FAST: A Framework for Automating Statistics–based Testing

Huey–Der Chu, John E Dobson
Centre for Software Reliability, Department of Computing Science,
University of Newcastle upon Tyne, NE1 7RU, UK

ABSTRACT

To achieve software quality, testing is an essential component in all software development. It involves the execution of a deterministic software system with test data and a comparison of the results with the expected output, which must satisfy the users' requirements. This accounts for over 25% of the cost of a software development. Therefore, automation has considerable potential. The quality programming which was introduced by Cho can automatically generate data for testing, based on a so–called 'SIAD tree' which is used to represent the hierarchical and "network" relation between input elements and also incorporates rules into the tree for using the inputs. However, it lacks a clear framework which would show how automated testing can be achieved. To address this problem, we present a Framework for Automating Statistics–based Testing (FAST), which is an extension of the testing concept in quality programming to achieve automated testing. In FAST, we propose a SOAD tree which is similar to the structure of the SIAD tree to describe the syntactic structure of the product unit and its defectiveness. Based on this tool, the inspection of test results can be automatically achieved by lexical and syntax analysis. The implementation of automated software testing for Command File Interpreter (CFI) software which incorporates the framework is also described.

Keywords: Software Quality, Automated Software Testing, Statistical Approach

1. INTRODUCTION

With the rapid development of computer technology, the applications of information systems have become more and more popular and products can be found everywhere. These cause the increasing dependence of most users on their information systems and the concomitant heavily increasing costs of failure. The production of high–quality information software systems is an important issue for the near future (Musa, 1990). Software quality is the degree to which a customer or user perceives the software as meeting his or her composite expectations (Deutsch and Willis, 1988). To achieve software quality, it is essential that the software is tested. This is not only a developmental activity for discovering product defects but also an independent assessment of software execution in an operating environment. It involves the execution of a deterministic software system with test data and a comparison of the results with the expected output which must satisfy the users' requirements. It is a very time–consuming and tedious activity and accounts for over 25% of the cost of software development (Ince, 1987; Myers, 1978; Norman, 1993). If the testing process could be automated, the cost of developing software could be reduced significantly.

It is a well known fact in the software industry that software of any complexity cannot be exhaustively tested and that a sample of the possible inputs must be relied on for the testing performed. The conventional testing techniques based on the deterministic method (Marre *et al.*, 1995) ask the tester to select these peculiar inputs to test peculiar cases by means of test criteria. It may discover many errors but may not provide much improvement in the product's quality. It is also accepted that errors can have significantly different effects on the failure rate of software and that a greater payoff comes from discovering and removing the errors with high failure rates during testing. Statistically based testing with random sampling driven from input probability distributions is uniquely effective at finding errors with high failure rates. The major advantages of using the statistical method for software testing are as follows (Curritt *et al.*, 1988; Whittaker and Tomason, 1994): 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 the analysis of test results.

Current statistical testing techniques involve exercising a piece of software by supplying it with test data that are randomly drawn according to a single, unconditional probability distribution on the software's input domain (Curritt *et al.*, 1988; Dyer, 1992; Thévenod–Fosse *et al.*, 1995). This distribution represents the best estimate of the operational frequency for the use for each input. This model is not sufficient effective for many types of software, because the probability of applying an input can change as the software is executed (Whittaker and Tomason, 1994). Cho

(1988) specifies the input domain of a software by means of a "*Symbolic Input Attribute Decomposition*" (*SIAD*) tree, which is a syntactic structure describing the characteristics of all possible input data. The SIAD tree is a way to achieve clarity, conciseness, completeness and measurability in the specification of input requirements. It enforces the development of well–defined requirements and imposes disciplines in both design and implementation. From a fault forecasting point of view, a comparative analysis (Thévenod–Fosse and Waeselynck, 1991) concluded that the best evaluation is provided by Cho's approach, particularly when few failures are observed during a test experiment.

Quality programming introduced by Cho (1988) can automatically generate data for testing, based on the SIAD tree, but it lacks a clear framework with which to tell us how to achieve automated testing. This paper proposes a Framework for Automating Statistics–based Testing (FAST) , which is an extension of the testing concept in quality programming to achieve automated testing.

In Section 2 of this paper, we survey the related work in software testing, particularly in statistics–based software testing. In Section 3, we present an automated software testing framework using a statistical approach and compare it with other testing methods. The implementation of automated software testing for Command File Interpreter (CFI) software incorporating a FAST is described in Section 4. Section 5 summarizes our research and offers suggestions for further study.

2 THE STATISTICAL APPROACH FOR SOFTWARE TESTING

2.1 Software Testing Techniques

Software testing has become the accepted method for detecting and removing errors and has played a dominant role in error detection. Existing software testing techniques are divided into two categories (Ould and Unwin, 1986; Roper, 1994): static and dynamic testing. Static testing techniques are concerned with the analysis and the checking of system representations such as the requirements documents, design diagrams and the software source code, either manually or automatically, without actually executing the software (Sommerville, 1996). In contrast to this, dynamic testing techniques are those that examine the software with a view to generating test data for execution by the software.

Static testing may include the reviews and walk–throughs (Vliet, 1994) 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 (Humphrey, 1988). 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 (Fagan, 1976) and has proved to be an effective technique for the design, code and test phases. On a more rigorous level, proof of correctness during design refinement offers some help. Proof of correctness (DeMillo *et al.*, 1979; Vliet, 1994) is a mathematical method of proving that a software meets its specification. 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 precondition) into the other (the postcondition).

Fagan (1986) reported that more than 60% of the errors in a software can be detected using informal software inspections. Mills *et a*l. (1987) suggest that a more formal approach, using mathematical verification, can detect more than 90% of errors in a software. However, static testing techniques can only check the correspondence between a software and its specification (verification), but they cannot demonstrate that the software is operationally useful. Therefore, dynamic testing techniques will still be needed, even though static testing techniques are becoming more widely used.

Dynamic testing techniques are generally divided into the two categories of black–box and white–box testing (Beizer, 1995; Sommerville, 1996), which correspond with two different software testing starting points: the internal structure of the software and the requirements specification. Basili and Selby (1987) conducted an experiment to compare the effectiveness of black and white–box testing. The results of this experiment found that, with professional programmers, black–box testing was more effective in discovering faults than the white–box testing. Poston and Sokolsky (Poston, 1996) applied both black and white–box testing to the Myers Triangle Problem

(Myers, 1979) to show how each kind of testing performs and the results that each produces. This experiment also showed that the black–box test cases revealed defects that were missed by those of the white–box test and test quality pertaining to defect detection was higher with the black–box testing than with the white–box testing. Hence, in this paper, the focus is to use a statistical approach to black–box testing. The model of dynamic testing is shown in Figure 1.

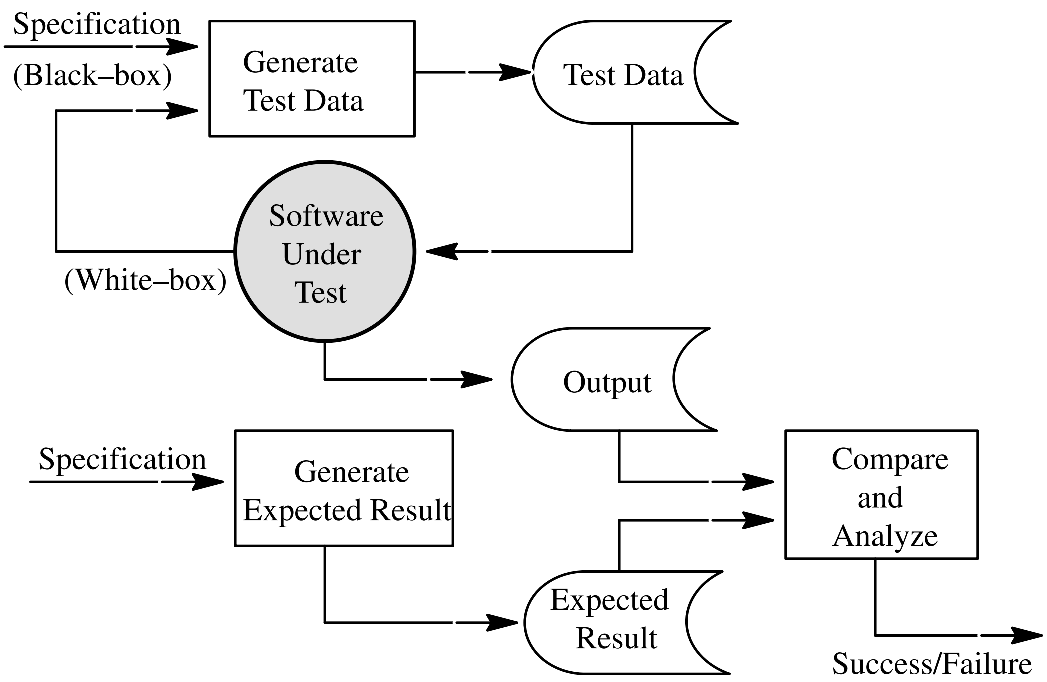

Figure 1: The model of dynamic testing

Static and dynamic testing techniques are complementary (Kit, 1995). 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 is satisfactory on its own has been pointed out by many leading researchers such as (Hamlet, 1992; Musa and Ackerman, 1989; Parnas *et al.*, 1990). The need to combine testing techniques is further substantiated when we consider the primary characteristics of each approach and find that each testing strategy addresses only a narrow set of concerns.

In this paper, we present a SIAD/SOAD tree to specify the syntactic structure of input data, product unit and product unit defectiveness. Testing can be completely automated by the statistical approach, from the generation of test data based on the SIAD tree (dynamic testing) to the inspection of test results based on the SOAD tree (static testing). (Hörcher and Peleska, 1995) describe a similar approach by Z specifications. The basic architecture of automated testing is shown in Figure 2.

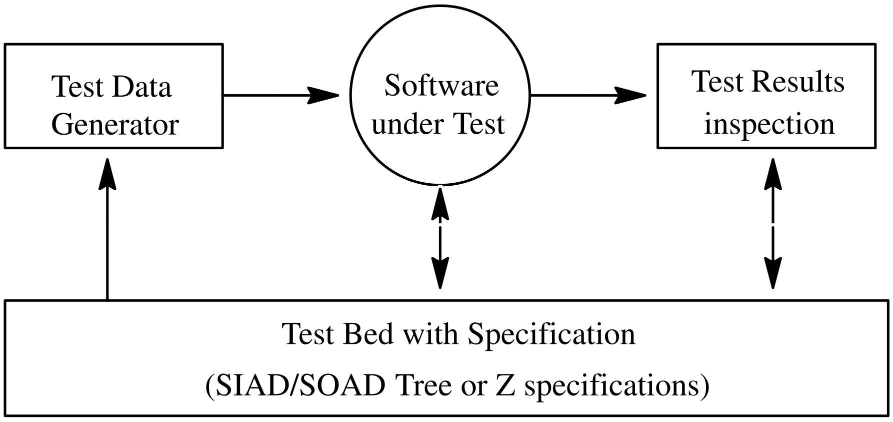

Figure 2: The basic architecture of automated testing

However, the method of test data selection in their approach was based on the deterministic testing method and two main issues of software testing (Cho, 1988; Zhu, 1996)–when to stop testing and how good the software is after testing–were only briefly discussed in this approach.

2.2 Statistical Software Testing

With a non–statistical approach, the determination of how much testing and in what order, is a subjective decision based on the tester's experience and the project's schedule. However, it cannot provide the user with a precise index in order to explain the quality of software. All we can do is to post the questions of when to stop testing a piece of software and how good the software is after testing. Statistical software testing involves exercising a piece of software by supplying it with test data that are randomly drawn according to a defined probability distribution on its input domain. It provides a scientific basis for making inferences from testing about operational environment. Therefore, if test data are randomly drawn from an input distribution representative of some particular user profile, statistical testing becomes an experimental way to determine whether or not a product meets its dependability requirements (Thévenod–Fosse and Waeselynck, 1991). The main benefit of statistical testing is that it allows the use of statistical inference to compute probabilistic aspects of the results of the testing process, such as reliability, mean time to failure (MTTF) and mean time between failures (MTBF). However, these techniques are insufficient for many types of software, because the probability of applying an input can change when the software is executed (Deck, 1996; Whittaker and Tomason, 1994).

In order to address this difficulty, we introduce the inverse concept presented by Cho (1988). Each execution of the software is considered equivalent to 'sampling' an output from the output population. The goal of software testing is to find certain characteristics of the population such as the ratio of the number of defective outputs in the population to the total number of outputs in the population. It uses the number of executions of the software to assess the software reliability, which is different from previously mentioned measures which use the execution time. Gathering the data for the error history of a piece of software requires a long period of time, and even then, the reliability measure is often difficult to quantify. However, a piece of software is not subject to deterioration such as wear, tear or burn, that is, the reliability of a piece of software is independent of time but dependent on the frequency and nature of software usage. Cho (1988) gives the following definition:

Software reliability is $1 - \theta$, *the probability that the software performs successfully, according to software requirements, independent of time.*

where θ is the defective rate of the software output population. This definition is a natural consequence of following the principles of software engineering with statistical quality control. From this point of view, determining the defective or non–defective outputs from software requires corresponding input data. The input domain is the source from which input data are constructed for the software. If the input domain is not well defined, the input data will not be properly constructed and will be of a poor quality. Following Cho, the approach adopted in this paper specifies the input domain of a software by means of a SIAD tree which is a syntactic structure representing the input domain of a piece of software in a form that facilitates construction of random test data for producing random output for quality inspection. The SIAD tree enforces the development of well–defined user requirements and imposes discipline in both design and implementation.

This approach also applies hypothesis testing to decide on the acceptance or rejection of a piece of software. The decision involves failure records during statistical testing and two different types of risk are taken into account: Firstly the acceptance of a false hypothesis (user's risk) and secondly the rejection of a true hypothesis (producer's risk). This method is outlined below.

2.3 Quality Programming

In quality programming as introduced by Cho, a statistical approach to control the quality of software is used. When a software is viewed as a factory, processing of input units into product units becomes conceptually equivalent to taking random product units from the software's population. If the product units in the population are all of a good quality, then the units taken will be of a good quality. A simplified view of the process of quality programming is shown in Figure 3.

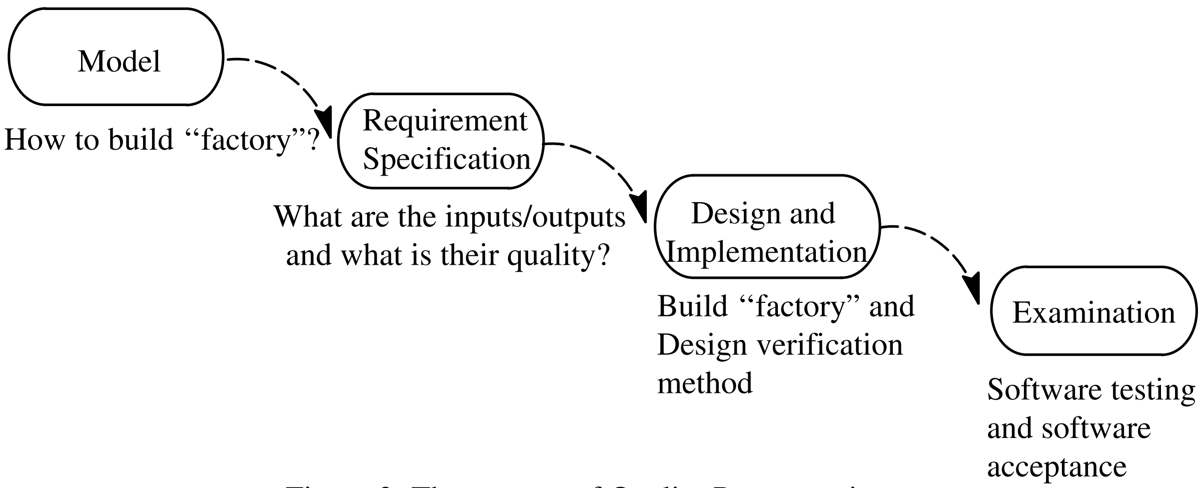

Figure 3: The process of Quality Programming

As shown, the process is divided into the following stages: model, requirements specification, design and implementation and examination. This process ensures that quality is built into software from the beginning of its development.

- Model: given a system to be developed, a model is developed to analyze and understand the problem. Models including a description of the problem and product to be generated by the software are built to form the basis of product design and the concept of the software being developed.

- Requirements specification: requirements are then generated as a result of the modeling activ– ity. Included in the requirements are a software and test requirements. Software requirements define the functions the software is to perform and the quality characteristics such as response time, throughput, understandability and portability. Test requirements define the product units and product unit defectiveness for statistical sampling, sampling methods for estimating the defective rate of the software population with which to judge software quality, statistical infer– ence methods and the confidence levels of software output population quality, the acceptable software defect rate and the generation methods of test input units.

- Design and implementation: with well–defined requirements, software development can be divided into two channels which can proceed concurrently: software design and implementa– tion and software test design and implementation. Top–Down programming and critical– module–first implementation methods are used in the software channel. The formulation of sampling plans are used in the test channel. During the design and implementation phases, interfaces between the channels are incorporated to ensure that quality is built into the software at every stage of development.

- Examination: software testing is performed again on a most–critical–module–first basis to ensure that the software is integrated on a secure–quality–part basis. If the software passes the test requirements delivery to the user takes place. The user employs quality control tools to determine the acceptability of the software output population, and this becomes the basis for accepting or rejecting the software.

3 THE SIAD/SOAD TREE

3.1 The Basic Concept Of The SIAD/SOAD Tree

One of the major problems in software development is ambiguity in requirements specification, particularly specification of input domain and product unit. The SIAD tree is a syntax structure representing the input domain of a piece of software in a form that facilitates construction of random test data for producing random output for quality inspection. It is used to represent the hierarchical relation between input elements and incorporate rules into the tree for using the inputs. In our approach, the specification of product unit is addressed by the “Symbolic Output Attribute

Decomposition" (SOAD) tree which is similar to the structure of the SIAD tree. Based on the SOAD tree, the test results of inspection can be automatically defined by lexical and syntax analysis.

Input is constructed from data of different characteristics that are called input attributes. Associated with each input attribute is a syntax structure. The structure can be decomposed into a lower level substructure and so on, until further decomposition is not possible. The lowest level substructure is called a basic element. If the basic element is numerical then the lower bound and the upper bound of the element are given under the element. The overall structure is a tree. The tree can be arranged as a linear list with the structure preserved by a set of symbols called the tree symbols. The list is called a symbolic input attributed decomposition (SIAD) tree. It is used to represent the hierarchical and "network" relation between input elements and incorporate rules into the tree for using the inputs.

The following example is a demonstration of a SIAD tree representing an input test data for a transaction in a grade report database system.

An example: a database system for a grade report

Consider a Grade Report database system that has three relations is shown in Figure 4.

STUDENT

student id	student name	
	first name	surname
945216775 :	Huey–Der :	Chu :

COURSE

course id	course name
CS2010 :	Data Base :

GRADE

student id	course id	score
945216775 :	CS2010 :	85 :

Figure 4: A database system for grade report

Query: Given a student id and several course id to get the grade report of figure 5.

A Grade Report for Huey–Der Chu		
Course id	Course name	Grade
CS2010	Data Base	85
CS2015	Algorithm	80
:	:	:

Figure 5: A Grade Report

According to this query, the input test data can be decomposed into student id and course id. A course id can be decomposed into course type and then course number. It shows that the decomposition of the components can be done at a number of levels. The results can be arranged in a tree, as shown in Figure 6.

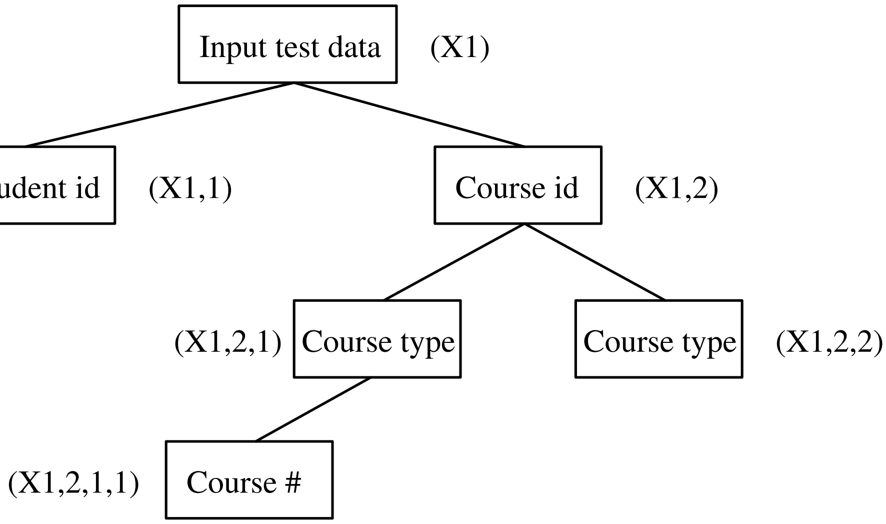

Figure 6: A tree structure of an input test data

The tree may be represented in a SIAD tree, as in Figure 7 (a). The tree has four columns: the index, the symbol, the element and the rule index. The indices are for sampling use. The symbols preserve the tree structure of the tree elements in Figure 6. A tree element is a node in Figure 6 with some explanation of the node. A rule index points to a rule that governs the use of the tree element in constructing an input unit.

index	symbol	element	rule index
1	X1	input test data	
2	X1,1	student id, K1 bytes	1, 6
3	X1,2	course id, K2 bytes	2
4	X1,2,1	course type, K3 bytes	3
5	X,1,2,1,1	course number, K4 bytes	4, 5
6	X1,2,2	course type, K3 bytes	3

(a)

rule index	rule description	subrule index
1	K1	1
2	K2 = K3+K4	2, 3
3	K3	2
4	K4	3
5	Only including characters 0,1,2,3,4,5,6,7,8,9	
6	Excluding characters + * / ' " < > $ & # @ ; :) (! ? =] [% £	

(b)

subrule index	subrule description	remark
1	$1 \leq K1 \leq 9$	length of student id
2	"CS", "AM", "ST"	type of course
3	$1 \leq K4 \leq 4$	length of course number

(c)

Figure 7: The SIAD tree of a grade report database system

The rules governing the use of tree elements in a SIAD tree can be listed as shown in Figure 7 (b). A rule index is used to identify the rule to be used. The rule description is the rule of interest. The sub–rule index indicates a sub–rule to supplement the use of the rule, as shown in Figure 7 (c). The symbol X1,2,2 is designed for constructing invalid or incomplete test data to test whether this software can detect input data error or not.

A product unit of a grade report in Figure 5 can be specified by the following SOAD tree:

index	symbol	element	rule index
1	X1	expected result	
2	X1,1	student name, K1 bytes	1
3	X1,1,1	first name, K2 bytes	2, 9,10
4	X,1,1,1,1	surname, K3 bytes	3, 9,10
5	X1,2	grade report, K4 bytes	4
6	X1,2,1	course id, K5 bytes	5, 10
7	X1,2,2	course name, K6 bytes	6, 9,10
8	X1,2,3	score, K7 bytes	7, 8

(a)

rule index	rule description	subrule index
1	K1 = K2+K3	1,2
2	K2	1
3	K3	2
4	K4 = K5+K6+K7	3,4,5
5	K5	3
6	K6	4
7	K7	5
8	integer	
9	Excluding characters 0,1,2,3,4,5,6,7,8,9	
10	Excluding characters + ∗ / ' " < > $ & # @ ; :) (! ? =] [% £	

(b)

subrule index	subrule description	remark
1	$2 \leq K2 \leq 20$	length of first name
2	$1 \leq K3 \leq 10$	length of surname
3	$1 \leq K5 \leq 10$	length of course id
4	$1 \leq K6 \leq 20$	length of course name
5	$1 \leq K7 \leq 4$	length of score value

(c)

Figure 8: The SOAD tree of a grade report database system

4 THE FRAMEWORK FOR AUTOMATED SOFTWARE TESTING

In quality programming as introduced by Cho, the generation of test data can be automatically achieved based on the SIAD tree, however, it lacks a clear framework which would indicate how automated testing is to be achieved. As an improvement, we propose a Framework for Automating Statistics–based Testing (FAST), which is an extension of the testing concept in quality programming to achieve automated testing. In FAST, we present the SOAD tree to represent the syntactic structure of the product unit and product unit defectiveness. Based on this tool, automated inspection of the product unit can be achieved. The FAST is shown in Figure 9.

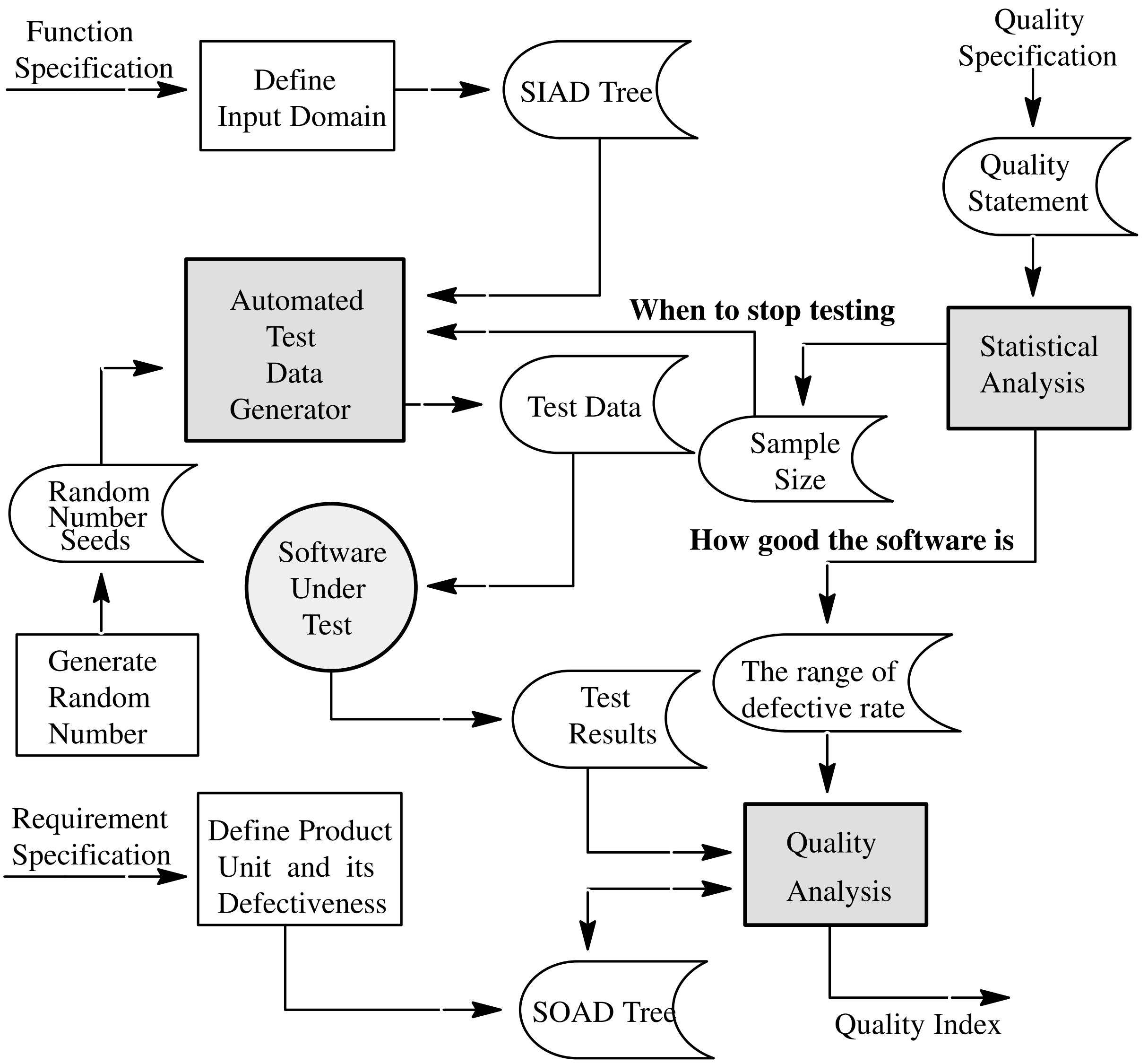

Figure 9: A framework for automating statistics–based testing

4.1 Test Requirements Specification

Test requirements are specified as the criteria for software testing and acceptance upon completion. The test requirements can be divided into two groups: functional and quality requirements. From functional requirements, we can define the input domain and the product units. To specify the quality requirements, some activities are followed to analyze the user's needs for quality, to convert the quality needs to requirements and to document the results of the software quality requirements analysis. These documents must clearly be validated by users since only they know what they want. From quality requirements, we can define the product unit defectiveness and specify the quality statement. Without the definitions of input domain , product unit and product unit defectiveness, it is not possible to control the quality of the data produced by a piece of software.

4.1.1 Define Input Domain

The input domain of a piece of software is the representation of all possible input data and rules governing the construction of input to be processed by the software. With a well–defined input domain, random test data can be constructed to test the software. The test data used in production is in fact a subset of all of the possible inputs that can be generated from this domain. The input domain of a piece of software can be represented by the SIAD tree introduced in Section III.

According to the different types of software applications, we can use a number of different types of SIAD trees (a detailed description of these trees is given in Cho). For example, in Liu *et al.* (1992), we apply the weighted and ruled SIAD trees for the Command File Interpreter (CFI) software, the regular SIAD tree for interface software in a relational database system and the regular SIAD tree for a LEX generator.

4.1.2 Define Product Unit

The purpose of a product unit is to specify the user's detailed output requirements (expected result). It is the foundation for applying statistical quality control principles. A collection of product units becomes a population from which various statistics can be studied to ascertain the quality of the product.

Conceptually, a piece of software maps an input onto an output. An output consists of a number of output data types. It may be expressed mathematically as

$$O = F(I)$$

where F is a function implemented in the software that transforms an input I into an output O. Or,

$$(O_{i1}, O_{i2}, \ldots\ldots, O_{i\mathrm{k}}) = F(I_{i1}, I_{i2}, \ldots., I_{i\mathrm{m}})$$

where i = 1,2,...., represents an input and an output

k = total number of output data types

m = total number of input data types

$(O_{i1}, O_{i2}, \ldots, O_{ik})$ is the ith output consisting of k pieces of output data $O_{i1}, O_{i2}, \ldots, O_{ik}$, which is transformed from the ith input $(I_{i1}, I_{i2}, \ldots, I_{im})$ consisting of m pieces of input data $I_{i1}, I_{i2}, \ldots, I_{im}$. Thus the output $(O_{i1}, O_{i2}, \ldots, O_{ik})$ is a product unit and the input $(I_{i1}, I_{i2}, \ldots, I_{im})$ is an input unit. Each product unit is produced from a unique input unit. Therefore, the definition of the product unit can also be given in terms of the input unit.

In the approach described by Cho, the definition of product unit is described in the requirement specification document, therefore, the inspection cannot be done automatically. In our approach, the specifiction of product unit is addressed by the SOAD tree. Based on the SOAD tree, the product unit of inspection can be automatically defined by lexical and syntax analysis.

4.1.3 Define Product Unit Defectiveness

Once the product unit definition is given, it is essential to define product unit defectiveness in judging the goodness of a product unit produced by the software being developed. The product unit of software is subject to inspection which determines whether or not the product unit satisfies the user's requirement. The result of the inspection may be divided into two categories: defective and non–defective. The classification depends on the unit's conformance to the requirements. For example, a clock is a product unit. The defectiveness of the unit is defined in terms of inaccuracy (seconds/day). The definition of product unit defectiveness can be described in the SOAD tree as the rules governing the inspection of the product unit.

4.1.4 Specify The Quality Statement

With the definitions of product unit and product unit defectiveness, the acceptance of a piece of software is defined in terms of the output population that the software generates. In FAST, the quality statements define software quality that is equivalent to $p\%$ of the output population as being non–defective (the confidence level). For example, if the user requires 95% of the output population to be non–defective, the acceptability of the population can be determined by statistical means as follows.

4.2 Statistical Analysis

Testing a piece of software is equivalent to finding the defect rate of the product unit population generated by the software. The defect 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 defect rate, denoted by θ, is $\theta = D/N$. Since it is impossible to obtain all N units, the best approach is to estimate by means of statistical sampling . If the population is conceptually shuffled, it provides a basis for applying the principle of binomial distribution sampling. The application of the distribution often arises when sampling from a finite population consisting of a finite number of units with replacement, or from an infinite population consisting of an infinite number of units with or without replacement. The probability of getting x defectives in a sample of n units taken from a population having a defect rate of θ is given by the binomial distribution:

$$\mathrm{b}(x) = \binom{n}{x} \theta^{x}(1 - \theta)^{n-x}$$

The mean and variance of the distribution are given by:

$$\mu = n\theta$$

$$\sigma^2 = n\theta(1 - \theta)$$

A sample of n units is taken randomly from the population. If it contains d defective units, then the sample defect rate, denoted by θ^0, is $\theta^0 = d/n$. If n is large enough, then the rate θ^0 can be used to estimate the product unit population defective rate θ. These two major testing issues are discussed in the following sections.

4.2.1 How Good The Software Is After Testing

The defect rate of the population can then be estimated from d. The estimate may be expressed in an interval called *100c%* confidence interval, where $0 \le c \le 1$. An approximation of the *100c%* confidence interval of the population defective rate may be computed by:

$$\left[\theta^0 - t_{n-1,\alpha/2}\frac{\sqrt{\theta^0(1-\theta^0)}}{n}\ ,\ \theta^0 + t_{n-1,\alpha/2}\frac{\sqrt{\theta^0(1-\theta^0)}}{n}\right] \qquad (4.1)$$

where $t_{n-1,\alpha/2}$ is called the value of the *Student t–distribution* at $n-1$ degrees of freedom and $\alpha = 1 - c$ is called a risk factor (In statistics, a binomial distribution can be approximated by a normal distribution). Formula (4.1) can be used to estimate the mean of the product unit population, denoted by μ . Once the value of μ is estimated, the product unit population defect rate θ can be computed by $\mu = n\theta$. If the value of θ is acceptable, then the product unit population is acceptable. The piece of software is acceptable only when the product unit population is acceptable. Therefore, the estimated product unit population defect rate θ can be viewed as the software quality index.

4.2.2 When To Stop Testing

The accuracy of the estimates depends on the sample size. In general, the larger the size, the more accurate the estimate. The value of n may be computed by the formula:

$$n = \frac{z^2(1-\theta)}{\alpha^2\theta} \qquad (4.2)$$

where α is the desired accuracy factor such that $|\theta - \theta^0| = \alpha\theta$, and z is the value of $z_{\alpha/2}$, which is the number of standard normal deviate in the normal distribution such that the area to its right under the normal curve is $\alpha/2$. The value of $z_{\alpha/2}$ is the same as $t_{n-1,\alpha/2}$ if n is large, e.g., $n \geq 30$. Since the population defect rate θ is unknown, the determination of n requires dynamic adjustment during sampling. An adjustment procedure, which is iterative in nature, is given as follows:

Step1: Take an initial sample of a small size, n_0 units (e.g., 50) from a software product population by executing n_0 input units.

Step2: Let θ_0^0 be the defect rate of the sample of size n_0,

Step3: Compute the sample size n_{i+1} by formula (4.2) as follows:

where θ_i^0 is the cumulative defect rate of the cumulative sample units n_i already taken after the ith iteration, for $i = 0,1,2,...,$

Step4: If $n_{i+1} > n_i$, then take $(n_{i+1} - n_i)$ additional units and repeat Step3 and Step4.

$$n_{i+1} = \frac{z^2(1 - \theta_i^{\,0})}{\alpha^2\theta_i^{\,0}}$$

Step5: Else stop. The total number of sample units taken is sufficient.

The final sample defect rate is then used to estimate μ and σ^2.

Whatever factory it is almost impossible to produce a defect–free product lot: therefore, the conformance of product quality is usually measured by the defect rate being less than an acceptable number, e.g., $\theta < 0.01$. With a statistical sampling method, a confidence level of 98% certainity can be imposed on the final value of the estimated defect rate.

4.3 Automated Test Data Generator

A SIAD tree can be used as a tool for describing the input domain of a piece of software and as a basis for automated test data generation through random sampling. The process of automated test data generation is as follows:

Step1: Determine the sample size *n* by statistical analysis,

Step2: Generate the number of test data *M* for each sample by a random number seed,

Step3: The construction of test data using a SIAD tree can be accomplished as follows:

3.1 Let *K* be the number of elements in the SIAD tree. Each element in the tree is indexed by a number ranging from 1 to *K*. A random number selected from [1,*K*] is produced by using a random number generator.

3.2 The element with its index equal to the random number is selected.

3.3 If the element has a parent in the SIAD tree, then backtrack to select it.

Step4: A total of *M* elements will be randomly sampled from a tree for designing test data.

For example, there are 6 elements in the SIAD tree of Figure 8. A test data includes one student id and several course id. The student id is generated from the index 2 of the SIAD tree. Two course id are to be chosen for a sample using random number generation producing 5 and 6. The elements in index 5 and 6 are drawn for constructing the test data with the student id. According to this process, the test data can be generated as Table 1:

Index	Tree Symbol	Tree Element	Remarks
2	X1,1	student id	Sampled element
4	X1,2,1	CS	Descriptive element
5	X1,2,1,1	0210	Sampled element
6	X1,2,2	AM	Sampled element

Table 1: A test data is drawn from SIAD tree

In this sampling, it takes two course id – 'CS0210' and 'AM'. The second course id is used to conduct the invalid test (incomplete data).

4.4 Quality Analysis

If the software output is defined in terms of the "product unit", then the output is a collection of product units called the output population of the software. For any non–trivial software, the population contains a very large number of units. The goal of software testing is to find certain characteristics of the population such as the ratio of the number of defective units in the population to the total number of units in the population. The ratio may be called the defect rate of the population and may be imposed on the software as the software quality index.

4.4.1 Inspect The Test Results

The result of executing an input by the software can be classified into two categories: defective and non–defective. Each product unit must be carefully analyzed for its conformance to the software requirements in order to reach the classification. The outcome of the analysis leads to classifying the output into either of the categories which, in turn, results in the acceptance or rejection of the software. Any unfair bias can increase the producer's risk of having good software rejected or can increase the user's risk of accepting poor software.

Test results can be inspected by manual, semi–manual or automatic means, which depend on software applications. A SOAD tree can be used as a tool for describing the expected result which satisfies the user's requirement and as a basis for analyzing the product unit automatically, particularly in non–numerical applications such as interpreter and updating a data base. It is a data structure containing a record for each output element with fields for the attributes of the output element.

Based on the SOAD tree, the process of automated test result analysis is shown as in Figure 10:

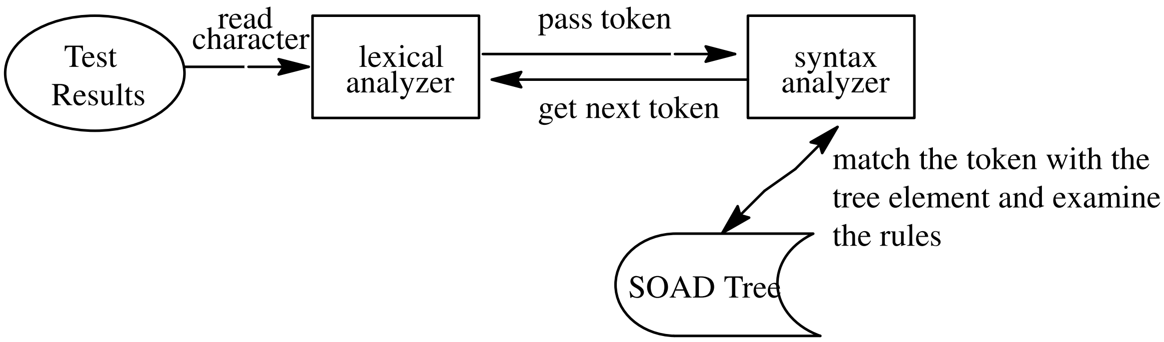

Figure 10: The process of test result analysis

The lexical analyzer is the first phase of inspection. Its main task is to read the characters of the test results and produce as an output a sequence of tokens. In this process, the syntax analyzer obtains a string of tokens from the lexical analyzer, as shown in Figure 7, and verifies that the string is defective or non–defective by matching the token with the tree element and examining the rules in the SOAD tree. According to the different types of software applications, the algorithm of inspection based on its SOAD tree can be separately designed. The main advantage of using the SOAD tree here is that we do not need a test oracle to computer expected results. The SOAD tree can be used directly for automatic inspection whether or not the results produced by the software are correct.

For example, the test result of the grade report part in Figure 5 produced as 'CS2010 Data Base A5' is defective because this type of score goes against the rule 8 in (b) of Figure 8.

4.4.2 Perform Statistical Inference

The sampling processing procedure discussed in 3.2 above represents the drawing of a product unit at random from a binomial distribution. If the number of defective product units in the sample is less than the tolerable number of defectives determined by the selected sampling plan, then the software can be delivered to users as acceptable. Otherwise, the developer should improve the quality by correcting the errors found during the test.

5 DISCUSSION

5.1 The Advantages Of FAST

The major advantages of FAST are: Firstly, testing can be completely automated using a statistical approach, from the generation of test data based on the SIAD tree to the inspection of test results based on the SOAD tree; secondly, changing distributions do not need to be acknowledged since the SIAD tree is static; thirdly, the software quality can be assessed using statistical techniques (such as sampling or inference); fourthly, the test data do not need to be stored for regression testing, as it only requires a small space in which to keep the random number seeds; fifthly, after the specification of requirements is developed, the generation of test data is independent from the software design and implementation; sixthly, we do not need the test oracle to compute expected results and finally, testing can be performed based on the user's actual execution of the software.

5.2 Comparison Of FAST And Other Testing Approaches

A comparison of FAST with Clearnroom (Dyer, 1992) and formal testing (deterministic testing from formal specification) presented by Hörcher and Peleska (1995) is shown in Table 2.

	Formal testing	Cleanroom	FAST
Test data selection	deterministic	random	random
Dependability evaluation	biased by the selective choice of the test inputs	unbiased by using the operational input profile	unbiased by using random sampling on output population
# test data	low	high	high
Test data generation	automatic	automatic	automatic
Output inspection	automatic	manual	automatic
Sampling	no	input domain	output population (SIAD tree)
Test data storage	yes	yes	no (keep random number seeds)
Reliability assessment	no	execution time	execution number

Table 2: Software testing method comparison

5.3 Proposal Of Strategy

In any software factory, it is difficult to attain fault–free software. If users require high–quality software, the cost of software development is correspondingly high. In comparing FAST and other testing methods, we find that there is more front–end test planning in FAST in developing the SIAD/SOAD tree, but this is effectively balanced by less test operation, since testing can be automatically achieved. We now discuss the relation between FAST and other testing techniques. Current software testing strategies use either conventional testing approaches, statistical testing approaches or both. The strategy of software testing advocated here is to use FAST on the most critical module and to use other conventional testing techniques on the remaining modules, or to use the conventional testing techniques first for removing the more easily discovered faults and use FAST for assessing the quality of the resulting software. The best way to mix the testing techniques is deduced from an analysis of their complementary features.

6 CONCLUSION

Summary Of Research

High–quality software is the trend of software development in the future. FAST, which is based on a statistical approach, has been proposed to help develop good quality and cost effective software. It addresses the two major software testing issues: when to stop testing and how good the software is after testing. FAST can automatically generate test data with an iterative sampling process which determines the sample size n; the software quality can be estimated with the inspection of test results which can be automatically achieved and the product unit of population defective rate θ which can be estimated from the sample defective rate θ^0. The basis of this framework is the definition of the input domain, of the product unit and of product unit defectiveness by the SIAD/SOAD tree.

Suggestions For Further Study

- Different types of software applications require their own special testing tools, especially in the SIAD/SOAD tree types. Therefore, it is necessary to develop a general tool which can build different SIAD/SOAD files for different applications.
- As a result of indeterminacy, it is difficult to know the possible execution behaviours of a distributed software, to exactly identify the execution behaviour to be tested and to control the software execution for testing a specific execution behavior. Our current work seeks to extend the SIAD/SOAD tree to define a test case, which consists of an input message plus a sequence of intermediate messages corresponding to messages in the distributed software and to resolve any non–deterministic choices that are possible during software execution.

Acknowledgement

The work of one author, H.D. Chu, was funded by the National Science Council in Taiwan from whom he received a fellowship to work toward a doctoral degree.

References

Basili, V.R. and Selby, R.W. (1987) Comparing the effectiveness of software testing strategies, *IEEE Transactions on Software Engineering*, **SE–13**(12), 1278–1296.

Beizer, B. (1995) *Black–Box Testing* (John Wiley & Sons, New York).

Cho, C.K. (1988) *Quality Programming – Development and Testing Software wih Statistical Qualty Contro*l (John Wiley & Sons, New York).

Curritt, P.A., Dyer, M. and Mills, H.D. (1986) Certifying the Reliability of Software, *IEEE Transactions on Software Engineering*, **SE–12**(1), 3–11.

Deck, M. (1996) Cleanroom practice: a theme and variations, in *Proceedings of the 9th International Software Quality Week* (Software Research Institute, San Francisco).

DeMillo, R.A., Lipton, R.J. and Perlis, A.J. (1979). Social Processes and the Proofs of Theorems and Programs. *Communications of the ACM*, **22**(5), 271–280.

Deutsch, M.S. and Willis, R.R. (1988) *Software Quality Engineering–A Total Technical and Management Approach* (Prentice Hall International , London).

Dyer, M. (1992) *The Cleanroom Approach to Quality Software Development* (John Wiley & Sons, New York).

Fagan, M.E. (1976). Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems journal*, **15**(3), 182–211.

Fagan, M. E. (1986). Advances in Inspections, *IEEE Transactions on Software Engineering*, **SE–12**(7), 744–751.

Hörcher, H. M. and Peleska, J. (1995). Using Formal Specifications to Support Software Testing. *Software Quality Journa*l, **4**, 309–327.

Humphrey, W.S. (1988). Characterizing the Software Process: a maturity framework. *IEEE Software*, **5**(2), 73–79.

Kit, E. (1995). *Software Testing in the Real World: improving the process* (Addison–Wesley Publishing Company, Workingham).

Ince, D. (1987) The automatic generation of test data, *Computer Journa*l, **30**(1), 63–69.

Liu, I C., Yang, R.D., Chu, H.D., Liu, F.H. and Chang, C.S. (1992) *Software Statistical Quality Assurance* (Technical Report, Institute for Information and Industry, Taiwan).

Marre, B., Thévenod–Fosse, P., Waeselynck, H., Gall, P.L. and Crouzet, Y. (1995). An Experimental Evaluation of Formal Testing and Statistical Testing. In *Predictably Dependable Computing Systems* (Springer, London), 273–281.

Mills, H.D., Dyer, M. and Linger, R. (1987). Cleanroom Software Engineering. *IEEE Software*, **4**(5), 19–25.

Musa, J.D. and Everett, W.W. (1990) Software–reliability engineering: technology for the 1990s, *IEEE Software*, **7**(6), 36–43.

Myers, G.J. (1978) *The Art of Software Testing* (John Wiley & Sons, New York).

Norman, S. (1993) *Software Testing Tools* (Ovum Ltd, London).

Ould, M. A. and Unwin, C. (1986). *Testing in Software Development* (Cambridge University Press, Cambridge).

Poston, R.M. (1996) *Automating Specification–Based Software Testing* (IEEE Computer Society Press, Los Alamitos).

Roper, M. (1994). *Software Testing*. (McGraw–Hill Book Company, London).

Sommerville, I. (1996) *Software Engineering*, 5th edn. (Addison–Wesley, Wkoingham).

Thévenod–Fosse, P. and Waeselynck, H. (1991) An investigation of statistical software testing, *Journal of Software Testing, Verification and Reliability*, **1**(2), 5–25.

Thévenod–Fosse, P., Waeselynck, H. and Crouzet, Y. (1995). Software Statistical Testing. In *Predictably Dependable Computing Systems* (Springer, London), 253–272.

Vliet, H. V. (1994). *Software Engineering: Principles and Practice* (John Wiley & Sons, New York).

Whittaker, J.A. and Thomason, M.G. (1994) A Markov Chain Model for Statistical Software Testing, *IEEE Transcations on Software Engineering*, **SE–20**(10), 812–824.

Zhu, H. (1996). A Formal Analysis of the Subsume Relation Between Software Testing Adequacy Criteria. *IEEE Transactions on Software Engineering*, **SE–22**(4), 248–255.