Program Reliability Estimation Tool

By
S. M. Reza Nejat

Software Engineering Research Group
McMaster University

Abstract

Testing is a very demanding procedure in software production, that takes a lot of effort, time and resources during both development and maintenance. Moreover, statistical testing is a very costly procedure, especially if high reliability requirements are placed on the software as in safety-critical, or safety-related software cases. The main question is when to stop testing, or how many tests are needed?

Singh *et al.* [49], using the method of the negative binomial, developed a procedure for quantifying the reliability of a module. According to their approach, the number of tests can be computed based on hypothesis testing. We implemented this method for a reliability estimation of a program.

In this work, a prototype black-box automated testing tool, called Program Reliability Estimation Tool (PRET) was developed as a statistical test generator and reliability estimation tool based on an operational profile (a proposed testing model) and negative binomial sampling.

The tool has a command line user interface. The inputs to the PRET are: an integer (0 or 1) to choose the usage (0: only generate test cases, 1: does the testing process), the test specification context file name, the data file name, the program under test name, and the oracle name.

PRET computes the number of test cases, generates test cases, runs the generated test cases, evaluates the result of each test run by using an oracle, and estimates the reliability of the program based on test results.

Acknowledgements

I would like to express my sincere and deepest thanks to Professor David L. Parnas for his guidance, insight, enthusiasm, encouragement and constant support throughout the preparation of this work.

I am very grateful to Dr. Roman Viveros and Dr. Ridha Khedri for their support, time and their helpful comments on this thesis.

I would like to thank Dr. Dennis Peters for many helpful and encouraging discussions.

I would like to thank Mr. Ross Steingrimsson for testing the tool and his useful suggestions.

I would like to thank Mrs. Doris Burns for all her help.

I would like to acknowledge the financial support of the Natural Sciences and Engineering Research Council (NSERC) and the Telecommunication Research Institute of Ontario (TRIO).

Finally, I would like to express my deepest appreciation to my wife, Monireh Eskandari, my daughter, Sara, and my son, sohail, for their endless patience and unending support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Issue

There are two reasons to test software: 1) to confirm the correct implementation of the software design (Structural testing [3]), and 2) to confirm that the software satisfies its specified requirements (Functional testing, [19]). Structural testing techniques are used by software developers, and are based on a detailed knowledge of the design and implementation of the software. White-box testing is another label for software structural testing [13]. Functional testing [19, 20] of a program is the process of determining whether or not the programming implementation satisfies its requirements specification. This method treats the program as a black box, and is concerned with the program's external behaviour [4, 13]. Functional testing techniques are generally used by testers and/or customers during the program development life cycle. Functional testing is mostly performed for the acceptance of a software or a program.

With black-box (functional) testing, an implementation is tested only with respect to its functional specification (i.e., the correct mapping of input to expected output is checked). There must be an input-output oracle (a mechanism that determines if the output from a program for the specific input is correct) available. The oracle determines if the output values are correct, and not whether they have been computed by correct method.

In black-box testing, software's code characteristics are not considered for selecting tests. One of the known variations of sampling strategies are statistical testing [20]. It

is testing a software or a program with a statistically representative of its operational use. The conceptually simplest method of determining the reliability of a computer system is by statistical testing. During acceptance testing if only the external behavior of a program is to be considered, statistical testing is the only known mathematically based method [20]. Tests are randomly selected according to the operational input distribution for the program. The operational input distribution is the frequency with which different elements of the input domain are selected when the program is in actual use [20].

Statistical techniques have been discussed for software testing but generally focus on structural testing [14]; they have also been introduced as functional testing methods [14, 29, 38, 49, 54]. They have been shown to be effective and promising techniques in measuring requirements satisfaction [13, 52, 53] specially for the final testing of software. Statistical techniques are the focus of this work.

Statistical methods for software functional testing have been advocated in cleanroom engineering [13]. Acceptance testing of a program developed using the cleanroom design methodology is done exclusively with test cases produced statistically [31].

Statistical methods have been highly used in safety-critical and safety-related software [38, 48]. The Canadian Nuclear Standard [6] requires this form of testing.

1.2 Motivation

This work develops a tool that generates statistical test cases for a program according to an operational profile and estimates the operational reliability, the failure rate, and an upper confidence bound for program failure rate. The tool uses the inverse binomial method (see section 3.9) for operational reliability estimation and a test oracle generated by the test oracle generator (TOG) [45] in order to evaluate the program output according to its specifications.

The need for estimating reliability for software motivated our interest in statistical testing and reliability. In fact, safety-critical and life-critical software require very high reliability. Estimating software reliability also requires statistical testing. These requirements were the driving force and motivation of this work. Black-box statistical testing is central to our approach.

The McMaster Software Engineering Research Group is developing documentation methods, utilizing mathematical functions and relations represented as tabular expressions [39] to describe a computing system's behaviour [43] for software product. Based on this methods, a set of tools (Table Tool System (TTS) [50]) has been developed that not only can store a syntactical representation of tabular expressions, but that also can store the table semantics and interpret them. A set of application tools has been developed on top of TTS kernel with different kinds of responsibility, such as evaluating each of the expressions in the specification.

In this regard, a tool that can accomplish black-box testing of a program based on specification and estimate its reliability after the program is developed according to the specification is needed. With TTS, the software can be tested through its development utilizing other testing tools plus this new application tool.

In safety-critical systems such as a nuclear power plant, the reliabilities of system and subsystems are computed based on system components or units (hardware and software). Computing reliability of a unit requires statistical testing that needs generating test input statistically. This new application tool is part of TTS and generates statistical test cases, estimates program reliability and accomplishes acceptance/conformance testing in TTS.

1.3 Scope

The purpose of this work is to design, implement and demonstrate a prototype black-box automated testing tool for a program, Program Reliability Estimation Tool (PRET), based on an operational profile (a proposed testing model) and negative binomial sampling (Singh *et al.* approach [49]). PRET will automatically generate test cases and, by implementing a test harness (see section 2.7.17), run the test case. Based on the results of the program execution with these test cases and results from the test oracle by applying the TOG, the tool will estimate reliability or failure rate, probability of type II error (see section 3.8.2) in the statistical testing of hypothesis and an upper confidence bound based on the negative binomial method. In this work, only testing of a memoryless program is considered. For a test case that consists of several random variables (program arguments), the probability distributions of all random variables should be identified.

1.4 The Tool

The inputs PRET takes are:

- a program name, that is the name of the executable program to be tested.
- an oracle name, that is the name of an executable program which is an oracle for the program under test
- an operational profile description which, consists of the identifier (or name), type, range, and probability distribution along with its parameters of each random variable (see section 2.7.13)
- other necessary information required by the testing model, such as: target failure rate, number of tolerated failure, maximum probability of type II error and confidence level.

The tool will do the following:

1. Compute the number of required test cases based on information such as target failure rate and probability of type II error.
2. Produce a sequence of test cases that satisfies the operational profile description
3. To obtain reliability metrics, repeatedly run a test harness that will:
   - test the program by running the test case
   - run the test oracle to get an *accepted* or a *rejected* result
4. Estimate the program reliability, failure rate and upper confidence bound
5. Determine that the test is passed or failed
6. Put the result in an output file.

To generate random numbers for simulating the values of random variables according to a given probability distribution, RANDLIB.C [5], the C routines library for Random Number Generation, is used. This library of C routines, generating non-uniform deviates was developed by the Texas University and tested for different cases.

1.5 Overview of Previous Work

A good survey on random testing is available in [55]. In the literature the operational profile has been described in two ways: using unconditional probability distribution [8, 12, 30, 51] and conditional probability distribution [7, 52, 53, 54]. In most cases the operational profile is described as an unconditional probability assigned to an individual input element. For a memoryless program, this definition is adequate to describe actual program usages. Whittaker [52, 53] and Woit [54] have proposed conditional probability for defining their operational profile for modules that have memory.

Whittaker [52, 53] defined module operational profile more specifically, in which the probability of issuing a module input depends on the input that was selected *immediately* prior to it.

Woit [54] defined her operational profile more generally, in which the probability of issuing a module input can depend on any of the inputs previously encountered.

For reliability estimation there are two general methods: reliability growth models and reliability models. In reliability growth models, the reliability of the next version of a software is partially predicted based upon data collected from the previous versions [33, 38, 56]. A short review of reliability growth models can be found in [54, 56], but they are not relevant to this work. In reliability models, the reliability of a software is estimated on the basis of trials of one version of the software only [30, 51, 56]. Research has been done on reliability estimation and several models have been developed. These models fall into two general categories of time-domain and data-domain.

In time-domain models, time is considered as the main parameter, and the reliability is measured as the mean time to failure and as the probability of failure in a given time interval [9, 15, 32, 33, 48].

Data-domain models measure reliability as the probability of obtaining a satisfactory output response from the software. These models have operational inputs rather than time as the main parameter and as the main indicator of the software failures. Data-domain models compute reliability as the ratio of number of successful runs to total number of runs [8, 29, 49, 51, 54, 56]. A survey, review and comparison of reliability models are available in [55, 56].

The work done at the Software Research Group of McMaster on operational profile and reliability estimation is summarised below.

Parnas *et al.* [38] applied the idea of statistical testing and showed the power of the method in measuring the reliability of software. They showed the potential of hypothesis testing of software to estimate module reliability.

Woit [54] addressed the issues of operational profile specification and test case generation and developed a technique to specify operational profile.

Li [29] used Woit's work on operational profile specification and reliability estimation model [54] to develop a reliability estimation tool for a module.

Peters [45] developed a program test oracle generator (TOG) based on formal program documentation. TOG uses a program specification to automatically generate a test oracle.

Abraham [1] used the general table model, developed the evaluation code generator (ECG) and improved Peters's approach to evaluate any tabular expression using that model.

Singh *et al.* [49], using the method of the negative binomial, improved Woit's binomial approach and developed a procedure for quantifying the reliability of a module.

1.6 Outline

This work consists of the following chapters:

1. Introduction
2. Terminology and Notation
3. Statistical technique and Reliability estimation methods
4. Program test oracle
5. Program reliability estimation tool design
6. PRET Module decomposition
7. Results and conclusions

Chapter 2 defines the terminology that is used in this work. Chapter 3 describes statistical sampling strategy for functional testing, reviews random number generation for simulating random variables and describes reliability estimation methods. Chapter 4 discusses the needs for oracle, and presents briefly Peter's TOG and Abraham's ECG for oracle generation. The external design of reliability estimation tool is discussed in chapter 5, and chapter 6 discusses the internal design and module decomposition. A module interface guide for the tool is given, and description of each module's access programs is presented. Chapter 7 presents the results of these methods, discusses future work and draws some conclusions. Appendix A gives the user's guide of the tool, Appendix B gives the example problem Triangle and Appendix C gives the Oracle for the example problem Triangle.

Chapter 2

Terminology and Notation

2.1 Synopsis

The notation and terminology in this work is defined in this chapter.

2.2 Basic Definitions

A *set* is a collection of different elements.
A *bag* is a collection of elements in which an element may occur any (finite) number.
Or a *bag* is a set with possible duplicated elements.
The following terminology is adopted from [40].
A *relation* is a set of tuples.
A *binary relation* R is a set of ordered pairs, such that $(x, y) \in R$.

- The *domain* of a binary relation R, denoted $\mathrm{Dom}(R)$, is the set of values that appear as the first component of elements of R.

$$x \in Dom(R) \;\; \textit{iff} \;\; \exists\, y : (x, y) \in R$$

- The range of a binary relation R, denoted $\mathrm{Range}(R)$, is the set of all values that appear as the second component of elements of R.

$$y \in Range(R) \;\; \textit{iff} \;\; \exists\, x : (x, y) \in R$$

A *function* F is a binary relation with an additional property: for each element x, $x \in Dom(\mathrm{F})$, there is a only one (x, y) in the function.
A *predicate* is a function whose range contains no members other than ***true*** and ***false***.
For a set, $\mathbf{R}$, the *characteristic predicate*, r, is a predicate such that $r(a) \Leftrightarrow a \in \mathbf{R}$.
The following convention are used in this work:
For two values a and b, a predicate p is defined as:

$$p(a, b) =^{df} r(a) \wedge r(b)$$

i.e.,

$$p(a, b) \Leftrightarrow a \in \mathbf{R} \wedge b \in \mathbf{R}$$

2.3 Predicate Logic

The predicate logic used in this work is based on that described in [40]. It differs from traditional logic in that it allows the use of partial functions but ensures that all predicates are total. The following terminology is adopted from [40].

2.3.1 Function Application

A *function application* is a function name together with its arguments. These arguments are terms.

Term

A *term* is either a constant, a variable, or a function application.

2.3.2 Primitive Relation

A small set of relations, such as <,>,=, <=, >=, are defined as being *primitive relations*.

2.3.3 Predicate Expression

All primitive relations are *predicate expressions*. If P and Q are predicate expressions, then $\forall x$: P, P $\wedge$ Q, and $\neg$P are predicate expressions, where x is the *index variable*.

2.3.4 Tabular Expressions

Tables are multi-dimensional expressions which are often easier to read and understand than equivalent, more traditional, scalar expression. Parnas in [39] describes a method to represent mathematical functions and relations using tables called *tabular expressions*. For a more detailed discussion refer to [22, 39, 42].

2.4 Relational Program Specifications

A digital computer can be viewed as a finite state machine(FSM) [41], that consists of a finite set of memory locations and input and output registers, each of which itself is a finite state machine. The state of a computer is a function of the states of all of its components. The following terminology used for programs was adopted from [41, 42].

A *Program*, P, is the mechanism introducing a pattern of state-change sequences in the machine.

An *execution* is a (possibly infinite) sequence of states of the machine.

A *starting state* is the first state in an execution.

A *terminating execution* is a finite execution.

A *stopping state* is the last state in a terminating execution.

The possible state-change sequence determined by the program, P, is called the *execution* of P.

An *execution summary* is a pair comprising the initial and final states of a terminating execution.

2.4.1 Limited Domain Relation

Parnas *et al.* [37, 41, 42] present the use of Limited Domain Relation ($LD-relation$) to specify programs. An $LD-relation$, L, is a pair (R_L, C_L) where R_L is a binary

relation and C_L , called *competence set*, is a subset of the domain of R_L.

An $LD - relation$, L, can be employed to specify a program by allowing R_L be the set of acceptable execution summaries and C_L be the set of starting states for which the program must terminate. Thus a program satisfies a specification, L, if and only if

- Every (starting state-stopping state) pair, for terminating program, is an element of R_L;
- for all starting states in C_L, the program will terminate.

2.5 Program Variables

As stated in [45] *program variables* can be viewed, from a behaviour point of view, as a finite state machine, that is a component of the computer.

A program variable *name* is a string of characters representing it in the program text (i.e. code). Each program variables has a *type* denoting its number of possible states. A *value* of a program variable describes its state. For example, a program variable with type *int* (in ANSI C), has at least 2^{16} possible states represented by the integer numbers between -32766 and 32767.

2.6 Module

A *module* is defined as an information hiding package of programs that implements objects of a particular type. The object, which is a finite state machine, communicates with the outside world through a set of access programs.

2.7 Testing Terminology

Software testing has an established terminology. The following four definitions, adopted from [4, 21] are employed in this work.

2.7.1 Failure

A *failure* is any observable misbehavior of any object, such as the falsification of a requirement or an unexpected processing by-product.

2.7.2 Fault

A *fault* is a design flaw that will result in failure exhibited by some object when an object is subjected to an appropriate input.

2.7.3 Passed

A test is *passed* if it is executed, the validation criteria are correctly applied, the actual outcome matches the predicted outcome.

2.7.4 Failed

A test is *failed* if it is executed and the actual and predicted outcome do not match.

2.7.5 Program Testing

In *program testing*, the program is treated as a memoryless program, in which the output produced from an input is not influenced by previous inputs. An input to a memoryless program can be considered to be an n-tuple of program argument values.

2.7.6 Module Testing

Module testing can be viewed as testing a program with memory. In a program with memory a sequence of inputs are involved and a sequence of output is produced. The output of the program not only depends on an input it receives, it also depends on the output produced by previous inputs.

2.7.7 Functional Testing

Functional testing of a program involves executing the program over selected input (test data), and comparing its output data with the expected correct output [20].

2.7.8 Random Testing

A *random testing* is the testing in which the test data is randomly selected in accordance with some predefined probability distribution on the input domain.

2.7.9 Operational Distribution

An *operational distribution* of a program is a probability distribution on the inputs of the program along with constraints such as the testing environment (i.e., conditions from testers, or constraints from operating systems or other applications).

2.7.10 Statistical Testing

Statistical testing of a program is a random testing in which the test data are selected according to the operational distribution for the program.

2.7.11 Failure Rate

The *failure rate* is the probability that the program with an input, selected statistically in accordance with a given operational distribution, will not produce correct output according to the program specifications.

2.7.12 Operational Reliability

Operational reliability is the probability that an input, selected at random according to a given operational distribution will not fail, i.e., the program produces correct(expected) output according to the program specification. Thus, operational reliability = 1 - (failure rate).

2.7.13 Random Variable

A *random variable* is a program variable whose value is selected randomly. A **random variable**, X, is defined as a tuple of a *name* or an *identifier* (id), a *type* (it), and a *value* (x).

$$X = (id,\ it,\ x)$$

2.7.14 Test Case

A *test case* T_X, is defined as a n-tuple of random variables, where n is the number of program's arguments.

$$T_X = (X_1, X_2, ..., X_n)$$

2.7.15 Test Sequence

A **test sequence** S_T, is defined as a bag of test cases.

$$S_T = \{\!|\ T_X\ |\!\}$$

2.7.16 Test Oracle

An *oracle* is defined in [20] as a function that for each input for a given program P, determines whether or not the output from P is "correct".

2.7.17 Test Harness

A *test harness* is a program that executes the program under test for a selected test case and executes the test oracle to verify the test results.

Chapter 3

Statistical Techniques and Reliability Estimation Methods

3.1 Synopsis

In this chapter, the program reliability estimation method is described. The method includes:

1. statistical testing technique and program test specification method used for specifying program usage and generating test cases based on test specification, and

2. the reliability estimation method used for computing number of test cases needed and estimating a program's reliability based on the test result from the execution of the test cases.

To generate a test case, we need to simulate random variables by using random numbers. Random number generation is described briefly. For reliability estimation, binomial and negative binomial sampling techniques are described. Negative binomial sampling is implemented in the PRET, to estimate program reliability.

3.2 Overview of Testing Techniques

There are many testing techniques that can be used to analyze software, and to demonstrate program properties. They can be classified into two broad categories:

1. **Deterministic (Systematic, Planned) Testing** (e.g. black- and white-box testing, stress testing, path testing, functional testing, regression testing, coverage metrics, ...) is testing in which the test data are selected according to some rules or guidelines based on certain assumption about the nature of faults and how they can be located, or in accordance with the adopted criteria.

2. **Probabilistic (Statistical) Testing** (e.g. random test cases, expected value, risk, probability, confidence limits, reliability estimate, reliability growth, ...) is the testing in which the test data are randomly selected in accordance with some predefined probability distribution on the input domain.

To get an impression of the difficulties laid in the field of program testing, an example, taken from Myers [34], will be presented here.

Mayers [34] assumes there is a program, "Triangle", that reads three integer values, that represent the lengths of the sides of a triangle. The program will determine whether the triangle is *scalene*, *isosceles*, *equilateral*, or none of these and output the result.

For a set of inputs to be good test cases for this program, Myers [34] suggests the following 14 different conditions to cover most possible cases to test for finding some (not all) possible errors and bugs. These conditions are:

1. Have a test case that represents a valid scalene triangle (note that test cases such as (1,2,3), (2,5,10) and (3,3,3) are not valid scalene triangles)

2. Have a test case that presents a valid equilateral triangle (3,3,3)

3. Have a test case that presents a valid isosceles triangle (a test case such as (2,2,4) would not be counted)

4. Have at least three test cases that present valid isosceles triangles such that you have tried the three permutations of two equal sides (e.g., (3,3,4); (3,4,3); (4,3,3))

5. Have a test case in which one side has a zero value
6. Have a test case in which one side has a negative value
7. Have a test case with three positive integers so that the sum of two is equal to the third (it is definitely a bug if the program says (1,2,3) is a scalene triangle)
8. Have at least three test cases such that you have tried the three permutations of category seven (e.g.,(1,2,3); (1,3,2); (3,1,2))
9. Have a test case with three positive integers so that the sum of two is less than the third
10. Have at least three test cases such that you have tried the three permutations of category nine (e.g.,(1,2,4); (1,4,2); (4,1,2))
11. Have a test case in which all sides are zero (i.e., (0,0,0))
12. Have at least one test case with non integer values
13. Have a test case with the wrong number of values
14. Last, but not least, specify the expected output for each case.

There is no doubt that this list is not able to detect all the errors or bugs which might be in such a program. As Parnas [44] mentioned, to ensure adequate coverage of all possible cases and requirements, one needs more than these 14 cases for this triangle problem. For example, corresponding to test cases 5 & 11, we can consider another test case: "Have a test case in which two sides have a zero value". So there can be a lot of test cases to be considered to cover and to validate every specified requirement under every possible operational condition and combination of input data.

To find all errors in the triangle problem, the exhaustive testing, which is the use of every possible input condition as a test case, should be accomplished. Hence, to create test cases only for (all) valid triangle, one would have to cover up to whatever is the maximum integer size. This will be a huge number of test cases. However, exhaustive testing is known to be impractical in most situations. Even to test the

triangle simple program, one would have to produce virtually an infinite number of test cases.

This example is to illustrate how difficult it is to develop a good test for a simple program. It is easy to imagine how much worse it can be to test a huge program.

To reduce the number of test cases, one way is to partition the possible inputs domain into a finite number of classes, for which equivalent results is expected from the execution of any value in the class. This method is called the equivalence partitioning method [34], and the classes are called equivalence classes [34]. Also, a boundary value analysis, which is the extension of partitioning technique and focuses on selecting values along the edges or boundaries of the equivalence classes, should be done. The above example can be used to illustrate the equivalence class and boundary value analysis.

The set "three equal-valued numbers having integer values greater than zero" should be an equivalence class in a correct triangle program. It might not be for an incorrect program.

For boundary value analysis, the following two conditions seem to fulfil the requirements:

- All integers have to be greater than zero.
- The sum of any two of these integers should be greater than the third one.

Besides the equivalence partitioning method [4, 13, 34], and the boundary value analysis method [13, 34], there are other techniques such as the category partition method, the cause-effect graphing method and the error-guessing method to create test cases for functional testing [4, 13, 14]. The category partition method [13] is based on test specification language. It is another variant of partitioning technique which employs an analysis of formal software specification and uses automatic tools to furnish full coverage of the specification with a minimum set of test cases. The cause effect graphing method [13] is a logic-based model, which tracks the relationship between software input to distinguish the input combinations needed for maximum requirement coverage. The error guessing method [13] depends on tester experience to segregate areas with more potential for causing software failures.

These are non-statistical functional testing methods which have been used and have been effective. But there are concerns with these methods about the lack of

strict way in prioritizing requirements and the subjectivity in the selection of input values [13, 14]. While statistical methods have not been widely used for functional testing [14], they may establish a more formal basis for inference about software quality and for building the confidence levels needed. Statistical methods depend upon the use of a probability distribution for selecting input values from the domain and for presenting a mathematical basis to acquire objectivity in the test samples.

3.3 Statistical Testing

The principal objective of statistical testing is to estimate the reliability of the software rather than to discover software faults. Statistical testing neither states that the tested object is correct, nor that it will work without any failure. The test result is a probabilistic one.

Statistical testing consists of the following steps:

- Determine the operational profile of the software or the program.
- Generate a set of test data corresponding to the operational profile.
- Apply these test cases to the program and record the output.
- Check the input-output pair with an oracle.
- After an adequate number of failures has been observed, the software reliability can be estimated.

This conceptually attractive approach to reliability estimation is not easy to apply in practice. The principle difficulties which arise are due to the uncertainties in determining the operational profile of the software, the high costs of generating a large enough data set for statistical testing and the problems of achieving statistically significant results when very high reliability requirements are placed on the software.

3.4 Model

Our approach to statistical testing is based on an operational input distribution from which test cases are selected. The operational input distribution relies on the proba-

bility distribution for program input based on the knowledge about the intended use of the program, its specification, and other considerations or constraints such as the testing environment. So identifying probability distributions and possible constraints in the form of relation among input arguments (input variables) will be the first task in this statistical method. The identified probability distribution plus the constraint define the structure of the operational input distribution model.

The operational input distribution comprises of:

1. A vector of input random variables $\boldsymbol{X} = (X_1, X_2, ..., X_n)$ and its probability density function $f(\boldsymbol{x})$. In the case of discrete random variables,

$$f_X(\boldsymbol{x}) = Pr\{X_1 = x_1, ..., X_n = x_n\}, \qquad \boldsymbol{x} = (x_1, ..., x_n) \in R_X \tag{3.1}$$

   where

$$R_X = \{(x_1, ..., x_n) : \quad \forall i \in [1, ..., n] \quad a_i \leq x_i \leq b_i\}$$

   is the range of the vector $\boldsymbol{x}$.

2. A constraint A ($A \equiv r(\boldsymbol{X})$, where $r(\boldsymbol{X})$ is a predicate).

So the testing model is: A probability distribution over $\boldsymbol{X}$ such that constraint A is satisfied.

If we represent constraint A as a conditioning event, we can find the conditional probability of $\boldsymbol{X}$, given A, as

$$f(\boldsymbol{x}|A) = \frac{f_X(\boldsymbol{x})}{Pr[A]} \qquad \boldsymbol{x} \in R_X \wedge R_A \quad where \quad R_A = \{x|A\} \tag{3.2}$$

If we denote $R_C = R_X \wedge R_A$, we will have:

$$f(\boldsymbol{x}|A) = \frac{f_X(\boldsymbol{x})}{Pr[A]} \quad \boldsymbol{x} \in R_C \tag{3.3}$$

Note that

$$\sum_{\boldsymbol{x} \in R_C} f(\boldsymbol{x}|A) = 1 \tag{3.4}$$

For every test case $I = (\boldsymbol{x}|A) = (x_1, x_2, ...)$, the probability is indicated by $f(I)$.

For example, consider two random variables X and Y with joint domain R. Their joint probability density function is denoted by $f_{X,Y}(x,y)$, for $(x,y) \in R$ where

$$0 \leq f_{X,Y}(x,y) \leq 1 \qquad \forall (x,y) \in R \tag{3.5}$$

$$\sum_x \sum_y \quad \sum_{(x,y)\in R} f_{X,Y}(x,y) = 1. \tag{3.6}$$

If we have a constraint, as conditioning event, the probability space will change from domain R to domain R_c. The joint probability density function of two random variables X and Y for all $(x,y) \in R_c$, denoted by $f_c(x,y)$, and their marginal probability density functions, denoted by $f_c(x)$ and $f_c(y)$ respectively, are given by:

$$f_c(x,y) = \frac{f(x,y)}{\sum_x \sum_{y|(x,y)\in R_c} f(x,y)} \qquad \forall (x,y) \in R_c \tag{3.7}$$

$$0 \leq f_c(x,y) \leq 1, \qquad \forall (x,y) \in R_c \tag{3.8}$$

$$\sum_x \sum_y \quad \sum_{(x,y)\in R_c} f_c(x,y) = 1. \tag{3.9}$$

$$f_c(x) = \sum_{y \in R_{c_y}} f_c(x,y) = \frac{\sum_{y\in R_{cy}} f(x,y)}{\sum_x \sum_{y(x,y)\in R_c} f(x,y)} \tag{3.10}$$

where $R_{c_y} = \{y : (x,y) \in R_c \text{ for some } x\}$

$$f_c(y) = \sum_{x \in R_{c_x}} f_c(x,y) = \frac{\sum_{x\in R_{cx}} f(x,y)}{\sum_x \sum_{y(x,y)\in R_c} f(x,y)} \tag{3.11}$$

where $R_{c_x} = \{x : (x,y) \in R_c \text{ for some } x\}$

Cases having more than two discrete variables will follow similar lines. For continuous random variables the summation will be replaced by integration.

3.4.1 Statistical Sampling

Based on the operational input distribution, N test cases, $I_1, I_2, ..., I_N$ should be generated randomly and with replacement. A test case I_i, $1 \leq i \leq N$, can be

generated in two ways: 1) determine $f_c(I_i) = f_c(x_{1i}, x_{2i}, ..., x_{ni})$, based on $f(I_i) = f(x_{1i}, x_{2i}, ..., x_{ni}), I_i \in R$ and the constraint, and generate the test case according to $f_c()$. 2) generate test case based on $f(I_i) = f(x_{1i}, x_{2i}, ..., x_{ni}), I_i \in R$, and then check it with the constraint. This generated case can be accepted as a test case if it satisfies the constraint. In later case, the danger is that we may have to generate a large number of test cases before one satisfies the constraint. To be in safer side, we implemented the former case. Therefore, to generate a test case, the following steps should be employed:

1. Identify the probability distribution, $f(\boldsymbol{x})$, over input sequence $\boldsymbol{X} = (X_1, X_2, ..., X_n)$.

2. Identify the constraint.

3. Determine the probability distribution $f_c(\boldsymbol{x})$ over input sequence $\boldsymbol{X} = (X_1, X_2, ..., X_n)$.

4. Generate case $I = (x_1, x_2, ..., x_n), I \in R_c$ according to probability distribution $f_c(\boldsymbol{x})$ over $\boldsymbol{X}$.

3.4.2 Simulating Random Variables

In this section, simulating random variables and generating random numbers are discussed. To generate test cases, simulating random variables is needed, that requires random number generators. A brief discussion about random number generation methods and an overview of random number generators are given here.

Random numbers are used to simulate random variables. A random variable with a given non-uniform distribution can often be generated from a uniformly distributed variate on $[0, 1]$. Truly random numbers can not be produced by computer algorithms [16]. These algorithms are in fact deterministic, although the sequence generated may appear random in that it does not display an obvious pattern. Such a sequence is called pseudo-random. A sequence of pseudo-random numbers is a finite series of numbers distributed in interval of $[0, 1]$, generated by some recurrence algorithm of type $x_n = f(x_{n-1})$.

Uniform random numbers, that are provided by almost any computer, are usually based on congruential algorithm. Congruential random number generators have the form

$$I_i = MOD_m(a.I_{i-1} + b) \tag{3.12}$$

Where a and b are suitable integer numbers, the starting integer I_0 is known as the seed and the integer m is approximately the largest integer representable on the machine. A congruential generator produces random integers between 0 and $m-1$. In order to produce random floating-point numbers uniformly distributed on $[0,1]$, the random integers must be divided by m.

A more difficult problem is to generate random numbers with non-uniform distribution, that is numbers with standard or non-standard distributions rather than uniform distribution. There are several methods for generating non-uniform random numbers such as: the inverse transform method, the composition method, the acceptance-rejection method, the polar method, etc. The inverse transform method is addressed here as an example to illustrate generation of non-uniform random numbers from uniform random numbers.

Inverse transform method

Assume a single random variable X has probability distribution $F(x)$. Since $F(x)$ is a nondecreasing function, the inverse function $F^{-1}(y)$ may be defined for any value of y between 0 and 1 as:

$$F^{-1}(y) = inf\{x : F(x) \geq y, \qquad 0 \leq y \leq 1\}. \tag{3.13}$$

If U is uniformly distributed over the interval $[0,1]$,

$$X = F^{-1}(U) \tag{3.14}$$

has distribution function $F(x)$. This is because

$$Pr(X \leq x) = Pr(F^{-1}(U) \leq x) = Pr(U \leq F(X)) = F(X) \tag{3.15}$$

For a random vector $\boldsymbol{X} = (X_1, ..., X_n)$ with given probability distribution function $F(\boldsymbol{x})$ consider a case that random variables $X_1, ..., X_n$ are independent. In order to

generate the random vector $\boldsymbol{X} = (X_1, ..., X_n)$ from probability distribution function $F(\boldsymbol{x})$, the inverse transform method can be applied to each variable separately:

$$X_i = F_i^{-1}(U_i), \qquad i = 1, ..., n \tag{3.16}$$

U_i is uniformly distributed random variate on [0,1].

For example, the exponential distribution is easily produced by inverse method. Consider an exponential distribution with the density function

$$g(t) = \frac{1}{r} e^{-\frac{t}{r}}$$

and with the cumulative distribution function

$$G(t) = \int_0^t g(\tau) d\tau = \int_0^t \frac{1}{r} e^{-\frac{\tau}{r}} d\tau = 1 - e^{-\frac{t}{r}}$$

Given

$$x_i = G(t_i) = 1 - e^{-\frac{t_i}{r}},$$

we can easily solve for t, Obtaining

$$t_i = G^{-1}(x_i) = -\ln(1 - x_i)$$

where x_i is uniform on $[0, 1]$, and t_i are exponentially distributed.

3.5 Overview of Random Number Generators

This section reviews random numbers generation, and discusses the random number generator selected. The design, testing and assessment of random number generators touches a broad range of mathematical topics, from number theory to probability theory, from numerical analysis to statistics. There is a large number of books and papers dealing with the generation of uniform and non-uniform random deviates.

In uniform random number generation, there are four publications that are practically essential and basic in this field, by Knuth [24], Niederreiter [35], L'Ecuyer [27], and Helleekalek [17]. The book of Knuth, known as "bible", is an all time-classic in the field of random numbers. The paper of L'Ecuyer surveys most of the basic concepts and results of random number generation. There are other publications too

in this field, such as [28], which is a very good review on uniform random number generators.

Having a good non-uniform random number depends on having a good uniform random number. As Hormann [18] stated: "The quality of non-uniform random numbers is not only influenced by the quality of the uniform generator that is used but also by the transformation method applied to the uniform random numbers". The book of Devroye [10] is the most complete book on generating non-uniform random variates. Many papers discuss the generation of non-uniform variates by different methods of transformation such as [11]. There are a lot of WWW sites, such as [57] that deal with uniform and non-uniform random number generation, and provides Fortran, C, and C++ procedures to create random numbers.

To generate test cases, we need to have a good non-uniform random generator. RANDLIB, a library of C routines for Random Number Generation, [5] is used for this purpose. This C-library is written and compiled by the department of Biomathematics of the University of Texas.

RANDLIB is a large library for random variate generation from many univariate and multivariate distributions. The bottom level routines provide 32 virtual random number generators. Each generator can provide 1,048,576 blocks of numbers, and the length of each block is 1,073,741,824. Any generator can be set to the beginning or end of the current block or to its starting value. Most users won't need the sophisticated capabilities of this package, and will desire a single generator. This single generator has period of 2.3×10^{18}.

The methods presented in the paper "Implementing a Random Number Package with Splitting Facilities" by L'Ecuyer and Cote [26], were used to develop base routines, and most of the code is a translation from the Pascal of the paper into Fortran and C. A detailed description of the RANDLIB.C library is given in [5].

3.6 Test Specification

A program operational input distribution (profile) description is proposed as a test specification. The program operational profile specification should completely define program usage, i.e. the set of input arguments, the space of each argument and the probability distribution (for this set) of choosing the value of all the input arguments.

The specification should be in such a format that can be used as an input for the test generator. The test generator automatically generates a sequence of test cases based on an operational input distribution specification.

Therefore, the test specification should define the test boundaries by describing the set of random variables (a random vector) with their range and the probability distribution that their values can be taken.

In more detail, let us assume that a random vector, denoted by $\boldsymbol{X}$, has the space $R_{\boldsymbol{X}}$, i.e. the set of values that $\boldsymbol{X}$ can take. Let $S_{\boldsymbol{X}}$ be the sequence containing the candidate values of $\boldsymbol{X}$ chosen with the input probability distribution $P(S_{\boldsymbol{X}})$. So a test case, denoted by T, is a set of values of $S_{\boldsymbol{X}}$ (with the probability distribution $P(S_{\boldsymbol{X}})$). The syntax of test specification is given in Appendix A

3.7 Program Reliability Estimation

As discussed in section 3.3, statistical testing requires a distribution from which test cases are selected. Let $P(I)$ denote the probability that a test case $I \in TS$ is selected according to the operational profile, where TS is the test set. Defining the indicator function $\omega(I)$ [8, 30, 51] by

$$\omega(I) = \begin{cases} 0 \text{ if I runs successfully,} \\ 1 \text{ if I fails to run.} \end{cases} \tag{3.17}$$

Let p be the *operational failure rate* of a program, denoting the probability that an execution of program running I will fail. Then using basic properties of probability one obtains [49, 51, 54]

$$p = E(\omega) = \sum_{I \in TS} \omega(I)P(I) = \sum_{I \in TS:\omega(I)=1} P(I). \tag{3.18}$$

The *operational reliability* q,

$$q = 1 - p \tag{3.19}$$

is defined as the probability that the program is successfully executed.

Although in most realistic applications TS is very large, we rarely have the luxury of knowing or calculating the exact value of p, hence of q, unless we do exhaustive testing. The best we can hope for is to calculate a good estimate of p or to test

statistical hypothesis about p, which requires the use of statistical methods. If a test case I from TS is generated at random from the operational profile distribution, then the outcome obtained by the test case is a Bernoulli trial with a probability of "failure" equal to p, and a probability of "success" equal to $q = 1-p$. In the following sections, the estimation of p is addressed.

3.8 Binomial Sampling

Let us assume that according to the operational profile distribution, N test cases, $I_1, I_2, ..., I_N$ are generated randomly and with replacement from TS, and are run to obtain the respective values $\omega(I_1), \omega(I_2), ..., \omega(I_N)$. The total number of failures, X_N, in N runs, namely

$$X_N = \sum_{i=1}^{N} \omega(I_i) \tag{3.20}$$

will have the *binomial distribution* with index N and parameter p. That is,

$$P(X_N = x) = \frac{N!}{x!(N-x)!} p^x (1-p)^{N-x}, \qquad x = 0, 1, 2, ..., N \tag{3.21}$$

with mean $E(X_N) = Np$ and variance $Var(X_N) = Np(1-p)$.

3.8.1 Point and Interval Estimation of Failure Rate

An estimator of p, under Binomial sampling is the proportion of failures $\hat{p}_N$, that is

$$\hat{p}_N = \frac{X_N}{N} \tag{3.22}$$

which is an unbiased estimator, $E(\hat{p}_N) = p$. $\hat{p}_N$ which is obtained from the test, is referred to as a point estimate of p and is not the true value of p. At the end, the main concern is having a low failure rate, which can be stated by calculating an upper confidence bound for p or by testing statistical hypotheses about p.

Following [49], an upper confidence bound for p, denoted by pu, at confidence level $1-\gamma$ is the largest value of p such that $P(X_N \leq x_{ob}) \geq \gamma$, where x_{ob} is the observed value of X_N and $0 \leq \gamma \leq 1$. pu is the unique root of the equation in p

$$\sum_{x=0}^{x_{ob}} \frac{N!}{x!(N-x)!} p^x (1-p)^{N-x} = \gamma, \tag{3.23}$$

which, in most cases, requires a numerical solution. If we have no failure, i.e. $x_{ob} = 0$, in N tests run, then the equation becomes $(1-p)^N = \gamma$, yielding

$$pu = 1 - \gamma^{1/N} \tag{3.24}$$

3.8.2 Test of Hypotheses on Failure Rate

Consider null and alternative hypotheses, denoted by H_0 and H_1 respectively, where

$$H_0 : \quad p \leq \theta \qquad \textit{and} \qquad H_1 : \quad p > \theta. \tag{3.25}$$

and θ is the largest value of the failure rate that the user is prepared to tolerate on the program.

The observed value of X_N, which is the number of failures in N runs of program test cases generated randomly according to the operational profile distribution, is the basis of the test. Following [49], the *critical region* of the test, $C = \{x_0 + 1, x_0 + 2, ..., N\}$ is the region of agreement with H_1, and the complement of C in the range of X_N , $C^* = \{0, 1, 2, ..., x_0\}$, is the region of agreement with H_0. x_0, the so called *critical point* of the test, should be determined.

Since there is a finite number of tests, N, and we have to accept H_0 or H_1 based on the observed count of failures, whether it is in C^* or in C region, there may be some misleading situations hence some risks too.

If the observed count of failures falls in C, while H_0 is indeed true, i.e.,$p \leq \theta$, suggesting H_0 be rejected even though H_0 is true. The probability of occurrence of this situation is called *false rejection risk* [49] or *producer's risk*, denoted by $\alpha_B = FRR_B(p; x_0, N)$, and given by

$$\begin{aligned} \alpha_B &= FRR_B(p; x_0, N) = P(X_N > x_0) = 1 - P(X_N \leq x_0) \\ &= 1 - \sum_{x=0}^{x_0} \frac{N!}{x!(N-x)!} p^x (1-p)^{N-x}, \qquad p \leq \theta. \end{aligned} \tag{3.26}$$

Similarly, when the observed count of failures falls in C^* while H_1 is true, i.e., $p > \theta$, H_1 will be rejected. The probability of occurrence of this one is called *false acceptance risk* or *user's risk*, denoted by $\beta_B = FAR_B(p; x_0, N)$, and given by

$$\begin{aligned}\beta_B &= FAR_B(p; x_0, N) = P(X_N \leq x_0) \\ &= \sum_{x=0}^{x_0} \frac{N!}{x!(N-x)!} p^x (1-p)^{N-x}, \qquad p > \theta. \end{aligned} \tag{3.27}$$

α and β are usually called *Type I* and *Type II* errors, respectively, in the literature.

3.9 Negative Binomial Sampling

Let us assume that, instead of running a fixed number of tests, N, testing is continued until a pre-specified number of failures, r say, is obtained. So, the total number of test cases, denoted by Y_r, is the random variable, run until the r failures are completed. Y_r has the *negative binomial distribution* with parameters r and p,

$$P(Y_r = y) = \left(\begin{array}{c} y-1 \\ r-1 \end{array} \right) p^r (1-p)^{y-r}, \qquad y = r, r+1, r+2, ... \tag{3.28}$$

with mean $E(Y_r) = r/p$ and variance $Var(Y_r) = r(1-p)/p^2$. More details can be found in [49].

3.9.1 Point and Interval Estimation of Failure Rate

An estimator of failure rate p, under negative binomial sampling, denoted by $\hat{p}_{NB}$, is

$$\hat{p}_{NB} = \frac{r-1}{Y_r - 1} \tag{3.29}$$

An upper confidence bound, pu, for the failure rate p at confidence level $1-\gamma$ is the largest value of p such that $P(Y_r \geq y_{ob}) \geq \gamma$, where y_{ob} is the observed value of Y_r and $0 \leq \gamma \leq 1$. As it was showed in [49], pu is the unique root of the equation in p

$$\sum_{y=r}^{y_{ob}-1} \left(\begin{array}{c} y-1 \\ r-1 \end{array} \right) p^r (1-p)^{y-r} = 1-\gamma, \tag{3.30}$$

Since $P(Y_r \geq y_{ob}) = P(X_{y_{ob}-1} \leq r-1)$, so pu can be determined by

$$\sum_{x=0}^{r-1} \left(\begin{array}{c} y_{ob}-1 \\ x \end{array} \right) p^x (1-p)^{y_{ob}-x-1} = \gamma, \tag{3.31}$$

which, in most cases, requires a numerical solution. For $r = 1$, the equation becomes $(1-p)^{y_{ob}-1} = \gamma$, yielding

$$pu = 1 - \gamma^{1/(y_{ob}-1)}. \tag{3.32}$$

3.9.2 Test of Hypotheses on Failure Rate

Consider the same null and alternative hypothesis as before in binomial mode. In negative binomial mode of sampling, the observed value of test cases, Y_r, with specified number of failures, r, is the basis of the test; that is the total number of test cases that result to r number of failures. The critical region of the test is $C = \{r, r+1, ..., y_0\}$, and its complement is $C^* = \{y_0 + 1, y_0 + 2, ...\}$, for appropriate choice of y_0 [49]. In negative binomial mode of sampling when testing statistical hypotheses about p, one needs to evaluate that in which region does Y_r fall, in C or in C^*. y_0, the so called *critical point* of the test, should be determined. For this C, the false rejection risk, and the false acceptance risk, respectively are

$$\begin{aligned} \alpha_{NB} &= FRR_{NB}(p; y_0, r) = P(y_r \leq y_0) \\ &= \sum_{y=r}^{y_0} \binom{y-1}{r-1} p^r (1-p)^{y-r}, \qquad p \leq \theta. \end{aligned} \tag{3.33}$$

$$\begin{aligned} \beta_{NB} &= FAR_{NB}(p; y_0, r) = P(Y_r \geq y_0 + 1) \\ &= 1 - \sum_{y=r}^{y_0} \binom{y-1}{r-1} p^r (1-p)^{y-r}, \qquad p > \theta. \end{aligned} \tag{3.34}$$

Since $P(Y_r \leq y_0) = 1 - P(Y_r > y_0) = 1 - P(X_{y_0} < r)$, and $P(Y_r \geq y_0 + 1) = P(Y_r > y_0) = P(X_{y_0} < r)$, thus

$$\alpha_{NB} = 1 - \sum_{x=0}^{r-1} \binom{y_0}{x} p^x (1-p)^{y_0-x}, \qquad p \leq \theta. \tag{3.35}$$

and

$$\beta_{NB} = \sum_{x=0}^{r-1} \binom{y_0}{x} p^x (1-p)^{y_0-x}, \qquad p > \theta. \tag{3.36}$$

For $r = 1$, they will be

$$\alpha_{NB} = 1 - (1-p)^{y_0}, \qquad r = 1. \tag{3.37}$$

and

$$\beta_{NB} = (1-p)^{y_0}, \qquad r = 1. \tag{3.38}$$

3.10 Summary

Statistical testing techniques were discussed in this section in two parts. First, the statistical test case generation have been discussed, and operational input distribution and test specification proposed. To generate test cases, we need to simulate random variables by using random numbers. Random numbers generation techniques and random numbers generators were briefly reviewed. In second part, the reliability estimation method was discussed and two sampling techniques, binomial and negative binomial, were reviewed.

In software reliability, the basis for statistical model is in the nature of the usage of the software. The statistical testing techniques for generating test cases is adopted for two reasons:

1. statistical techniques permit an accurate simulation of program's usage environment;

2. statistical techniques give a mathematical basis for producing statistical inferences from the testing.

Non-statistical testing methods are to be subjective in test case selection. These techniques, using selected test cases, can only provide anecdotes from testing.

For reliability estimation, the negative binomial method is implemented. With negative binomial technique, the stopping rule is clear: the statistically generated test cases will be executed until either the number of failures observed passes the allowable number of failures, in which case the program has failed the test, or all test cases have been executed and number of failures has not reached the allowable number of failures, in which case the program has passed the test.

After generating test cases, running these cases and evaluating the output according to program specification with an input-output oracle, generated by TOG, the reliability metrics can be computed.

Chapter 4

Test oracle

4.1 Introduction

Functional testing, particularly with statistical test sampling, relies on the availability and accessibility of a test oracle that can be used to evaluate and ascertain the correctness of the program output for a particular test input. In our work in statistical functional testing and reliability estimation, we need to have an input-output oracle to be able to verify automatically the testing results. The tools, TOG and ECG were developed by SERG that can be used to automatically manage the program output verification in functional testing.

In this section, program design documentation, the Test Oracle Generator (TOG) and the Evaluation Code Generator (ECG) will be briefly described. TOG & ECG produce an oracle from program design documentation. A detailed discussion of TOG can be found in [45] and of ECG can be found in [1].

4.2 Test oracle

An oracle [19] is any means, (i.e., a program, a program specification, a process, a table of examples, a body of data, or simply the programmer's knowledge of how a program should work) used to specify the expected outcome of a test. There are different kinds of oracle depending on different kinds of testing. The most common oracle in functional testing is input-output oracle, that determines whether the returned

output is correct for a given input or not.

In statistical testing, because of the large number of test cases, output verification can not be done manually, although it may be done for some systematic testing. A test oracle that is capable of verifying the program output in accordance with the program specification and that can be used automatically is required.

4.3 Test Oracle Generator

The TOG is a prototype automated test oracle generator tool that, given a relational program specification [37] using tabular expressions [39], will produce a program that will act as an oracle. This oracle program will take an input-output pair from the program under test as an input and will return *true* if it satisfies the specification relation, or *false* if it does not. The relational program documentation that is used by TOG, as an input, describes the intended behaviour of a program in terms of its effect on the data structure.

4.4 Program Documentation Method

The relational program documentation method used in TOG is briefly reviewed here and is based on that described in [36]. This documentation that is the input to TOG is design documentation for a single program. It is written using LD-relation (see section 2.4.1) to document program behavior and is reffered to as the specification relation [37]. The LD-relation is given by the characteristic predicate, domain and competence set (section 2.4.1). Each expression in specification should be total in order to have a *true* or *false* value. In program specification, often partial functions are used. In this regards the predicate logic described in [40], which allows partial function, is used. Notice that each predicate expression used in this method, itself is total.

Functions and relations are presented using multidimensional tabular expressions described in [39]. The tabular expressions have often been found easier to read and understand. These expressions are suited for describing relations in program specifications. A tabular expression is constructed from scalar expressions (non-tabular, in conventional form expression) and grids (indexed sets of cells containing terms or

predicate expressions). Figure 4.1 illustrates an example of two dimensional table, a normal predicate table [39], adopted from [46]. The table in Figure 4.1 is to be interpreted as follows: select a row and a column (their header cell expressions are true), evaluate the predicate expression in the main grid of that row and column. An equivalent convention form of the same expression is given below the table in Figure 4.1.

	$x > 0$	$x \leq 0$
$y \leq 10$	$x + z < y$	$z > 5$
$y > 10$	$z < 5$	$z > x + 2$

$$
\begin{array}{l}
((x > 0) \wedge (y \leq 10) \wedge (x + z < y)) \vee \\
((x > 0) \wedge (y > 10) \wedge (z < 5)) \vee \\
((x \leq 0) \wedge (y \leq 10) \wedge (z > 5)) \vee \\
((x \leq 0) \wedge (y > 10) \wedge (z > x + 2))
\end{array}
$$

Figure 4.1: Normal Predicate Table and Equivalent Scalar Expression.

4.4.1 Documentation Components

The program documentation consists of: constants, variables, program specification, auxiliary predicate definitions, auxiliary function definitions, inductive predicate definitions and user definitions. The detailed description of all the computation components can be found in [45]. The program specifications are briefly presented here.

Program Specifications

A (relational) program specification consists of three components: 1) The *program invocation* that gives the name and type of the program and lists all of its argument program variables; 2) The *external variable list* lists all other program variables; 3) The *specification relation* defines the $LD - relation$ that specifies the behaviour of the program. Figure 4.2, taken from [46], illustrates an example of how the program documentation might be presented. It describes a program 'find' that searches an integer array 'B' in order to find a value presented by 'x', and returns its index in 'j'.

It also employs a boolean variable 'present' to indicate whether a match is found or not.

Program Specification

void
find(int B[N], int x, int j, bool present)

external variables:

$D_{find} = \underline{true}$
$C_{find} = \underline{true}$

$R_{find}(,) =$

	$(\exists i,\ bRange(i) \wedge$ $`B[i] = `x)$	$(\forall i,\ bRange(i) \Rightarrow$ $\neg(`B[i] = `x))$
$j`\|$	$`B[j'] = `x$	$\underline{t}rue$
$present' =$	$TRUE$	$FALSE$

$\wedge NC(`B, `x, B', x')$

Auxiliary Predicate Definitions

$$NC(\textbf{\textit{int}}\ \text{`a[]}, \textbf{\textit{int}}\ \text{`b}, \textbf{\textit{int}}\ \text{a'[]}, \textbf{\textit{int}}\ \text{b'})$$

$$\doteq (\forall i, bRange(i) \Rightarrow `a[i] = a'[i]) \wedge (`b = b')$$

Inductively Defined Predicates

$$bRange(\textbf{\textit{int}}\ i) \doteq \begin{cases} \boldsymbol{I} = \{0\} \\ G(i) = i + 1 \\ Q(i) = i < (N - 1) \end{cases}$$

User Definitions $\#include \quad$ "$defs.h$"

$\#define \quad N \quad 10$ **/* size of array to search */**

Figure 4.2: Example Specification.

4.5 Test Oracle Generation

As mentioned in [46], "a program that evaluates the characteristic predicate of the specification relation can be used as an oracle." The prototype TOG tool generates an oracle in the form of "C" language procedure; and it is part of the table tool system, TTS [50]. Its initial version is explained in [45]. Then, it is extended to handle more types of tabular expressions. More details about extension of TOG can be found in [1].

4.5.1 Oracle Internal Design

The oracle generated in the TOG is a *compiled* version of the program specification translated into "C" code (executable form). Each of the oracle boolean valued access programs evaluates a predicate expression. This expression can be scalar expression (i.e., non-tabular expression) or tabular expression. Since each expression may comprise one or more subexpressions, the TOG generates the C code by traversing the syntax tree of the expressions in a depth-first order (i.e., innermost subexpression is translated first).

4.6 Evaluation Code Generation

The TOG can evaluate a limited class of tabular expressions. Abraham [1] developed a TTS [50] application tool, the Evaluation Code Generator (ECG), to evaluate expressions from specification, that consists of a collection of expressions including tabular expression. As Abraham said: *"The design of the evaluation code is an adaption and enhancement of the oracle code produced by the prototype Test Oracle Generator (TOG) tool"* [1]. The Evaluation Code Generator (ECG) tool uses a general semantic model [1, 22] to interpret a wide class of tables. It consists of some of the modified TOG modules and Generalized Table Semantics (GTS) module, which is responsible to store knowledge of how to interpret tabular expressions. The ECG generates code which will evaluate each of the expressions in the specification.

Actually, the ECG produces a C program routine for each expression defined in given program specification. The generated code will generate output values for

given input data only if the expression defines a function. If an expression defines the characteristic predicate of a relation, not a function, the generated code can be used to determine if the given input and output data satisfy the relation.

The evaluation code generator (ECG) reuses many of the evaluation modules from the TOG, but generalizes the evaluation routines to cover a wider class of tables. More details can be found in [1]. The original TOG required three expressions in the relational program documentation, (the program relation, the competence set and the domain) to be specified. It generated three routines, *inRelation*, *inComSet* and *inDomain* from these expressions. These routines were the only routines that could be invoked and evaluated by the oracle user. In the modified version, these three expressions are optional and the routines generated from "auxiliary" function and predicate expressions are externally available, so these expressions may be explicitly evaluated.

4.7 Using ECG

After this enhancement and modification, the Test Oracle Generator, which now is called the Evaluation Code Generator, takes a context file, explained in more detail in [50], from TTS and a plain C header file as its inputs. All data structures should be defined in the input C header file. The code will produce a C^{++} header file and a C^{++} file as its output.

As it was mentioned in section 4.5.1, the TOG, and in the same manner the ECG, evaluate two kind of expressions. Figure 4.3 illustrates the structure of any input expression applicable to the TOG, and the ECG.

As Figure 4.3 shows, the expression root should be the "define" symbol. The left branch is function definition. The right branch is function with n, $(n > 0)$, arguments. The function definition will have all variables in the expression as its arguments.

The syntax of each input expressions applicable to the TOG, and so to the ECG, can be shown as:

```
defined (function definition, function).
```

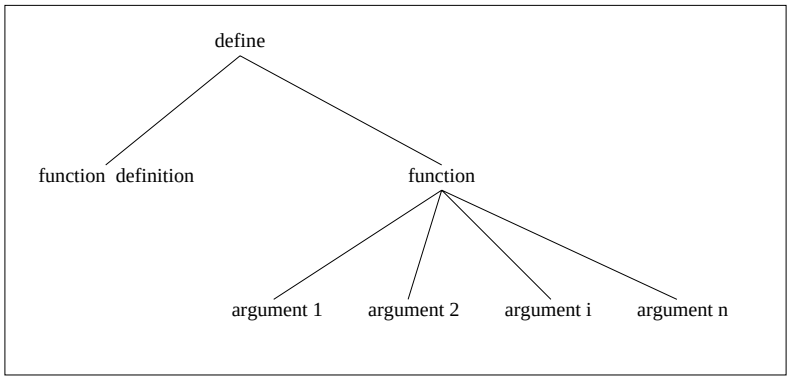

Figure 4.3: Expression Structure

4.8 Summary

To have an oracle that can automatically be used to verify the correctness of test results is required when statistical testing is accomplished. In fact, an oracle that can automatically evaluate the program's output regarding to specific input is essential in statistical testing and so in reliability estimation. TOG and its new form ECG can be used to generate a program oracle. They use program specifications in the form of a context file and produce procedures and code which evaluate each expression in the program's specification. We will use them for test evaluation purposes.

Chapter 5

Program Reliability Estimation Tool Design

5.1

This chapter describes the requirements and design of the prototype PRET.

5.2 Requirements

The requirements of the Program Reliability Estimation Tool are that it accepts a test specification as described in Chapter 3 in the format of a context file and a data file both in the forms described in detail in Appendix A, a program and an oracle. The PRET generates test cases and estimates reliability of a program as described in Chapter 3. The PRET produces a test case file which contains all the generated test cases, and an output file which contains all the input information, required reliability metrics computed and the final test result.

A context file contains a group of one or more expressions together with a symbol table. The context file is created using the context manager and table construction tool (TCT), both part of table tool system (TTS) [50]. A detailed description of these tools in TTS is given in [50]. The PRET is TTS compatible. Figure 5.1 shows the input/output of the program reliability estimation tool.

It should be noticed that ECG (or new TOG) should be run to get the oracle.

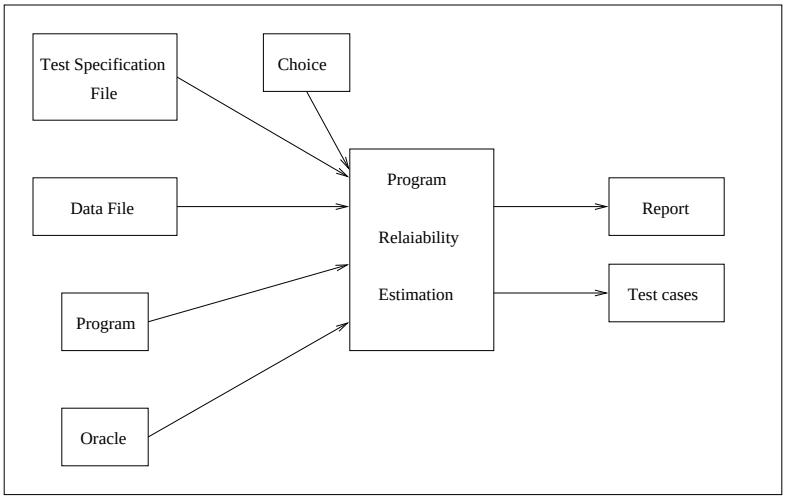

Figure 5.1: Input/Output of Program Reliability Estimation

As mentioned before, TOG gets a program specification in the context format as an input. So a context file for the program specification should be ready to use or should be prepared as described in section 5.1 of [1].

5.3 Assumption

It is assumed that the expressions used in the test specifications have been input and saved using the table holder module (see section 6.2.11). This assumption affects the Test Specification module, which is described in section 6.2.3. The same assumption is applied for the TOG and the ECG and affects some of their modules too.

Also, it is assumed that both specifications (program's and test's) used are proper.

5.4 Environmental Quantities for PRET

All the variables (controlled & monitored) controlled by the PRET are given in Table 5.1

Table 5.1: Monitored/Controlled Variables

Variable	Description	Mon./Con.	Value Set	units
P	Program under test	Monitored	string	
PTO	Program test oracle	Monitored	string	
ds	Operational input distribution specification	Monitored	string	
TS	Test Sequence (Sequence of N test cases)	Controlled	seqT	
q	Program reliability ($q = 1 - p$)	Monitored	real	
θ	Target failure rate	Monitored	real	
tf	Maximum number of tolerated failures	Monitored	integer	
ff	Number of failures found in the test	Controlled	integer	
ci	Critical point	Controlled	integer	
β	Maximum probability of type II error	Monitored	real	
α	Maximum probability of type I error	Controlled	real	
cl	confidence level(1 - γ)	Monitored	real	
pu	Upper confidence bound on failure rate	Controlled	real	

5.5 Brief Description of PRET

In this section the informal description of the PRET is given.

Through the user interface, the tool will get test specifications and related data. A testing option, and all monitored variables in accordance to chosen option are provided through test spect file and data file. The tool will compute the value of all the cotrolled variables and generate two files:

1. *test file* contains all the generated test cases.

2. *report file* contains the test option (or condition), the values of all monitored variables (i.e., random variables identifier, type, range, probability distribution and probability parameters, and all reliability related input data such as

$\theta, tf, \beta, \gamma$, all as an input data), and the values of all controlled variables, like N, ff, pu, and the test result.

According to the reliability estimation model, based on negative binomial technique, there are two kinds of test results:

1. The number of failures found, ff, in the test is not greater than the maximum number of the tolerant failures, or the number of tests accomplished is not less than critical point.

2. The number of failures found, ff, in the test is greater than the maximum number of the tolerant failures, or the number of tests accomplished to obtain that maximum number of tolerated failures is less than critical point. In this case, the program should be rejected.

5.6 User Interface

The user interface to the PERT is a command line interface. The command line syntax is as follows:

pret[*choice*][*testspecFile*][*dataFile*] [*program*][*oracle*]

Options:

choice	Choose the usage, 0 : only generate test cases; 1 : does the testing process.
testspecFile	Gets all test information from *testspecFile*.
datafile	Gets all necessary information from *datafile*.
program	Use the name to run the program under test.
oracle	Use the name to run the oracle.

5.7 Input Format

The input to the PRET consists of the following:

1. A "TTS context file", see [50], that contains the expressions that make up the test specification. The expressions are:
   - **Random Variables** For each random variable used in the test specification, an expression must be given containing just that variable. This gives the set of variables.
   - **Ranges** Range of each random variable must be named and be defined as separate predicate expression, as defined in section 2.2.
   - **Parameters** Parameters of probability distribution governing the values of each random variable must be named and defined as separate predicate expression, as defined in section 2.2.
2. A "data file", which contains all necessary data related to reliability estimation, option of computation and some related parameters such as β or θ in accordance with the chosen option.

A complete description of the input format is given in Appendix A.

5.8 Anticipated Changes

The following items regarding the PRET are anticipated as future changes within the useful life of the tool.

- Test specification file format.
- Format of information in data file.
- Test specification format.
- Reliability estimation method.
- Hypotheses testing method.
- User interface.

5.9 Test Generation Design

The function of the test generation in the PRET is to generate test cases for the program under test (PUT) according to test specification. It is managed by the test manager module. The test case will be run on the program and on the oracle.

The input of the test generator is the test specification (ds). The output of the test generation is a test case. The following algorithm is used for test generation.

```
Get the number of the program argument variables, n.
Get variable's related information:  id, itype, prob.
Repeat variables number times:
       Select an argument variable.
       Generate a random value based on probability distribution.
       Put the value in test case.
put the test case in test sequence.
```

5.10 Test Manager Design

The function of the test manager is to manage the whole process of testing procedure and reliability computation. Through the test manager, the number of required tests is computed, the test case is generated and is passed to test harness, the test result is gathered, and the test report is generated. The algorithm used by the test management is:

```
Get inputs from the main module.
Process the test specification.
Compute the number of required tests, N = ci.
Initialize the test sequence.
Do for N(test cases) times:
   Generate a test case.
   Pass it to test harness.
   Put it in the test sequence.
```

```
   Collect the result from test harness.
Collect the number of failures found.
Compute reliability related values.
Generate test report based on the data acquired.
```

5.11 Test Harness Design

Test harness can be used to automatically apply a selected test case to both a program and the program oracle and to verify the correctness of the resulting test output. The test harness supplies a place for the PRET to call program under test(PUT) and the desired oracle. The output of oracle evaluation of each test case is returned through the test harness.

To use the oracle produced by TOG, as mentioned in [45], a program (oracle harness program) which calls the oracle procedures and run them should be written. It should contain the oracle header file. This oracle harness program and oracle code should be compiled and linked to produce an executable program. This executable oracle harness program is referred to as "oracle" from now, and will be used by test harness. As stated in [45] each program under test needs a separate oracle.

The program under test, PUT, and the oracle are called through the test harness. The test harness gets and applies generated test cases to the program and the oracle and executes them for verifying the program output and returns the result. The only restriction is that both the program and the oracle should have command line and start with command line. All input variables for program under test and oracle should be presented as their command arguments.

The program test harness supplies a link between program under test, oracle, and the PRET. Figure 5.2 shows the program test harness algorithm.

5.12 Summary

The external design of the Program Reliability Estimation Tool (PRET) is discussed. The requirements of the tool and the environmental quantities are introduced. The

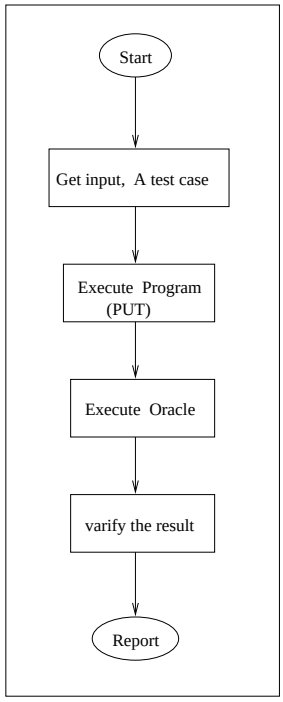

Figure 5.2: Test Harness Algorithm

algorithm of test generation, test manager and test harness design are presented. In next chapter, the external design and module decomposition of the PRET is discussed.

Chapter 6

Module Decomposition of PRET

6.1 Synopsis

In this chapter, the implementation of the PRET is described. The PRET is implemented by a set of modules, according to the "separation of concerns" or "information hiding" principle. Each module encapsulates a set of design decisions. Some of the modules are further decomposed into submodules that encapsulate more detailed design decisions. This encapsulation makes the design of the tool easier to understand, and makes it easier to change and maintain the tool.

6.2 PRET Module Interface Guide

Figure 6.1 shows the module uses relation which illustrates the system design and the first level module decomposition for the PRET. Module A is said to use Module B if some programs in Module A rely on the correct behavior of some programs in Module B to accomplish their task [1, 36, 45]. The module uses relation is obtained based on the program uses relation discussed in [36].

The user interface of the PRET is used to interact with the PRET. The test manager module receives commands and inputs from the user interface and manages the execution of the test specification, test generation, test storage, reliability, and test harness. The Table Holder is used by the PRET, but it is not a PRET module.

A general description of all the PRET modules and their access programs are

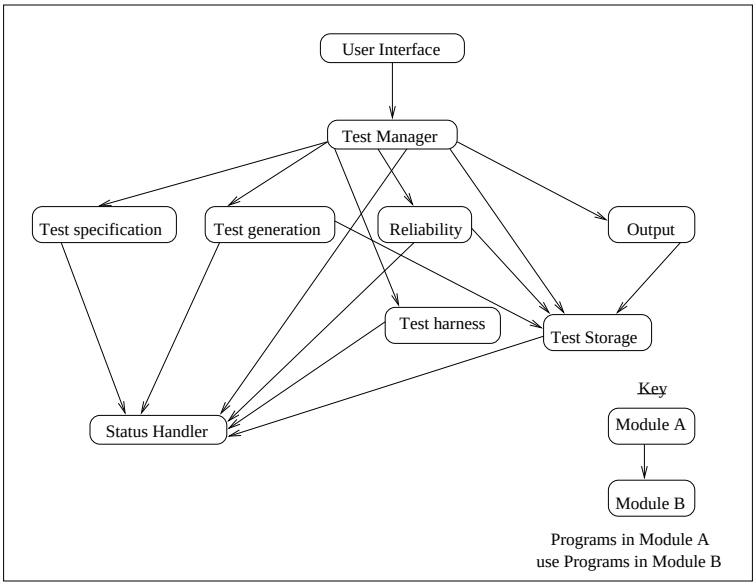

Figure 6.1: Module Uses Relation

covered in this section.

6.2.1 User Interface (main.c)

Secret: Encapsulates the format and interpretation of command line arguments and the sequence of module calls.

Service: It acts as the main controlling module and lets the PRET interact with its user. This module is responsible for getting the values of the monitored variables: the program under test, the test specification, and all other monitored variables (the data file), and the oracle. It uses the Test Manager module to manage the testing procedure.

6.2.2 Test Manager (testmanage.c)

This module is used by main controller module. It manages the testing procedure and reliability computation.

Secret: Encapsulates the way that the whole testing process is managed and that other modules cooperate during testing.

Service: It uses the specification module to load and process test data and test specification, storage module to create test case storage, generation module to generate test cases, the reliability module to compute all controlled variables related to the negative binomial methods, and the output module to generate the test report.

Table 6.1: Test Manager Module Access Programs

Program Name	Type	Arguments	Description
testmanage			Manage test process

6.2.3 Test Specification (testspec.c)

The Test Specification module is responsible for providing access to the program test specification information and reliability related data. Table 6.2 gives the test specification module access programs and their descriptions.

Secret: Encapsulates the format of the specification file.

Service: It extracts test specification from a file and stores it in a way that can be retrieved by the Test Generation Module.

Table 6.2: Test Specification Module Access Programs

Program Name	Type	Arguments	Description
testspecOpen		CHandle FILE*	Read the Test Specification from the Context File and reliability information from Data File
testspecNextVar	bool		Make the next random variable the current one
testspecGetVarId	Id		Returns the Id of current random variable
testspecGetVarName	char	Id	Returns the name of current random variable
testspecGetVarType	char		Returns the type of current random variable
testspecGetVarProb	int		Returns the probability of current random variable
testspecGetVarRange	bool	Id double	Returns the range of current random variable
testspecGetVarProPara	bool	Id int utype int utype	Return the parameter of the distribution of current random variable
readDataFile	bool	FILE*	Read the data File to get info of random variable
testdataGetReliData	bool	int double double double	Get the reliability related data

6.2.4 Symbol Information(infoP.c)

Information about the symbols (identifiers) in the test description expression are stored in a symbol table. The information are grouped into classes, which are required along with other default classes to interpret test expressions.

Secret: Encapsulates the symbol table used to store the information needed for random variables (such as: probability distribution, or distribution parameters) and the portion of the Information module interface that deals with the symbol data.

Service: Access the symbol table and the information stored in it.

Table 6.3: Symbol Information Module Access Programs

Program Name	Type	Arguments	Description
infoInit			Initialize the module
infoSetSymTable		SymTbl	Sets the symbol table
infoGetSymTable	SymTbl		Gets the symbol table
infoSetSymClass		SymTbl	Sets the Class in the symbol table
infoGetSymData	Infoptr	Id Name	Gets the information from the symbol table for a given id in a given class

6.2.5 Test Generation (testgen.c)

The Test Generation module is responsible for generating test cases based on the information from test specification.

Secret: Encapsulates the algorithm used for generating test cases.

Service: Generates test set for PUT according to test specification. It uses the test case module to store generated test case, and uses random number generator to generate nonuniform random values. It is used by test manager to generate test case.

Table 6.4: Test Generation Module Access Programs

Program Name	Type	Arguments	Description
testcaseGen	bool	randvarDS Testcase	Generates test cases

6.2.6 Test Storage

The Test Storage module is responsible for storing the generated test cases and test sequence. It is divided into the following submodules.

6.2.6.1 Test Case Storage (testcase.c)

Secret: Encapsulates the data structure used to store generated test case.

Service: Responsible for generating the test case storage. It is used by Test Manager module and Test generation module.

6.2.6.2 Test Sequence Storage (testsequence.c)

Secret: Encapsulates the data structure used to store generated test sequence.

Service: Responsible for generating the test sequence storage. It is used by Output Module.

6.2.7 Reliability (rliest.c)

This module is responsible for computing reliability metrics (see Chapter 3).

Secret: Encapsulates the algorithm to compute all related controlled variables acording to the negative binomial methods.

Table 6.5: Test Case Storage Module Access Programs

Program Name	Type	Arguments	Description
testcaseCreate	Testcase*		Create testcase object
testcaseDestroy		Testcase*	Destroy testcase object
testcaseAddVar		Testcase* varType*	Add variable to the test case
testcaseSetVarVal		Testcase Id utype	Sets the value for variable in the test case
testcaseGetvar	varType	Testcase* Id	Get the value of variable from the test case
testcaseDelVar		Testcase* Id	Delete variable from the test case
testcaseSetSize		Testcase* int	Set number of variables in the test case
testcaseGetSize	int		Get the size of test case

Table 6.6: Test Sequence Module Access Programs

Program Name	Type	Arguments	Description
testsequInit			Initialize the module
testsequAdd		Testcase*	Add testcase to sequence
testsequGet	Testsequptr		Return test sequence
testsequDel			Del test case from test sequence
testsequSetVal		Testcase int	Put test case at position "int"
testsequGetVal	Testcase	int	Get test case from position "int"
testsequGetSize	int		Get the size of test sequence
testsequGetPos	int	Testcase*	Gets position of test case

Service: Compute confidence level, upper confidence bound, probability of type II error, probability of type I error, and total number of test.

Table 6.7: Reliability Module Access Programs

Program Name	Type	Arguments	Description
Beta	double	int N int tf double θ	Compute the probability of type II error, β, according to the test of hypothesis, (Eq: 3.36)
Alpha	double	int N int tf double θ	Compute the probability of type I error, α, according to the test of hypothesis, (Eq: 3.35)
Gamma	double	int N int tf double θ	Compute the probability of upper confidence bound for failure rate, (Eq: 3.31)
critPoint	int	int tf double θ double β	Compute number of test cases based on allowable failures, failure rate and β, (Eq: 3.36)
uppConfBound	double	int N int tf double γ	Compute upper confidence bound for failure rate, (Eq: 3.31)
numFailAllow	int	int N double β double θ	Compute number of allowable failures, (Eq: 3.36)
Theta	double	int N int tf double β	Compute target failure rate, θ, (Eq: 3.36)

6.2.8 Output (output.c)

This module is responsible for reporting the testing results.

Secret: Data structure used to store evaluated result and method of reporting it.

Service: Responsible for output, generating the test report which should contain the test condition, the values of all monitored variables, the test result, and the value of all controlled variables.

Table 6.8: Output Module Access Programs

Program Name	Type	Arguments	Description
ReportCreate	Report		Create report object
ReportDestroy		Report	Destroy the report object
ReportOutput		Report	Print output

6.2.9 Status and Error Handler (errorP.c)

The Status and Error Handler module for PRET, P_error, provides a means for the access programs of the other PRET modules to indicate to their callers the status of the invocation and operation, and provides the general reason if a failure occurs. All PRET access programs set the status token, P_token on each invocation. Table 6.10 contains a list of all the status token used in the PRET, along with the interpretation of each of them.

Secret: Encapsulates the way of reporting the result of calling other modules in PRET.

Services: Monitoring and reporting the status of other modules invocation.

Table 6.9: Status and Error Handler Module Access Programs

Program Name	Type	Arguments	Description
initPErr			
setPErr		P_Token	Sets the status token to t
getPErr	P_Token		Returns the current status token
getStrPErr	*char	P_Token	Return the string associated with status token t

Table 6.10: PRET Status Token

Token	Interpretation
P_Success	No error.
P_noInfo	Information not set in symbol table
P_infoTblErr	Symbol table is not set
P_infoCreatClassErr	Symbol class is not set
P_UnknownPErr	Unknown type of error
P_infoAllocFailErr	Memory allocation failure
P_testcaseAllocErr	Memory allocation for test case failure
P_testcaseDestroyed	Test case destroyed
P_Notestcase	Test case not set
P_ExcessVar	Variable number does not match required
P_NotVar	Variable is not set
P_invalidVarId	Error in variable id
P_notRanVarErr	Not a random variable
P_specFileErr	Error in specification module
P_THErrErr	Error in Table Holder module
P_dataFileErr	Error in data file
P_dataptrAllocErr	Memory allocation for data pointer failure
P_normalParameterErr	Parameter error in normal distribution
P_disTagErr	Error in distribution tag
P_rangeErr	Error in variable range
P_noRangeErr	Error in rang expression
P_numTestErr	Error in number of test
P_noFailure	Zero failure number
P_betaErr	Error in beta calculation
P_rongFB	Error in failure rate or beta
P_numfailErr	error in number of failures

6.2.10 Test Harness (pharness.c)

Secret: Algorithm used for calling program under test (PUT) and oracle.

Service: executes program under test (PUT), and oracle and pass the result to the test manger.

Table 6.11: Test Harness Module Access Programs

Program Name	Type	Arguments	Description
do_testing_harness	bool	Testcase * char * char *	

6.2.11 Table Holder (libtblhold.a)

The Table Holder module, which is the central part of TTS, is responsible for storing expressions structure. The components of a test specifications that are mathematical expressions are stored using the table Holder module, which is not a PRET module. This module is described in detail in [25, 50].

6.2.12 Context Manager Module (libcm.a)

The Context Manager module allows grouping of expressions in a context. The specification file input to PRET is a context file, that is created using the context Manager as stated in [50].

Chapter 7

Results and Conclusions

7.1 Synopsis

This chapter focuses on the application and limitation of the method described in this work. The Program Reliability Estimation Tool (PRET) is developed as a statistical test generator and reliability estimation tool by combining the test specifications and reliability estimation method based on negative binomial sampling.

In order to evaluate the method used in this work, the program of the triangle example, discussed in Chapter 4, is used. A brief description of the program to be tested, a discussion of the testing procedure and results of the testing are included here.

7.2 Trial Application

The program to be tested is a triangle program, *Triangle*, which gets three numbers (real or integer) and gives the kind of triangle: equilateral, isosceles, scalene, or not a triangle. Specifications and the source code of the triangle program, *Triangle*(), are given in Appendix B.

7.3 Test Procedure

The methods described in Chapter 3 are intended for statistical testing and estimating reliability of individual programs, not modules. Test cases are generated statistically according to test specification, program is tested by applying these test cases, and the program output is verified against program specification by oracle. The PRET accomplishes automatically all of these. To do so, an oracle code should be obtained by using the TOG, as discussed in Chapter 4. To run the PRET, testers should perform the following tasks:

1. Prepare program test specification using TTS.

2. Develop an oracle program, compile it along with the oracle code obtained from TOG to get an executable oracle.

3. Run the PRET. The PRET generates test cases, runs the test harness for each case which gives back pass or fail answer, and determines the desired reliability metrics.

7.4 Testing Results

For testing the Triangle program, different suites of testing were considered and used as an input for PRET in the form of specification file and data file. These suites are presented in Table 7.1.

Table 7.1: Test Suite Description

Suite	tf	β	θ	γ
I	2	.1	.01	.0001
II	2	.002	.01	.0001
III	2	.0002	.001	.001
IV	2	.00001	.001	.001
V	2	.00001	.0005	.001
VI	2	.00001	.0004	.001
VII	2	.00001	.0002	.001

For each suite number of test cases and upper confidence bound on failure rate were computed, test cases were generated, number of failures were measured, test status were reported, and failure rate and reliability after testing were computed. All these are done automatically by the PRET. Table 7.2 illustrates the results of the testing.

Table 7.2: Summary of Triangle Test Results

No.	Suite	N	pu	ff	status
1	I	389	.0312	0	passed
2	II	1172	.0117	0	passed
3	III	10999	.00084	0	passed
4	IV	14232	.00065	0	passed
5	V	28468	.00033	0	passed
6	VI	35586	.00026	0	passed
7	VII	71178	.00013	0	passed

The testing was accomplished with different cases such as triangle with three integer sides, three real sides and three mixed sides with different probability distributions.

Since the triangle program has been carefully inspected and tested, it didn't show any failure. To verify that the testing procedure does detect errors when they occur, some errors were introduced in triangle program. This was done by introducing small changes into triangle program code so as to slightly alter its behavior in a way that it no longer satisfies its specification. Each changed version of triangle function was tested separately using the same suit which was used for unchanged one. Table 7.3 illustrates the result for suite III in two different cases. In case A, all sides of the triangle considered have integer uniform distribution. In case B, all sides considered have real value with normal distribution.

7.5 Discussion

The PRET was used to evaluate the Triangle program with different testing suites. The results are given in Table 7.2. To use the PRET, the following considerations

Table 7.3: Test Results for Triangle Program with Errors

Suite	Triangle Case	N	No. of test done	ff	test status
III	A	10999	7	3	failed
III	B	10999	3	3	failed

should be followed.

7.5.1 Specifications

Two different specifications should be produced: test specification for generating test cases and program specification for producing oracle. The generated test cases are only good as the specifications from which they were produced, and so also is the oracle. So having precise and good specifications is essential.

7.5.2 Test Data File

Data file contains all parameters for reliability estimation process. Test data file should be prepared based on data file format explained in Appendix A.

7.5.3 Test Harness Construction

Test Harness executes both program under test (PUT) and the program's oracle. Since program takes starting state description as arguments and oracle takes starting and stopping state descriptions as arguments, the value of starting and stopping state is copied to variables by the test harness before executing the program or program's oracle. So the test harness and also the program's oracle both depend on the program's data structure.

Since different programs with different data structures are tested, different oracles are needed. Each test harness will execute two separate programs, a program under test and the program's oracle. Each tester should develop an oracle for the program to be tested. As an illustration, oracle for the Triangle problem is given in Appendix C.

The test harness implementation is discussed in Section 5.11. As mentioned before, a tester can develop a test harness and use it. In order to use your own test harness, the test harness interface should remain unchanged. In order to get an executable PRET you should compile and link the rest of the PRET code and your test harness.

7.6 Future Work

The PRET has been used for an example problem, Triangle problem, which demonstrated that the method is viable in this case. Experience with using it for a wide variety of industrial program applications is needed to draw better conclusions about its viability.

The current version of PRET is for testing memory-less programs. Further work is needed to cover other kind of programs, like programs with memory. It also suffers from all limitations that the TOG has. The detailed descriptions of these limitations can be found in Section 6.2 of [1] and Section7.2 of [45].

The program testing method can be extended to apply to real-time programs. Therefore, PRET can be extended to have the capability of testing real-time programs. The monitor generated by Dennis Peters [47] can be used as an oracle.

As a prototype, the syntax of test specification does not allow the specification to have explicitly dependent variables. Adding this to the PRET in future, makes PRET more powerful.

Other statistical methods, besides negative binomial or hypotheses testing, can be integrated into PRET if the same concept of test case is used. There is no need to make any change to the TOG as oracle generator.

To generate test case, only the standard probability distributions are used, which are covered in RANDLIB library (see Section 3.5). One other work is expanding that library to cover more probability distributions from different practical cases.

7.7 Conclusions

The Program Reliability Estimation Tool (PRET) developed in this work combined the test specification introduced in Chapter 3 and the reliability estimation method based on negative binomial technique (see Section 3.9).

The development of PRET prototype has shown that it is feasible to automatically generate test cases from relational test documentation and automatically evaluate the test results with the oracle which is produced according to relational program documentation. The formal test specification offers a way to specify formally the program usage. A test case generator can be implemented in accordance with the specification methods. All this makes statistical testing very practical. A program's reliability can be estimated based on the test specification, estimation methods and a generated oracle based on program specification.

The success of the PRET implementation shows the power of program test specification in testing and the negative binomial technique in reliability estimation. In reliability estimation and statistical testing, having an oracle to evaluate automatically the testing results is essential. This is due to the large number of tests needed in testing that requires automated output checking.

The PRET, in its present form, can be applied to a wide range of memoryless program, if a suitable oracle for the program can be provided. Like all tools it has its limitations. Program under test should terminate and also have explicit input and output. Since it is a functional testing tool, and not structural, it can be used for small, medium and big programs considering the availability of required oracle, that can be used automatically. To generate only test cases, an oracle is not required. Therefore it is not a limitation.

The experience of applying these methods has shown the following benefits for the program reliability estimation process:

- Fewer test cases production, hence reduces testing time and cost,
- Faster test analysis, hence reduces cost, and
- Better reliability estimation, hence increases safety and value.

In addition, if an oracle can be generated for a module from the module documentation, this method of estimating reliability can be applied to it. Also, it should be noticed that the test specification should be changed to cover program with memory. But the reliability part of the PRET does not need to be changed.

Having reliability of a unit, like a function or access program, a fault tree analysis can be performed to quantify software system reliability, especially in safety-critical

systems and characterise the life-threatening conditions and their likelihood of occurrences.

Also in large program and software, the importance or usage of units in that program can be weighted by a function or a parameter. Therefore the reliability of software can be computed from the reliability of units weighted by usage weight functions. The optimization analysis can be done on software reliability also. If the weight function can be related to unit failure cost, the cost optimization can be performed. Also the number of tests of each unit can be related to weight function. So cost benefit analysis and optimization can be performed based on usage function, number of tests and cost of testing regarding budget constraint.

Appendix A

PRET User's Guide

A.1 Synopsis

The PRET is implemented as a component tool of the table tool system (TTS), described in detail in [50]. In this appendix, we describe how the PRET can be used to generate test cases and estimate reliability for a program from a test specification, represented as expressions in a TTS context file, and a data file represented in a format given below.

A.2 Test Spec Input Format

A TTS context file contains an ordered list of named expression and related symbol information. The expressions in the context file are evaluated by Test Specification module. The contents of the TTS context file representing the test specification of the triangle problem (see section 7.2) is given in Table A.2. The context file is created using the context manager and Table Construction Tool (TCT) of TTS. A detailed description of all the tools in TTS is given in [50].

A.2.1 Random Variables

An expression is required to identify each random variable in the test specification. The name of each such expression is of the form "**Ran** *name*", where *name* is the name of the random variable. The expression is simply the variable itself.

For each random variable the following symbol information is required:

Name The random variable name

Tag varTag (= 2)

CType The random variable type

Probability Probability is given by an integer number, representing the probability distribution that governs the values of random variable. Table A.1 gives the probability distribution and the integer number representing each of them.

Range Range is represented by an integer number, the "id" number of range expression for the random value.

Parameters Parameters is represented by an integer number, the "id" number of the parameter expression for the probability distribution of the random variable.

Table A.1: Probability Presentation

Number	Probability
1	Integer Uniform
2	Real Uniform
3	Normal
4	Beta
5	Chi-square
6	Exponential
7	F
8	Gammma
9	Binomial
10	Negative Binomial
11	Poisson

A.2.2 Range

Range of each random variable must be defined as an expression. The name of each expression is of the form "**Range** *name*", where *name* is the name of the random

variable. The expression is the predicate, as defined in section 2.2, with two arguments that gives the range of random variable. The expression is simply in the form "R(a,b)", where a and b represents the values, lower and upper, in range.

For each variable range, the following symbol information is required:

Name The symbol "R"
Tag varTag (= 3)
Arity
CType Type of range values

A.2.3 Parameter

Parameters of each probability distribution governing the values of random variables can be defined as an expression. The name of each expression is of the form "**Paramet** *name*", where *name* is the name of the random variable. The expression is the predicate, as defined in section 2.2, with two arguments that gives the parameters of probability distribution of each random variable. The expression is simply in the form "P(c,d)", where c and d represents the values of first and second parameters in probability distributions.

For each variable parameters, the following symbol information is required:

Name The symbol "P"
Tag varTag (= 3)
Arity
CType Type of parameters values

A.3 Data File Input Format

The data in data file has free format and in following way:
RLP
$< tf > < \beta > < \theta > < \gamma >$

Table A.2: Triangle Test Specification TTS Context File

Name	Expression
Ran x	x
Ran y	y
Ran z	z
Range x	R(0,100)
Range y	R(0.0, 100.0)
Range z	R(,)
Paramet x	P(,)
Paramet y	P(,)
Paramet z	P(,)

A.4 User's Interface

The PRET uses a simple command line user interface, as described in section 5.6. The input required by PRET is the file names for:

1. An integer (0, or 1), representing the choice
2. the test specification context file name,
3. the data file name,
4. the program name, and
5. the oracle name.

Appendix B

Triangle Example

B.1 Synopsis

This appendix gives the complete specifications and code for a small program, triangle, taken from myers book [34]. The specification is developed so that can be adopted by TTS and can be used by TOG to make an oracle.

B.2 Program Function

int Triangle(float a, float b, float c)

external variables:

$D_{Triangle} = \underline{true}$

$C_{Triangle} = \underline{true}$

$R_{Triangle} =$

$H_1 \wedge H_2$ G	negz	$\neg$ negz	
imp	0	0	$\wedge NC(a,b,c)$
$\neg$ impl $\wedge$ equ	0	1	
$\neg$(impl $\vee$ equ) $\wedge$ iso	0	2	
$\neg$(impl $\vee$ equ $\vee$ iso)	0	3	

- Definitions:

  equ $\equiv ((a = b) \wedge (a = c) \wedge (b = c))$

  iso $\equiv ((a = b) \vee (a = c) \vee (b = c))$

  impl $\equiv ((a + b \leq c) \vee (a + c \leq b) \vee (b + c \leq a))$

  negz $\equiv ((a \leq 0) \vee (b \leq 0) \vee (c \leq 0))$

- Return Values:

Return	Triangle Type
0	Not a triangle
1	Equilateral
2	Isoscele
3	Scalene

B.3 Triangle Program Code

```
 /*
* all needed functions:
* max- find bigger value between two values.
* equal- find two floating values are equal or not.
* tritype- find type of triangle.
*/

 int max(float a, float b)
{
     return(a > b) ? a : b;
}

int equal(float a, float b, float eps)
{
     return (fabs(a − b) <= eps * max(fabs(a), fabs(b))) ? 1 : 0;
}
```

```
 int tritype(float x, float y, float z, float ep)
{
    int retval = 0;

   if( (x > 0) &&  (y > 0) &&  (z > 0) )
     {
      if((x + y > z) &&  (x + z > y) &&  (y + z > x))
       {
        if((equal(x, y, ep)) &&  (equal(x, z, ep)) &&  (equal(y, z, ep)))
         {
           retval = 3;
         }
        else if((equal(x, y, ep)) ||  (equal(x, z, ep)) ||  (equal(y, z, ep)))
          {
           retval = 2;
          }
        else
          {
           retval = 1;
          }
      }
      else
        {
         retval = 0;
        }
      }
     else
      {
       retval = 0;
      }
return retval;
}
```

Appendix C

Oracle

C.1 Synopsis

This appendix gives the oracle for a small program, triangle, taken from myers book [34]. The oracle is developed by using the procedure obtained from TOG based on the program's, triangle, specifications explained in Appendix B.

C.2 Program Oracle

```
/*****************************************************************
* program : triorac.c
* to identify kind of triangle, using oracle from program specification
* This is a seperate program.
*****************************************************************/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "myoracle.h"

int main(int argc, char *argv[])
{
```

```
    int retval;
    float x = 0.0;
    float y = 0.0;
    float z = 0.0;

    x = atof(argv[1]);
    y = atof(argv[2]);
    z = atof(argv[3]);

/* invok Triangle function from myoracle.c to identify kind of triangle */

retval = Triangle(x, y, z);
return retval;
}
```

C.2.1 Oracle procedure

In this section. Oracle procedures and functions, filed as myoracle.c, obtained from the TOG for triangle problem is given.

```
    /*
————————————————————
— end user definitions
————————————————————-
*/
#include "indPred.h"
#include "Table.h"
#include "mtoracle.h"

/*————————-Begin body————————*/

bool
negz(double a /* 151 */, double b /* 152 */, double c /* 153 */)
```

```
{
return ((c <= 0) || ((a <= 0) || (b <= 0)));
}

bool
imp(double a /* 151 */, double b /* 152 */, double c /* 153 */)
{
return ((b + c < a) || ((a + b < c) || (a + c < b)));
}

bool
iso(double a /* 151 */, double b /* 152 */, double c /* 153 */)
{
return (((a == b) || (a == c)) || (b == c));
}

bool
equ(double a /* 151 */, double b /* 152 */, double c /* 153 */)
{
return (((a == b) && (a == c)) && (b == c));
}

bool
TriTable_guard_1_1(va_list ar)
{
double a = va_arg(ar, double );
double b = va_arg(ar, double );
double c = va_arg(ar, double );
return (imp( a, b, c) && negz( a, b, c));
}

int
TriTable_val_1_1(va_list  ar)
```

```
{
double a = va_arg(ar, double );
double b = va_arg(ar, double );
double c = va_arg(ar, double );
return 0;
}

bool
TriTable_guard_1_2(va_list  ar)
{
double a = va_arg(ar, double );
double b = va_arg(ar, double );
double c = va_arg(ar, double );
return (imp( a, b, c) && !(negz( a, b, c)));
}

int
TriTable_val_1_2(va_list  ar)
{
double a = va_arg(ar, double );
double b = va_arg(ar, double );
double c = va_arg(ar, double );   return 0;
}

bool
TriTable_guard_2_1(va_list  ar)
{
double a = va_arg(ar, double );
double b = va_arg(ar, double );
double c = va_arg(ar, double );
return ((!(imp( a, b, c)) && equ( a, b, c)) && negz( a, b, c));
}
```

```
int
TriTable_val_2_1(va_list  ar)
{
double a = va_arg(ar, double );
double b = va_arg(ar, double );
double c = va_arg(ar, double );
return 0;
}

bool
TriTable_guard_2_2(va_list  ar)
{
double a = va_arg(ar, double );
double b = va_arg(ar, double );
double c = va_arg(ar, double );
return ((!(imp( a, b, c)) && equ( a, b, c)) && !(negz( a, b, c)));
}

int
TriTable_val_2_2(va_list  ar)
{
double a = va_arg(ar, double );
double b = va_arg(ar, double );
double c = va_arg(ar, double );
return 1;
}

bool
TriTable_guard_3_1(va_list  ar)
{
double a = va_arg(ar, double );
double b = va_arg(ar, double );
double c = va_arg(ar, double );
```

```
return
((!((imp( a, b, c) || equ( a, b, c))) && iso( a, b, c)) && negz( a, b, c));
}

int
TriTable_val_3_1(va_list  ar)
{
double a = va_arg(ar, double );
double b = va_arg(ar, double );
double c = va_arg(ar, double );
return 0;
}

bool
TriTable_guard_3_2(va_list ar)
{
double a = va_arg(ar, double );
double b = v_arg(ar, double );
double c = va_arg(ar, double );
return ((!((imp( a, b, c) || equ( a, b, c))) && iso( a, b, c)) && !(negz( a, b, c)));
}

int
TriTable_val_3_2(va_list ar)
{
double a = va_arg(ar, double );
double b = va_arg(ar, double );
double c = va_arg(ar, double );
return 2;
}

bool
TriTable_guard_4_1(va_list ar)
```

```
{
double a = va_arg(ar, double );
double b = va_arg(ar, double );
double c = va_arg(ar, double );
return (!(((imp( a, b, c) || equ( a, b, c)) || iso( a, b, c))) && negz( a, b, c));
}

int
TriTable_val_4_1(va_list ar)
{
double a = va_arg(ar, double );
double b = va_arg(ar, double );
double c = va_arg(ar, double );
return 0;
}
bool
TriTable_guard_4_2(va_list ar)
{
double a = va_arg(ar, double );
double b = va_arg(ar, double );
double c = va_arg(ar, double );
return (!(((imp( a, b, c) || equ( a, b, c)) || iso( a, b, c))) && !(negz( a, b, c)));
}

int
TriTable_val_4_2(va_list ar)
{
double a = va_arg(ar, double );
double b = va_arg(ar, double );
double c = va_arg(ar, double );
return 3;
}
```

```
Table<int> TriTable;

int
Triangle(double a /* 151 */, double b /* 152 */, double c /* 153 */)
{
return TriTable( 1, a, b, c);
}

void
initOracle()
{
Index index(2);
index[0] = 1;
index[1] = 1;
TriTable.setGuardCell(index, TriTable_guard_1_1);
TriTable.setValCell(index, TriTable_val_1_1);
index[0] = 1;
index[1] = 2;
TriTable.setGuardCell(index, TriTable_guard_1_2);
TriTable.setValCell(index, TriTable_val_1_2);
index[0] = 2;
index[1] = 1;
TriTable.setGuardCell(index, TriTable_guard_2_1);
TriTable.setValCell(index, TriTable_val_2_1);
index[0] = 2;
index[1] = 2;
TriTable.setGuardCell(index, TriTable_guard_2_2);
TriTable.setValCell(index, TriTable_val_2_2);
index[0] = 3;
index[1] = 1;
TriTable.setGuardCell(index, TriTable_guard_3_1);
TriTable.setValCell(index, TriTable_val_3_1);
index[0] = 3;
```

```
index[1] = 2;
TriTable.setGuardCell(index, TriTable_guard_3_2);
TriTable.setValCell(index, TriTable_val_3_2);
index[0] = 4;
index[1] = 1;
TriTable.setGuardCell(index, TriTable_guard_4_1);
TriTable.setValCell(index, TriTable_val_4_1);
index[0] = 4;
index[1] = 2;
TriTable.setGuardCell(index, TriTable_guard_4_2);
TriTable.setValCell(index, TriTable_val_4_2);
return ;
}
```

Bibliography

[1] Ruth Abraham, "Evaluating Generalized Tabular Expression in Software Documentation", CRL Report No. 346, February 1997.

[2] W. Bartussek and D.L. Parnas, "Using Assertion About Traces to Write Abstract Specifications for Software Modules", Proceeding of 2nd Conference of European Cooperation in Informatics, Venice, 1978.

[3] B. Beizer, *Software Testing Techniques.* Van Nostrand Reinhold Co., New York, 1983.

[4] B. Beizer, *Black-Box Testing* . John Wiley & Sons Inc., 1995.

[5] Barry W. Brown, James Lovato, Kathy Russell and John Venier, "RANDLIB.C, Library of C Routines for Random Number Generation", Department of Biomathematics, The University of Texas, http://odin.mda.uth.tmc.edu/pub/source/.

[6] CANADA, Standard for Software Engineering of Safety-critical Software", AECL CANDU. 2251 Speakman Drive, Mississauga, Ontario, Canada, L5K 1B2. Revision 0, December 1990.

[7] Sanping Chen and Shirley Mills, "A Binary Markov Process Model for Random Testing", IEEE Trans. on Software Engineering, Vol 22, No. 3 , pp. 218-223, March 1996.

[8] R.C. Cheung, "A user-oriented software reliability model", IEEE Trans. Software Engineering, Vol SE-6, No. 2, pp. 118-125, March 1980.

[9] P. Allen Currit, Michael Dyer, and Harland D. Mills, "Certifying the Reliability of Software", IEEE Trans. Software Engineering, Vol. SE-12, No. 1, January 1986

[10] Luc Devroye, *Non-uniform random variate generation.* Springer-Verlag, New York, 1986.

[11] Luc Devroye, "Random Variate Generation in One Line of Code", 1996 Winter Simulation Conference Proceedings, J.M. Charnes, D.J. Morrice, D.T. Brunner and J.J. Swain eds, ACM, pp. 265-272, 1996.

[12] J.W. Duran, and S.C. Ntafos, "An Evaluation of Random Testing", IEEE Trans. Software Engineering, Vol SE-10, No. 4, pp. 438-444, July 1984.

[13] Micheal Dyer, *The Cleanroom Approach to Quality Software Development.* John Wiley & Sons Inc., 1992.

[14] Micheal Dyer, "Distribution-Based Statistical Sampling: An Approach to Software Functional Test", J. System Software, Vol 20, pp. 107-114 , 1993.

[15] J.H.Poore, Harlan D. Mills, David Mutchler, "Planning and Certifying Software System Reliability", IEEE Software, Vol. 10, No. 1, pp. 88-99, 1993.

[16] Micheal T Heath, *Scientific Computing: An Introductory Survey.* WCB/McGraw-Hill Companies, 1997.

[17] P. Hellekalek, "Good random number generators are (not so) easy to find", Proc. Second {Mathematics and Computers in Simulation} Symposium on Mathematical Modelling, Vien, Vol. 46, 1998, pp. 485-505.

[18] Wolfgang Hormann, "The quality of Non-uniform Random Numbers", Operations Research Proceedings 1993, pp. 329-335, Berlin, 1994, Springer Verlag.

[19] William E. Howden, "A Functional Approach to Program Testing and Analysis", IEEE Transaction on Software Engineering, Vol SE-2, No.10, October 1986.

[20] William E. Howden,*Functional Program Testing and Analysis.* McGraw-Hill, 1987.

[21] IEEE *Software Engineering Standards.* New York: IEEE, 1994. Collection of ANS/IEEE standards on software engineering.

[22] R. Janicki, "Towards a Formal Semantics of Parnas Tables", 17th International Conference on Software Engineering, IEEE Computer Society, Seattle, WA, April 1995, pp. 231-240.

[23] J.G. Kalbfleisch, *Probability and Statistical Inference.* Springer-Verlag, New York, 1985.

[24] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison_Wesley, Reading, MA, 3rd edition, 1997.

[25] C. Krasnor, and D. L. Parnas, "The Table Tool System: The Table Holder", CRL Report No. 300, Telecommunications Research Institute of Ontario (TRIO), May 1995.

[26] P. L'Ecuyer and S. Cote, "Implementing a Random Number Package with Splitting Facilities". ACM Transactions on Mathematical Software 17:1, pp 98-111.

[27] P. L'Ecuyer, "Uniform Random Number Generation", Annals of operations research, Vol 53, 1994, pp. 77-120.

[28] P. L'Ecuyer, "Uniform Random Number Generator: A Review", Proceedings of the 1997 Winter Simulation Conference, IEEE Press, December 1997, pp. 127–134.

[29] Chunming Li, "Documentation Based Software Reliability Estimation Tool", CRL Report No. 337, December 1996.

[30] H.D. Miller, L.J. Morell, L.E. Noonan, S.K. Park, D.M. Nicol , B.W. Murrill, and J.M. Voag, "Estimating the probability of failure when testing reveals no failures", IEEE Trans. Software Engineering, Vol 18, No. 1 , pp 33-42, January 1992.

[31] H. D. Mills, M. Dyer, R. C. Linger, "Cleanroom Software Engineering", IEEE Software, pp. 19-25, September 1987.

[32] John D. Musa, "A Theory of Software Reliability and its Application", IEEE Trans. Software Engineering, Vol. SE-1, No. 3, September 1975.

[33] J.D. Musa, A. Jannino, and K. Kamoto, *Software reliability: mesurement, prediction, application.* McGrawHill, New York, 1990.

[34] Glenford J. Myers, *The Art of Software Testing.* John Wiley & Sons, 1979.

[35] H. Niederreiter, *Random Number Generation and Quasi_ Monte Carlo Methods.* SIAM, Philadelphia, 1992.

[36] D.L. Parnas, "On a 'Buzzword': Hierarchical Structure", Proceedings of the IFIP Congress 1974, North Holland 1974, pp. 336-339.

[37] D.L. Parnas, "A Generalized Control Structure and Its Formal Definition", Communications of the ACM, Vol 26, No. 8, August 1983, pp. 572-581.

[38] D.L. Parnas, J Van Schouwen, S.P. Kwan, "Evaluation Standards for Safety Critical Software", Communications of the ACM, Vol 33, No. 6, pp. 636-648, June 1990.

[39] D.L. Parnas, "Tabular Representation of Relations", CRL Report No. 260, Telecommunications Research Institute of Ontario (TRIO), November 1992.

[40] D.L. Parnas, "Predicate Logic for Software Engineering", IEEE Trans. Software Engineering, Vol 19, No. 9, September 1993, pp. 856-862.

[41] D.L. Parnas, "Mathematical Description and Specification of Software", Proceedings of IFIP World Congress 1994, Vol I, August 1994, pp. 354-359.

[42] D.L. Parnas, J. Madey and M. Iglewski, "Precise Documentation of Well-Structured Programs", IEEE Trans. on Software Engineering, Vol 20, No. 12, December 1994, pp. 948-976.

[43] D.L. Parnas and J. Madey, "Functional Documentation for Computer Systems", Science of Computer Programming, Elesevier, Vol. 25, No. 1, October 1995, pp. 41-61

[44] D.L. Parnas, Personal conversation.

[45] Dennis Peters, "Generating a Test Oracle from Program Documentation", CRL Report No. 302, April 1995.

[46] Dennis K. Peters and David L. Parnas, "Using Test Oracles Generated from Program Documentation", IEEE Trans. on Software Engineering, Vol 24, No. 3, pp 161-173, March 1996.

[47] Dennis Peters, "Deriving Real-Time Monitors From System Requirements Documentation", SERG Report No. 383, January 2000.

[48] Norman F. Schneidewind, "Reliability Modeling for Safety- Critical Software", IEEE Trans. Reliability, Vol. 46, No. 1, pp. 88-98, 1997.

[49] Balwant Singh, Roman Viveros, David Parnas, "Estimating Software Reliability Using Inverse Sampling", CRL Report No. 351, August 1997.

[50] Software Engineering Research Group, "Table Tool System Developer's Guide", CRL339, Telecommunications Research Inst. of Ontario, January 1997.

[51] Thomas A. Thayer, Myron Lipow, and Eldred C. Nelson, *Software Reliability.* , North-Holland Publishing Company, New York, 1978.

[52] James A. Whittaker, and J.H. Poor, "Markov Analysis of Software Specifications", ACM Trans. on Software Engineering and Methodology" Vol 2, No. 1, pp. 93-106, January 1993.

[53] James A. Whittaker, and Michael Thamason, "A Markov Chain Model for Statistical Software Testing", IEEE Trans. on Software Engineering, Vol 20, No. 10, PP 812-824, October 1994.

[54] Denise M Woit, "Operation Profile Specification, Test case Generation, and Reliability Estimation for Modules", CRL Report No. 281, February 1994.

[55] Denise M Woit, "Realistic Expectations of Random Testing", CRL Report No. 246, May 1992.

[56] Denise M Woit, "Estimating Software Reliability With Hypothesis Testing", CRL Report No. 263, April 1993.

[57] "http://lib.stat.cmu.edu/" or "http://lib.stat.cmu.edu/ general/Utexas/".