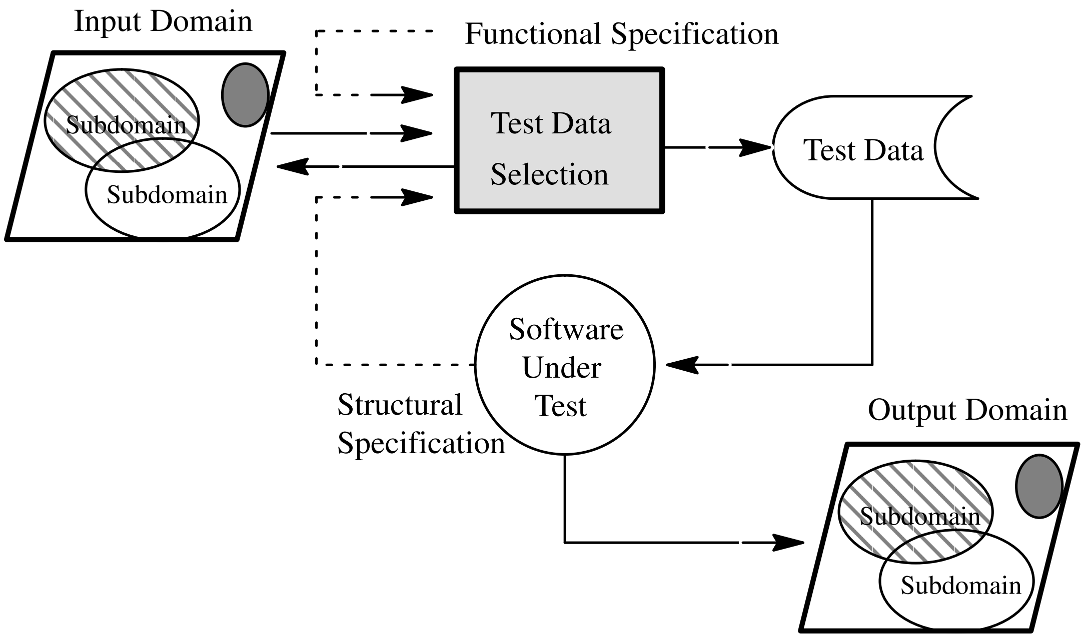


Figure 4: The basic model of test data selection

In general, however, the subdomains formed may not be all disjointed, and hence they are not the true partitions in the formal mathematical sense. Figure 4 also shows a software system as a mapping of an input to an output set. The software responds to the inputs by producing an output or a set of outputs. Some of the inputs (shown in the shaded circle in Figure 4) cause system failures where erroneous outputs are generated by the software.

There are three basic strategies for selecting test data [19]:

(1) Functional testing: essentially, test data is selected according to the reference functional specification.

(2) Random testing: essentially, test data is selected according to the way in which software is operated. (in fact, input domain is generally partitioned according to the expected use of software).

(3) Structural testing: essentially, test data is selected according to the reference structural specification.

The first two strategies are both referred to as black–box testing and the third as white–box testing. A variety of testing techniques fall within each of the above basic strategies. When the focus of dynamic testing is fault removal, that is, fault–finding rather then reliability assessment, the tester is faced with the problem of selecting a subset of the input domain that is suitable for revealing the actual but unknown faults. Then, the methods for generating test inputs proceed according to one of two principles: either deterministic or probabilistic [20, 21].

### 3.2.1 Deterministic Testing

Deterministic methods for generating test inputs usually take advantage of information on the target software in order to provide guides for selecting test cases, the information being depicted by means of test criteria [22]. Both functional and structural testing strategies use a systematic means to determine subdomains. They often use peculiar inputs to test peculiar cases. Given a criterion, the type of test input generation is deterministic: input test sets are built by selecting one element from each subdomain involved in the set proper to the adopted criterion. This approach to the selection of test data is commonly referred to as partition testing [23 – 25]. In fact, most of the testing strategies subdivide the input domain into overlapping subdomains and thus do not form a true partition of the input domain, so the term subdomain–based testing or simply subdomain testing was suggested by a number of authors [26, 27]. However, their intention is to provoke failure behaviour and success hinges on the assumption that we can identify subdomains with a high failure probability. Unfortunately, exercising each subdomain defined from such criteria only once is far from being enough therefore the corresponding test set is relevant to expose faults, since a real limitation of current criteria is their imperfect connection with faults. Though this is a good strategy for fault detection, it does not necessarily inspire confidence.

### 3.2.2 Random Testing

Random testing strategy is the conventional probabilistic method for generating test inputs. It is assumed that the tests are randomly selected according to the operational distributed for the software. This is the probability distribution describing the frequency with which different elements of the input domain are selected when the software is in actual use. Without this assumption the results are not useful. It consists in generating random test data based on a uniform distribution over the input domain [23]: this is an extreme case of the black–box testing method, since no information related to the target piece of software is considered except for the range of its input domain. An advantage of using random testing is that quantitative estimates of a software's operational reliability may be inferred. From another viewpoint, random testing can be viewed as a degenerate form of subdomain testing in the sense that there is only one "subdomain," the entire input domain. Thus, random testing does not bear the overheads of subdividing the input domain and of keeping track of which subdomains have been tested or not [24].

Statistical testing is based on an unusual definition of random testing [22]: it aims to provide a "balanced" coverage of a model of the target software, no part of the model being seldom or never exercised during testing. With this approach, the method for generating statistical test patterns combines information provided by a model of the target software, that is, by a test criterion with a practical way of producing large sets of patterns, that is, a random generation. The set of statistical test patterns are then defined by two parameters, which have to be determined according to the test criterion used. These are as follows:

(1) The test profile or input distribution, from which the patterns are randomly drawn.

(2) The test size or equivalently the number of input patterns that are generated.

The major advantages of using the statistical method for software testing are [29, 30]: Firstly, testing can be performed based on the user's actual utilization of the software, secondly, It allows the use of statistical inference techniques to compute probabilistic aspects of the testing process and thirdly, in many applications, testing can be completely automated, from the generation of test data to analysis of test results.

### 3.3 Adequacy Criterion

We consider here only dynamic testing because a static testing cannot determine whether a piece of code is or is not reachable. On the other hand, the software is run on one or more input values and the outputs produced are assessed. There are other ways of validating a software which do not involve running the software on test cases including static analysis and formal verification, but our concern here is only with ways of assessing the adequacy of dynamic testing [31, 32].

A software test data adequacy criterion is a means for determining whether a test set is sufficient or "adequate, " for testing a given software or not by means of test criteria [32, 33]; if the test set is inadequate then more tests are added to the test set and the entire process is repeated. Otherwise, the software is released. A model of the test data adequacy is shown in Figure 5.

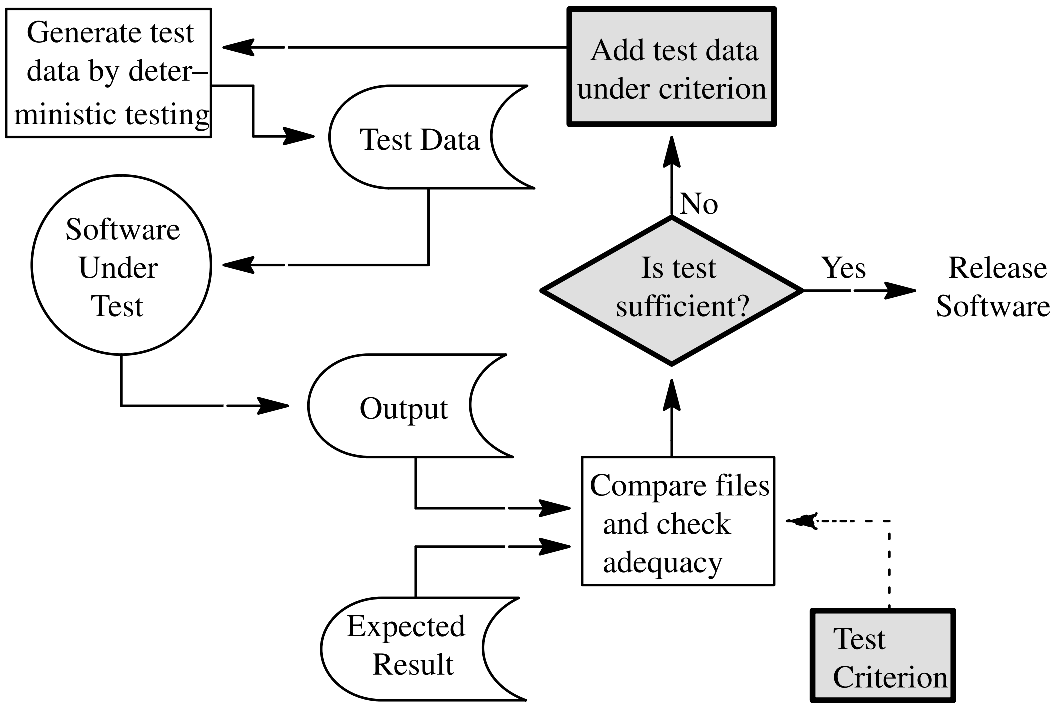


Figure 5: A model of the test data adequacy

Since Goodenough and Gerhart's pioneering work [34], which pointed out the central problem of software testing is "what is a test data adequacy criterion," various adequacy criteria have been proposed, investigated and used in practice [10], including control flow adequacy criteria, data flow adequacy criteria, program text–based criteria and mutation adequacy criteria. Examples of such "test data adequacy criteria" might be 'to ensure coverage of all branches in the software,' 'to execute each path between a definition and a use of a variable,' 'to ensure coverage of all equivalence classes in the input domain' or 'to test each feature at its extreme points and at some intermediate point' [33, 35].

In the testing process, a test data adequacy criterion is only invoked when the tests no longer expose faults. Even though no faults are exposed by the test set, the software may not be correct. Here is an example:

```
Display "Did you pass your English test?"
Input score
If score > 70
  Display "Pass"
Else
  Display "Failure"
```

Support one test set consists of two test cases: 80 and 50. We get the right answers! Furthermore, we have achieved 100% statement coverage, branch coverage and 100% true path coverage. Since we have just tested this program more throughly than any real–life module is ever tested and have exercised every path, we should be able to feel extremely confident. However, the answer is that our software actually does not work at all well and gives the wrong answer for every score which is between 60 and 70 because all such scores should pass the test which is 60.

Therefore, one measure of the worth of an adequacy criterion is the confidence that we can justifiably have in the correctness of the software, given that no faults were exposed by a test set satisfying the criterion.

### 3.4 Exit Criterion

Random testing is a software testing process in which the objective is to measure the reliability of the software rather than to discover software faults. The user of a system is interested in the probability of failure–free behaviour. Following this line of thought, random testing which has a high fault–revealing power provides a method for determining test data sets, in spite of a tricky link between the adopted criterion and the actual faults [2, 22]. Exercising each subdomain defined from an imperfect criterion only once is far from being enough to provide an efficient test set. An obvious improvement consists in exercising each subdomain several times. In the other words, the execution of a large number of test cases that represent typical usage scenarios is required.

The exit criterion can be based on a reliability measure when the test set has been selected randomly from an appropriate probability distribution over the input domain.

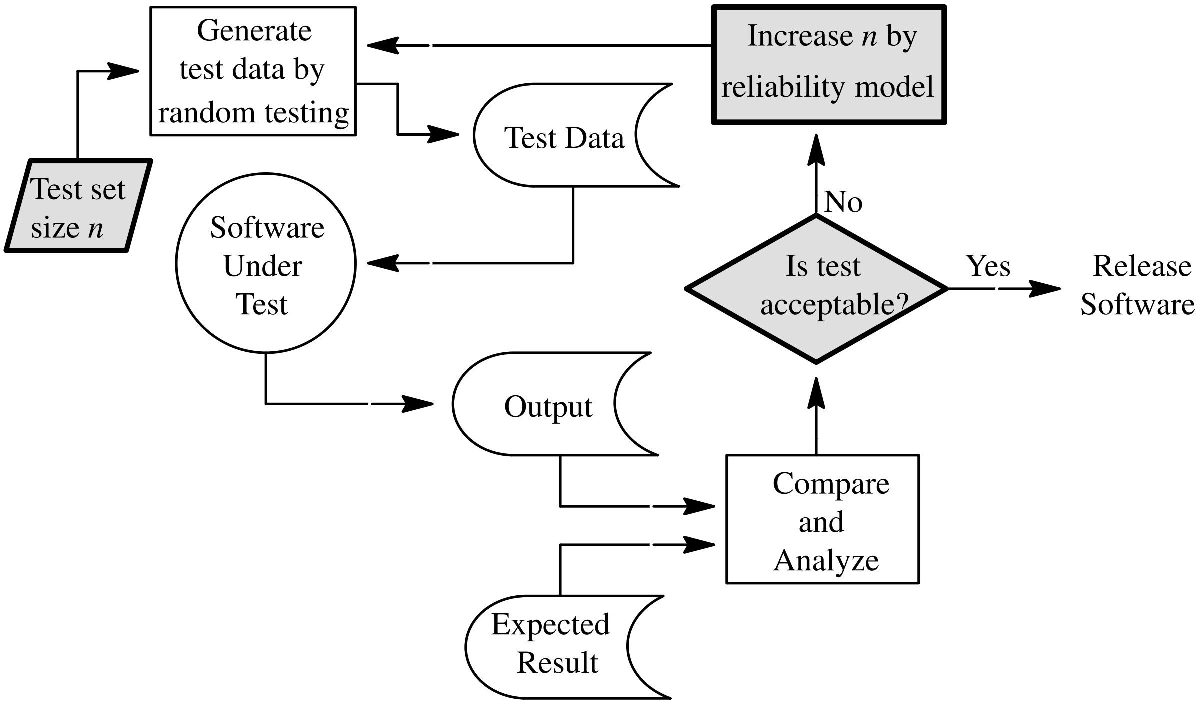


Figure 6: A model of the test data adequacy

The basic procedure shown in Figure 6 is

- to determine the size $n$ of test set and select test cases from an input distribution,
- to execute the software under test,
- to record the amount of execution time between failure [7] or estimate the defective rate of the output population [36],
- to continue testing until the selected model shows that the required failure–intensity level has been met to the desired level of confidence like "we can accept the testing to say with 95% confi–dence that the probability of 1,000 CPU hours of failure–free operation in a probabilistically de–fined environment is at least 0.995."

### 3.5 Test Quality Measurement

#### 3.5.1 Software Reliability Measures

A number of software reliability measures have been defined and they have been used in software reliability analysis [9]. Testing a piece of software is equivalent to finding the failure rate of the product unit population generated by the software. The failure rate is defined as the ratio of the number of product units that are defective to the total number of product units that the software has generated. The total number of product units, denoted by $N$, of any non–trivial piece of software ranges from extremely large to infinite, but can still be treated as an object of statistical interest. Although impossible in practice, it can be conceptually assumed that all $N$ units have been produced and analyzed. Each of them can be classified as defective or non–defective. If there are $D$ units that are defective, then the product unit population failure rate, denoted by $\theta$, is $\theta = D/N$. Since it is impossible to obtain all $N$ units, the best approach is to estimate  by means of statistical sampling. This definition is applicable to conventional batch processing environments and real–time systems dealing with discrete operations. For real–time systems dealing with continuous streams of data, a more realistic index is mean time between failures (*MTBF*) or mean time to failure (*MTTF*). It is expressed as $MTBF = t/F$ where $t$ is the predetermined total running time and $F$ is the total number of failures in the interval $[0,t]$. The failure rate is then $u = 1/MTBF$. In software systems, components do not wear out and after a single failure usually remain operational. Therefore, *MTBF* or *MTTF* are only useful in software reliability assessment when the system is stable and no changes are being made to it. In this case, it provides an indication of how long the software will remain operational before a failure occurs.

### 3.5.2 Test Data Adequacy Criteria As Continuous Measures

Test data adequacy criteria are measures of the quality of testing, that is, they measure the ability of a test set to reveal a particular feature of a software. Hence, a degree of adequacy is associated with each test set. Indeed, the coverage of code as a percentage is often used as an adequacy measure. Therefore, test sets are not simply classified into good or bad. Thus, an adequacy criterion $C$ can be formally defined to be a function $C$ from a program $p$, a specification $s$ and a test set $t$ to a real number, the degree of adequacy [10]. The greater the real number is the more adequate the testing.

Test adequacy criteria is divided into fault–based testing and error–based testing by the underlying testing approach. Fault–based testing focuses on the detection of faults in the software. An adequacy criterion of this approach is some measure of the ability of test sets to detect the faults in the software. Error–based testing requires test cases to check the program on certain error–prone points identified by the testers' previous empirical knowledge about software errors.

### 3.5.3 Experimental Works

Basili and Selby [37] looked at code reading by stepwise abstraction (static testing), functional testing using equivalence partitioning and boundary value analysis (black–box testing) and statement testing (white–box testing) in three aspects of software testing–fault detection effectiveness, fault detection cost and classes of faults detected. They found that, among the professional programmers, code reading detected more faults and had a higher fault detection rate than both functional and statement testing. Functional testing discovered more faults than statement testing, but the detection rate was the same. Among the less–experienced programmers, in one group statement testing was outperformed by the other techniques, and in the other group no difference was found. Code reading detected more interface faults and functional testing detected more control faults.

The efficiency of deterministic and random testing methods has been exemplified by experimental work performed on a program which is a piece of a software from the nuclear field using mutation analysis [38]. Mutation analysis [39] is a technique for the measurement of test data adequacy. In practice, a tester interacts with an automated mutation system to determine the adequacy of the current set of test data and to improve that test data. This forces the tester to test for specific types of faults. These faults are represented by a set of simple syntactic changes to the test program. These changed programs are mutants of the original and a mutant is killed by distinguishing the output of the mutant from that of the original program. High mutation scores were observed in this work. In fact, deterministic test sets left alive only two of the entire set of 1345 mutants and random test sets killed all of them.

Although the detection of more failures does not always help us identify more faults, in practice it is often desirable to reveal as many failures as possible. Intuitively, the more failures the testing reveals, the more information we are likely to get about the faults and the higher the chance of detecting more faults. From this point of view, a best or most effective testing strategy is one which results in a test suit that can discover as many failures in the program as possible. However, we believe that there is probably no single "best" measure. Whichever one is the most appropriate to use depends very much on the purpose of the testing. Different measures based on different intuition should complement each other to provide a more comprehensive understanding of deterministic and random testing.

# 4 SOFTWARE TESTING STRATEGY

## 4.1 Comparison Of Software Testing Techniques

### 4.1.1 Static Versus Dynamic

Experimental evaluation of code inspections and code walkthroughs has found these static testing techniques to be very effective in finding 30% to 70% of logic design and coding errors in a typical software [9, 40]. Mills et al. [41] suggest that a more formal approach, using mathematical verification, can detect more than 90% of the errors in a program. Gilb and Graham [42] also found that static testing was more effective and less expensive than dynamic testing in discovering software faults.

Basili and Selby [39] compared the effectiveness of software testing techniques as regards detecting faults, the associated costs and the kinds of faults found. They experimented using four different software systems and a variety of programmer expertise. They found that, among the professional programmers, code reading detected more faults and had a higher fault detection rate than both functional and statement testing. Functional testing discovered more faults than statement testing, but the detection rate was the same. Among the less–experienced programmers, in one group statement testing was outperformed by the other techniques, and in the other group no difference was found. Code reading detected more interface faults and functional testing detected more control faults.

Lauterbach and Randall [43] compared branch testing, functional testing, random testing and some static analysis techniques including code reviews. On average, code reviews were found to be the most effective, but in many instances were out–performed by branch testing. Of the testing techniques, the highest coverage level was achieved by branch coverage.

#### 4.1.2 Black–box Versus White–box

Poston and Sokolsky [44] applied both black and white–box testing to the Myers Triangle Problem [1] to show how each kind of testing performs and the results that each produces. Experimental results from Poston and Sokolsky were:

1. The white–box testing approach produced test cases that exercised or covered 100 percent of the statements, branches and control flow paths in the code. So TQ in terms of code coverage was high. However, those test cases did not find four defects in the code, so TQ in terms of defect or failure detection was low. In addition, the example showed that white–box testing did not address requirements, so TQ in terms of the requirements coverage was unknown.

2. The black–box testing approach produced test cases that exercised 100 percent of the requirements and probed large samplings of the input domain and output range. TQ relative to requirements coverage was high. When the black–box test cases were exercised, they caused the execution of 100 percent of the statements, branches, and control flow paths in the code. Therefore, TQ relative to code coverage was high. In addition, the black–box test cases found defects that were missed by the white–box test cases. TQ pertaining to defect detection was higher with black–box testing than with white–box testing.

#### 4.1.3 Deterministic Versus Random

The empirical work on the comparison of deterministic and statistical (random) testing with respect to the percentages of revealed faults was presented by Thévenod–Fosse and Waeselynck [21]. For purposes of comparison, program faults have been divided into two classes, 'regular' faults and 'marginal' faults. A simple example in this work illustrates the difference between them. More generally, regular faults are those which lead to failure due to the incorrect behaviour of the software in respect of any one test criterion used to select test inputs, while the remainder are marginal. In the case of regular faults, selected deterministic input data are appropriate for ensuring that they are uncovered while random data provide only a high, but always less than 1, probability of revealing them. Hence, it can reasonably be expected that statistical testing reveals a lower percentage of regular faults. On the other hand, in the case of marginal faults, no data specifically aimed at revealing these faults has been purposely included in the set of deterministic patterns. The probability of revealing these faults is then an increasing function of the number of executions. Therefore, statistical testing should reveal a higher percentage of marginal faults. Since a real limitation of current test criteria is their lack of connection with the actual faults, most of the faults are likely to be marginal. Hence, the fault revealing power of statistical testing should not be ignored.

### 4.2 Proposal Of Strategy

As shown in the previous Section (4.1) there are two points of view about the relative merits re static versus dynamic testing. Some researchers have suggested that static testing techniques should completely replace dynamic testing techniques in the verification and validation process and that dynamic testing is unnecessary. However, static testing can only check the correspondence between a program and its specification but it cannot demonstrate that the software is operationally useful. Although static testing techniques are becoming more widely used, dynamic testing is necessary for reliability assessment, performance analysis, user interface validation and to check that the software requirements are what the user really wants. Today, if we want to prevent all the faults that we can and expose those that we cannot prevent, we must review, inspect, read, do walkthroughs and then test, that is, use the static testing first and the dynamic testing later.

In current dynamic testing the strategies used are deterministic test data, random test data or both. In practice, the choice of the strategy is most often related to various factors such as available testing tools, time limit, allocated budget, personal's cultural background and usage. Since random testing makes minimal use of the available knowledge about the software, one might expect it to be very ineffective when compared with deterministic testing techniques. However, a number of practitioners still propose the use of random testing, especially for the final testing of software. The dynamic testing strategy advocated here combines deterministic and random testing. The way to mix the two testing techniques is deduced from their complementary features, that is, to use the deterministic testing techniques first for removing the more easily discovered faults and to use the random testing techniques later for assessing the reliability of the resulting software. The strategy proposal for software testing is shown in Figure 7.

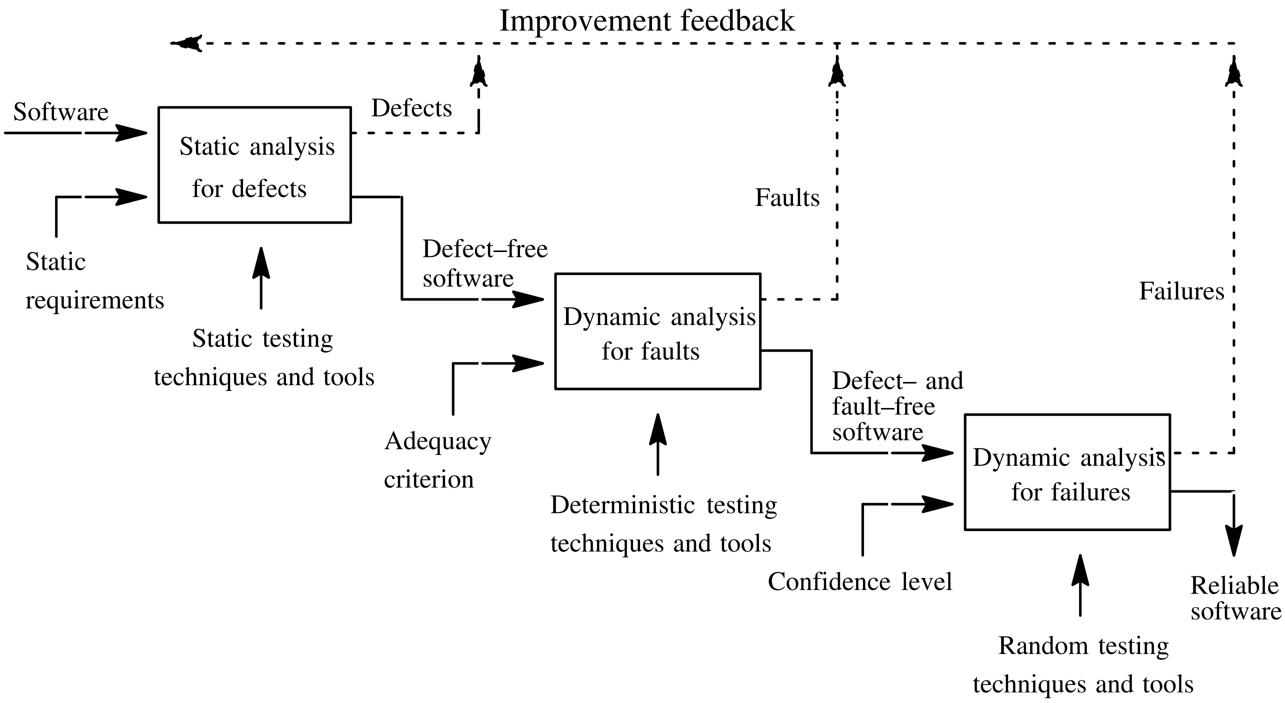


Figure 7: The strategy proposal for software testing

## 5 CONCLUSION

To achieve software quality, software testing is an essential component in all software development. Software testing is characterized by the existence of many methods, techniques and tools, that must fit the test situation, including technical properties, goals and restrictions. There is no single ideal software testing techniques for assessing software quality. Therefore, we must ensure that the testing strategy is chosen by the combination of testing techniques at the right time on the right work products. From this viewpoint, a scheme for evaluating software testing techniques is presented to the classification and evaluation of software testing techniques. A strategy proposal for software testing is also advocated in this paper. We expect that the proposal will provide a guide-line to testers in the development of applications.

**Acknowledgments**

The Author would like to thank Professor John Dobson for his constructive suggestions which significantly improved this paper. Additionally, the author is funded by the National Science Council in Taiwan from whom he received a fellowship to work toward a doctoral degree.

## References


[1] MYERS, G.: 'The art of software testing' (John Wiley & Sons, 1978)

[2] VLIET, H.: 'Software engineering: principles and practice' (John Wiley & Sons, 1994)

[3] GELPERIN, D., and HETZEL, B.: 'The growth of software testing', *Comm. ACM*, 1988, **31**, (6), pp. 687–695

[4] IEEE: 'IEEE standard for software test documentation: IEEE/ANSI standard 829–1983' (IEEE, 1983)

[5] KIT, E: 'Software testing in the real world: improving the process' (Addison–Wesley, 1995)

[6] HAMLET, D.: 'Special section on software testing', *Comm. ACM*, 1988, **31**, (6), pp. 662–667

[7] MUSA, J., and ACKERMAN, A.: 'Quantifying software validation: when to stop testing?', *IEEE Softw.*, 1989, May, pp. 19–27

[8] PARNAS, D., SSHOUWEN, J., and KWAN, S.: 'Evaluation of safety–critical software', *Comm. ACM*, **33**, (6), pp. 636 – 651

[9] DEMILLO, R., MCCRACKEN, W., MARTIN, R., and PASSAFIUME, J.: 'Software testing and evaluation' (The Benjamin/Cummings, 1987)

[10] ZHU, H., HALL, P., and MAY, J.: 'Software test coverage and adequacy', technical report, The Open University, 1994

[11] OULD, M., and UNWIN, C.: 'Testing in software development' (Cambridge Univ. Press, 1986)

[12] ROPER, M.: 'Software testing' (McGraw–Hill, 1994)

[13] SOMMERVILLE, I.: 'Software engineering' (Addison–Wesley, 1996)

[14] HUMPHREY, W. 'Managing the software process' (Addison–Wesley, 1989)

[15] FAGAN, M.: 'Design and code inspections to reduce errors in program development', *IBM Syst. j.*, 1976, **15**, (3), pp. 182–211

[16] DEMILLO, R., LIPTON, R., and PERLIS, A.: 'Social processes and the proofs of theorems and programs', *Comm. ACM*, 1979, **22**, (5), pp. 271–280

[17] BEIZER, B.: ‘Black–box testing: techniques for functional testing of software and systems’ (John Wiley & Sons, Inc., 1995)

[18] RAPPS, S., and WEYUKER, E.: ‘Selecting software test data using data flow information’, *IEEE Trans. Softw. Eng.*, 1985, **11**, (4), pp. 367–375

[19] BERTOLINO, A.: ‘An overview of automated software testing’, J. Syst. Softw., 1991, **15**, pp. 133–138

[20] LAPRIE, J.: ‘Dependability – its attributes, impairments and means’, In *Predictably dependable Computing Systems*, (Springer, 1995), pp. 3–18

[21] THÉVENOD–FOSSE, P., and WAESELYNCK, H.: ‘An investigation of statistical software testing. *J. Softw. Testi., Veri. and Reli.*, 1991, **1**, (2), pp. 5–25

[22] THÉVENOD–FOSSE, P., and WAESELYNCK, H., and CROUZET, Y.: ‘Software statistical testing’, In *Predictably dependable computing systems*. (Springer, 1995), pp. 253–272

[23] DURAN, J., and NTAFOS, S.: ‘An evaluation of random testing’. *IEEE Trans. Softw. Eng.*, 1984, **10**, (4), pp. 438–444

[24] HAMLET, D., and TAYLOR, R.: ‘Partition testing does not inspire confidence’, *IEEE Trans. Softw. Eng.*, 1990, **16**, (3), pp. 202–213

[25] WEYUKER, E., and JENG, B.: ‘Analyzing partition testing strategies’, *IEEE Trans. Softw. Eng*, 1991, **17**, (7), pp. 703–711

[26] FRANKL, P., and WEYUKER, E.: ‘A formal analysis of the fault–detecting ability of testing methods’, *IEEE Trans. Softw. Eng.*, 1993, **19**, (3), pp. 202–213

[27] CHEN, T., and YU, Y.: ‘On the expected number of failures detected by subdomain testing and random testing’, *IEEE Trans. Softw. Eng.*, 1996, **22**, (2), pp. 109–119

[28] CHEN, T., and YU, Y.: ‘On the relationship between partition and random testing’, *IEEE Trans. Softw. Eng.*, 1994, **20**, (12), pp. 977–980

[29] CURRIT, P., DYER, M., and Mills, H.: ‘Certifying the reliability of software’, *IEEE Trans. Softw. Eng.*, 1986, **12**, (1), pp. 3–11

[30] WHITTAKER, J., and THOMASON, M.: ‘A markov chain model for statistical software testing, *IEEE Trans. Softw. Eng.*, 1994, **20**, (10), pp. 812–824

[31] BEIZER, B.: 'Software testing techniques' (Van Nostrand Reinhold, 1990)

[32] WEYUKER, E.: 'Axiomatizing software test data adequacy', *IEEE Trans. Softw. Eng.*, 1986, **12**, (12), pp. 1128–1138

[33] PARRISH, A., and ZWEBEN, S.: 'Analysis and refinement of software test data adequacy properties', *IEEE Trans. Softw. Eng.*, 1991, **17**, (6), pp. 565–581

[34] GOODENOUGH, J., and GERHART, S.: 'Toward a theory of test data selection', *IEEE Trans. Softw. Eng.*, 1975, **1**, (2), pp. 156–173

[35] SPILLNER, A.: 'Test criteria and coverage measures for software integration testing', *Softw. Qual. J.*, 1995, **4**, pp. 275–286

[36] CHO, C.: 'Quality programming: developing and testing software with statistical quality control' (John Wiley & Sons, 1988)

[37] BASILI, V., and SELBY, R.: 'Comparing the effectiveness of software testing strategies', *IEEE Trans. Softw. Eng.*, 1987, **13**, (12), pp. 1278–1296

[38] MARRE, B., THÉVENOD–FOSSET, P., WAESELYNCK, H., GALL, P., & CROUZET, Y.: 'An experimental evaluation of formal testing and statistical testing', In *Predictably de–pendable computing systems*. (Springer, 1995), pp. 273–281

[39] DEMILLO, R., LIPTON, R., and SAYWARD, F.: 'Hints on test data selection: help for the practicing programmer', *IEEE Comp.*, 1978, **9**, (4), pp. 34–41

[40] FAGAN, M.: 'Advances in inspections', *IEEE Trans. Softw. Eng.*, 1986, **12**, (7), pp. 744–751

[41] MILLS, H., DYER, M., and LINGER, R.: 'Cleanroom software engineering', *IEEE Softw.*, 1987, Apr., pp. 19–25

[42] GILB, T., and GRAHAM, D.: 'Software inspection' (Addison–Wesley, 1993)

[43] LAUTERBACH, L., and RANDALL, W.: 'Experimental evaluation of six test techniques'. Proc. COMPASS 89, 1989, pp. 36–41

[44] POSTON, R.: 'Automating specification–based software testing' (IEEE Computer Society, 1996)