Software Reliability Using Markov Chain Usage Model

Submitted by

Md. Shazzad Hosain

M. Sc. Engineering Student

Department of Computer Science and Engineering

Student ID: 040205045P

A thesis submitted to the Department of Computer Science and Engineering in partial fulfillment of the requirements for the degree of Masters of Science in Engineering in Computer Science and Engineering

Supervised by

Dr. Md. Shamsul Alam

Professor, Department of CSE, BUET

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

DHAKA, BANGLADESH

AUGUST 2005

The thesis **"Software Reliability Using Markov Chain Usage Model"**, submitted by Md. Shazzad Hosain, Roll No. 040205045P, Registration No. 95405, Session April 2002, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory for the partial fulfillment of the requirements for the degree of Master of Science in Engineering (Computer Science and Engineering) and approved as to its style and contents. Examination held on August 24, 2005.

Board of Examiners

1. ______________ 24/8/05
Dr. Md. Shamsul Alam
Professor & Head
Department of CSE
BUET, Dhaka – 1000

Chairman
(Supervisor)

2. ______________
Dr. Md. Shamsul Alam
Professor & Head
Department of CSE
BUET, Dhaka – 1000

Member
(Ex-officio)

3. ______________
Dr. A. S. M. Latiful Hoque
Associate Professor
Department of CSE
BUET, Dhaka – 1000

Member

4. ______________
Dr. Md. Saidur Rahman
Assistant Professor
Department of CSE
BUET, Dhaka – 1000

Member

5. Haider 24-08-05
Dr. Md. Haider Ali
Associate Professor
Department of CSE
University of Dhaka

Member
(External)

Declaration

I, hereby, declare that the work presented in this thesis is the outcome of the investigation performed by me under the supervision of Dr. Md. Shamsul Alam, Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka. I also declare that no part of this thesis and thereof has been or is being submitted elsewhere for the award of any degree or diploma.

27/8/05

(Md. Shazzad Hosain)

Acknowledgement

First I express my heartiest thanks and gratefulness to Almighty Allah for His divine blessings, which made me possible to complete this thesis successfully.

I feel grateful to and wish to acknowledge my profound indebtedness to Dr. Md. Shamsul Alam, Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology. Deep knowledge and keen interest of Dr. Md. Shamsul Alam in the field of software reliability influenced me to carry out this project and thesis. His endless patience, scholarly guidance, continual encouragement, constructive criticism and constant supervision have made it possible to complete this thesis.

I would like to thank the members of the graduate committee, Dr. A. S. M. Latiful Hoque, Associate Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dr. Md. Saidur Rahman, Assistant Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology and Dr. Md. Haider Ali, Associate Professor, Department of Computer Science and Engineering, University of Dhaka for their helpful suggestions and careful review of this dissertation. Lastly I would like to convey gratitude to all my course teachers here whose teaching helps me a lot to start and complete this thesis work.

// Abstract

Statistical testing gives us opportunity to have statistical inferences such as reliability, mean time to failure (MTTF) etc. for software systems and Markov chain usage model gains it's credibility in this field. Markov chain usage model has several benefits. It allows generating test sequences from usage probability distributions, assessing statistical inferences based on analytical results associated with Markov chains and also to derive stopping criterion of the test process. But the main problem in this process is to model software behavior in a single Markov chain. For large software systems the model size i.e. the number of sates become unwieldy and it becomes infeasible to apply this method in generating test cases as well as measuring reliability.

Two Markov models called usage chain and testing chain are developed from the example software. The discriminant value of the two chains is determined to analyze software reliability. As the software becomes more complex the model size grows quickly, which is known as state explosion problem. To overcome this problem we present a technique to measure software reliability by combining the ideas drawn from stochastic modeling, statistical testing using Markov chain usage model and component based software testing. We have taken example from database based application software, find its modules, in this case forms, and measure reliability of each forms using Markov chain usage model. We then analyze system reliability from those form's reliabilities according to their usage probabilities. Our experimental efforts lead us to a more practical and effective approach for software reliability and quality assurance.

Contents

Declaration	I
Acknowledgement	II
Abstract	III
List of Figures	VIII
List of Tables	X
1. Introduction	1
1.1 Introduction	1
1.2 Literature Review	3
1.3 Problems of Existing Works	5
1.4 Scope of the Thesis	6
2. Testing Techniques	8
2.1 Issues of Software Quality	8
2.2 Software Testing	9
2.3 Testing versus Debugging	10
2.4 Function versus Structure	11
2.4.1 Flow Graphs and Path Testing	11
2.4.1.1 Control Flow Graphs	12
2.4.1.2 Notational Evolution	13
2.4.1.3 Path Testing	17
2.4.1.4 Fundamental Path Selection Criteria	17
2.4.1.5 Path Testing criteria	17
2.4.1.6 Loop Testing	18
2.4.2 Transaction Flow Testing	20
2.4.2.1 Transaction Flows	21
2.4.2.2 Get the Transaction Flows	22
2.4.2.3 Path Selection	22
2.4.3 Data Flow Testing	22
2.4.3.1 Data Object State and Usage	23
2.4.3.2 Data Flow Anomalies	23
2.4.3.3 Static vs. Dynamic Anomaly Detection	25

2.5 Software Testing Stages 26
2.5.1 General Testing Stages 26
2.5.1.1 Unit Testing 27
2.5.1.2 Component Testing 27
2.5.1.3 Integration Testing 27
2.5.1.4 System Testing 27
2.5.2 Specialized Testing Stages 28
2.5.2.1 Stress Testing 28
2.5.2.2 Survivability Testing 29
2.5.2.3 Recovery Testing 29
2.5.2.4 Security Testing 29
2.5.2.5 Compatibility Testing 30
2.5.2.6 Performance Testing 30
2.5.3 User-Involved Testing Stages 31
2.5.3.1 Usability Testing 31
2.5.3.2 Alpha Testing 31
2.5.3.3 Beta Testing 32
3. Reliability Models **33**
3.1 Introduction 33
3.2 State – Based Models 34
3.2.1 LittleWood Model 34
3.2.2 Laprie Model 35
3.2.3 Kubat Model 36
3.2.4 Gokhale et. al. Model 36
3.3 Path – Based Models 37
3.3.1. Shooman Model 38
3.3.2 Krishnamurthy & Mathur Model 38
3.3.3 Yacoub, Cukic and Ammar Model 39
3.4 Additive Models 40
3.4.1 Xie & Wholin Model 40
3.4.2 Everett Model 40
3.5 Input Domain Models 41

3.5.1 Nelson Model 41

3.5.2 Weiss & Weyuker Model 41

3.6 Reliability Growth Models 43

3.6.1 Software Reliability Growth Model Types 45

4. Statistical Testing **49**

4.1 Statistical Testing 49

4.2 Markov Chain Model for Statistical Software Testing 50

4.2.1 The Usage Markov Chain 52

4.2.2 Analysis of The Usage Chain 57

4.2.3 Constructing the Testing Chain 57

4.2.4 Incorporating Failure Data 59

4.2.5. Analytical Results for the Testing Chain 60

4.2.5.1 An Analytical Stopping Criterion 60

4.2.5.2 Measuring reliability, mean time to failure and the impact of failure 62

4.3 Effectiveness of Statistical Software Testing 65

5. Arc-based Reliability **67**

5.1 Introduction 67

5.2 The Miller Reliability Model 67

5.3 Single Use Reliability and Single Action Reliability 68

5.3.1 Testing Records 69

5.3.2 Arc Failure Rate Calculation 70

5.3.3 Single Action Reliability Estimator 70

5.3.4. Miller Model 71

5.4 Single Action Reliability 71

6. Stopping Criteria **76**

6.1 The Euclidean Distance 76

6.2 The Kullback Discriminant 78

6.3 The Sayre Long Run Arc Occupancies 78

6.4. Our Stopping Criterion 79

7. Conclusion **83**

7.1 Introduction 83

7.2 Contributions 83
7.3 Suggestions for Further Research 84
References 86

List of Figures

1.1	Failure Rates	1
1.2	Classification of Software Reliability Models	4
2.1	Flowgraph Elements	12
2.2	Program Example (PDL)	14
2.3	One-to-one Flowchart for Fig. 2.2 Example	15
2.4	Control Flowgraph for Fig. 2.2 Example	16
2.5	Simplified Flowgraph Notation	16
2.6	Classes of loops	19
3.1	Example defect detection data	44
3.2	Concave and S-Shaped Models	46
4.1	Selection Menu	52
4.2	Department Entry Form First State	53
4.3	Department Entry Form Second State	53
4.4	Usage Markov Chain for the Software	54
4.5	Plot of $D(U, T)$	61
4.6	Test Sequence versus Reliability	63
4.7	Expected Number of Steps Between Failures	64
4.8	Stationary Probabilities of States of the Testing Chain	65
4.9	Statistical Testing vs Random Testing (fault lies on high probability path)	65
4.10	Statistical Testing vs Random Testing (fault lies on low probability path)	66
5.1	Search Books Form 1	72
5.2	Search Books Form 2	73
5.3	Usage Markov Chain of Search Books	74
5.4	Single Action Reliability	74
5.5	Single Action Reliability Variance	75
6.1	Euclidean Distance, Example Model	77
6.2	Euclidean Distance, Testing Chain A	77
6.3	Euclidean Distance, Testing Chain B	77
6.4	Example Model, Convergence of Testing Chain to Usage Chain	80
6.5	Convergence of Testing Chain to Usage Chain, Successively Updated Testing Record	81

6.5 Convergence of Testing Chain to Usage Chain, Successively Updated Testing Record 82

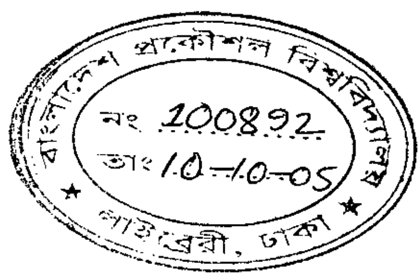
List of Tables

3.1 Software Reliability Growth Model Examples 46

3.2 Software Reliability Model Assumptions 47

4.1 Transition Probabilities for the Example Usage Chain 55

4.2 Some Standard Analytical Results for Markov Chains 56



Chapter 1

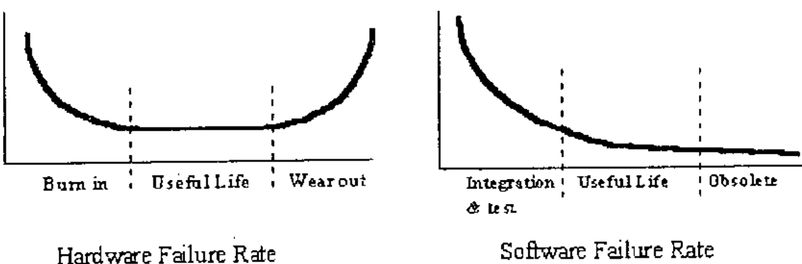
Introduction

1.1 Introduction

Like hardware reliability, software reliability is based on modes of failure. Hardware modes of failure – wear, design flaws, and unintentional environmental phenomena – are more tangible because hardware is a physical entity. In fact, it is this very physical quality that prompts hardware designers to assume that hardware cannot be perfect. Ironically, the same designers often assign perfect reliability to a software component because it can't "wear out," for example.

But software does have a mode of failure, which is based on the assumption that design and development are not perfect process. The mistakes made during these processes manifest as faults in the code, which are revealed as inputs are processed. That is failure occurs when the software does not perform according to specification for an input history.

It is important to recognize that there is a difference between hardware failure rate and software failure rate. For hardware, as shown in Fig. 1.1, when the component is first manufactured, the initial number of faults is high but then decreases as the faulty components are identified and removed or the components stabilize. The component then enters the useful life phase, where few, if any faults are found. As the component physically wears out, the fault rate starts to increase.



Hardware Failure Rate Software Failure Rate

Fig. 1.1: Failure Rates

Software however, has a different fault or error identification rate. For software, the error rate is at the highest level at integration and test. As it is tested, errors are identified and removed. This removal continues at a slower rate during its operational use; the number of errors continually decreasing, assuming no new errors are introduced. Software does not have moving parts and does not physically wear out as hardware, but is does outlive its usefulness and becomes obsolete.

To quantify software reliability in a meaningful way, software use must be modeled as a random process in which a use is selected according to some probability distribution, or use distribution. Reliability then becomes the probability that the software will perform according to specification for a randomly selected use. When the software fails to meet specification during use, a failure occurs.

Reliability can be a useful metric. We can use it to help guide software development. We can also use it to assess a program's fitness for use by conducting experiments to establish empirical evidence of quality.

Reliability can be defined in two different ways. Reliability as a function of time, perhaps the more traditional definition, addresses the design of software that will operate according to specification for a period of time. But we can also use a simpler definition – reliability is the probability that a randomly chosen use (test case) will be processed correctly. Using this latter definition the mean time to failure is the average number of uses between failures. MTTF and reliability can be related mathematically in the models.

1.2 Literature Review

The work on software reliability models started in 70's, the first model being presented in 1972. Today the number of existing models exceeds hundred with more models developed every year. Still there does not exist any model that can be applied in all cases. Models that are good in general are not always the best choice for a particular data set, and it is not possible to know in advance what model should be used in any particular case [1, 2].

Since software reliability models are used in different phases of the Software Development Life Cycle (SDLC), the reliability models are broadly classified under the following categories:

Early prediction models uses characteristics of the software development process from requirement to design and test, and extrapolate this information to predict the behavior of software during operation [3, 4, 5].

Software reliability growth models (SRGM) captures failure behavior of software during testing and extrapolates it to determine its behavior during operation. Hence this category of models uses failure data information and trends observed in the failure data to derive reliability prediction. The SRGMs are further classified as Concave models and S-shaped models [1, 6, 7]. The different types of SRGMs are shown in Fig. 1.3. Goel-Okumoto model is one of the most widely used SRGM [8]. In this model, the failure arrival process is assumed to be non-homogeneous Poisson process (NHPP). The expected cumulative failures, called the mean function $m(t)$ in NHPP, over time t is given by the formula: $m(t) = N(1-e^{-bt})$, where the model constants N (total number of defects in the system) and b (model curvature) need to be estimated from the observation data.

Input domain based models use properties of the input domain of the software to derive a correctness probability estimate from test cases that executed properly [1, 9]. Nelson model [10] is one of the most widely used input domain reliability models and it can be obtained as: $R = 1 - \frac{f}{n} = \frac{n-f}{n}$. When usage time t_i is available for each hit i, the summary reliability measure, mean-time-between-failures (MTBF), can be calculated as: $MTBF = \frac{1}{f}\sum_i t_i$.

Architecture based models put emphasis on the architecture of the software and derives reliability estimates by combining estimates obtained for the different modules of the software. The architecture based software reliability models are further classified into State based models, Path based models and Additive models [11, 12]. The details of some examples of this type of models are given in chapter 3.

Other reliability models are known as hybrid models [1]. Hybrid black box models combine the features of input domain based models and software reliability growth models. Hybrid white box models use selected features from both white box models and black box models. However these models consider the architecture of the system for reliability prediction, therefore these models are considered in hybrid white box models.

However, another alternative model for reliability measurement is Markov chain usage model, though it is not yet offered as a complete reliability model for software [13, 14]. It is the kind of state based reliability model under the category of architecture based reliability model shown in Fig. 1.3. Littlewood model [15] is one of the earliest models of this type. An irreducible Semi-Markov Process (SMP) models the software architecture. This was the generalization of the previous work [16] that describes software architecture with continuous time Markov chain (CTMC).

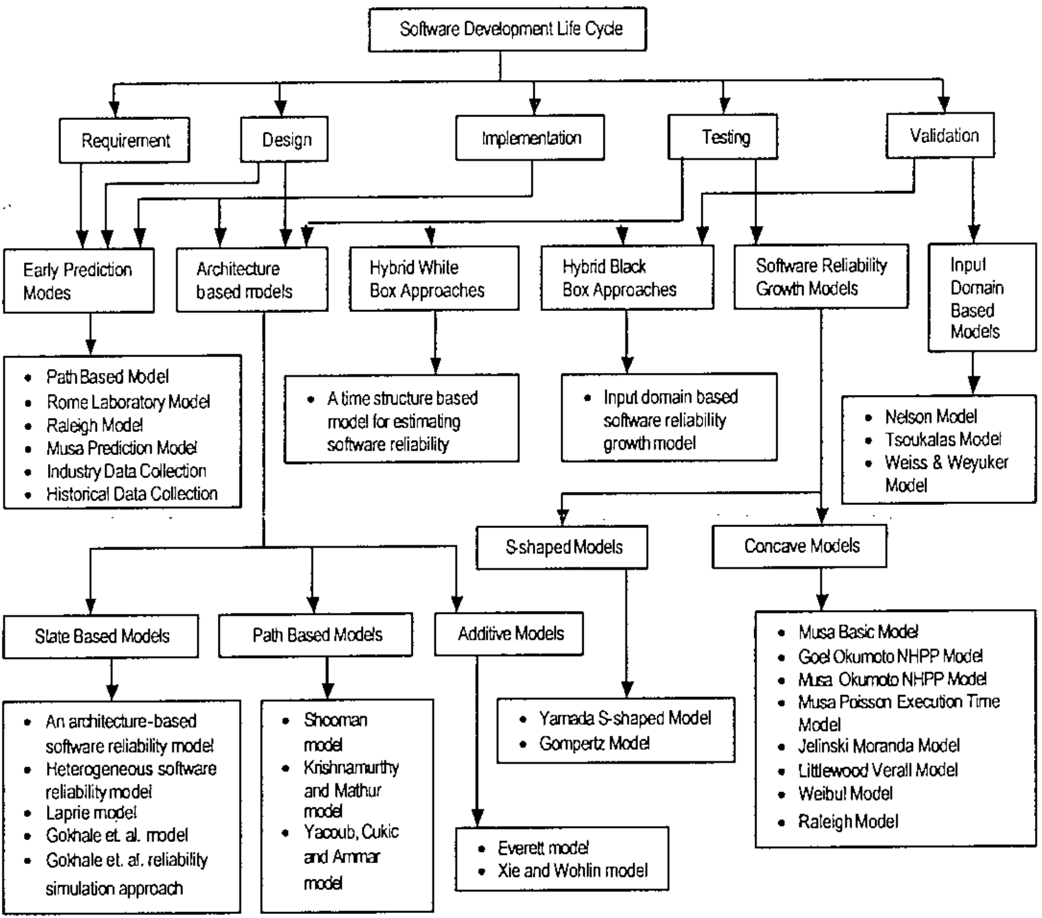


Fig. 1.2: Classification of Software Reliability Models [1]

Reliability measurement according to software usage [20] is another interesting area found in the literature. Several methods have been found but two of them get popularity. One is operational profile (OP) method and the other is Markov chain usage model.

Operational profile [7, 21, 22] is simply the description of expected product usage. An ideal operational profile method would be like this [21]:

- Set up every customer's system in the test lab (all at once).
- Use each system exactly like those customers.
- Count failures and track usage.
- Compute and model the resulting metrics.

On the other hand Markov chain usage model have used successfully in several applications [23, 24, 25], involving both real-time embedded systems and user-oriented applications. Whittaker [13] developed an irreducible finite state Markov chain called usage chain from the software behavior and another Markov chain called testing chain to encode the testing history while measuring reliability and other statistical inferences like mean-time-to-failure etc. Latter on Kirk Sayre developed arc-based reliability models [26] combining the Miller reliability model [27] and Markov chain usage model. This reliability is known as single action reliability.

1.3 Problems of Existing Works

The problems of existing works are summaries as follows:

i) Present works on reliability using Markov chain usage models only give a set of equations for analytical purpose. The examples taken to explain the model were small ones. So it is not clear whether it is applicable for large software systems.

ii) If the software system is large the number of states in the Markov chain becomes large and it becomes infeasible to generate test cases and measure reliability.

iii) Stochastic modeling of software is not well defined. For larger systems the Markov chain is developed in a hierarchical fashion by selecting a primary modeling mode, creating a Markov chain for it (which becomes the top-level in the hierarchy) and then adding the remaining operational modes by expanding states in the top-level model.

iv) There are hundreds of reliability models in literature but not a single model is suitable for all applications. So there is a lack of practical and effective approach to measure software system reliability according to its usage behavior.

1.4 Scope of the Thesis

Since statistical testing based on Markov chain usage model is not feasible for large software system we propose a hybrid approach that combines the ideas from statistical testing, stochastic testing and component bases software testing. The objectives of the thesis are summarized as follows:

i) Develop two models called usage model and testing model from software specification.

ii) Measure reliability of the example software.

iii) Analyze the discriminant value of two models and determine the stopping point of software testing and release of software.

iv) Find the complexities in measuring reliability for large software system.

v) Propose a methodology of single action reliability analysis with improved technique to determine the similarity between usage chain and testing chain.

vi) Find a more practical and effective approach for software testing and quality assurance.

The remaining chapters of this thesis are organized as follows:

- Chapter 2 provides background knowledge on issues of software quality, different types of testing techniques and testing stages or levels.
- Chapter 3 presents some architecture based software reliability models. Two state based models such as Littlewood model and Gokhale model; two path based model such as Shooman model and Krishnamurthy model and one additive model Xie and Wholin model are presented.
- Chapter 4 presents a detailed description of modeling and measuring software reliability of our example software. This chapter shows how Markov chain usage model is used to measure reliability, mean-time-to-failure and to find stopping criteria.

- Chapter 5 shows how Miller reliability model is combined to Markov chain usage model in measuring reliability and finding an analytical stopping criterion. In this chapter we measure reliability of a form/partition of our example software.
- Chapter 6 presents the existing methodologies of determining stopping criterion and the reasons we choose Sayre's criterion as our stopping criterion. It also gives the experimental results from our example software.
- Finally in chapter 7 contributions, limitations and future works of the research are presented. 9

Chapter 2

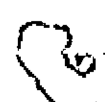
Testing Techniques

2.1 Issues of Software Quality

Quality is defined as the bundle of attributes present in a commodity and, where appropriate, the level of the attribute for which the consumer (software users) holds a positive value. Defining the attributes of software quality and determining the metrics to assess the relative value of each attribute are not formalized processes. Compounding the problem is that numerous metrics exist to test each quality attribute. Because users place different values on each attribute depending on the product's use, it is important that quality attributes be observable to consumers. However, with software there exist not only asymmetric information problems (where a developer has more information about quality than the consumer), but also instances where the developer truly does not know the quality of his own product. It is not unusual for software to become technically obsolete before its performance attributes have been fully demonstrated under real-world operation conditions. As software has evolved over time so has the definition of software quality attributes. McCall et. al. [28] first attempted to assess quality attributes for software. His software quality model characterizes attributes in terms of three categories: product operation, product revision, and product transition. In 1991, the International Organization for Standardization (ISO) adopted ISO 9126 as the standard for software quality (ISO, 1991).

It is structured around six main attributes listed below (sub-characteristics are listed in parenthesis):

1. Functionality (suitability, accurateness, interoperability, compliance, security)
2. Reliability (maturity, fault tolerance, recoverability)
3. Usability (understandability, learnability, operability)
4. Efficiency (time behavior, resource behavior)
5. Maintainability (analyzability, changeability, stability, testability)



6. Portability (adaptability, installability, conformance, replaceability)

Although a general set of standards has been agreed on, the appropriate metrics to test how well software meets those standards are still poorly defined. Publications by IEEE (1988, 1996) have presented numerous potential metrics that can be used to test each attribute. These metrics include

1. Fault density,
2. Requirements compliance,
3. Test coverage, and
4. Mean time to failure.

The problem is that no one metric is able to unambiguously measure a particular quality attribute. Different metrics may give different rank orderings of the same attribute, making comparisons across products difficult and uncertain.

2.2 Software Testing

Software testing is the process of applying metrics to determine product quality. Software testing is the dynamic execution of software and the comparison of the results of that execution against a set of pre-determined criteria. "Execution" is the process of running the software on a computer with or without any form of instrumentation or test control software being present. "Pre-determined criteria" means that the software's capabilities are known prior to its execution. What the software actually does can then be compared against the anticipated results to judge whether the software behaved correctly [29].

In many respects, software testing is an infrastructure technology or "infra-technology." Infra-technologies are technical tools, including scientific and engineering data, measurement and test methods, and practices and techniques that are widely used in industry [30]. Software testing infra-technologies provide the tools needed to measure conformance, performance, and interoperability during the software development. These tools aid in testing the relative performance of different software configurations and mitigate the expense of reengineering software after it is developed and released. Software testing infra-technologies also provide critical information to the software user

regarding the quality of the software. By increasing quality, purchase decision costs for software are reduced.

2.3 Testing versus Debugging

Testing and debugging are often lumped under the same heading, and it's no wonder that their roles are often confused: for some, the two words are synonymous; for others, the phrase "test and debug" is treated as a single word. The **purpose of testing** is to show that a program has bugs. The **purpose of debugging** is to find the error or misconception that led to the program's failure and to design and implement the program changes that correct the error. Debugging usually follows testing, but they differ as to goals, methods, and most important, psychology [29]:

1. Testing starts with known conditions, uses predefined procedures, and has predictable outcomes; only whether or not the program passes the test is unpredictable. Debugging starts form possibly un-known initial conditions, and the end cannot be predicted, except statistically.
2. Testing can and should be planned, designed, and scheduled. The procedures for, and duration of, debugging cannot be so constrained.
3. Testing is a demonstration of error or apparent correctness. Debugging is a deductive process.
4. Testing proves a programmer's failure. Debugging is the programmer's vindication.
5. Testing, as executed, should strive to be predictable, dull, constrained, rigid and inhuman. Debugging demands intuitive leaps, conjectures, experimentation, and freedom.
6. Much of testing can be done without design knowledge. Debugging is impossible without detailed design knowledge.
7. An outsider can often do testing. An insider must do debugging.
8. Although there is a robust theory of testing that establishes theoretical limits to what testing can and can't do, debugging has only recently been attacked by theorists – and so far there are only rudimentary results.

9. Much of test execution and design can be automated. Automated debugging is still a dream.

2.4 Function versus Structure

Test can be designed from a functional or a structural point of view. In functional testing the program or system is treated as a black box. It is subjected to inputs, and its outputs are verified for conformance to specified behavior [29, 31]. The software's user should be concerned only with functionality and features, and the program's implementation details should not matter. Functional testing takes the user's point of view.

Structural testing does look at the implementation details. Such things as programming style, control method, source language, database design, and coding details dominate structural testing; but the boundary between function and structure is fuzzy. Good systems are built in layers – from the outside to the inside. The user sees only the outermost layer, the layer of pure function. Each layer inward is less related to the system's functions and more constrained by its structure; so what is structure to one layer is function to the next. For example, the user of an online system doesn't know that the system has a memory – allocation routine. For the user, such things are structural details. The memory – management routine uses a link – block subroutine. The memory – management routine's designer writes a "functional" specification for a link – block subroutine, thereby defining a further layer of structural detail and function. At deeper levels, the programmer views the operation system as a structural detail, but the operation system's designer treats the computer hardware logic as the structural details.

There's no controversy between the use of structural versus functional tests: both are useful, both have limitations; both target different kinds of bugs. Functional tests can, in principle, detect all bugs but would take infinite time to do so. Structural tests are inherently finite but cannot detect all errors, even if completely executed. The art of testing in part is in how you choose e between structural and functional tests. Some structural and functional testing techniques are discussed below.

2.4.1 Flow Graphs and Path Testing

Path testing is the name given to a family of test techniques based on judiciously selecting a set of test paths through the program. If the set of paths is properly chosen, then we have achieved some measure of test thoroughness. For example, pick enough paths to assure that every source statement has been executed at lease once. Path testing is most applicable to new software for unit testing. It requires complete knowledge of the program's structure (i.e., source code). Programmers to unit-test their own code most often use it. The effectiveness of path testing rapidly deteriorates as the size of the software aggregate under test increases. Path testing is rarely, if ever, used for system testing. For the programmer, it is the basic test technique.

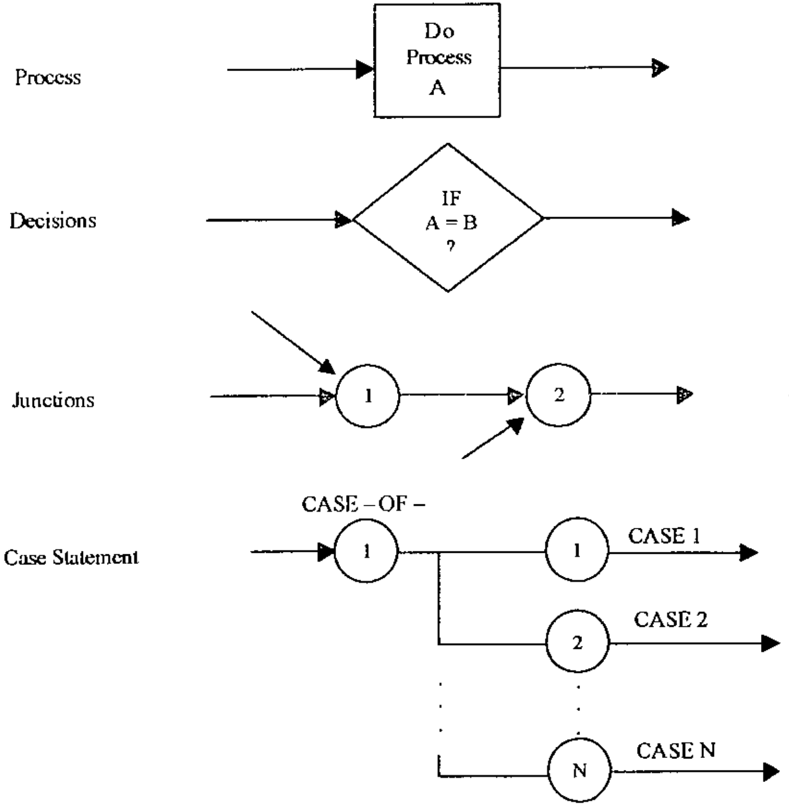


Fig. 2.1: Flowgraph Elements

2.4.1.1 Control Flowgraphs

The control flowgraph (or flowgraph alone when the context is clear) is a graphical representation of a program's control structure. It uses the elements shown in Fig. 2.1: process blocks, decisions and junctions. The control flowgraph is similar to the earlier flowchart, with which it is not to be confused.

Process block: a process block is a sequence of program statements uninterrupted by either decisions or junctions. Formally, it is a sequence of statements such that if any one statement of the block is executed, then all statements thereof are executed. Less formally, a process block is a piece of straight-line code. A process block can be one source statement or hundred. The point is that, bugs side, once a process block is initiated, every statement within it will be executed. The term "process' will be used interchangeably with "process block". From the point of view of test cases designed from control flowgraphs, the details of the operation within the process are unimportant if those details do not affect the control flow. If the processing does affect the flow of control, the effect will be manifested at a subsequent decision of case statement.

Decisions and case statements: A decision is a program point at which the control flow can diverge. Machine language conditional branch and conditional skip instructions are examples of decisions. The FORTRAN IF and the Pascal IF-THEN-ELSE constructs are decisions, although they also contain processing components. A case statement is a multi-way branch or decision. Examples of case statements include a jump table in assembly language, the FORTRAN-computed GOTO and assigned GOTO, and the Pascal CASE statement. From the point of view of test design, there are no fundamental differences between decisions and case statements.

Junctions: A junction is a point in the program where the control flow can merge. Examples of junctions are: the target of a jump of skip instruction in assembly language, a label that is the target of GOTO, the END-IF and CONTINUE statements in FORTRAN, and the Pascal statement labels, END and UNTIL.

2.4.1.2 Notational Evolution

The control flowgraph is a simplified (i.e., more abstract0 representation of the program's structure. To understand its creation and use, we'll go through an example, starting with Fig. 2.2 – a little horror written in a FORTRAN – like program design language (PDL). The first step in translating this to a control flowgraph is shown in Fig. 2.3, where we have the typical one-for-one classical flowchart. In Fig. 2.4 we merged the process steps and replaced them with the ingle process box. We now have a control

flowgraph. But this representation is still too busy. We simplify the notation further to achieve Fig. 2. 5 where for the first time we can really see what the control flow looks like.

```
                    CODE*  (PDL)

       INPUT X, Y
       Z := X + Y
       V := X - Y
       IF Z >= 0 GOTO SAM
JOE:      Z := Z -1
SAM:   Z := Z + V
       FOR U = 0 TO Z
       V(U), U(V) := (Z + V)*U
       IF V(U) = 0 GOTO JOE
       Z := Z - 1
       IF Z = 0 GOTO ELL
       U := U + 1
       NEXT U
       V(U - 1) := V(U + 1) + U(V - 1)
ELL:      V(U + U(V)) := U = V
       IF U = V GOTO JOE
       IF U > V THEN U := Z
       Z := U
       END
```

Fig. 2.2: Program Example (PDL)

Fig. 2.5 is the way we usually represent the program's control flow-graph. There are two kinds of components: circles and arrows that join circles. A circle with more than one arrow leaving it is a decision; a circle with more than one arrow entering is a junction. We call it e circles nodes and the arrows links. Note also that the entry and exit are also denoted by circles and are thereby also considered to be nodes. Nodes are usually numbered or labeled by using the original program labels. The link name can be formed from the names of the nodes it spans. Thus a link from node 7 to nod e 4 is called link (7, 4), whereas one from node 4 to node 7 is called link (4, 7). For parallel links between a pair of nodes, (nodes 12 and 13 in Fig. 2.5) we can use subscripts to denote each one or some unambiguous notation such as (12, 13, upper) and (12, 13, lower). And alternate way to name links that avoids this problem is to use a unique lowercase letter for each like in the flowgraph.

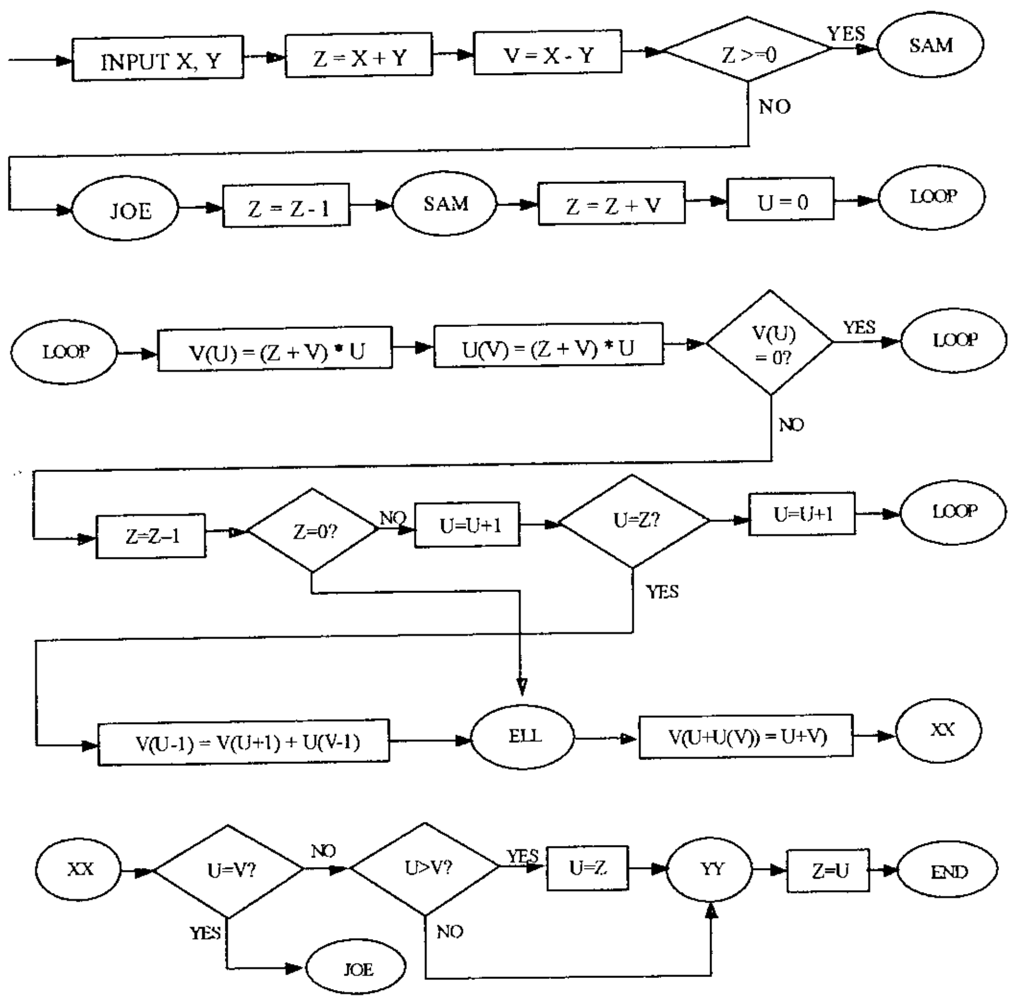


Fig. 2.3 : One-to-one Flowchart for Fig. 2.2 Example

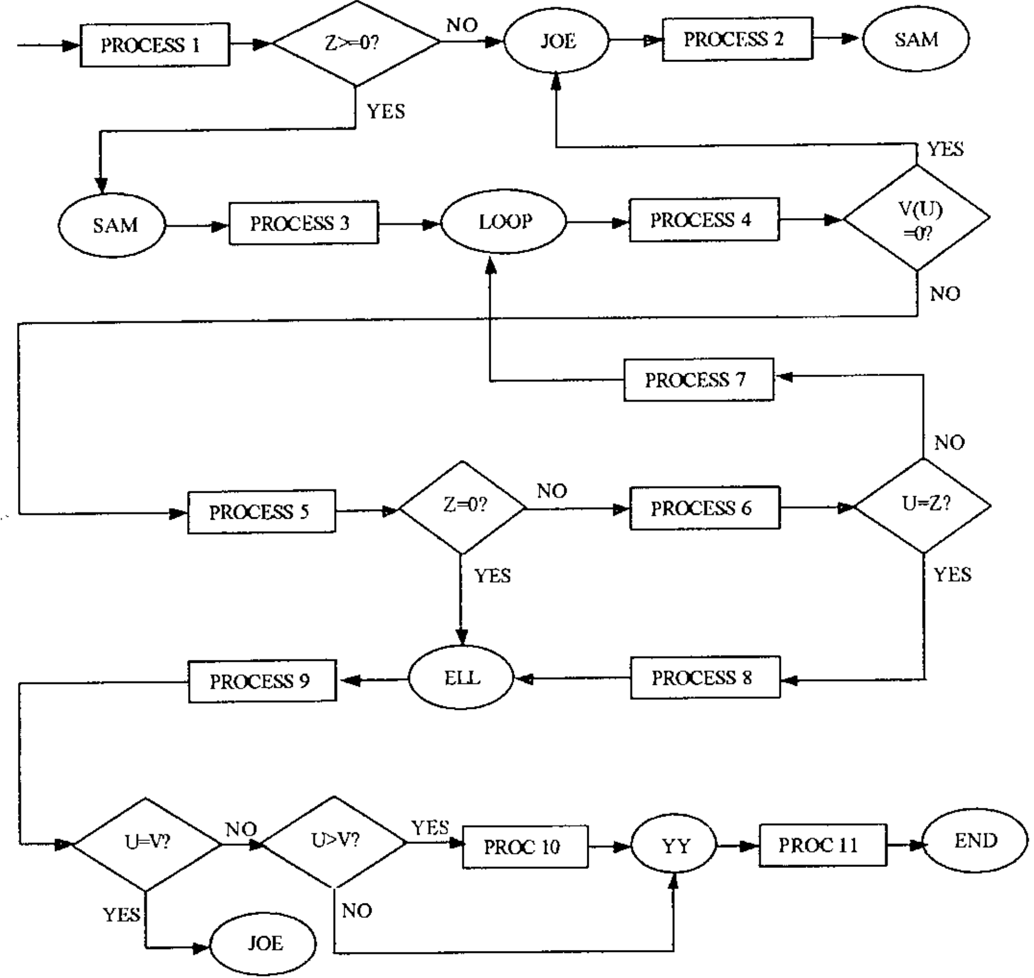


Fig. 2.4: Control Flowgraph for Fig. 2.2 Example

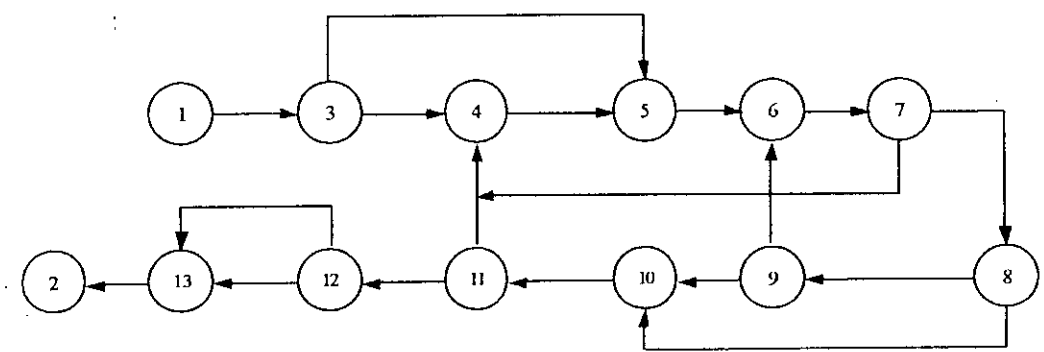


Fig. 2.5: Simplified Flowgraph Notation

2.4.1.3 Path Testing

A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit. A path may go through several junctions, processes, or decisions, one or more times. Paths consist of segments. The smallest segment is a link – that is, a single process that lies between two nodes (e.g., junction – process - decision). A path segment is a succession of consecutive links that belongs to the same path. The length of a path is measured by the number of links in it and not by the number of instructions or statements executed along the path. The name of a path is the name of the nodes along the path. For example, the shortest path from entry to exit in Fig. 2.5 is called "(1,3,5,6,7,8,10,11,12,13,2)". The terms entry/exit path and compete path are also used in the literature to denote a path that starts at an entry and goes to an exit. Our interest in entry /exit paths in testing are pragmatic because: (1) It's difficult t set up and execute paths that start at an arbitrary statement; (2) it's hard to stop at an arbitrary statement without setting traps or using patches and (3) entry/exit paths are what we want to test because we use routines that way.

2.4.1.4 Fundamental Path Selection Criteria

There are many paths between the entry and exit of a typical routine. A lavish test approach might consist of testing all paths, but that would not be a complete test, because a bug could create unwanted paths or make mandatory paths un-executable. And just because all paths are right doesn't mean that the routine is doing the required processing along those paths. Such possibilities aside for the moment, how might we define "complete testing"?

1. Exercise every path from entry to exit.
2. Exercise every statement or instruction at least once.
3. Exercise every branch and case statement, in each direction, at least onee.

2.4.1.5 Path Testing Criteria

Any testing strategy based on paths must at least both exercise every instruction and take branches in all directions. A set of tests that does this is not complete in an absolute sense, but it is complete in the sense that anything less must leave something untested.

We have, therefore, explored three different testing criteria or strategies out of a potentially infinite family of strategies. They are:

1. *Path Testing* (P_∞) – Execute all possible control flow paths through the program: typically, this is restricted to all possible entry/exit paths through the program. If we achieve this prescription, we are said to have achieved 100% path coverage. This is the strongest criterion in the path-testing strategy family: it is generally impossible to achieve.

2. *Statement Testing* (P_1) – Execute all statements in the program at least once under some test. If we do enough tests to achieve this, we are said to have achieved 100% statement coverage. An alternate, equivalent characterization is to say that we have achieved 100% node coverage. This is the weakest criterion in the family; testing less than this for new software is unconscionabk and should be criminalized.

Branch Testing (P_2) – Execute enough tests to assure that every branch alternative has been exercised at least once under some test. If we do enough tests to achieve this prescription, then we have achieved 100% branch coverage. An alternative characterization is to say that we have achieved 100% link coverage. For structured software, branch testing and therefore branch coverage strictly includes statement coverage.

2.4.1.6 Loop Testing

Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests. Loop testing is white box testing i.e. structural testing technique that focuses exclusively on the validity of loop construct. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops and unstructured loops. The loops are shown in Fig. 2.6.

Simple loops: The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.

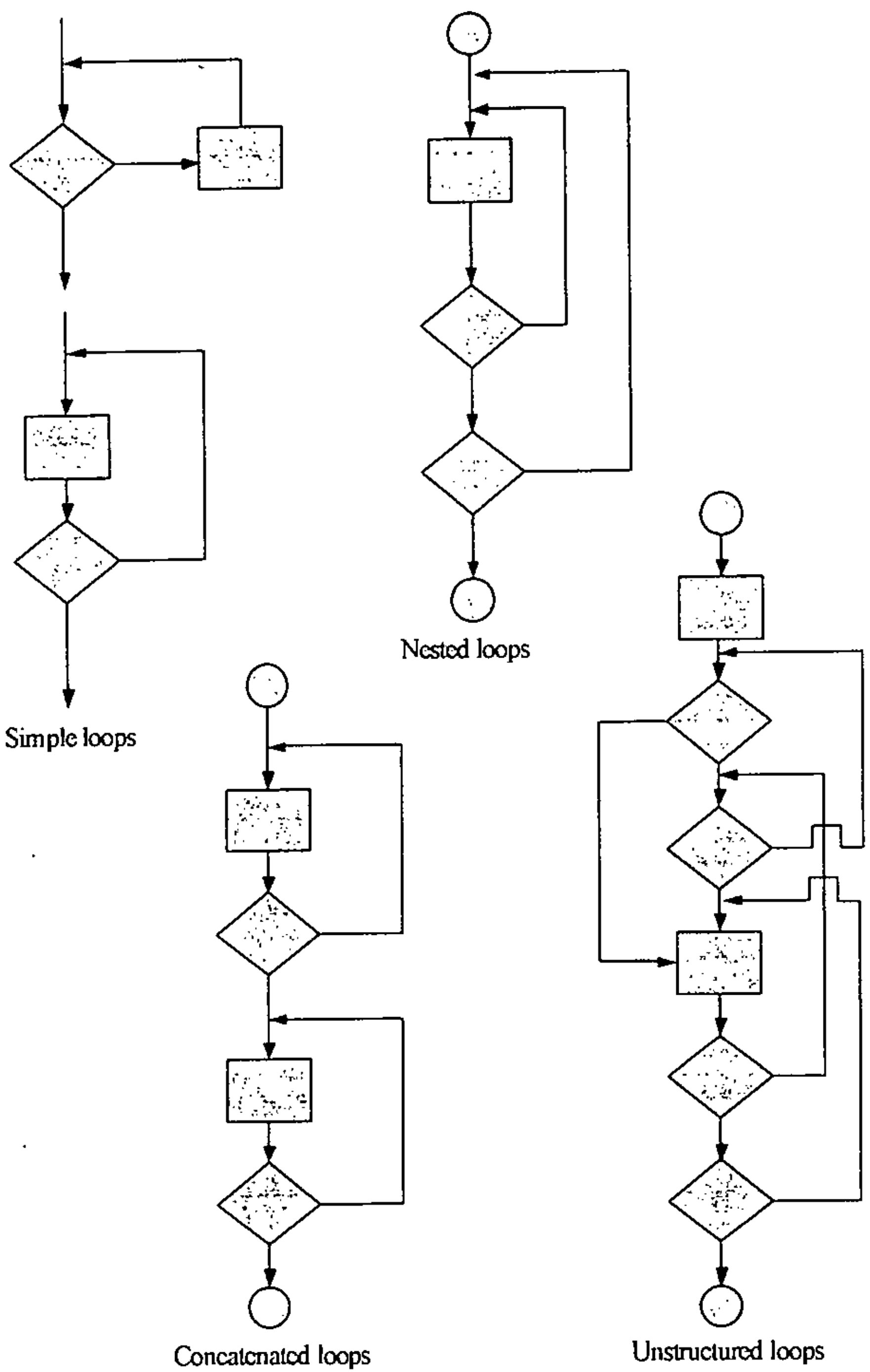


Fig. 2.6: Classes of loops

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. m passes through the loop where $m < n$.
5. $n - 1, n, n + 1$ passes through the loop.

Nested loops: if we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. Beizer [29] suggests an approach that will help to reduce the number of test:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g. loop counter) values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
4. Continue until all loops have been tested.

Concatenated loops: concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop *1* is used as the initial value for loop *2*, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

Unstructured loops: Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.

2.4.2 Transaction-Flow Testing

The control flowgraph was introduced as a structural model. Here the same conceptual components and methods over a different kind of flowgraph, the transaction flowgraph – this time though, to create a behavioral model of the program that leads to functional testing. The transaction flowgraph is, if you will, a model of the structure of

the system's behavior. Transaction flows and transaction-flow testing are to the independent system tester what control flows and path testing are to the programmer.

2.4.2.1 Transaction Flows

A transaction is a unit of work seen from a system user's point of view. A transaction consists of a sequence of operations, some of which are performed by a system, persons, or devices that are outside of the system. Transactions begin with **birth** – that is they are created as a result of some external act. At the conclusion of the transaction's processing, the transaction is no longer in the system, except perhaps in the form of historical records. A transaction for an online information retrieval system might consist of the following steps or **tasks**:

1. Accept input (tentative birth)
2. Validate input (birth)
3. Transmit acknowledgement to requester
4. Do input processing
5. Search file
6. Request directions form user
7. Accept input
8. Validate input
9. Process requester
10. Update file
11. Transmit output
12. Record transaction in log and cleanup (death)

The user sees this scenario as a single transaction. From the system's point of view, the transaction consists of twelve steps and ten different kinds of subsidiary tasks.

Most online systems process many kinds of transaction. For example, an automatic bank teller machine can be used for withdrawals, deposits, bill payments and money transfers. Furthermore, these operations can be done for a checking account, savings account, vacation account, Christmas club and so on. Although the sequence of

operations may differ form transaction to transaction, most transactions have common operations. For example, the automatic teller machine begins every transaction by validating the user's card and password number. Tasks in a transaction flowgraph correspond to processing steps in a control flowgraph. As with control flows, there can be conditional and unconditional branches and junctions.

2.4.2.2 Get the Transaction Flows

Complicated systems that process a lot of different complicated transactions should have explicit representations of the transaction flows, or the equivalent, documented. Transaction flows are like control flowgraphs, and consequently we should expect to have them in increasing levels of detail. It is correct and effective to have subflows analogous to subroutines in control flow-graphs, although there may not be any processing module that corresponds to such subflows.

2.4.2.3 Path Selection

Select a covering set of paths based on functionally sensible transactions as you would for control flowgraphs. Confirm these with the designers. Having designed those (easy) tests, now we do exactly the opposite of what we should have done for unit tests. We try to find the most tortuous, longest, strangest path from the entry to the exit of the transaction flow. We create a catalog of these weird paths; go over them not just with the high level designer who laid out the transaction flows, but also with the next-level designers who are implementing the modules that will process the transaction. It can be gratifying experience, even in a good system. The act of discussing the weird paths will expose missing interlocks, duplicated interlocks, interface problems, programs working at cross-purposes, duplicated processing – a lot of stuff that would otherwise have shown up only during the final acceptance tests, or worse, after the system was operating.

2.4.3 Data-Flow Testing

Data-flow testing is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects. For example, pick enough paths to assure that every data object has been initialized prior to use or that all defined objects have been used for something.

2.4.3.1 Data Object State and Usage

Data objects can be created, killed and/or used. They can be used in two distinct was: in a calculation or as part of a control flow predicate. The following symbols denote these possibilities:

d – defined, created, initialized, etc.

k – killed, undefined, released.

u – used for something.

c – used in a calculation.

p – used in a predicate.

Defined – An object is **defined** explicitly when it appears in a data declaration or implicitly (as in FORTRAN) when it appears on the left-hand side of an assignment statement.

Killed or Undefined – An object is **killed** or **undefined** when it is released or otherwise made unavailable, or when its contents are no longer known with certitude. For example, the loop control variable in FORTRAN is undefined when the loop is exited; release of dynamically allocated objects back to the availability pool is 'killing' or 'un-defining'; return of records; the old top of the stack after it is popped; a file is closed. Define and kill are complementary operation. That is, they generally come in pairs and one does the opposite of the other. When you see complementary operations on data objects it should be a signal to you that a data-flow model and therefore data-flow testing methods, might be effective.

Usage – A variable is used for computation (c) when it appears on the right-hand side of an assignment statement, as a pointer, as part of a pointer calculation, a file record is read or written, and so on. It is used in a predicate (p) when it appears directly in a predicate (for example, IF A>B …), but also implicitly as the control variable of a loop, in an expression used to evaluate the control flow of a case statement, as a pointer to an object that will be used to direct control flow.

2.4.3.2 Data-Flow Anomalies

An anomaly is denoted by a twocharacter sequence of actions. For example, ku means that the object id killed and then used (possible in some languages), whereas dd means that the object is defined twice with out an intervening usage. There are nine possible twoletter combinations for d, k and u. some are bugs, some are suspicious and some are okay.

dd – probably harmless but suspicious. Why define the object twice without an intervening usage?

dk – probably a bug. Why define the object without using it?

du – the normal case. The object is defined and then used.

kd – normal situation. An object is killed and then redefined.

kk – harmless but probably buggy. Did we want to be sure it was really killed?

ku – a bug. The object doesn't exist in the sense that its value is undefined or indeterminate. For example, the loop-control value in a FORTRAN program after exit from the loop.

ud – usually not a bug because the language permits reassignment at almost any time.

uk – normal situation.

uu – normal situation.

In addition to the above two-letter situations there are six single-letter situations. We'll use a leading dash to mean that nothing of interest (d, k, u) occurs prior to the action noted along the entry-exit path of interest and a trailing dash to mean that nothing happens after the point of interest to the exit.

$-k$: possibly anomalous because from he entrance to this point on the path, ,the variable had not been defined. We're killing a variable that does not exit; but note that the variable might have been created by a called routine or might be global.

$-d$: okay. This is just the first definition along this path.

$-u$: possibly anomalous. Not anomalous if the variable is global and has been previously defined.

$k-$: not anomalous. The last thing done on this path was to kill the variable.

d-: possibly anomalous. The variable was defined and not used on this path; but his could be a global definition or within a routine that defines the variables for other routines.

u-: not anomalous. The variable was used but not killed on this path. Although this sequence is snot anomalous, it signals a frequent kind of bug. If d and k mean dynamic storage allocation and return respectively, this could be an instance in which a dynamically allocated object was not returned to the pool after use – not a bug if we expect some other routine to return it.

The single-letter situations do not lead to clear data-flow anomalies but only the possibility thereof. Also, whether or not a single-letter situation is anomalous is an integration testing issue rather than a component testing issue because the interaction of two or more components is involved.

2.4.3.3 Static versus Dynamic Anomaly Detection

Static analysis is analysis done on source code without actually executing it. Dynamic analysis is done on the fly as the program is being executed and is based on intermediate values that result from the program's execution. Source-code system error detection is the archetypal static analysis result, whereas a division by zero warning is the archetypal dynamic analysis result. If a problem, such as a data-flow anomaly, can be detected by static analysis methods, then it does not belong in testing – it belongs in the language processor.

There's actually a lot more static analysis for data flow anomalies going on in current language processors than we might realize at first. Languages, which force variable declarations, can detect $-u$ and ku anomalies and optimizing compilers can detect some (but not all) instances of dead variables. The run-time resident portion of the compiler and/or the operating system also does dynamic analysis for us and therefore helps in testing by detecting anomalous situations. Most anomalies are detected by such means; that is, we don't have to put in special software or instrumentation to detect an attempt, say to read a closed file, but we do have to assure that we design tests that will traverse paths on which such things happen.

2.5 Software Testing Stages

Aggregated software testing activities are commonly referred to as software testing phases or stages [32]. A software testing stage is a process for ensuring that some aspect of a software product, system, or unit functions properly. The number of software testing stages employed varies greatly across companies and applications. The number of stages can range from as low as 1 to as high as 16 [32].

For large software applications, firms typically use a 12-stage process that can be aggregated into three categories:

1. General testing stages include subroutine testing, unit testing, new function testing, regression testing, integration, and system testing.
2. Specialized testing stages consist of stress or capacity testing, performance testing, platform testing and viral protection testing.
3. User-involved testing stages incorporate usability testing and field-testing.

After the software is put into operational use, a maintenance phase begins where enhancements and repairs are made to the software. During this phase, some or all of the stages of software testing will be repeated. Many of these stages are common and well understood by the commercial software industry, but not all companies use the same vocabulary to describe them. Therefore, as we define each software stage below, we identify other names by which that stage is known.

2.5.1. General Testing Stages

General testing stages are basic to software testing and occur for all software [32]. The following stages are considered general software testing stages:

1. Unit testing
2. Component testing
3. Integration testing
4. System testing

2.5.1.1. Unit Testing

A unit is the smallest testable piece of software, by which I mean that it can be compiled or assembled, linked, loaded, and put under the control of a test harness or driver. A unit is usually the work of one programmer and it consists of several hundred or fewer, lines of source code. Unit testing is the testing we do to show that the unit does not satisfy its functional specification and / or that its implemented structure does not match the intended design structure. When our tests reveal such faults, we say that there is a unit bug.

2.5.1.2. Component Testing

A component is an integrated aggregate of one or more units. A unit is a component, a component with subroutines it calls is a component, etc. by this (recursive) definition, a component can be anything from a unit to an entire s system. Component testing is the testing we do to show that the component does not satisfy its functional specification and/or that its implemented structure does not match the intended design structure. When our tests reveal such problems, we say that there is a component bug.

2.5.1.3. Integration Testing

Integration is a process by which components are aggregated to create larger components. Integration testing is testing done to show that even though the components were individually satisfactory, as demonstrated by successful passage of component tests, the combination of components are incorrect or inconsistent. For example, components A and B are both passed their component tests. Integration testing is aimed as showing inconsistencies between A and B. examples of such inconsistencies are improper call or return sequences, inconsistent data validation criteria, and inconsistent handling of data objects. Integration testing should not be confused with testing integrated objects, which is just higher-level component testing. Integration testing is specifically aimed at exposing the problems that arise from the combination of components. The sequence, then, consists of component testing for components A and B, integration testing for the combination of A and B and finally, component testing of the "new" component (A, B).

2.5.1.4. System Testing

A system is big component. System testing is aimed at revealing bugs that cannot be attributed to components as such, to the inconsistencies between components, or to the

planned interactions of components and other objects. System testing concerns issues and behaviors that can only be exposed by testing the entire integrated system or a major part of it. System testing includes testing for performance, security, accountability, configuration sensitivity, start-up, and recovery.

2.5.2. Specialized Testing Stages

Specialized software testing stages occur less frequently than general software testing stages and are most common for software with well-specified criteria. The following stages are considered specialized software testing stages:

1. Stress, capacity, or load testing
2. Error-handling/survivability testing
3. Recovery testing
4. Security testing
5. Compatibility testing
6. Performance testing

2.5.2.1. Stress Testing

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example

a) Special tests may be designed that generate ten interrupts per second, when one or two is the average rate

b) Input data rates may be increased by an order of magnitude to determine how input functions will respond

c) Test cases that may cause thrashing in a virtual operating system are designed

d) Test cases that may cause excessive hunting for disk-resident data are created.

Essentially the tester attempts to break the program.

A variation of stress testing is a technique call sensitivity testing. In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation. Sensitivity testing attempts to

uncover data combinations within valid input classes that may cause instability or improper processing.

2.5.2.2. Survivability Testing

Perfect software, imperfectly deployed, or deployed in such a way that is vulnerable to failure or attack is of no more value than imperfect software that fails of its own accord. A truly useful metric for distributed, service-based software must measure both the quality of the software itself (the traditional role) and the quality of its configuration vis a vis the underlying infrastructure and the kinds of threats to which the software and infrastructure are subject. In the real world, systems can fail for a variety of reasons other than code and specification errors (e.g., a virus might corrupt the file system that the software relies upon). Thus, rather than ask simply whether the specification and code are correct, it is necessary to ask how likely it is that the system will continue to provide the desired functionality, or failing this, something approaching it. A *survivable* system [33] is one in which actions can be taken to reconfigure applications in the event of partial failures to achieve functionality approximating the functionality of the original system. The usefulness of a survivable system can be judged in several ways: how useful is what it is doing now?; how useful is it likely to be in the future?; if it breaks, can it be repaired so that it can again do something useful?

2.5.2.3. Recovery Testing

Many computer-based systems must recover from faults and resume processing within a pre-specified time. In this case, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.

Recovery testing is a system test that forces the software to fail in variety of ways and verifies that recovery is properly performed. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

2.5.2.4. Security Testing

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal

penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport; disgruntled employees who attempt to penetrate for revenge; dishonest individuals who attempt to penetrate for illicit personal gain.

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. To quote Beizer [29]: "The system's security must, of course, be tested for invulnerability from frontal attack – but must also be tested for invulnerability form flank or rear attack".

During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Anything goes! The tester may attempt to acquire passwords through external clerical means; may attack the system with custom software designed to breakdown any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.

Given enough time and resources, god security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

2.5.2.5. Compatibility Testing

Testing to ensure compatibility of an application or Web site with different browsers, operating systems, and hardware platforms. Compatibility testing can be performed manually or can be driven by an automated functional or regression test suite.

2.5.2.6. Performance Testing

For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. Performance testing s designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as white-box tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource

utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

2.5.3. User-Involved Testing Stages

For many software projects, the users and their information technology consultants are active participants at various stages along the software development process, including several stages of testing. Users generally participate in the following stages.

1. Usability testing
2. Lab or alpha testing
3. Field or beta testing
4. Acceptance testing

2.5.3.1. Usability Testing

For a large number of products, it is believed that the usability becomes the final arbiter of quality. This is true for a large number of desktop applications that gained market share through providing a good user experience. Usability testing needs to not only assess how usable a product is but also provide feedback on methods to improve the user experience and thereby gain a positive quality image. The best practice for usability testing should also have knowledge about advances in the area of Human Computer Interface.

2.5.3.2. Alpha Testing

Alpha testing is the software prototype stage when the software is first able to run. It will not have all the intended functionality, but it will have core functions and will be able to accept inputs and generate outputs. An alpha test usually takes place in the developer's offices on a separate system. In-depth software reliability testing, installation testing, and documentation testing are not done at alpha test time, as the software is only a prototype. Alpha tester feedback forms are not used, although the developer does request feedback on specific aspects of the software [31].

2.5.3.3. Beta Testing

The idea of a Beta is to release a product to a limited number of customers and get feedback to fix problems before a larger shipment. For larger companies, such as IBM, Microsoft and Oracle, many of their products are used internally, thus forming a good beta audience. Techniques to best conduct such an internal Beta test are essential for us to obtain good coverage and efficiently use internal resources. This best practice has everything to do with Beta programs though on a smaller scale to best leverage it and reduce cost and expense of an external Beta [31].

An opportunity that a beta program provides is that one gets a large sample of users to test the product. If the product is instrumented so that failures are recorded and returned to the vendor, they would yield an excellent source to measure the mean time between failure of the software. There are several uses for this metric. Firstly, it can be used as a gauge to enhance the product's quality in a manner that would be meaningful to a user. Secondly, it allows us to measure the mean time between failure of the same product under different customer profiles or user sets. Thirdly, it can be enhanced to additionally capture first failure data that could benefit the diagnosis and problem determination. Microsoft has claimed that they are able to do at least the first two through in3strumented versions that they ship in their betas.

Chapter 3

Reliability Models

3.1 Introduction

Software reliability can be considered to be the probability that the software will operate successfully. Because the measurement of software reliability is in principle the modeling of a deterministic process by a probabilistic one, a problem to be solved is what event should be considered random. When the measurement of reliability involves units of time, such as the time to the next failure of the software, we refer to the implicit definition of reliability as *time-dependent*. Otherwise, reliability measurement involves occurrences of some other event of interest, such as a successful run of the program, and we say the definition is *time-independent*.

A number of analytical models have been proposed to address the problem of quantifying the software reliability, one of the most important metrics of software quality. However, a great deal of this research effort has been focused on modeling the reliability growth during the debugging phase [8, 34, 35]. These so called black-box models treat the software as a monolithic whole, considering only its interactions with the external environment, without an attempt to model the internal structure. Their main common feature is the assumption of some parametric model of the number of failures over a finite time interval or of the time between failures. Failure data obtained while the application is tested are then used to estimate model parameters or to calibrate the model. We discuss some models here. The models are broadly classified into five categories.

1. State – based models
2. Path – based models
3. Additive models
4. Input domain models
5. Reliability growth models

5.2 State – Based Models

This class of models uses the program flow graph to represent the architecture of the system assuming that the transfer of control between modules has a Markov property [11, 12]. This means that given the knowledge of the module in control at any given time, the future behavior of the system is conditionally independent of the past behavior. The architecture of software has been modeled as a discrete time Markov chain (DTMC), continuous time Markov chain (CTMC), or semi Markov process (SMP). These can be further classified into irreducible and absorbing, where the former represents an infinitely running applications, and the latter a terminating one.

3.2.1 Littlewood Model [11, 15]

This is one of the earliest, yet a fairly general architecture - based software reliability model.

Architecture: It is assumed that software architecture can be described by an irreducible SMP, thus generalizing the previous work [16], which describes software architecture with CTMC. The program comprises a finite number of modules and the transfer of control between modules is described by the probability $p_{ij} = Pr$ {program transits from module i to module j}. The time spent in each module has a general distribution $F_{ij}(t)$ with a mean sojourn time μ_{ij}.

Failure behavior: Individual modules, when they are executing, fail with constant failure rates v_i. The transfers of control between modules (interfaces) are themselves subject to failure; when module i calls module j there is a probability λ_{ij} of a failure's occurring.

Solution method: The interest is focused on the total number of failures of the integrated-program in time interval $(0, t)$, denoted by $N(t)$, which is the sum of the failures in different modules during their sojourn times, together with the interface failures. It is possible to obtain the complete description of this failure point process, but since the exact result is very complex, it is unlikely to be of practical use. The asymptotic Poisson process approximation for $N(t)$ is obtained under the assumption that failures are very infrequent. Thus, the times between failures will tend to be much larger than the

times between exchanges of control, that is, many exchanges of control would take place between successive program failures. The failure rate of this Poisson process is given by

$$\sum_{i} a_i v_i + \sum_{i,j} b_{ij} \lambda_{ij}$$

where a_i represents the proportion of time spent in module i, and b_{ij} is the frequency of transfer of control between i and j. These terms depend on p_{ij}, v_i, λ_{ij}, μ_{ij} and the steady state probabilities of the embedded Markov chain π_i.

3.2.2 Laprie Model [18]

This model is a special case of Littlewood model and the result, although obtained in a different way, agrees with those given in [15].

Architecture: The software system is made up of n components and the transfer of control between components is described by CTMC. The parameters are the mean execution time of a component i given by $1/\gamma_i$, and the probability q_{ij} that component j is executed after component i given that no failure occurred during the execution of component i.

Failure behavior: Each component fails with constant failure rate λ_i.

Solution method: The model of the system is an $n + 1$ state CTMC where the system is up in the states i, $0 \leq i \leq n$ (component i is executed without failure in state i) and the $(n + 1)$th state (absorbing state) being the down state reached after a failure occurrence. The associated generator matrix between the up states B is such that $b_{ii} = -(\gamma_i + \lambda_i)$ and $b_{ij} = q_{ij}\gamma_i$, for $i \neq j$. The matrix B can be seen as the sum of two generator matrices such that the execution process is governed by B' whose diagonal entries are equal to $-\gamma_i$ and its off-diagonal entries to $q_{ij}\gamma_i$, and the failure process is governed by B'' whose diagonal entries are zero.

It is assumed that the failure rates are much smaller than the execution rates, that is, the execution process converges towards steady state before a failure is likely to occur. As a consequence, the system failure rate becomes $\lambda_{eq} \sim \sum_{i=1}^{n} \pi_i \lambda_i$, where the steady state probability vector $\pi = [\pi_i]$ is the solution of $\pi B' = 0$. This result has a simple physical

interpretation having in mind that π_i is the proportion of time spent in state i when no failure occurs.

3.2.3 Kubat Model [19]

This model considers the case of software composed of M modules designed for K different tasks. Each task may require several modules and the same module can be used for different tasks.

Architecture: Transition between modules follow a DTMC such that with probability $q_i(k)$ task k will first call module i and with probability $p_{ij}(k)$ task k will call module j after executing in module i. The sojourn time during the visit in module i by task k has the pdf $g_i(k,t)$. Thus, the architecture model for each task becomes a SMP.

Failure model: The failure intensity of a module i is α_i.

Model solution: The probability that no failure occurs during the execution of task k while in module i is

$$R_i(k) = \int_0^{\infty} e^{\alpha_i t} g_i(k,t) dt .$$

The expected number of visits in module i by task k, denoted by $a_i(k)$, can be obtained by solving

$$a_i(k) = q_i(k) + \sum_{j=1}^{M} a_j(k)\, p_{ji}(k) .$$

The probability that there will be no failure when running for task k is given by

$$R(k) = \prod_{i=1}^{M} [R_i(k)]^{a_i(k)}$$

and the system failure rate becomes $\lambda_s = \sum_{k=1}^{K} r_k \left[1 - R(k)\right]$, where r_k is the arrival rate of task k.

3.2.4 Gokhale et. al. Model [11]

The novelty of this work lies in the attempt to determine software architecture and component reliabilities experimentally by testing the application.

Architecture: The terminating application is described by an absorbing DTMC. The trace data produced by the coverage analysis tool called ATAC [36] during the testing is used to determine the architecture of application and compute the branching probabilities p_{ij} between modules. The expected time spent in a module j per visit, denoted by t_j ,is computed as a product of the expected execution time of each block and the number of blocks in the module.

Failure behavior: The failure behavior of each component is described by the enhanced non-homogeneous Poisson process model using time-dependent failure intensity $\lambda_j(t)$ determined by block coverage measurements during the testing of the application.

Solution method: The expected number of visits to state j, denoted by V_j is computed by

$$V_j = \pi_j(0) + \sum_{i=1}^{n} V_i p_{ij}$$

where $\pi(0)$ denotes the initial state probability vector.

The reliability of a module j, given time-dependent failure intensity $\lambda_j(t)$ and the total expected time spent in the module per execution $V_j t_j$, is given by

$$R_j = e^{-\int_0^{V_j t_j} \lambda_j(t)\,dt}$$

and the reliability of the overall application becomes $R = \prod_{j=1}^{n} R_j$.

3.3 Path – Based Models

This class of models is based on the same common steps as the state-based models, except that the approach taken to combine the software architecture with the failure behavior can be described as a path-based since the system reliability is computed considering the possible execution paths of the program either experimentally by testing or algorithmically.

3.3.1 Shooman Model [37]

This is one of the earliest models that consider reliability of modular programs, introducing the path-based approach by using the frequencies with which different paths are run.

Architecture: This model assumes the knowledge of the different paths and the frequencies f_i with which path i is run.

Failure behavior: The failure probability of the path i on each run, denoted by q_i characterizes the failure behavior.

Method of analysis: The total number of failures n_f in N test runs is given by $n_f = Nf_1q_1 + Nf_2q_2 + \ldots + Nf_iq_i$, where Nf_i is the total number of traversals of path i. The system probability of failure on any test run is given by

$$q_0 = \lim_{N \to \infty} \frac{n_f}{N} = \sum_{j=1}^{i} f_i q_j.$$

3.3.2 Krishnamurthy and Mathur Model [38]

This method first involves computing the path reliability estimates based on the sequence of components executed for each test run, and then averaging them over all test runs to obtain an estimate of the system reliability.

Architecture: Components and their interfaces are identified, and a sequence of components along different paths is observed using the component traces collected during the testing.

Failure behavior: Each component is characterized by its reliability R_m.

Method of analysis: The component trace of a program P for a given test case t, denoted by M (P, t), is the sequence of components m executed when P is executed against t. The reliability of a path in P traversed when P is executed on test case $t \in T$ is given by

$$R_t = \prod_{\forall_m \in M(P,t)} R_m$$

under the assumption that individual components along the path fail independently of each other. The reliability estimate of a program with respect to a test set T is

$$R = \frac{\sum_{\forall_{t \in T}} R_t}{|T|}.$$

An interesting case occurs when most paths executed have components within loops and these loops are traversed a sufficiently large number of times. Then if intra-component dependency is ignored individual path reliabilities are likely to become low, resulting in low system reliability estimates. In this work intra-component dependency is modeled by "collapsing" multiple occurrences of a component on an execution path into k occurrences, where $k>0$ is referred as the degree of independence. However, it is not clear how one should determine a suitable value of k.

An alternative way to resolve the issue of intra-component dependency is proposed in [39]. The solution of dependency characterization of a component that is invoked inside a loop m times with a fixed execution time spent in the component per visit relies on the time dependent failure intensity of a component.

3.3.3 Yacoub, Cukic and Ammar Model [58]

This reliability analysis technique is specific for component-based software whose analysis is strictly based on execution scenarios. A scenario is a set of component interactions triggered by specific input stimulus and it is related to the concept of operations and run-types used in operational profiles.

Architecture: Using scenarios, a probabilistic model named Component Dependency Graph (CDG) is constructed. A node n_i of CDG models a component execution with an average execution time EC_i. The transition probability PT_{ij} is associated with each directed edge that models the transition from node n_i to n_j. CDG has two additional nodes, start node and termination node.

Failure behavior: The failure process considers component reliabilities RC_i and transition reliabilities RT_{ij} associated with a node n_i and with a transition from node n_i to n_j respectively.

Method of analysis: Based on CDG a tree-traversal algorithm is presented to estimate the reliability of the application as a function of reliabilities of its components

and interfaces. The algorithm expands all branches of the CDG starting from the start node. The breadth expansions of the tree represent logical "OR" paths and are hence translated as the summation of reliabilities weighted by the transition probability along each path. The depth of each path represents the sequential execution of components, the logical "AND", and is hence translated to multiplication of reliabilities. The depth expansion of a path terminates when the next node is a terminating node (a natural end of an application execution) or when the summation of execution time of that thread sums to the average execution time of a scenario. The latest guaranties deadlock avoidance for loops between two or more components.

3.4 Additive models

This class of models does not consider explicitly the architecture of the software. Rather, they are focused on estimating the overall application reliability using the component's failure data. It should be noted that these models consider software reliability growth. The models are called additive since under the assumption that non-homogeneous Poisson process (NHPP) can model component's reliability, the system failure intensity can be expressed as the sum of component failure intensities.

3.4.1 Xie and Wholin Model [40]

This model considers a software system composed of n components, which may have been developed in parallel and tested independently. If the component reliabilities are modeled by NHPP with failure intensity $\lambda_i(t)$ then the system failure intensity is $\lambda_s(t)=\lambda_1(t)+\lambda_2(t)+...+\lambda_n(t)$, and the expected cumulative number of system failures by time t is given by

$$\mu_s(t)=\sum_{i=1}^{n}\mu_i(t)=\int_0^t\sum_{i=1}^{n}\lambda_i(\tau)d\tau .$$

When this additive model is used the most immediate problem is that the starting time may not be the same for all components, that is, some components may be introduced into the system later. In that case, the time has to be adjusted appropriately to consider different starting points for different components.

3.4.2 Everett Model [59]

This approach considers the software made out of components, and addresses the problem of estimating individual component's reliability. Reliability of each component is analyzed using the Extended Execution Time (EET) model whose parameters can be determined directly from properties of the software and from the information on how test cases and operational usage stresses each component. Thus, this approach requires keeping track of the cumulative amount of processing time spent in each component.

When the underlying EET models for the components are NHPP models, the cumulative number of failures and failure intensity functions for the superposition of such models is just the sum of the corresponding functions for each component.

3.5 Input Domain Models

In case of input-domain based models, the reliability of the software is measured by exercising the software with a set of randomly chosen inputs. The ratio of the number of inputs that resulted in successful execution to the total number of inputs gives an estimate of the reliability of the software product. Two important input domain reliability models Nelson model and Weiss & Weyuker model is described bellow.

3.5.1 Nelson Model [10]

It is one of the most widely used input domain reliability models. According to this model, if a total number of f errors are recorded (referred to as failures in software reliability engineering, denoting behavioral deviations) for n hits, the estimated reliability R is calculated as:

$$R = 1 - \frac{f}{n} = \frac{n-f}{n}.$$

When usage time t_i is available for each hit i, the summary reliability measure, mean-time-between-failures (MTBF), can be calculated as:

$$MTBF = \frac{1}{f}\sum_i t_i$$

when the usage time t_i is not available, we can use the numbers of hits as the rough time measure. In this case,

$$MTBF = \frac{n}{f}.$$

3.5.2 Weiss & Weyuker Model [60]

Weiss & Weyuker [60] have partitioned the input domain into some equivalence classes with respect to the behavior of the system under test. This approach mainly reduces the number of test cases with respect to the input domain. According to this model reliability of a program is dependent only on the a priori probability distribution of the operational input domain, on the properties of the program, and on the end user's notion of tolerable discrepancies between the actual and intended program behaviors. To this end, the definition incorporates

1. The operational distribution of the input space
2. The actual discrepancy between the functional behavior of the program and its specification, and
3. Parameterization by the *tolerance function*, which specifies the tolerable discrepancy between the functional behavior of the program and its specification at each possible input point.

In order to assess the reliability of a particular program P for a specification S, one must perform the following steps.

1) Determine what the operational environment will be.
2) Define approximation to the operational distribution by using existing data and/or a probabilistic analysis. Additionally, one may add an optional step, as follows.

2′) Define a partition of the input domain and assign operational probabilities to the cells of the partition in accordance with the estimated operational distribution.

3) Determine a metric on the output space, and document its definition with ample justification that it reasonably models the structure of that space.
4) Select a set of test cases for the purpose of reliability testing. In particular, errors will not be corrected as they are found. Some of the factors influencing the selection process include maximizing confidence and minimizing the cost of testing.
5) Determine a tolerance function for this set of test.
6) Determine a measure of confidence for the reliability estimate that will be obtained from this sample set.
7) Run the tests.

8) Calculate the reliability estimate. This requires determining the α-discrepancy at each test point.

9) Publish the reliability estimate along with the documentation of all of its parameters, confidence estimates, etc.

Step 4) requires selecting a test set T. If a test set $T = \{t_i\}$ is chosen for reasons other than its representativeness of the operational profile, such as for its value in exposing errors or exercising certain program structures, then the representativeness of the operational profile, such as for its value in exposing errors or exercising certain program structures, then the representativeness of each test case of T must be *explicitly* incorporated into the *estimate* of reliability obtained form T. to this end, at each $t \in T$ a weight $p_T(t)$ is assigned that imparts a degree of representativeness to t. formally, p_T is an arbitrary probability function such that $p_T(t) = 0$ if $t \notin T$.

Having established these weights and completed stes 5) - 7), the reliability estimate R is determined form the following formula.

$$R = 1 - \sum_{t \in T} \frac{p_T . d_\alpha (S_P, P, t)}{\alpha(t)}$$

Where $d_\alpha(S_P, P, t)$ is known as the α*-discrepancy* between S_P and P at t. The function α is known as *tolerance* function and $\alpha(t)$ is the *tolerance* allowed at t. For any specification S with domain D and "don't care" set U and any program P, S_P is defined by

$$S_P(n) = \begin{cases} S(n) & if \quad n \in D \\ P(n) & if \quad n \in U \end{cases}$$

A specification S of a program is supposed to prescribe the output of the program for each input. However, three cases can arise for a given input n:

1. No output is specified
2. Exactly one output is specified, or
3. More than one output is specified.

If case 1) occurs for a given input n, then that input is a "don't care".

3.6 Reliability Growth Models

Developing reliable software is one of the most difficult problems facing the software industry. Schedule pressure, resource limitation, and unrealistic requirements can all negatively impact software reliability. Developing reliable software is especially hard

when there is interdependence among the software modules as in the case with much of the existing software. It is also a hard problem to know whether or not the software being developed is reliable. After the software is shipped, its reliability is indicated by from customer feedback, problem reposts, system outages, complaints or compliments and so forth. However then it is too late; software vendors need to know whether their products are reliable before they are shipped to customers. Software reliability growth models attempt to provide that information.

As mentioned earlier reliability is usually defined as the probability that a system will operate without failure for a specified time period under specified operating conditions. Reliability is concerned with the time between failures or tits reciprocal, the failure rate. But software reliability growth models (SRGMs) repot on defect detection rate rather than failure rate. Defect detection is usually a failure during a test, but test software may also detect a defect even though the test continues to operate. Defects can also be detected during design reviews or code inspections, but SRGMs do not consider those sorts of activities. Time in a test environment is a synonym for amount of testing, which can be measured in several ways. Defect detection data consists of a time for each defect or group of defects and ca be plotted as shown in Fig. 3.1.

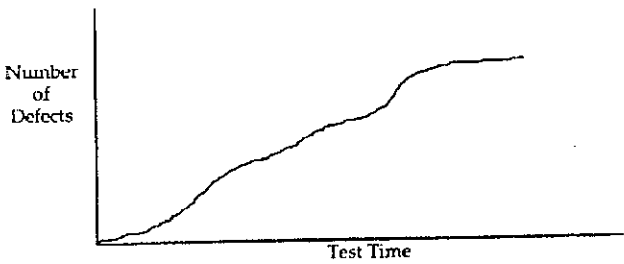


Fig. 3.1: Example defect detection data

A cumulative plot of defects vs. amount of testing such as Fig. 3.1 should show that the defect discovery rate decreases as the amount of testing increases. The theory is that each defect is fixed as it is discovered. This decreases the number of defects in the code, so the defect discovery rate should decrease (the length of time between defect discoveries should increase). When the defect discovery rate reaches an acceptably low value, the software is deemed suitable to ship. However, it is difficult to extrapolate from defect discovery rate in a test environment to failure rate during system operation, primarily because it is hard to extrapolate from test time to system operation time. Instead

SRGMs look at the expected quantity of remaining defects in the code. These residual defects provide an upper limit on the number of unique failures the customers could encounter in field use.

Software reliability growth models are statistical interpolation of defect detection data by mathematical functions. The functions are used to predict future failure rates or the number of residual defects in the code. There are many types of software reliability growth models as described in successive sections.

3.6.1 Software Reliability Growth Model Types

Software reliably growth models have been grouped into two classes of model – concave and S-shaped. These two model types are shown in Fig. 3.2. The most important thing about both models is that they have the same asymptotic behavior, i.e., the defect detection rate deceases as the number of defects detected (and repaired) increases, and the total number of defects detected asymptotically approaches a finite value. The theory for this asymptotic behavior is that:

1. A finite amount of code should have a finite number of defects. Repair and new functionality may introduce new defects, which increase the original finite number of defects. Some models explicitly account for new defect introduction during test while others assume they are negligible or handled by the statistical fit of the software reliability growth model to the data.
2. It is assumed that the defect detection rate is proportional to the number of defects in the code. Each time a defect is repaired, there are fewer total defects in the code, so the defect detection rate decreases as the number of defects detected (and repaired) increases. The concave model strictly follows this pattern. In the S-shaped model, it is assumed that the early testing is not as efficient as later testing, so there is a ramp-up period during which the defect detection rate increases. This could be a good assumption if the first QA tests are simply repeating tests that developers have already run or if early QA tests uncover defects in other products that prevent QA from finding defects in the product being tested. For example, an application test may uncover OS defects that need to be corrected before the application can be run. Application test hours are accumulated; but defect data is minimal because OS defects don't count as part of the application test data. After

the OS defects are corrected, the remainder of the application test data (after the inflection point in the S-shaped curve) looks like the concave model.

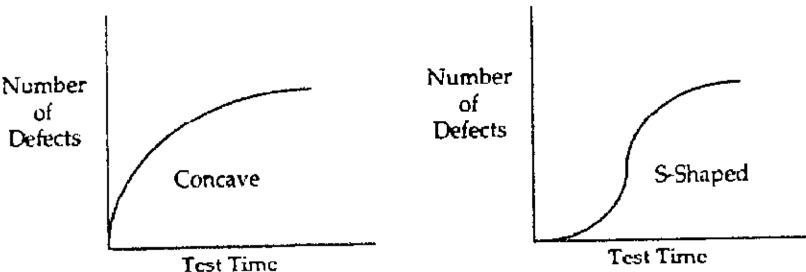


Fig. 3.2: Concave and S-Shaped Models

There are many different representations of software reliability growth models. The models show the expected number of defects at time t and is denoted $\mu(t)$, where t can be calendar time, execution time or number of test executed. An example equation for $\mu(t)$ is the Goel-Okumoto (G-O) model:

$\mu(t) = a(1-e^{-bt})$, where

a = expected total number of defects in the code

b = shape factor = the rate at which the failure rate decreases, i.e., the rate at which we approach the total number of defects.

Table 3.1: Software Reliability Growth Model Examples

Model Name	Model Type	$\mu(t)$	Ref.	Comments
Goel-Okumoto (G-O)	Concave	$a(1-e^{-bt})$ $a \geq 0, b > 0$	[7]	Also called Musa model or exponential model
G-O S-Shaped	S-Shaped	$a(1-(1+bt)e^{-bt})$ $a \geq 0, b > 0$	[61]	Modification of G-O model to make it S-shaped (Gamma function instead of exponential)
Hossain-Dahiya/ G-O	Concave	$a(1-e^{-bt})/(1+ce^{-bt})$ $a \geq 0, b > 0, c > 0$	[62]	Solves a technical condition with the G-O model. Becomes same as G-O as c approaches 0.
Gompertz	S-Shaped	$a(b^{c^t})$ $a \geq 0, 0 \leq b \leq 1, 0 < c < 1$	[63]	Used by Fujitsu, Numazu Works
Pareto	Concave	$a(1-(1+t/\beta)^{1-\alpha})$ $a \geq 0, \beta > 0, 0 \leq \alpha \leq 1$	[64]	Assumes failures have different failure rates and failures with highest rates removed first

Model Name	Model Type	$\mu(t)$	Ref.	Comments
Weibull	Concave	$a(1-e^{-bt^c})$ $a \geq 0, b > 0, c > 0$	[7]	Same as G-O for c = 1
Yamada Exponential	Concave	$a(1-e^{-r\alpha(1-e^{-\beta t})})$ $a \geq 0, r\alpha > 0, \beta > 0$	[65]	Attempts to account for testing effort
Yamada Raleigh	S-Shaped	$a(1-e^{-r\alpha(1-e^{(-\beta t^2/2)})})$ $a \geq 0, r\alpha > 0, \beta > 0$	[65]	Attempts to account for testing effort
Log Poisson	Infinite Failure	$(1/c)\ln(c\alpha t+1)$ $c > 0, \alpha > 0$	[7]	Failure rate decreases but does not approach to 0

The Goel-Okumoto (G-O) model is a concave model, and the parameter "a" would be plotted as the total number of defects in Fig. 3.2. The Goel-Okumoto model has 2 parameters. However other models can have 3 or more parameters. For most models, $\mu(t) = aF(t)$, where a is the expected total number of defects in the code and $F(t)$ is a cumulative distribution function. Note that $F(0) = 0$, so no defects are discovered before the test starts, and $F(\infty) = 1$, so $\mu(\infty) = a$ and a is the total number of defects discovered after an infinite amount of testing. Table 3.1 provides a list of the models. A derivation of the properties of most of these models can be found in [7].

The Log Poisson model is a different type of model. This model assumes that the code has an infinite number of failures. Although this is not theoretically true, it may be essentially true in practice since all the defects are never found before the code is rewritten, and the model may provide a good fit for the useful life of the product.

The models all make assumptions about testing and defect repair. Some of these assumptions seem very reasonable, but some are questionable. However, we give a list and discussion of these assumptions in the following Table 3.2.

Table 3.2: Software Reliability Model Assumptions

Assumption	Reality
Defects are repaired immediately when they are discovered	Defects are not repaired immediately, but this can be partially accommodated by not counting duplicates. Test time may be artificially accumulated if a non-repaired defect prevents other defects from being found.
Defect repair is perfect	Defect repair introduces new defects. The new defects are les likely to be discovered by test since the retest for the repaired code is not usually as comprehensive as the original testing.
No new code is	New code is frequently introduced throughout the entire test

Assumption	Reality
introduced during QA test	period, both defect repair and new features. This is accounted for in parameter estimation since actual defect discoveries are used, but may change the shape of the curve, i.e., make it less concave.
Defects are only reported by the product testing group	Defects are reported by lots of groups because of parallel testing activity. If we add the test time for those groups, we have the problem of equivalency between and hour of QA test time and an hour of test time from a group that is testing a different product. Restricting defects to those discovered by QA can accommodate this, but that eliminates important data. This problem means that defects do not correlate perfectly with test time.
Each unit of time (calendar, execution, number of test cases) is equivalent	This is certainly not true for calendar time or test cases as discussed earlier. For execution time, "comet" tests sometimes are more likely to find defects, so those tests create more stress on a per hour basis. When there is a section of code that has not been as thoroughly tested as other code, e.g., a product that is under schedule pressure, tests of that code will usually find more defects. Many test are rerun to ensure defect repair has been done properly, and these reruns should be less likely to find new defects. However, as long as test sequences are reasonably consistent from release to release, this can be accounted for if necessary from lessons learned on previous release.
Tests represent operational profile	Customers run so many different configurations and applications that it is difficult to define an appropriate operational profile. In some cases, the sheer size and transaction volume of the production system makes the operational environment impractical to replicate. The tests contained in the QA test library test basic functionality and operation, error recovery, and specific areas with which we have had problems in the past. Additional tests are continually being added, but the code also learns the old tests, i.e., the defects that the old tests would have uncovered have been repaired.
Failures are independent	Our experience is that this is reasonable except when there is a section of code that has not been as thoroughly tested as other code, e.g., a product behind schedule that was not thoroughly unit tested. Tests run against this section of code may find a disproportionate share of defects. In [12] there are detailed discussions on independence assumption.

Chapter 4

Statistical Testing

4.1 Statistical Testing

Harlan Mills (IBM Fellow who invented Clean Room software engineering) invented the concept of statistical testing in 1987 [41, 42]. The central idea is to use software testing as a means to assess the reliability of software as opposed to a debugging mechanism. This is quite contrary to the popular use of software testing as a debugging method. Therefore one needs to recognize that the goals and motivations of statistical testing are different fundamentally. There are many arguments as to why this might indeed be a very valid approach. The theory of this is buried in the concepts of Clean Room software engineering and is worthy of a separate discussion. Statistical testing needs to exercise the software along an operational profile and then measure interfailure times that are then used to estimate its reliability. A good development process should yield an increasing mean time between failu2res every time a bug is fixed. This then becomes the release criteria and the conditions to stop software testing.

Most *systematic testing* methods have been aroused from the idea of coverage [29, 43]. Some aspect of a program is considered as a potential source of failure, and the systematic method attempts to show that this aspect will not cause failure. For example, a statement could be wrong, and if it is never executed during testing, the fault remains unrevealed. Therefore we may want to measure line coverage during testing. Or similarly, we may want to make sure that every one of the functions of the systems is executed at least once.

However, every testing method (except exhaustive testing for *batch programs*) is less than perfect. Testing reveals a part of the software faults, yet some remain undetected. It has therefore been suggested [13, 43, 44, 45] that testing should take into account use patterns the software will encounter in its intended environment. The argument is that by testing according to use, the faults found by imperfect methods are more likely to be the important ones, the ones most users would encounter. In statistical prediction, the

argument that test should follow user patterns is vital. If this is not the case, then the tests are not a representative sample and all statistical conclusions are invalid.

Statistical testing, in contrast to other systematic testing method, makes no claims to cover anything. One might therefore expect that statistical testing can't compete with systematic testing in exposing faults. But, however, this has been proven wrong by several studies: under assumptions not unfavorable to systematic methods, they are not much better at finding faults than statistical testing [46, 47].

The black box approach [13, 29] to the software testing process unfolds as follows. Given a program P with intended function f and input domain d, the objective is to select a sequence of entries from d, apply them to P, and compare the response with the expected outcome indicated by f. Any deviation form the intended function is designated as failure. It is assumed that f is well defined and completely specified, so that any deviation is unambiguously detected and a failure is explicitly noted. The history of the test at some time n s a sequence of inputs $d_0 d_1 d_2 .. d_{n-1}$ and a corresponding sequence of zero or more failures, each of which is uniquely identified with the particular input d_i at which the failure was observed.

Statistical testing follows the black box model with two important extensions. First, sequences form d are stochastically generated based on probability distribution that represents a profile of actual or anticipated use of the software. Second, a statistical analysis is performed on the test history that enables the measurement of various probabilistic aspects of the testing process. Thus, one can view statistical testing as a sequence generation and analysis problem. A solution to the problem is achieved by constructing a generator to obtain the test input sequences and by developing an informative analysis of the test history.

4.2 Markov Chain Model for Statistical Software Testing

Statistical testing of software establishes a basis for statistical inferences about a software system's expected field quality. We describe a method for statistical testing based on a Markov chain model of software usage. The significance of the Markov chain is twofold. First, it allows test input sequences to be generated from multiple probability

distributions, making it more general than many existing techniques. Analytical results associated with Markov chains facilitate informative analysis of the sequence before they are generated, indicating how the test is likely to unfold. Second, the test input sequences generated from the chain and applied to the software are themselves a stochastic model and are used to create a second Markov chain to encapsulate the history of the test, including any observed failure information. The influence of the failures is assessed through analytical computations of the chain. We also derive a stopping criterion for the testing process based on a comparison of the sequence generating properties of the two chains.

Statistical testing process can be carried out in three major steps [13, 43].

Step 1: Construct the statistical models based on actual usage scenarios and related frequencies.

Step 2: Use these models for test case generation and execution.

Step 3: Analyze the test results for reliability assessment and predictions, and help with decision-making.

In Markov chain based statistical testing software usage behavior is modeled as a finite state, discrete parameter, time homogeneous Markov chains. It is known as *usage Markov chain* or in short *usage model* [13]. The usage model consists of elements from d, the domain of the intended function, and a probabilistic relationship defined on these elements. A test input is a finite sequence of inputs form domain d probabilistically generated form the usage model. The statistical properties of the model lend insight into the expected makeup of the sequences for test planning purposes.

As the test sequences are applied to the software, the results are incorporated into a second model. This *testing model* or the *testing Markov chain* [13] consists of the inputs executed in the test sequences, plus any failures discovered while applying the sequences to the software P. in other words, it is a model of what has occurred during testing. The testing model also allows analysis of the test data in terms of random variables appropriate for the application. For example, we may measure the evolution of the testing model and decide to stop testing when tit has reached some suitable "steady state".

4.2.1 The Usage Markov Chain

A usage chain for a software system consists of states, i.e., externally visible modes of operation that must be maintained in order to predict the application of all system inputs, and state transitions that are labeled with system inputs and transition probabilities. To determine the state set, one must consider each input and the information necessary to apply the input. It may be that certain software modes cause an input to become more or less probable (or even illegal). Such a mode represents a state or set of states in the usage chain. Once the states are identified, we establish a start state, a terminate state (for bookkeeping purposes), and draw a state transition diagram by considering the effect of each input from each of the identified states. The Markov chain is completely defined when transition probabilities are established that represent the best estimate of real usage.

Consider a simple selection menu pictured in Fig. 4.1. Though it is simple it has the salient features of database based application software. The input domain consists of up arrow key, down arrow key and enter key that select the items. The up arrow key and down arrow key moves the cursor from one item to next, and wraps from top to bottom on an up-arrow and from bottom to top on a down-arrow key.

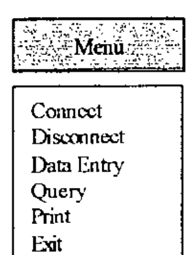


Fig. 4. 1: Selection Menu

The first item "Connect" is used to establish connection to a database server. The connect window has two options, Ok and Cancel. The Ok button establishes a connection to the specified server with proper authentication and the Cancel button returns to previous state. Once the connection is established the next four items, Disconnect, Data Entry, Query and Print can be selected to perform their respective functions. If connection is not established, selecting these 8items give no response. As Connect state, Disconnect state has also Ok and Cancel button to disconnect from database server or not. From the other three options we enter another screen only for Data Entry state for simplicity and

assume that the same thing could be done for other states. For data entry state we enter in a new screen, which could insert or update department record to database. The screen is pictured in Fig. 4.2.

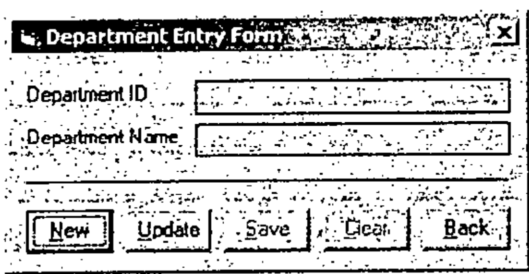


Fig. 4. 2: Department Entry Form First State

Initially New, Update and Back button are enabled and the other controls are disabled. Selecting data entry from menu displays this screen and the control focus goes on to "New" button. The tab key will shift the focus on the next enabled button, and will rotate right when the focus is on right-most button. If "New" button is pressed, "New", "Update" and "Back" button will be disabled and the disabled controls will be enabled. In that case the screen will look like Fig. 4.3. The same thing will happen if "Update" button is pressed. Now, if "Save" button is pressed, data provided in the text boxes will be updated to database. If "Clear" button is pressed the screen will go to its initial state i.e. "New" state and "Back" button returns to "Data Entry" state.

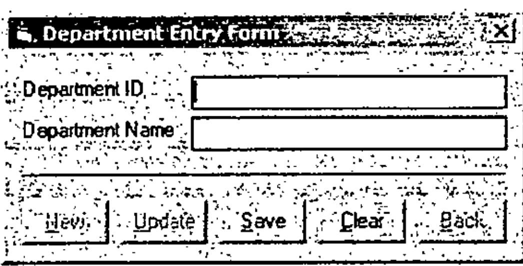


Fig. 4. 3: Department Entry Form Second State

In this example, there are two items of interest when applying menu inputs. First, the current cursor location must be maintained to determine the behavior of the "Enter" key. Second, whether connection to database is established or not to determine which of the menu items are available.

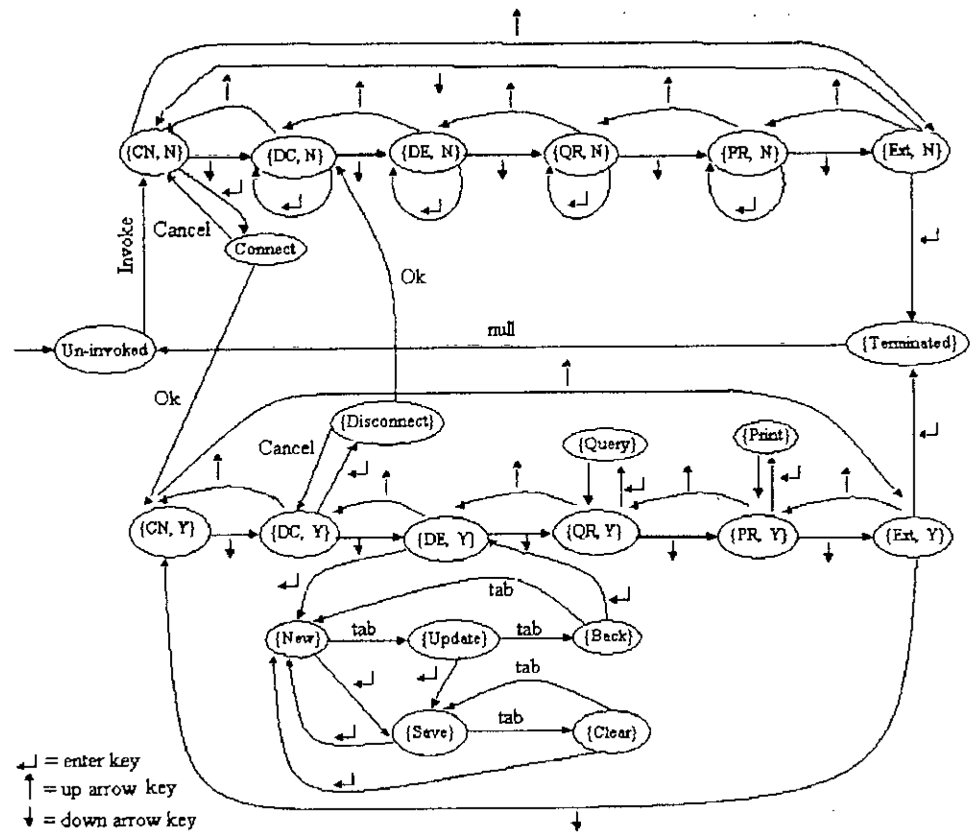


Fig. 4. 4: Usage Markov Chain for the Software

These two items of information are organized as the following usage variable:

1. *Cursor location* (which is abbreviated CL and takes on values CN, DC, DE, QR, PR or Exit for each respective menu item), and
2. *Connection status* (which is abbreviated CS and takes on the values Y or N).

The state set therefore consists of the following: {(CL = CN, CS = N), (CL = DC, CS = N), (CL = DE, CS = N), (CL = QR, CS = N), (CL = PR, CS = N), (CL = Ext, CS = N), (CL = CN, CS = Y), (CL = DC, CS = Y), (CL = DE, CS = Y), (CL = QR, CS = Y), (CL = PR, CS = Y), (CL = Ext, CS = N)}. In addition, we include states that represent placeholders for other system screens, as well as start and end states that represent the software in its "not invoked" mode. The state transitions are depicted in Fig. 4.4 in a graphical format.

Table – 4.1: Transition Probabilities for the Example Usage Chain

SL#	From state	Trans. Stimuli	To state	Est. Prob.
1	Un-Invoked	*Invoke*	{CL=CN, CS=No}	1.00
2	{CL=CN, CS=No}	↓	{CL=DC, CS=No}	0.10
		↑	{CL=Ext, CS=No}	0.10
		↵	{Connect}	0.80
3	{CL=DC, CS=No}	↓	{CL=DE, CS=No}	0.33
		↑	{CL=CN, CS=No}	0.34
		↵	{CL=DC, CS=No}	0.33
4	{CL=DE, CS=No}	↓	{CL=QR, CS=No}	0.33
		↑	{CL=DC, CS=No}	0.34
		↵	{CL=DE, CS=No}	0.33
5	{CL=QR, CS=No}	↓	{CL= R, CS=No}	0.33
		↑	{CL= E, CS=No}	0.34
		↵	{CL= R, CS=No}	0.33
6	{CL=PR, CS=No}	↓	{CL=Ext, CS=No}	0.34
		↑	{CL=QR, CS=No}	0.33
		↵	{CL=PR, CS=No}	0.33
7	{CL=Ext, CS=No}	↓	{CL=CN, CS=No}	0.34
		↑	{CL=PR, CS=No}	0.33
		↵	{Terminated}	0.33
8	{CL=CN, CS=Yes}	↓	{CL=DC, CS=Yes}	0.50
		↑	{CL=Ext, CS=Yes}	0.35
		↵	{CL=CN, CS=Yes}	0.15
9	{CL=DC, CS=Yes}	↓	{CL=DE, CS=Yes}	0.70
		↑	{CL=CN, CS=Yes}	0.15
		↵	{Disconnect}	0.15
10	{CL=DE, CS=Yes}	↓	{CL=QR, CS=Yes}	0.25
		↑	{CL=DC, CS=Yes}	0.25
		↵	{Data Entry New}	0.50
11	{CL=QR, CS=Yes}	↓	{CL=PR, CS=Yes}	0.25
		↑	{CL=DE, CS=Yes}	0.25
		↵	{Query}	0.50
12	{CL=PR, CS=Yes}	↓	{CL=Ext, CS=Yes}	0.25
		↑	{CL=QR, CS=Yes}	0.25
		↵	{Print}	0.50
13	{CL=Ext, CS=Yes}	↓	{CL=CN, CS=Yes}	0.15
		↑	{CL=PR, CS=Yes}	0.35
		↵	{Terminated}	0.50
14	{Connect}	Ok	{CL=CN, CS=Yes}	0.85
		Cancel	{CL=CN, CS=No}	0.15
15	{Disconnect}	Ok	{CL=DC, CS=No}	0.60
		Cancel	{CL=DC, CS=Yes}	0.40
16	{Data Entry	Tab	{Update}	0.40

SL#	From state	Trans. Stimuli	To state	Est. Prob.
	New}	↵	{Save}	0.60
17	{Update}	Tab	{Back}	0.50
		↵	{Save}	0.50
18	{Back}	Tab	{Data Entry New}	0.20
		↵	{CL=DE, CS=Yes}	0.80
19	{Save}	Tab	{Clear}	0.20
		↵	{Data Entry New}	0.80
20	{Clear}	Tab	{Save}	0.50
		↵	{Data Entry New}	0.50
21	{Query}	Query Data	{CL=QR, CS=Yes}	1.00
22	{Print}	Print Data	{CL=PR, CS=Yes}	1.00
23	{Terminated}	Null	Un-Invoked	1.00

A path from the initial "Un-invoked" state to the final "Terminated" state represents a single execution of the software. In order to generate sequence statistically, probability distributions are established over the exit arcs at each state that simulates expected field usage. Several methods can be employed to extract this information, including subjective evaluation based on expert opinions, survey of target customers, and measurement of actual usage patterns. We assigned the probabilities by expert opinion and Table-I lists each transition for the example chain in Fig. 4.4.

Table – 4.2: Some Standard Analytical Results for Markov Chains

Results	Equation for Prob. or Mean	Interpretation of Mean
Recurrent chain		
Stationary distribution, π	$\pi_j = \sum_i \pi_i U_{ij}$ (1)	π_j is the asymptotic appearance rate of state j in a large number of sequences from U.
Recurrent time for state j	$m_{jj} = \frac{1}{\pi_j}$ (2)	The mean number of state transitions between occurrences of state j in a large number of sequences from U.
Number of occurrences of state i between occurrences of state j	$m_{jj}\pi_i = \frac{\pi_i}{\pi_j}$ (3)	The mean number of occurrences of state i between occurrences of state j.

First passage time	$m_{ij} = 1 + \sum_{k \neq j} U_{ik} m_{kj}$ (4)	The mean number of state transitions until state j occurs from state i.
Absorbing chain (for initial state j)		
Single sequence prob. for state j	$y_{ij} = U^a_{ij} + \sum_{k \neq t} U^a_{ik} y_{kj}$ (5)	The probability that state j occurs in a single sequence (i.e., from the initial state to the absorbing state).
Number of sequences to occurrence of state j	$h_j = \frac{1}{y_{ij}}$ (6)	The mean number of states until state j occurs.
Single sequence prob. for arc j, k	$z_{jk} = y_{ij} U_{jk}$ (7)	The probability that arc j, k occurs in a single sequence (i.e., from the initial state to the absorbing state).
Number of sequences to occurrence of arc j, k	$h_{jk} = \frac{1}{z_{jk}}$ (8)	The mean number of states until arc j, k occurs.
Number of occurrences of state j in a single sequence	$m(j \mid i) = \sum U^a_{ik} m(j \mid k) + \begin{cases} 1 & \text{if } i=j \\ 0 & \text{if } i \neq j \end{cases}$ (9)	The mean number of occurrences of state j in a single sequence.

4.2.2 Analysis of the Usage Chain

The fact the usage model is a Markov chain allows software testers to perform significant analysis that gives insight how the test is likely to unfold. The details of the underlying mathematics can be found in [48]; however, we have included Table II to summarize some useful results. This analysis is used to gain insight into how the test will likely unfold so that testers can proceed in an informed manner. The insight gained through the analysis can be used to aid test planning and preparation.

4.2.3 Constructing the Testing Chain

Usage chain U has stationary transition probabilities; i.e., they do not change throughout the test. However, probabilities in testing chain T are updated, and tracking T's evolution is an inherent part of monitoring the statistical testing process. Let $s_1, s_2, \ldots, s_m$ denote the set of test sequences in the order generated by U and applied to

software P. The corresponding series of testing chains $T_0, T_1, ..., T_m$ describes the evolution of T during testing and is constructed as follows.

Before any sequence is input to P, the test history is empty. The initial chain T_0 is a copy of the usage chain U, with all arc probabilities set to 0. Assume first that no software failures occur. T_1 is obtained from T_0 by incrementing arc frequencies along the path of states from "*Un-invoked*" to "*Terminated*" in s_1. Similarly, T_2 is obtained from T_1 by sequence s_2, and, in general, T_i is obtained from T_{i-1} by s_i. In the way, frequency counts on arcs in T_i are always obtained from specific sequences applied to software P. These arc frequencies are converted to relative frequency probabilities whenever computation with T_i's state transition probabilities is required.

The testing chin's arc counts are reset when fixes are applied to P. Thus, as the software changes, a new testing chain is created to model only the sequences applied on that version. In this manner, the testing chain remains an accurate model of the testing experience of the current software version. An additional formulation is to maintain a testing chain that is not rest between fixes and incorporates testing experience across different software versions. This latter testing chain is really a model of the *process* of error discovery and fault removal, whereas the former series of chains represents each successive version of the software product. Either interpretation can provide valuable feedback about software development activity.

What can be said about the series $T_0, T_1, ..., T_m$? If no failures are detected, the evolution of T is dictated solely by sequences from U. The Strong Law of large numbers for Markov chains [55] guarantees (with probability 1) that these sequences $s_1, s_2, ..., s_m$ will become statistically typical of U when enough are generated. This means that convergence of T to U is certain, because the relative frequencies on T's arcs will converge to the probabilities on U's arcs. A key point is that the test history T is statistically typical of the usage chain U if and only if convergence is achieved.

In other words, U is a fixed reference toward which T_i evolves at an expected rate with statistical variation that depends on factors such as the source entropy of U [14]. This evolution is well controlled and predictable in statistical terms.

4.2.4 Incorporating Failure Data

Suppose now that failures do occur and that the jth failure f_j is detected during input of sequence s_i to P. To incorporate this failure event into the test history, a new state labeled f_j is placed in Markov chain T_i exactly as it was ordered in s_i. The arcs to and from the new state f_j have frequency count 1. If f_j is catastrophic failure, then the run of software P is aborted, and the arc form f_j goes to "Terminated"; otherwise, the test sequence can continue, and the arc from f_j goes to the next state in s_i. In this way, T_i is maintained as a Markov chain that incorporates both the underlying structure of the source of test sequences, U, and the frequency count history of sequences-plus-failures as testing evolves.

Convergence of T to U is adversely affected by failures of software P during testing. To achieve convergence when failures have been observed, the relative frequency probabilities on arcs to failure states in T_i must approach 0. In this way, the probabilities on the non-failure arcs are still forced to converge to the corresponding (nonzero) values in U. If even one failure occurs, this can be accomplished only when P responds to more test sequences without exhibiting failures. Thus, failures automatically impose additional testing to overcome their adverse impact on the convergence of T to U.

The testing chain, T, is a model of the current test history and is useful for computing properties of descriptive random variables as shown in the next section. An alternative would be to obtain statistics directly from the set of sequences executed; however, T incorporates explicitly the structure of the usage chain, which is only implicit in the sequences. In other words, each sequence is accorded different status according its specific attributes; e.g., sequences can vary in length and probability and thus contribute a different amount of information into the statistical testing experiment. The testing chain incorporates each event of each sequence, recognizing the probabilistic relationship between states and arcs established in the usage chain. Any computation based on T incorporates this information as well. Thus, T is an important model for the identification and derivation of measures that describe the statistical testing process. See [14] for proofs concerning specific attributes of testing chains.

4.2.5 Analytical Results for the Testing Chain

In this section, the testing chain, T, is used to obtain analytical results to answer two questions. First, at what point does the test history become representative of usage (as defined by U); second, how does each failure impact the testing process?

4.2.5.1 An analytical stopping criterion

Stopping criteria for statistical software testing can be as simple as choosing some target reliability [49, 50, 51, 52], and testing until the estimate of the reliability meets or exceeds the target. However, the estimate of the reliability meets or exceeds the target. However, the usage-to-testing-chain approach suggests an analytic stopping criterion based directly on the statistical properties of the usage and testing chains. The usage chain is a model of ideal testing of the software; i.e., each arc probability is established with the best estimate of actual usage, and no failure states are present. The testing chain, on the other hand, is a model of a specific test history, including failure data. Thus, the usage chain represents what would occur in the statistical test in the absence of failures, and the testing can represents what has occurred. Dissimilarity between the two models is therefore a useful measure of the testing process. When the dissimilarity is small, the test history is an accurate picture of the usage model.

The *log likelihood ratio* [53, 54] known as *discriminant* is used to measure how two stochastic processes relate to each other. If two stochastic processes tend to converge each other the numerical value of discriminant tends to zero and if both are same than the value is zero. This value is computed for two arbitrary ergodic stochastic processes λ_0 and λ_1 [52] as follows:

$$D(\lambda_0,\lambda_1)=\lim_{n\to\infty}\frac{1}{n}[\log_2 p(d_0d_1...d_{n-1}\mid\lambda_0)-\log_2 p(d_0d_1...d_{n-1}\mid\lambda_1)] \qquad (1)$$

where $p(d...|\lambda)$ denotes the probability with which stochastic process λ generates sequence d. Although $D(\lambda_0,\lambda_1)$ cannot be directly computed for arbitrary process λ_0 and λ_1, it can be computed for Markov chains U and T [14] as follows:

$$D(U,T)=\sum_{ij}\pi_i p_{ij}\log_2\frac{p_{ij}}{\bar{p}_{ij}} \qquad (2)$$

where π is the stationary distribution of U, p_{ij} is the probability of a transition from i to j in U, and $\hat{p}_{ij}$ is the corresponding probability in T. Each $\hat{p}_{ij}$ that corresponds to a nonzero p_{ij} must be greater than zero in order for $D(U,T)$ to be defined. $D(U,T)$ is non-negative and equal to zero if and only if $p_{ij} = \hat{p}_{ij}$ for all i, j [53].

To monitor the testing process, *D (U, T)* can be computed with each sequence applied to the software after *T* becomes fully defined. A downward trend in the values of *D (U, T)* signifies growing similarity of the two models. Usage chain *U* never changes; however, *D (U, T)* reflects the impact of each additional sequence on the stochastic characteristics of the testing chain. D (U, T), for example, can rise when no failures are observed if a sequence reinforces some low-probability event. Of course, a rise is expected when a failure occurs. When the discrimination drops below some predefined threshold and experiences little change for an extended period, it is implied that additional test sequences will not significantly impact the statistics of the testing model, and testing can stop.

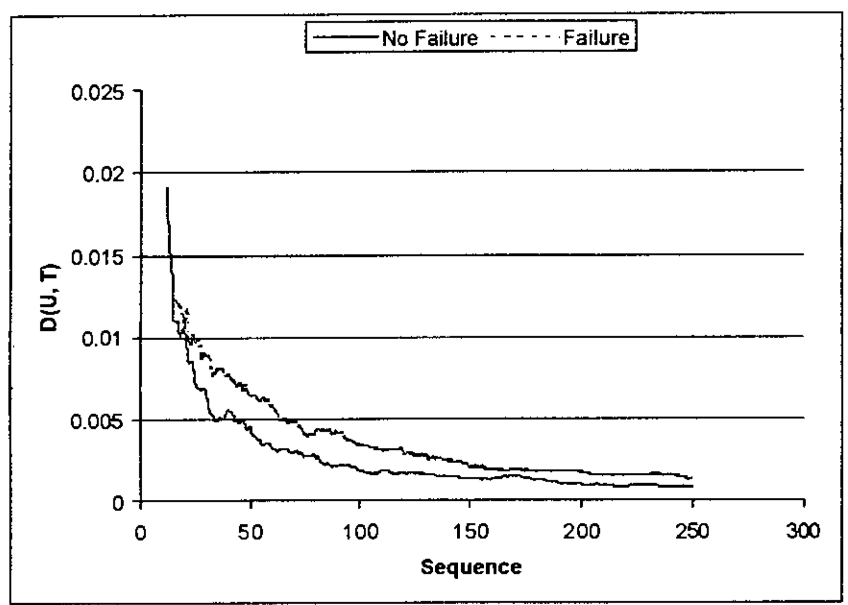


Fig. 4.5: Plot of *D (U, T)*

We have written a program for our example software and calculate the discriminant according to equation (2) and plot it in Fig. 4.5. The figure shows two plots of *D (U, T)*. The solid line depicts behavior of *D (U, T)* with no failures and the dotted line depicts a

sequence with four failures. When testing chain grows quite similar to usage chain i.e. the test history reflects the actual usage pattern, the value of *D (U, T)* becomes very small. Test should stop at this point. Whenever a failure occurs the value of *D (U, T)* increases significantly so additional tests require minimizing that effect [13]. It is important to stress that analysis of *D (U, T)* should involve trends in the values of the function over time rather than any single value at some specific point in time.

4.2.5.2 Measuring reliability, mean time to failure and the impact of failure

We compute two characteristics of the testing chain that give insight into the effect of the failures. The first is the probability of a failure free realization of the testing chain, denoted R, computed by using a standard result from Markov chain theory. The second is the expected number of steps between failure states, denoted M, which requires a new computation.

R and M can be computed directly from the testing chain T at any time during the testing of software P, even when only a single sequence has been input to P. It must be emphasized that R is a probability and M is an expected value conditioned on the test history encoded as T. these values gain credibility as statistical measures as the discrimination D *(U, T)* becomes relatively small, for this indicates that T is becoming statistically typical of software P's response to the input sequences form usage chain U.

The probability, R, of a failure-free realization of the testing chain is the probability that a realization of T beginning with "*Un-invoked*" and ending with the first occurrence of "*Terminated*" will not contain a failure state. To compute R, each failure state and "*Terminated*" are made absorbing states. R is the probability that absorption occurs at "*Terminated*", given "*Un-invoked*" as the start state [13, 48]; namely, as follows:

$$R_{Un-inTerm} = \widehat{p}_{Un-inTerm} + \sum_{j \in \tau} \widehat{p}_{Un-in\,j} R_{jTerm} \qquad (3)$$

where τ is the set of transient (non-absorbing) states.

Fig. 4.6 depicts a plot of R for 250 sequences. Failures on high probability paths will cause a sharper decrease in R, because the failures are probability-weighted according to their location in chain [13]. Note that $R = 1$ when no failure states exist in T. because it is a conditional probability, R gains credibility as D *(U, T)* gets small. From that plot after

250 sequences of test if the value of R is 0.976 than we can say that a randomly selected test sequence has 97.6% chance to execute successfully without failure.

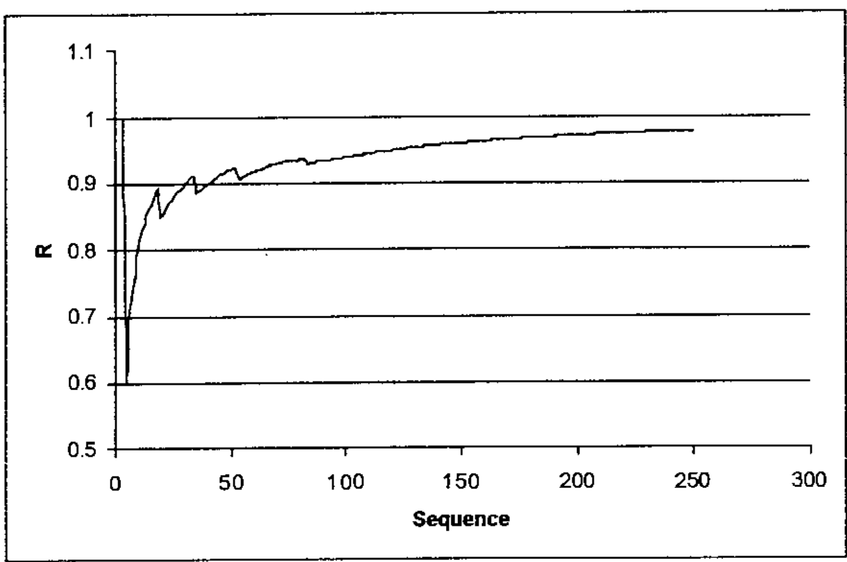


Fig. 4. 6: Test Sequence versus Reliability

The expected number of steps between failures is the expected number of state transitions encountered between occurrences of failure states in the testing chain. This value is computed [13, 14] as follows:

$$M = \sum_{i \in f_1,\ldots,f_m} v_i \left(\sum_{j \in u_1,\ldots,u_n} \hat{p}_{ij}(m_j + 1) \right) \qquad (4)$$

where v_i is the conditional long-run probability for failure state f_i, given that the process is in a failure state, m_j is the mean number of steps until the first occurrence of any failure state from j, $u_1,\ldots,u_n$ is the set of usage chain states, and $f_1,\ldots,f_m$ is the set of failure states. Fig. 4.7 is a plot of M for 250 sequences generated from our example software.

Also some additional information can be found from the usage and testing Markov chain. For example as the test process advances we can compute stationary probability of each state of the testing Markov chain which shows the amount of time spent in any state

200892

in the long run. We compute the stationary probabilities and are shown in the following graph.

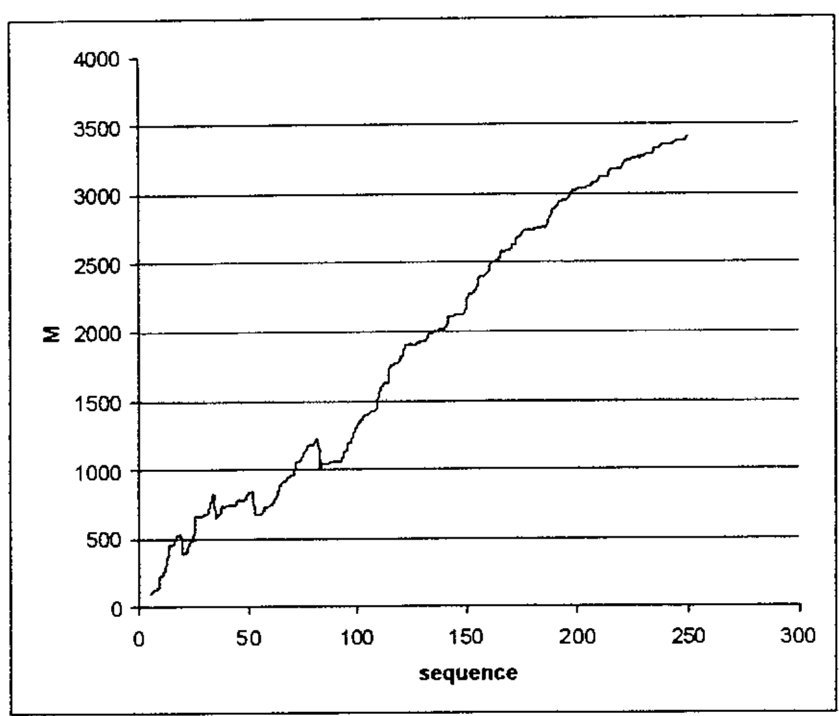


Fig. 4. 7: Expected Number of Steps Between Failures

We describe a sequence generation and analysis technique for statistical testing using Markov chains. We discuss the construction of a Markov chain as a sequence generator for statistical testing and show how analytical results associated with Markov chains can aid in test planning. An important aspect of this method is that the test sequences generated and applied to the software are used to create a second Markov chain to encapsulate the history of the test, including any observed failure information. The influence of the failures is assessed through analytical computations on this chain. We also find a stopping point for the testing process based on a comparison of the sequence generating properties of the two chains.

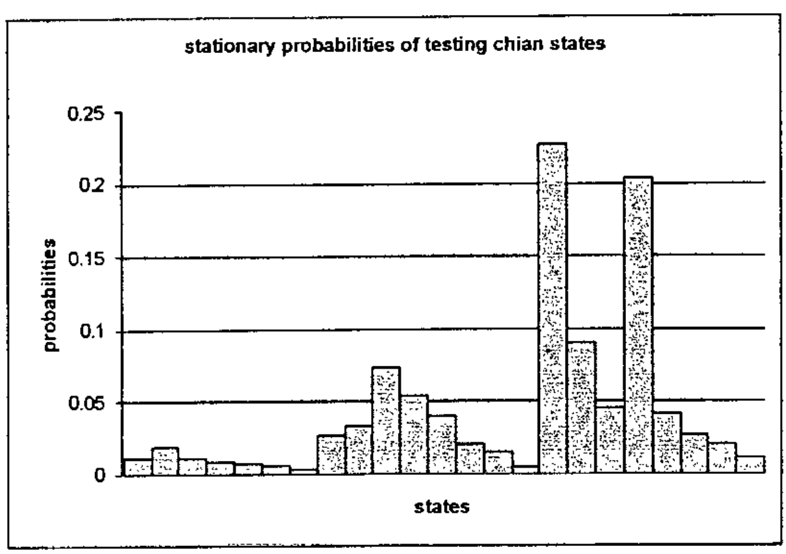


Fig. 4. 8: Stationary Probabilities of States of the Testing Chain

4.3 Effectiveness of Statistical Software Testing

One of the major shortfalls of statistical testing is the lack of evidence of the effectiveness of statistical testing compared to other methodologies, such as structural testing [66], random testing etc. Here we show the effectiveness of statistical testing over random testing. We assign equal probabilities to each exiting arcs from a state, generate test cases that represent random testing and measure reliability to compare the test processes.

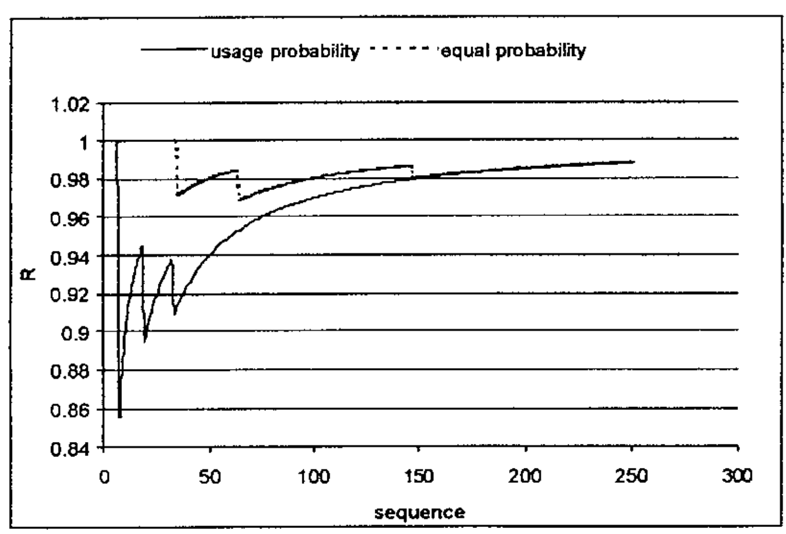


Fig. 4.9: Statistical Testing vs Random Testing (fault lies on high probability path)

From Fig. 4.9 we find that if the fault lies on the path of heavy usage probability than it reveals early in statistical testing while the fault reveals lately in random testing. If we set the target reliability to 0.98 we see from Fig. 4.9 that random testing may not reveal one bug. But if the fault lies on the less usage probability path than random testing reveals the fault early than statistical testing but this does not jeopardize our test effort as the same number of bugs are revealed by statistical testing before attaining the desired reliability and this is shown in Fig. 4.10.

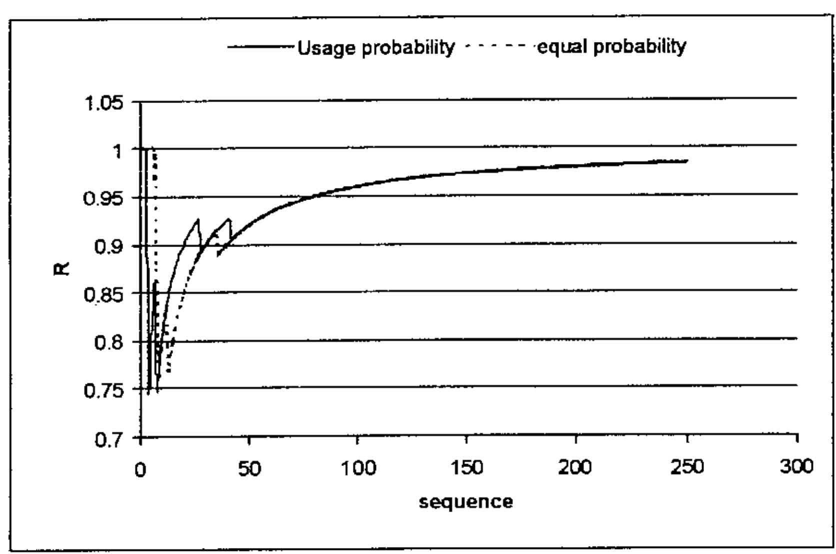


Fig. 4.10: Statistical Testing vs Random Testing (fault lies on low probability path)

Chapter 5

Arc-based Reliability

5.1 Introduction

Markov chain usage model that we have used to measure reliability in previous chapter has several benefits. It allows generating test sequences from usage probability distributions, assessing statistical inferences based on analytical results associated with Markov chains and also to derive stopping criterion of the test process. But the main problem in this process is to model software behavior in a single Markov chain. For large software systems the model size i.e. the number of sates become unwieldy and it becomes infeasible to apply this method in generating test cases as well as measuring reliability. Our main goal is to measure reliability of the entire software system. But a software system evolves over time. Whenever a change is made or new functionalities are added we have to go through the same procedure repeatedly to measure reliability. This is also a drawback of the previous technique. So we find an alternative approach to measure software reliability by combining the ideas drawn from partition testing, statistical testing using Markov chain usage model and component based software testing. Again we have taken example from database based application software, find its partitions or modules, in this case individual forms, separately so that it improves sampling efficiency [20, 26] and measure reliability of each forms according to Markov chain usage model. We assume that the forms are independent. When the individual forms reliabilities are measured we can calculate the entire software system's reliability using the usage probabilities of the forms.

5.2 The Miller Reliability Model

The Miller reliability model [27] can be used in conjunction with usage models to define software reliability estimators. The Miller model is based on Bayesian statistics and allows the user of the model to take advantage of prior knowledge of the system under test [27]. The Miller model assumes that the possible failure rates of the software have a standard beta distribution [56]. In the Miller model the expected value of the

reliability R of the system under test is $E(R)=1-\left(\frac{f+a}{f+s+a+b}\right)$, where s is the number of successful tests run, f is the number of failures, and a and b are parameters representing prior information about the software failure rate. For the case of no prior information about the software failure rate $a=b=1$.

The variance of R is.

$$Var(R)=\frac{(f+a)(s+b)}{(f+s+a+b)^2(f+s+a+b+1)}$$

While the Miller model can be used to calculate the expected value of the overall reliability for the system, it can also be used when the testing domain is partitioned into equivalence classes (also called blocks or bins). Reliability can then be calculated for each block of the partition as well as for the entire system [26].

5.3 Single-Use Reliability and Single-Action Reliability

The Whittaker model [14] estimates software reliability in terms of test cases as "uses", where a use is an executed sequence of actions from *(Un-Invoke)* to *(Terminate)*. For example, a use of word processing software could be to invoke the software, load a document, print the document, and then exit the software. This view of software reliability, the *single-use reliability*, defines the reliability as the probability of the software executing a randomly selected use without a failure. A use is considered to have a failure if at least one failure occurs during the execution of that use. While preserving this definition of reliability, an alternative approach to its estimation is given that does not necessarily yield $R=1.0$ when random testing reveals no failures. This definition is needed because Cleanroom development and testing often leads to testing results where no failures are seen in random testing. Current reliability estimators in use today, such as the Whittaker model and the sampling theory based model [57], do not provide a meaningful variance in the absence of failures. Therefore it is impossible to define a confidence interval around the estimated reliability, which in turn means it is impossible to assess the trustworthiness of the estimated reliability.

The *single-action reliability* [26] is introduced to provide an estimate of the probability that a single user action, a single state transition in the usage model, will occur

without failure. For example, a single user action involving a word processor might be loading a file.

Field experience shows many testing situations in which pre-test information is known or asserted in terms of individual arcs of the usage model. Both of these reliability estimates make use of this type of information.

Note that it is possible for the single-use reliability and the single-action reliability of a system to be quite different. This is because the single-use reliability depends strongly on the length of a typical use of the software. The longer the typical use of the software, the more chances the software has to fail. Therefore it is possible for software with a high single-action reliability to have relatively low single-use reliability. For example, consider a model where the probability of each individual user action succeeding is 0.99 and every use is 100 steps long (however rare such a model might be). The single-action reliability would be 0.99 since any given step has 0.99 reliability. However, for a use to succeed every user action must succeed; therefore the single-use reliability is $(0.99)^{100} = 0.366$, a much lower value than the single-action reliability.

5.3.1 Testing Records

Five matrices are needed to compute the single-action: the usage model transition matrix **U**, a success matrix **S**, a failure matrix **F**, and matrices of parameters of prior information, **A** and **B**. The transition matrix contains the arc transition probabilities of the usage model. It is created when the model is created.

A success matrix contains the counts of the number of times that each transition has been taken successfully during testing. Note that the success matrix does not contain information about failure states and that normalization of the success matrix would yield the *testing chain* only in the case of no failures.

The record of failures is maintained separately from the record of successes. The failure matrix **F** contains the counts of the number of times that a transition has failed during testing. If a failure is encountered while executing a test case that does not permit testing to continue (a halting failure), then no transitions beyond the failure will be counted in the testing record in either the success matrix or the failure matrix. Since those transitions were not executed during the test, they cannot be counted as successes or failures. Testing will continue with the sequence if possible.

The general process followed in testing using this reliability measure is as follows:

1. Generate sequences from the usage model.
2. Run the sequences until *(Terminate)* unless there is a halting failure.
3. Update the count of successful transitions in S.
4. Update the count of failed transitions in F.
5. Using the Miller failure rate calculation, estimate the failure rate of each arc in the model. The arc failure rate is defined to be zero if the state transition probability is zero.
6. Estimate the single-action reliability, E (R_a).

5.3.2. Arc Failure Rate Calculation

Following Miller, the expected values and variances of the arc failure rate random variables are computed using the Beta distribution [26].

$$E(F_{(i,j)}) = \frac{f_{i,j} + a_{i,j}}{f_{i,j} + s_{i,j} + a_{i,j} + b_{i,j}}$$

$$Var(F_{(i,j)}) = \frac{(f_{i,j} + a_{i,j})(f_{i,j} + s_{i,j} + b_{i,j})}{(f_{i,j} + s_{i,j} + a_{i,j} + b_{i,j})^2 (f_{i,j} + s_{i,j} + a_{i,j} + b_{i,j} + 1)}$$

5.3.3 Single Action Reliability Estimator

The single-action failure rate can be viewed as the probability of failure of a randomly selected transition from the convergent sequence. In terms of the Miller model, the probability associated with each state is the long run probability of the arc that defines that state. The long run arc probabilities of U are defined by $\pi(i, j) = \pi(i)u_{i,j}$.

Theorem: For each arc in a usage model U; the long run arc probability is equal to the probability of selecting the state under **R** identified with the arc, i.e.

$$p_{i,j} = \pi(i)u_{i,j}.$$

Proof: Because the Markov chain representing the usage model is ergodic, in the convergent sequence the probability of selecting an arbitrary arc approaches that arc's long run probability [48]. Selecting an arc at random and taking the sequence beginning with the most recent *(Invoke)* is equivalent to selecting the state. Therefore, the long run

arc probability is equal to the probability of selecting the state under **R** identified with the arc.

Theorem: The expected value of the single-action reliability is $E(R_a) = 1 - E(F_a)$, where $E(F_a) = \sum_i \left(\sum_j p_{i,j} E(F_{i,j}) \right)$ is the expected value of the single-action failure rate.

Proof: The probability associated with block (i, j) is $p_{i,j}$. The failure rate associated with block (i, j) is $F_{(i,j)}$. The expected value of the single-action failure rate follows by the Miller model.

5.3.4. Miller Model

Because the single-action reliability is the sum of random variables, by the central limit theorem the random variable representing the single-action reliability has an approximately normal distribution. Therefore, given the expected value and variance of the single-action reliability it is possible to compute a c% confidence interval for the single action reliability.

5.4 Single Action Reliability

From our example software we have taken a complex form named "Search Books". It is a part of Library Management Software. Through this form a borrower can search books according to Author, Title, Call Number, Accession Number and Keywords, and can reserve books for a specified time period. Fig. 5.1 shows the window of "Search Books".

Borrower selects search type from the combo box, inserts string in the text box and clicks on `"Search"` button to get the result. User can also log in to the system using his/her borrower id and password. If the process succeeds the `"Change Password"`, `"Add"`, `"Remove"` and `"Reserve"` buttons, and the `"Issued Books"`, `"Reserved Books"` radio buttons are enabled. The form now looks like in Fig. 5.2.

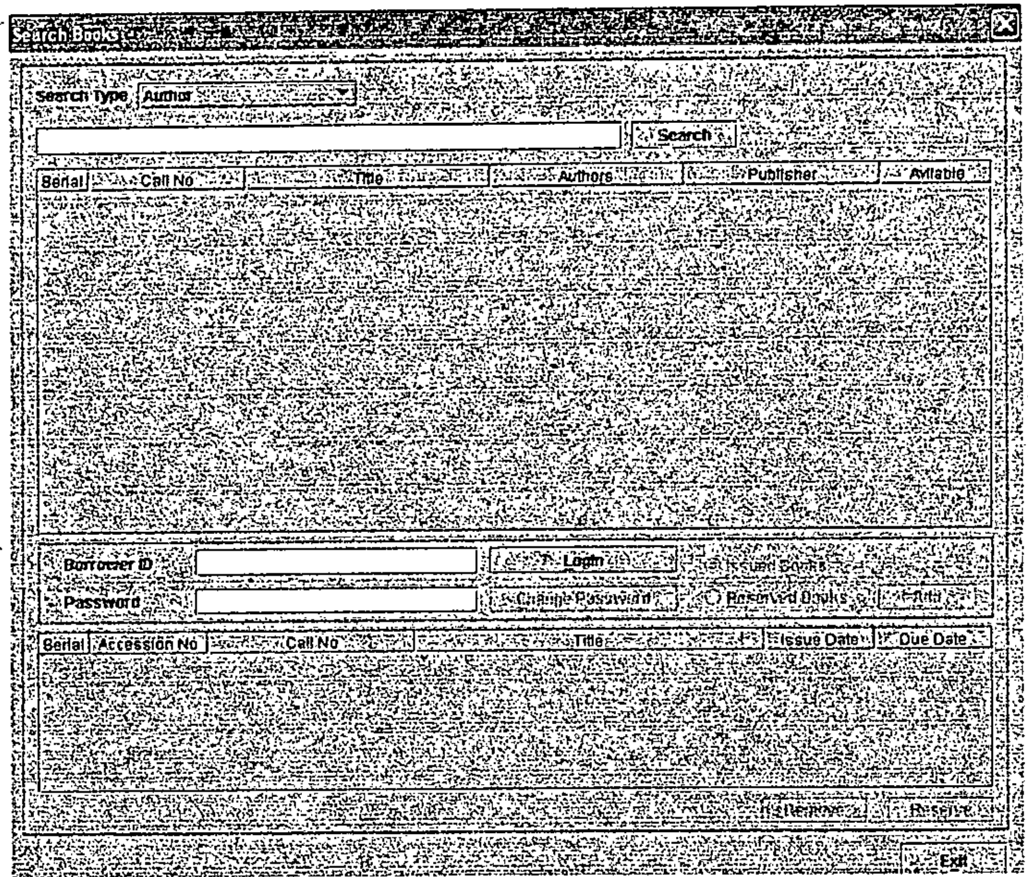


Fig. 5.1: Search Books Form 1

If borrower clicks on "Issued Books" radio button then the system displays the books information that the borrower borrowed from library. If he wants to reserve books then he selects "Reserved Books" radio button, selects a book from the searched book list and clicks on "Add" button to add the book in the second list as in Fug. 5.2. The status is set to 'pending'. He can remove a book from the list by clicking on "Remove" button. When the list is complete borrower reserves the books using the "Reserve" button and the status is set to 'reserved'. The behavior of this form is modeled as a Markov chain as shown in Fig 5.3 and probabilities are assigned to each arc according to expert judgment as are assigned in previous chapter. For simplicity of the graphical representation we omit some unimportant arcs.

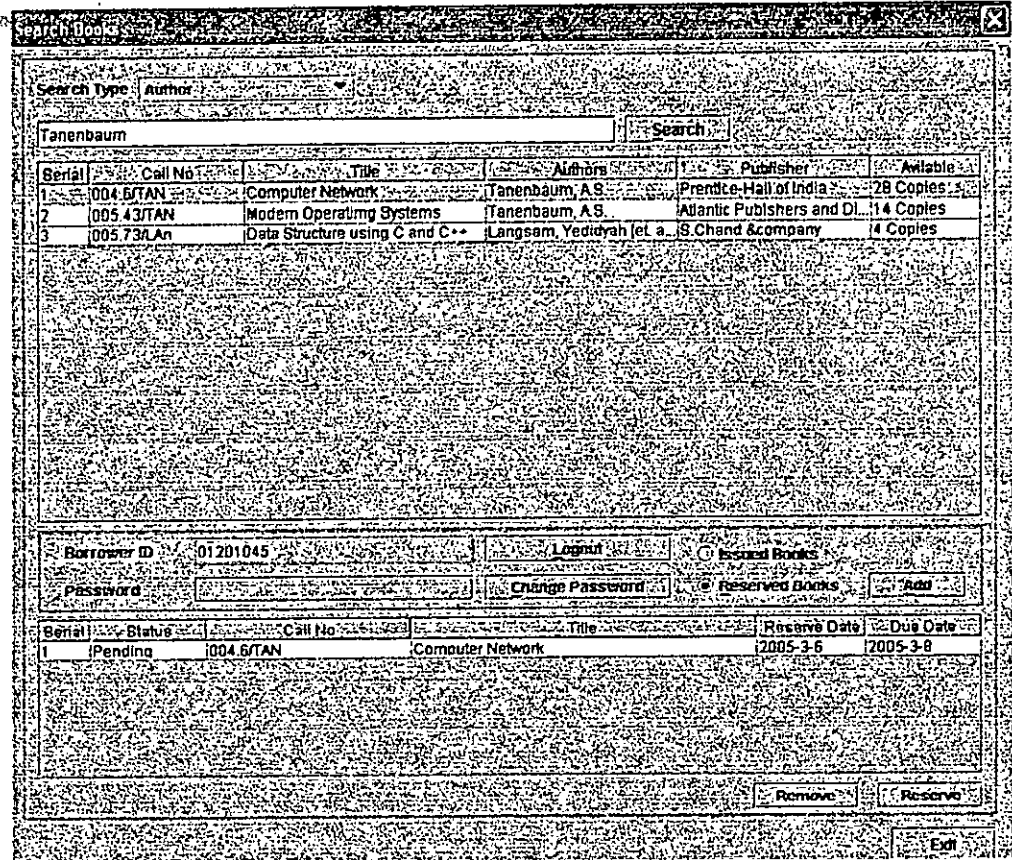


Fig. 5.2: Search Books Form 2

Assigning each arc an "actual" failure rate testing was simulated. Taking random walks from "Un-invoked" to "Terminated" of the usage model based on the transition matrix generated the simulated test sequences. At each transition it was randomly determined whether a failure occurred; success and failure matrices were updated as discussed earlier.

All arcs were assumed to have a failure rate of 0.01 and no prior information was used in the arc failure rate estimates, i.e., all elements of **A** and **B** equal 1. A graph of the single action reliability is shown in Fig. 5.4.

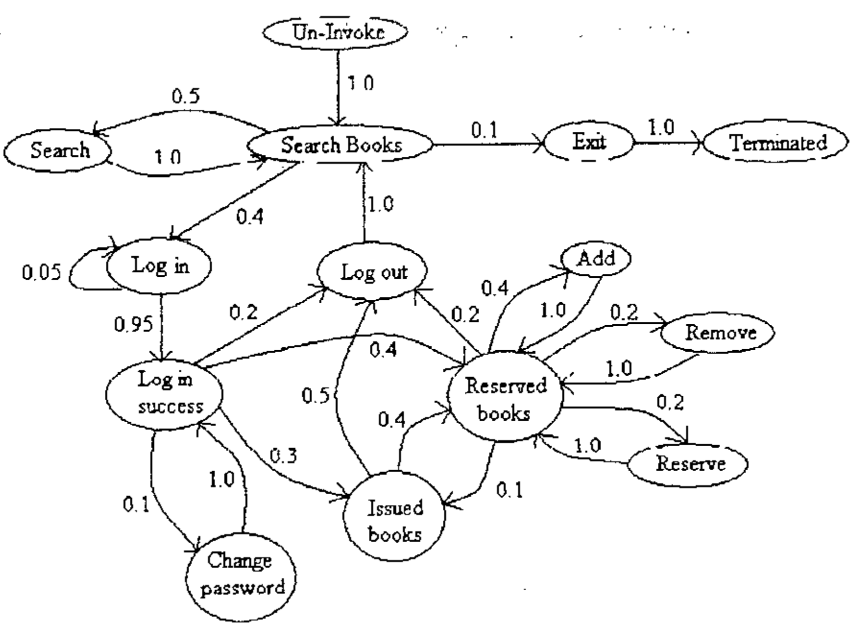


Fig. 5.3: Usage Markov Chain of Search Books

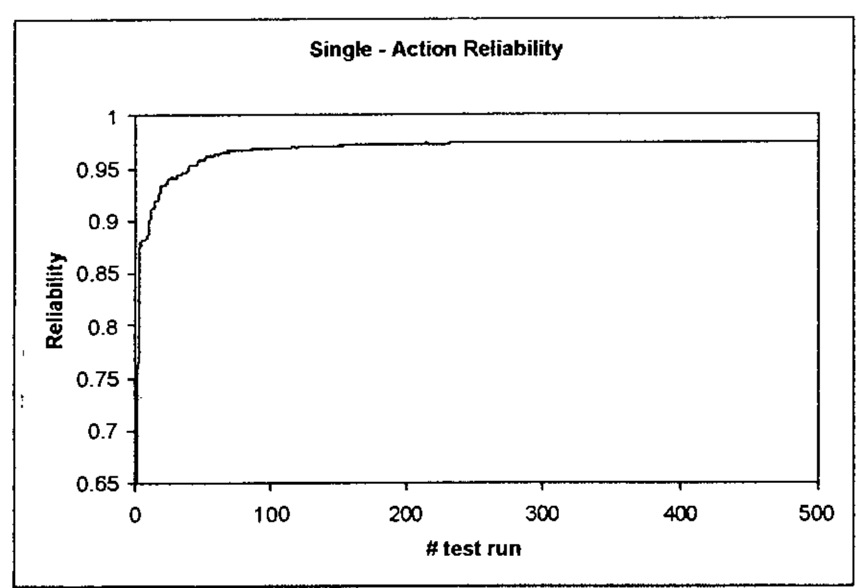


Fig. 5.4: Single-Action Reliability

Through the application of statistical sampling theory it is possible to compute the probability that the testing chain will remain essentially unchanged if more test cases are run. When estimating the mean of a population through sampling it is possible to estimate

the variance of the sample mean. Given a sample of size, the variance of the sample mean provides information on how the sample mean might vary from sample to sample. If the variance of the sample mean is small, repeated drawings of samples of size are likely to yield the same sample mean.

The variance of the single -action reliability estimator is shown in Fig. 5.5.

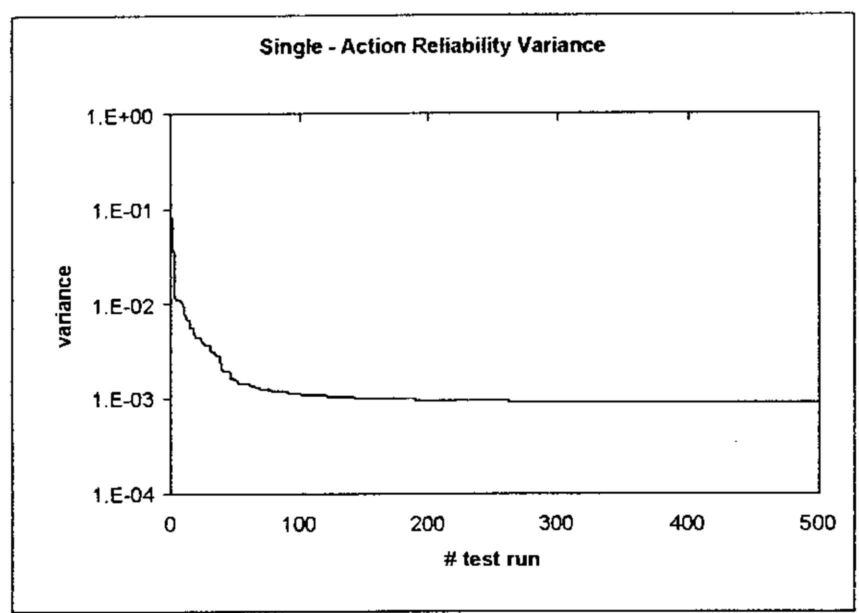


Fig. 5.5: Single Action Reliability Variance

The stopping criterion is determined for this process and is discussed in the next chapter. Software consists of a number of forms. Single action reliability estimator determines reliability R_i of every form. The system reliability $R = \sum_i p_i * R_i$ is computed from the form's reliabilities, where p_i is the usage probability of each form.

Chapter 6

Stopping Criteria

Currently, three methods are used to compare testing experience with expected use of the software, the Euclidean distance between the usage chain and the testing chain, the Kullback discriminant from the usage chain to the testing chain and Krik Sayre's long run arc occupancy. These methods compare the current testing chain with the usage chain and provide an indication of the degree to which the testing chain matches the usage chain. However, the Kullback discriminant and Krik Sayre's long run arc occupancy provide a more accurate indication of the similarity of the usage chain and the testing chain.

6.1. The Euclidean Distance

The Euclidean distance is computed as $\sqrt{\sum_{i,j}(u_{i,j} - t_{i,j})^2}$ where $u_{i,j}$ and $t_{i,j}$ are the probabilities of going from state (*i*) to state (*j*) in the usage chain and the testing chain, respectively.

As stated earlier, the Euclidean distance can be an inaccurate measure of the similarity of two usage models. For example, consider the usage model shown in Fig. 6.1.

Now suppose that two different testing chains with extreme differences resulted from two separate testing experiments. The testing chains are shown in Fig. 6.2 and Fig. 6.3.

In testing chain **A** the probabilities of the arcs exiting (*Invoke*) match the corresponding arcs in the usage model. However, the probabilities of the arcs in the cloud containing the majority of the model structure of testing chain **A** do not match the corresponding arcs in the usage model. In testing chain **B** the situation is reversed. The arcs exiting (*Invoke*) in testing chain **B** do not match the corresponding arcs in the usage chain but the probabilities of the arcs in the cloud in testing chain **B** do match the probabilities of the corresponding arcs of the usage model. Because testing chain **B** has more arc probabilities in common with the usage model than testing chain **A**, the Euclidean distance will indicate that testing chain **B** is much closer to the usage chain than testing chain **A**, or in other words the testing performed to create testing chain **B** will

be interpreted as being more representative of the expected use of the software than the testing experience represented by testing chain A. If the Euclidean distance is interpreted in this manner the software organization runs the risk of wasting time fixing relatively unimportant faults uncovered through testing of the cloud and runs the risk of missing important faults that would be exposed through testing of the transition from *(Invoke)* to *(State A)*. Thus, the Euclidean distance has the potential for misleading interpretation.

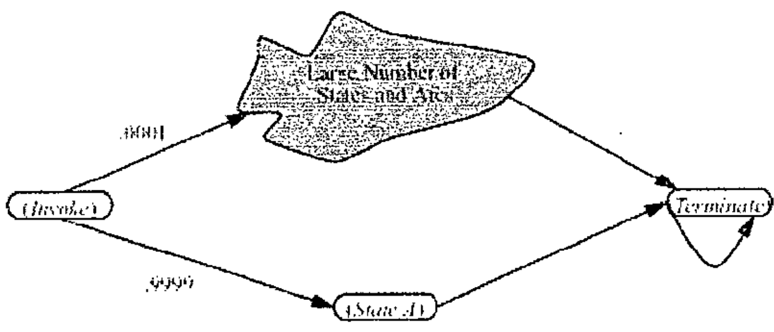


Fig. 6.1: Euclidean Distance, Example Model

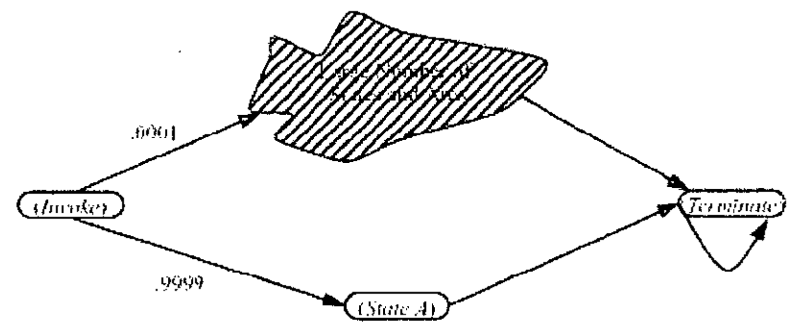


Fig. 6.2: Euclidean Distance, Testing Chain A

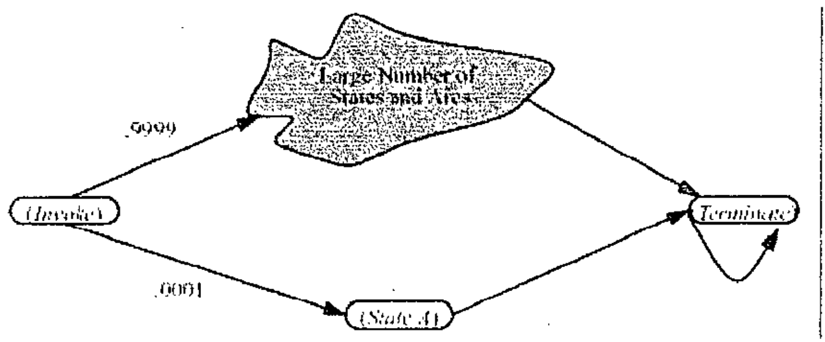


Fig. 6.3: Euclidean Distance, Testing Chain B

6.2. The Kullback Discriminant

The Kullback discriminant [53] is the expected value of the log-likelihood ratio of two stochastic processes, i.e. $K(U,T)=\lim_{n\to\infty}\frac{1}{n}\left[\log\left(\frac{\Pr[X_0,X_1,...,X_n \mid U]}{\Pr[X_0,X_1,...,X_n \mid T]}\right)\right]$. In the specific case of comparing the usage chain to the testing chain, $X_0, X_1,..., X_n$ is a sequence of length n generated by the usage chain and $K(U,T)=\sum_{i=1}^{u}\pi(i)\sum_{j=1}^{u}u_{i,j}\log\left(\frac{u_{i,j}}{t_{i,j}}\right)$.

A problem arises in the computation of the discriminant when one or more arcs in the usage chain have not been covered in the testing chain. This leads to a division by zero in the discriminant calculation. Therefore, the discriminant is not defined unless all arcs in the usage chain have been covered during testing.

6.3. The Sayre Long Run Arc Occupancies

According to Sayre the testing chain is considered to heave converged if $\Pr[\forall_i \forall_j (|\pi(i,j)-\hat{\pi}(i,j)| \leq \varepsilon_{i,j})] > p$, i.e. the probability that all the long run arc occupancies of the testing chain will be approximately equal to the long run arc occupancies of the usage chain is greater than p, if an equal number of tests were to be run again. Henceforth, the testing chain will be termed *approximately equal* to the usage chain if all of the long run arc occupancies as estimated from the testing record are approximately equal to the long run arc occupancies of the usage chain. This probability is estimated through simulation.

The simulation is performed by repeating iterations of generating a fixed number, n, of sequences, updating the testing chain, and checking to see if the resulting testing chain is approximately equal to the usage chain. The probability of the testing chain being approximately equal to the usage chain after the generation of n sequences is estimated as the proportion of times that the testing chain and usage chain were approximately equal to the total number of times the generation of n sequences was simulated.

In more detail, given **S** and **F** as the initial value of the testing record, n, the number of sequences to generate, U, the usage chain, and j, the number of simulation iterations, the simulation proceeds as follows:

```
Count_Of_Equal = 0
for p=1 to j do
     t_temp = S + F
     for q = 1 to n do
          s = Generate_Sequence(U)
          Update t_temp with s
     end for
     if (Estimate_Long_Run_Arc_Occ(t_temp) is approximately
     equal to Calc_Long_Run_Arc_Occ(U)) then
          Count_Of_Equal = Count_Of_Equal + 1
     end if
end for
Probability = Count_Of_Equal/j
```

As j becomes sufficiently large, `Probability` will approach the true probability of the testing chain being approximately equal to the usage chain after the generation of n sequences.

6.4. Our Stopping Criterion

Neither the Kullback discriminant nor the Euclidean distance directly check whether the testing chain has followed a testing activity to converge to the usage chain. They simply provide a number used by the testing engineer to assess the degree to which the testing chain is currently in some sense equal to the usage chain. So we use Sayre's long run arc occupancies technique to measure the stopping point of the test process. The graph in Fig. 6.4 shows the probability of approximate equalty of the testing chain, **T**,

and the usage chain, U. During the simulation the testing record was initialized with an empty testing record, i.e., the same base of testing experience was used to compute the probability of approximate equality after running $n = 1 \ldots 1000$ additional tests. In this example the $\varepsilon_{i,j}$ for each arc (i, j) was set to 20% of the actual long run occupancy of the arc. Thus, the testing chain is considered to be approximately equal to the usage chain if all long run arc occupancies as estimated from the testing record are within 20% of their actual values. Values were computed every ten test cases

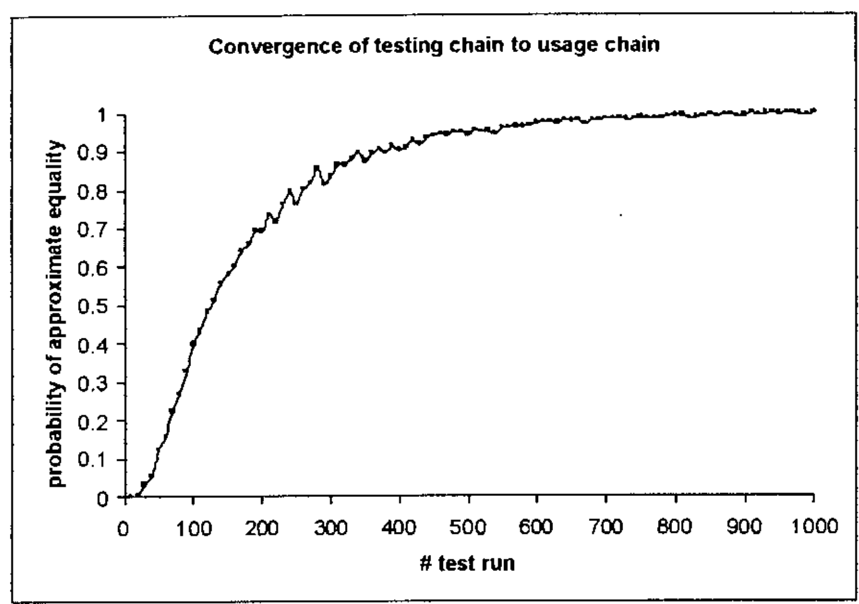


Fig. 6.4: Example Model, Convergence of Testing Chain to Usage Chain

Given that no specific prior testing was performed, when testing from the example model there is approximately a 50% chance of the testing chain being approximately equal to the usage chain after running 190 test cases. After running 1000 test cases there is a 99.5% chance of the testing chain being approximately equal to the usage chain.

In Fig. 6.4 the graph of the probability of approximate equality given a fixed base of testing experience seems to be smoothly increasing, with a number of local rough points. These rough points in the graph will disappear if the number of iterations in the simulation is increased. Given a fixed base of testing experienee, i.e., the simulation is initialized every time with the same testing record, the probability of approximate equality increases monotonically as the number of tests run increases.

There are two basic ways of using the probability of approximate equality, (1) calculating the probability of approximate equality given a fixed prior testing record and varying the number of additional tests to run, or (2) calculating the probability of approximate equality given a fixed number of additional tests to run based on a successively updated testing record. Discussion up to this point has centered around the calculation of the probability of approximate probability given a fixed prior testing record and varying the number of additional tests to run. The probability of approximate equality is monotonically increasing in this case.

Now we are concerned with the second use of the probability of approximate equality. The testing record used in the estimation of the probability of approximate equality is updated after each executed test case. Given this updated base of testing experience, the probability of approximate equality after running some number of additional tests is estimated.

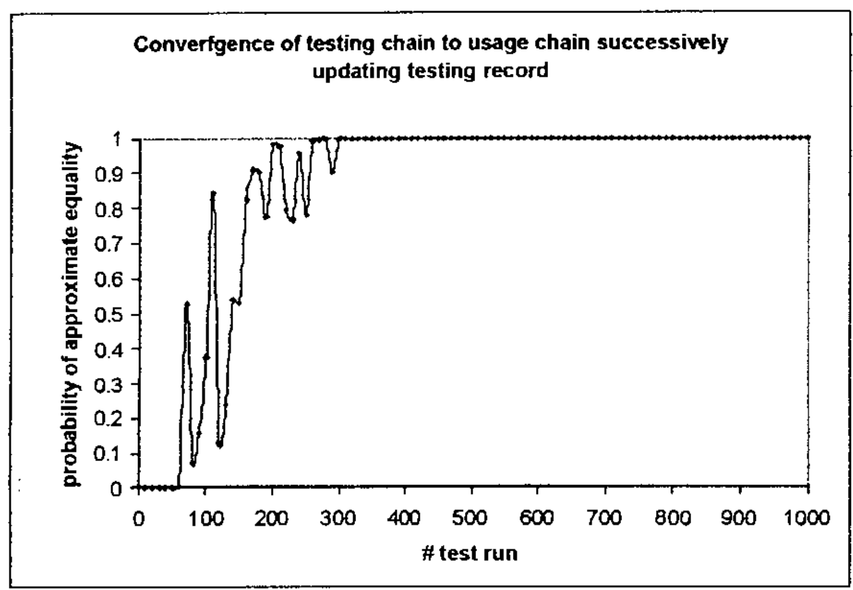


Fig. 6.5: Convergence of Testing Chain to Usage Chain, Successively Updated Testing Record

Fig. 6.5 and Fig. 6.6 displays the probability that the testing chain will be approximately equal to the usage chain after running $n1$ additional test cases, given that n specific test cases have already been run. Two different random seeds are used to generate the two graphs. The probabilities presented illustrate the case for $n1 = 10$ and n going from 0 to 1000.The testing record used in the simulation of the probability of

approximate equality is updated after each test case is run, i.e., the base of testing experience is evolving over time.

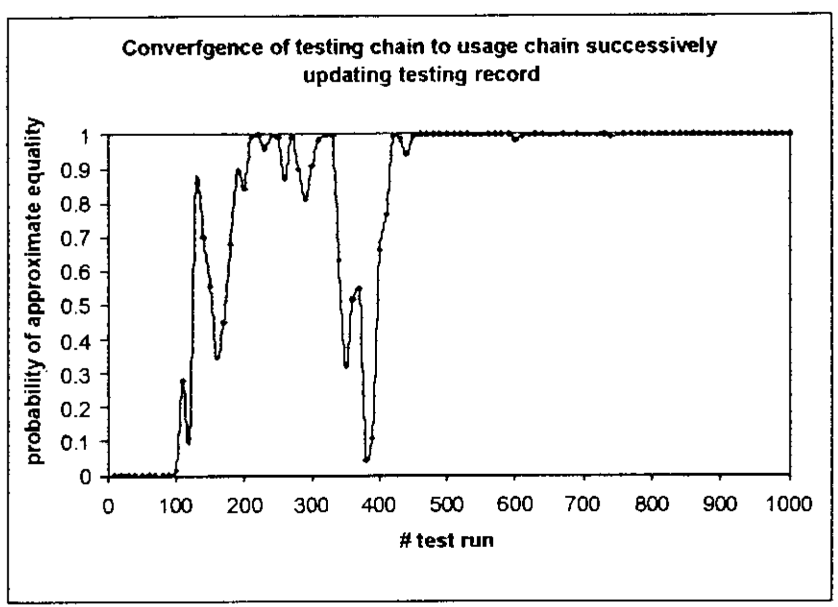


Fig. 6.6: Convergence of Testing Chain to Usage Chain, Successively Updated Testing Record

The graph of Fig. 6.5 and Fig. 6.6 do not follow an orderly curve. Until the testing record has stabilized to a certain degree, the probability of approximate equality of the testing chain and usage chain after running the next ten test cases given the updated testing experience is quite sensitive to the current state of the testing record. While testing chain **T** will eventually converge to usage chain **U**, **T** does not converge monotonically to **U**.

A planning analysis, using Fig. 6.4 might suggest that 700 test cases will need to be run before the probability of approximate equality nears 0.95. Fig. 6.5 & Fig. 6.6 suggest that given the actual testing performed, the probability of approximate equality consistently exceeds 0.95 after 500 test cases have been run. Therefore, if the probability of approximate equality being greater than 0.95 was used as a stopping criterion, it may make sense to stop testing after 500 test cases have been run.

Chapter 7

Conclusion

7.1 Introduction

The statistical reliability assessment of software requires that the random test cases be generated from the operational profile or usage pattern of the system. An operational profile or usage model consists of a logical description of the possible states of the system and of a statistical or probabilistic model describing how often certain states of sequences of states are encountered. If the model doesn't correspond statistically to the actual usage of the system, the reliability estimates based on the statistical testing is erroneous. In this thesis we have considered statistical testing approach and reliability assessment in general. We have shown how stochastic modeling [55] can be applied to the software-testing problem. Although choice for a model could have been any number of stochastic processes, Markov chains were used because they have been shown to be successful in practice and because of their potential to provide valuable analytical feedback.

In this concluding chapter, the contributions and limitations of the research are presented, and propose future research tasks aimed at addressing the limitations.

7.2 Contributions

Statistical software testing promises a solution to the increased testing burden caused by the ever-increasing complexity of today's software systems; however, the complexity of these systems makes it more difficult to provide a model to use as a basis for statistical testing. The flat operational profile and Whittaker's Markov model leads to enormous models when capturing the usage of these complex systems and makes it infeasible to generate test cases randomly and asses reliability form the enormous model. In our thesis we have shown a different approach while measuring software reliability. We combined the ideas of stochastic modeling, statistical testing and component based software testing to measure reliability of software system. We also find a stopping criterion to stop testing, i.e. the number of test cases that should be run before releasing software.

Taking an application form as a software component and modeling it as a Markov chain gives us several benefits. Traditional Markov modeling or stochastic modeling would incorporate additional states to previous model and would require repeated testing effort to find reliability. But in our approach it needs only to model the new form and find its reliability to measure system reliability that is less cumbersome. Thus our approach considers the impact of software change or software evolution. We choose single action reliability model to measure the individual form's reliability, which enables us to use pre-test information that was not possible in Whittaker model. Testing savings can be realized if accurate pre-test reliability information is available. We also apply Sayre's long run arc occupancy to measure the similarity of usage model and testing model. Thus extensive simulation can be used in test planning. A testing organization now has considerable ability to tailor the reliability estimation to the situation in order to make testing more efficient. Using partition-testing techniques in conjunction with usage models and arc-based reliability models increase testing efficiency. These accomplishments allow for larger systems to be modeled more concisely and compactly while providing statistical testing's benefits of effective, efficient testing, reliability estimation and quantified business decisions.

7.3 Suggestions for Further Research

A formal relationship between the similarity of the testing chain and the usage chain and the estimated reliability should be established. Field evidence shows that a high degree of similarity between the testing chain and the usage chain indicates that the reliability estimated from the testing experience is accurate. However, a more formal relationship is needed. The distributions of various random variables based on the usage model (sequence length, number of sequences to cover all states or arcs, etc.) should be studied. Knowledge of the distribution underlying these random variables will allow for increased accuracy in test planning and model validation.

In our approach we simply come up the idea of divide and conquer method. But as the model increases with the growing complexities of software, researchers have developed the Requirements State Machine Language without Events ($RSML^{-e}$), a state-based modeling language that will serve as the basis for the parallelism-capable statistical modeling [66]. As a formal language $RSML^{-e}$ can be used to accurately describe requirements and create a state-based model of a system's behavior. This addresses the

statistical component of the operational profile, adding probabilities to the structural model and generating test cases from the parallelism-enhanced operational profile.

In this thesis we show how stochastic modeling can be applied to the software-testing problem. Although the choice for a model could have been any number of stochastic processes, Markov chains were used because they have been shown to be successful in practice and because of their potential to provide valuable analytical feedback. Our given example is a small one and the future work can focus on more complex software like banking software, where transactions are very much important. This will increase the volume of state space. Future work can also investigate methods to mechanically enumerate the state space of the model from the operational modes. We envision the development of the operational modes and a set of constraints defining possible states to be the task of a human tester and then an algorithm would generate the full state space. It remains to be seen how general such an algorithm could be and whether we could embed arc information so that the entire Markov chain could be constructed.

The issue of generating test cases randomly from the model is an important one. Currently, we generate test cases based only on the probabilities assigned to each arc. An extension of this idea would be to dynamically change the probabilities as new information surfaces during test. For example, software that has artificial intelligence capabilities may change its branching probabilities form one state to another. Also, we might decide that a particularly buggy section of code needs additional testing and then raise the probabilities associated with that part of the model. Thus the model would adapt to the demands of testing by learning failure patterns and adjusting probabilities to get better coverage of specific parts of the model.

In our research effort we focus on database based application software and take only functional requirements in consideration to measure reliability. But today security aspects of database especially authentication, authorization are some major issues. Can we say that our system is highly reliable if security is poor? No research effort is given to this direction. A formal approach could be developed that incorporates security issues and other kinds of failure like network falure etc. in measuring software reliability.

References

[1] Ch. Ali Asad, Muhammad Irfan Ullah and Muhammad Jaffar-Ur Rehman, "An Approach for Software Reliability Model Selection", *Proceedings of the 28th Annual International Computer Software and Applications Conference*, Hong Kong, China, September 2004.

[2] Denton A. D., "Accurate Software Reliability Estimation", Master of Science Thesis, Colorado State University, Fort Collins, Colorado, Fall 1999.

[3] C. Smidts, R. W. Stoddard and M. Stutzke, "Software Reliability Models: An Approach to Early Reliability Prediction", *IEEE Transactions on Reliability*, vol. 47(3), pp. 268 – 278, 1998.

[4] H. Sing, V. Cortellessa, B. Cukic, E. Gunel and V. Bharadwaj, "A Bayesian Approach to Reliability Prediction and Assessment of Component Based System", *Proceedings of 12th International Symposium on Software Reliability Engineering (ISSRE)*, Hong Kong, China, November 2001.

[5] V. Cortellessa, H. Sing and B. Cukic, "Early Reliability Assessment of UML Based Software Models", *3rd International Workshop on Software Performance*, Rome, Italy, July 2002.

[6] A. Wood, "Software Reliability Growth Models", *Tandem Computers*, TR 96.1, Part No. 130056, September 1996.

[7] J. D. Musa, *Software Reliability Engineering*, New York: McGraw – Hill, 1998.

[8] A. L. Goel, "Software Reliability Models: Assumptions, Limitations and Applicability", *IEEE Transaction on Software Engineering*, vol. 11(12), pp. 1411 – 1423, December 1985.

[9] S. S. Gokhale, P. N. Marinos and K. S. Trivedi, "Important Milestones in Software Reliability Models", *Proceedings of Software Engineering and Knowledge Engineering*, Lake Tahoe, NV, pp. 345 – 352, 1996.

[10] E. Nelson, "Estimating Software Reliability from Test Data", *Microelectronics and Reliability*, vol. 17(1), pp. 67 – 73, 1978.

[11] S. Gokhale, W. E. Wong, K. S. Trivedi and J. R. Horgan, "An Analytical Approach to Architecture-Based Software Reliability Prediction", *Proceedings of IEEE International Computer Performance and Dependability Symposium*, pp 13 – 22, September 1998.

[12] K. Goseva - Popstojanova and K. S. Trivedi, "Architecture Based Software Reliability", *Proc. of International Conference on Applied Stochastic System Modeling*, Kyoto, Japan, March 2000.

[13] J. A. Whittaker and M. G. Thomason, "A Markov chain model for statistical software testing", *IEEE Transactions on Software Engineering*, vol. 20(10), pp. 812 – 824, October 1994.

[14] J. A. Whittaker, "Markov chain techniques for software testing and reliability analysis", Ph.D. dissertation, Dept. of Computer Science, University of Tennessee, Knoxville, USA, 1992.

[15] B. Littlewood, "Software Reliability Model for Modular Program Structure", *IEEE Transaction on Reliability*, vol. 28(3), pp. 241 – 246, 1979.

[16] B. Littlewood, "A Reliability Model for Systems with Markov Structure", *Applied Statistics*, vol. 24(2), pp 172 – 177, 1975.

[17] R. C. Cheung, "A User – Oriented Software Reliability Model", *IEEE Transactions on Software Engineering*, vol. 6(2), pp. 118 – 125, 1980.

[18] J. C. Laprie, "Dependability Evaluation of Software Systems in Operation", *IEEE Transactions on software Engineering*, vol. 10(6), pp. 701 – 714, 1984.

[19] P. Kubat, "Assessing Reliability of Modular Software", *Operational Research Letters*, vol. 8, pp. 35 – 41, 1989.

[20] K. Sayre and J. H. Poore, "Partition testing with usage models", *Information & Software Technology*, vol. 42(12), pp. 845 – 850, 2000.

[21] B. D. Juhlin, "Implementing operational profiles to measure system reliability", *Proceedings of 3rd International Symposium on Software Reliability Engineering*, vol. 7(10), pp. 286 – 295, October 1992.

[22] M. Gittens, H. Lutfiyya and M. Bauer, "An Extended Operational Profile Model", *Proceedings of 15th International Symposium on Software Reliability Engineering*, vol. 2(5), pp. 314 – 325, November 2004.

[23] K. Agrawal and J. A. Whittaker, "Experiences in Applying Statistical Testing to a Real-time Embedded Software System", *Proceedings of Pacific Northwest Software Quality Conference*, pp. 154 – 170, 1993.

[24] J. A. Whittaker and J. H. Poore, "Markov Analysis of Software Specifications", *ACM Transactions on Software Engineering Methodology*, vol. 2, pp. 93 – 106, January 1993.

[25] F. Zhen and C. Peng, "A System Test Methodology Based on the Markov Chain Usage Model", *Proceedings of 8th International Conference on Computer Supported Cooperative Work and Design*, pp. 160 – 165, 2003.

[26] Kirk Sayre, "Improved Techniques for Software Testing Based on Markov Chain Usage 9Models", Ph.D. dissertation, Dept. of Computer Science, University of Tennessee, Knoxville, USA, December 1999.

[27] K.W. Miller, et. al., "Estimating the Probability of Failure When Testing Reveals No Failures", *IEEE Transactions on Software Engineering*, Vol. 18, pp. 33-42, January 1992.

[28] J. McCall, P. Richards and G. Walters, "Factors in Software Quality", NTIS AD-A049-014, 015, 055, November 1977.

[29] B. Beizer, *Software Testing Techniques*, second ed. Boston, Mass. Int'l Thomason Computer Press, 1990.

[30] G. Tassey, *The Economics of R&D Policy*, Westport, CT: Quorum Books, 1997.

[31] R. S. Pressman, *Software Engineering: A Practical Approach*, fifth edition, McGraw Hill, 2001.

[32] C. Jones, *Software Quality-Analysis and Guidelines for Success*, Boston: International Thompson Computer Press, 1997.

[33] D. Wells, "Survivability in Object Services Architectures", Annual Report, Object Services and Consulting Inc., www.objs.com/survivability/, 1998.

[34] C. V. Ramamoorthy and F. B. Bastani, "Software Reliability – Status and Perspectives", *IEEE Transactions on Software Engineering*, vol. 11 (12), pp. 354 – 371, 1982.

[35] W. Farr, "Software Reliability Modeling Survey", *in Handbook of Software Reliability Engineering*, pp. 71 – 117, McGraw-Hill, 1996.

[36] J. R. Horgan and S. London, "ATAC: A Data Flow Coverage Testing Tool for C", *Proceedings of 2nd Symposium on Assessment of Quality Software Development Tools*, vol. 2(10), 1992.

[37] M. Shooman, "Structural Models for Software Reliability Prediction", *Proceedings of 2nd International Conference on Software Engineering*, pp. 268 – 280, 1976.

[38] S. Krishnamurthy and A. P. Mathur, "On the Estimation of Component Based Software Systems", *Proceedings of 9th International Symposium on Software Reliability Engineering*, pp. 192 – 201, 1998.

[39] S. Gokhale and K. Trivedi, "Dependency Characterization in Path-Based Approaches to Architecture Based Software Reliability Prediction", *Proceedings of Symposium on Application – Specific Systems and Software Engineering Technology*, pp. 86 – 89, 1998.

[40] M. Xie and C. Wohlin, "An Additive Reliability Model for the Analysis of Modular Software Failure Date", *Proceedings of 6th International Symposium on Software Reliability Engineering*, pp. 188 – 194, 1995.

[41] R. H. Cobb and H. D. Mills, "Engineering Software Under Statistical Quality Control", *IEEE Software*, vol. 7(6), pp. 44 – 54, November 1990.

[42] S. J. Prowell, R. C. Linger and S. Prowell, "Cleanroom software Engineering: Developing Software Under Statistical Quality Control", Addison-Wesley, 1999.

[43] Chaitanya Kallepalli and Jeff Tian, "Measuring and modeling usage and reliability for statistical web testing", *IEEE transactions on software engineering*, vol. 27 (11), pp. 1023 – 2001, November 2001.

[44] J. D. Musa, "Operational Profiles in Software Reliability Engineering", *IEEE Software*, vol. 10 (2), pp. 14-32, 1993.

[45] Haapanen Pentti, Pulkkinen Urho and Korhonen Jukka, "Usage models in reliability assessment of software-based systems", STUK-YTO-TR 128, Helsinki, pp. 1-48, April 1997.

[46] J. Duran and J. Wiorkowski, "Quantifying Software Validity by Sampling", *IEEE Transactions on Reliability*, vol. 29, no. 2, pp. 141-144, 1984.

[47] S. Karlin and H.M. Taylor, *A First Course in Stochastic Processes*, second ed., Academic Press, New York, 1975.

[48] J. G. Kemeny and J. L. Snell, *Finite Markov Chains*. New York: Springer – Verlag, 1976.

[49] P. A. Currit, M. Dyer and H. D. Mills, "Certifying the correctness of software", *IEEE Trans. Software Eng.*, vol. 12(1), pp. 3 – 11, Jan. 1986.

[50] R. Hamlet, "Testing software for software reliability", Technical Report, TR – 91 – 2, rev. 1, Department of Computer Science, Portland, OR, USA, March 1992.

[51] K. Diegrist, "Reliability of systems with Markov transfer of control", *IEEE Transaction on Software Engineering*, vol. 14, pp. 1049 – 1053, July 1988.

[52] B. H. Juang and L. R. Rabiner, "A probabilistic distance measure for hidden Markov models", AT & T Tech. J., vol. 64(2), pp. 391 – 408, Feb. 1985.

[53] S. Kullbck, *Information theory and statistics*, New York: Wiley, 1958.

[54] J. L. Doob, *Stochastic Processes*, New York: Wiley, 1953.

[55] J. A. Whittaker, "Stochastic Software Testing", *IEEE Annals of Software Engineering*, vol. 4, pp. 115 – 131, 1997.

[56] N.L. Johnson, S. Kotz and N. Balakrishnan, *Continuous Univariate Distributions*, John Wiley and Sons, New York, 1995.

[57] W. G. Cochran, *Sampling Techniques*, John Wiley and Sons, New York, 1977.

[58] S. M. Yacoub, B. Cukic, H. H. Ammar, "Scenario-based Reliability Analysis of Component-based Software", in *Proc. of the 10th Intarnational Sumposium of Softwaer Reliability Engineering*, pp.22 – 31, 1999.

[59] W. Everett, "Software Component Reliability Analysis", in the *Proceeding of Symposium on Application-Specific Systems and Software Engineering Technology*, pp. 204 – 211, 1999.

[60] S. N. Weiss and E. J. Weyuker, "An Extended Domain-Bases Model of Software Reliability", *IEEE Transaction on Software Engineering*, vol.14, no. 10, pp.1512-1524, October 1988.

[61] Yamada, Shigeru, Mitsuru Ohba and Shunji Osaki, "S-shaped Reliability Growth Modeling for Software Error Detection", *IEEE Transaction on Reliability*, vol. R-32, pp. 475 – 484, December 1983.

[62] Hossain, Syed and Ram Dahiya, "Estimating the Parameters of a Non-Homogeneous Poison-Process Model of Software Reliability, *IEEE Transaction on Reliability*, vol. 42, no. 4, pp. 604 – 612, December 1993.

[63] Kececioglu, Dimitri, *Reliability Engineering Handbook*, Volume 2, Prentice-Hall, 1991.

[64] Littlewood B., "Stochastic reliability Growth: A Model for Fault Removal in Computer Programs and Hardware Design", *IEEE Transaction on Reliability*, vol. R-30, pp. 313 – 320, December 1981.

[65] Yamada, Shigeru, Hiroshi Othera and Hiroyuki Narihisa, "Software Reliability Growth Models with Testing Effort", *IEEE Transaction on Reliability*, vol. R-35, no. 1, pp. 19 – 23, April 1986.

[66] Robert J. Weber, "Statistical Software Testing with Parallel Modeling: A Case Study", *Proceedings of the 15th International Symposium on Software Reliability Engineering*, 2004.