To the Graduate Council:

We are submitting a dissertation written by James A. Whittaker entitled "Markov Chain Techniques for Software Testing and Reliability Analysis." We have examined the final copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jesse H. Poore, co-Major Professor

Michael G. Thomason, co-Major Professor

We have read this dissertation
and recommend its acceptance:

Accepted for the Council:

Associate Vice Chancellor
and Dean of The Graduate School

MARKOV CHAIN TECHNIQUES FOR SOFTWARE TESTING AND RELIABILITY ANALYSIS

A Dissertation

Presented for the

Doctor of Philosophy Degree

The University of Tennessee, Knoxville

James A. Whittaker

May 1992

Copyright © James A. Whittaker, 1991
All rights reserved

To Sharon

Who told me I could

Acknowledgements

This dissertation is the result of several years of theoretical and empirical study. Along the way, there were many people who contributed in various ways to its completion. First, I thank Dr. Jesse Poore for suggesting the problem and for supplying the opportunity for me to work as part of his research group. His guidance in this work has been invaluable. Dr. Michael Thomason deserves special recognition for his contributing ideas during this research. I thank Dr. Thomason for the time he spent patiently answering my many questions. I am honored to have these two as my co-advisers on this project and look forward to working with them in the future.

I am grateful to Dr. David Mutchler, Dr. Mary Leitnaker, and Dr. Harlan Mills for their time in reading this dissertation as well as for their helpful suggestions. I extend my thanks to the members of the various research teams and study groups that I was involved with at UT. Special thanks go to Charlie Guthrie, Hailong Mao, and David Primeaux who helped me endure some particularly trying times.

Finally I thank my family for their support and encouragement. My most personal thanks go to my wife, Sharon, for believing in me and waiting for me.

Abstract

Cleanroom Software Engineering [40] is a new methodology that has evolved from structured programming into a promising technology for high quality software development. Cleanroom has three major components: specification, design with verification, and statistical certification testing. This dissertation describes a new approach to statistical testing by modeling software usage and the testing process as finite state, discrete parameter Markov chains. Using the software specification document as a guide, a Markov chain is constructed which models the *usage* of the specified software. This time homogeneous chain is used to compute stochastic properties of pertinent usage random variables before any code development begins and to generate a set of "statistically typical" test sequences. These sequences, along with any failure data they produce upon execution, are used as a training set for a second Markov chain which models the behavior of the software during testing. This second chain is updated as testing progresses and is used to compute software quality measures, such as the reliability and mean time between failure at any stage of the testing process. Comparison of the two chains is by an information theoretic discriminant function based on the ergodic properties of the stochastic processes. Among its uses this comparison yields an analytical stopping criterion for the testing process. The latter chain is updated based upon appropriate expected values to obtain a third chain which is used to predict future software quality. The model presented is a complete certification strategy encompassing usage modeling, statistical testing, and reliability analysis.

Table of Contents

Chapter Page

1. Introduction 1

1.1 Statistical Software Testing 1

1.2 Outline and Overview 6

2. Mathematical Principles - Markov Chains 8

2.1 Introduction 8

2.2 Basic Concepts 8

2.3 Irreducible Markov Chains 11

2.3.1 Asymptotic Behavior 12

2.3.2 Further Computations on Irreducible Markov Chains 14

2.3.3 Typical Sequences 15

2.4 Absorbing Markov Chains 17

2.5 Additional Results 18

3. The Usage Analysis 20

3.1 Introduction 20

3.2 Constructing the Usage Model 22

3.3 Analytical Results from the Usage Chain 27

4. The Reliability Analysis 36

4.1 Introduction 36

4.2 Constructing the Testing Markov Chain 37

4.2.1 Failure Data 39

4.2.2 Properties of the Testing Markov Chain 41

4.3 A Stopping Criterion 45

4.4 Analytical Results from the Testing Chain 50

4.5 Predicting Future Software Behavior 54

5. A Case Study 58

5.1 Introduction 58

5.2 The Specification 59

5.3 The Usage Analysis 63

5.4 The Reliability Analysis 75

6. Concluding Remarks 83

6.1 Synopsis 83

6.2 Future Work 85

Bibliography 88

Vita 94

List of Figures

Figure 1.1 : Overview of the Testing Model 7

Figure 3.1 : Top Level View of Software Usage 22

Figure 3.2 : A Hypothetical Window 24

Figure 3.3 : Expansion of the Top Level Usage Diagram 25

Figure 3.4 : Structural Phase - Constructing the Usage Markov Chain 25

Figure 4.1 : The Frequency Model Updated With Sequence One 38

Figure 4.2 : The Frequency Model Updated With Sequences One and Two 39

Figure 4.3 : The Frequency Model Updated With All Five Sequences 40

Figure 4.4 : Adding a Failure State 41

Figure 5.1 : The CreMain Window 60

Figure 5.2 : Plot of *R* Data for Test Cases 1-100 80

Figure 5.3 : Plot of *MTBF* Data for Test Cases 1-100 81

Figure 5.4 : Plot of *D* Data for Test Cases 50-100 81

List of Tables

Table 3.1 : Example Software Specification 24

Table 3.2 : Statistical Phase - Assigning the Transition Probabilities 28

Table 3.3 : Analytical Results from the Irreducible Usage Chain 32

Table 3.4 : Analytical Results from the Absorbing Usage Chain 34

Chapter 1

Introduction

1.1 Statistical Software Testing

A growing concern among software users and engineers is the quality of the software with which they must interact. The introduction of software technology to medical, military, and scientific applications makes these concerns particularly poignant [35,48]. One response of the software engineering community is the introduction of formal testing methods to the software development process. A formal testing strategy, coupled with rigorous design and verification methods, forms the basis for developing high quality software. However, comprehensive methods for conducting formal software tests are scarce.

A formal software testing strategy should accomplish two goals [14]. First, it should establish statistically valid data that can be used to accurately measure software quality. This is important in order to add meaning to the test. Test cases that are applied to the software and yield no measurable confidence in the software's quality are simply anecdotes and serve no scientific purpose (although they may serve an important social purpose). Second, since it is generally impossible to determine whether or not all errors have been removed from a program, failures should be sought in the order of their frequency. That is, testing should locate failures in the same order that a typical user will encounter them. The errors that are not removed will then tend to be the least likely to be encountered in actual use of the software.

Statistical software testing, in which test cases are randomly generated based upon a probability distribution which represents actual (or intended) use of the software, accomplishes these goals. Duran and Ntafos [20] show that random testing (statistical testing in which the probability distribution of inputs is assumed to be uniform) is generally as effective at finding failures as partition testing and much less expensive to perform. Partition testing is a popular testing method in which the input space of a program is partitioned according to some heuristic and test cases are drawn uniformly from each subset of the partition [45]. This scheme seeks to minimize the number of test cases necessary by choosing only one test case from each subset. However, the authors show that effective partitioning is significantly more difficult and expensive to perform than simple random testing with a large number of test cases. Furthermore, the ability of random testing in finding failures is often superior. Hamlet and Taylor [24] suggest optimal partitioning schemes and sampling techniques based on input equivalence classes. They show that partition testing is advantageous only in the presence of strong knowledge of the software, i.e., knowledge of the failure rate in each subset. Random testing could regain the advantage simply by an inexpensive increase in the number of test cases.

Duran and Wiorkowski [21] compared random testing to path testing in which the structure of the program is considered when constructing test cases. Test cases are made to cover paths through the software. The authors argue that this is a very difficult task and show that it yields no increase in failure location ability over simple random testing while being significantly more costly.

Sexton [56] continues the idea of random testing by showing the power of the method in measuring the reliability of software. She applies the technique of hypothesis testing to software in order to determine the probability that a given program passes N successive tests. Similarly, Parnas [48] argues that software which runs in safety-critical situations must be randomly tested

in order to truly gain confidence in its ability to perform correctly under diverse usage conditions.

Currit, Dyer, and Mills [14] introduce statistical testing as the basis for obtaining a high operational software reliability. In statistical testing, test cases are either randomly generated according to some appropriate usage distribution or sampled directly from actual user data. Such usage testing allows inference of quality measures from the testing environment to the operational environment because test cases are constructed from the users' point of view. The authors point out that statistical testing tends to locate failures in the order of their contribution to the mean time to failure of the software.

Currently, no comprehensive methods exist for modeling software usage, generating random test cases, or conducting statistical tests. This research establishes a model for statistical testing by treating the testing process as a random experiment. A random experiment requires the complete characterization of the sample space and its associated probability distribution, the definition of the appropriate event space, and a method of computing properties of descriptive random variables [12]. The sample space is the input domain of the software which results from the enumeration of stimuli in the specification document. The selection of points from the sample space is governed by some unknown probability distribution that must be defined. In statistical testing, the events of interest are *sequences of stimuli* that represent an execution history or use of the software. These sequences constitute the event space of the specified software and are obtained by defining an ordering on the points in the sample space. The sequences are ultimately the important attribute of the random experiment for they represent the test cases for the software and the means by which the software is certified. Statistical descriptions of the sequences are desirable in order to gain insight into the makeup of the test cases and the expected number needed to certify the software. These descriptions are obtained by defining random variables that describe the profile of the entire set of sequences that will be used to certify the software.

The statistical testing experiment is centered around sequences of events and as such must be modeled as a stochastic process. Thus, we define a stochastic model to guide test case generation and to describe the testing process. In this research, the use of the finite state, discrete parameter Markov chain[1] to model software usage and to conduct statistical testing is explored. The states of the Markov chain represent entries from the input domain of the software, classes of entries, response indicators, and non-action states. The arcs of the chain define an ordering that determines the event space, or sequences, of the experiment. Furthermore, the ergodic nature of the chain induces a probability distribution on the sample space (called the *usage distribution*) and facilitates the computation of properties of pertinent random variables described by probabilities underlying the experiment. The Markov model facilitates comprehensive analysis of the testing process including a data-driven reliability model.

There are many other generation or sampling techniques from which to choose; however, the Markov chain is a good choice for several reasons. First, there is a well developed body of theoretical results associated with Markov chains. This strong theoretical basis allows deep characterization of the software testing process. Second, the Markov chain is a powerful, tractable mathematical model. Properties of many random variables are readily computable and have a solidly intuitive interpretation. The ergodic properties of the model give the effect of long run activity to the analytical results. Third, Markov chains are proven in practice. Although Markov chains are a highly constrained stochastic process, they perform well on models for real data in diverse applications [36,62]. Finally, Markov chains are easily automated. Once a Markov usage model is constructed, any number of test cases can be automatically obtained from the model. Analytical descriptions of the test cases are available before any testing begins.

---

[1]Hereafter, *finite state* and *discrete parameter* are understood unless different characteristics are explicitly stated.

Some aspects of software reliability have been investigated by other researchers using Markov chains. Trivedi and Shooman [63] use a model with "up" states and "down" states. The transitions between the states are derived from assumptions regarding error repair rates and failure occurrence rates. They are able to compute software reliability and availability from their chain. Siegrist [60] develops a Markov model (originally proposed by Cheung [10]) for software systems constructed from modules. He computes the system reliability (given the reliability of each module) using a Markov model with the states of the chain representing software modules. A single failure state is accessible from each module state according to the given module reliability. Thus, he reduces the problem of computing reliability to whether the process is absorbed into the legitimate termination state as opposed to the failure state.

Whenever the finite state Markov chain is the stochastic model for software usage, the specification must describe either a finite input language or one of sufficient structure to be modeled by a Markov chain. The language generating capability of the finite state Markov chain is discussed in detail in Chapter 3. The implication of a specification that cannot be modeled by a Markov chain is that characteristics of the process and its associated random variables may not be computable. For example, the source entropy (discussed in Chapter 2) for an arbitrary stochastic process requires the evaluation of a joint probability function that may be difficult or impossible to obtain. From a software engineering point of view, specifications that can be modeled by a finite state Markov chain facilitate testability of the code and imply a degree of simplicity of design and verification. Engineers ignoring the underlying stochastic model of software usage subject themselves to these problems.

The absolute applicability of the Markov chain to certain types or classes of software is not the focus of this research. Such an analysis will require formal specification documents to be available. The specification must then be classified by the type (regular, context-free, etc.) of

formal language that it represents. Further research in formal specification must be pursued to make this analysis possible. (Even so, the problem of classifying a formal language is, in general, undecidable [26].) However, the Markov model has proven useful in modeling the usage of various types of software [50-53,55] and thus warrants further investigation. Even if other models are specified, approximation by Markov chains should be considered. In classical information theory, for example, Markov chains have long been used to obtain acceptable, approximate solutions to problems otherwise not solvable at all [4].

1.2 Outline and Overview

This research presents a new approach to software usage modeling and statistical testing using Markov chains. Figure 1.1 depicts an overview of the model. Using the software specification document as a guide, a Markov chain is constructed which models the usage of the specified software. This time homogeneous chain is used to compute stochastic properties of pertinent usage random variables before any code development begins and to generate a set of "statistically typical" test sequences. These sequences, along with any failure data, are used as a training set for a second Markov chain which models the behavior of the software during testing. This second chain is updated as testing progresses and is used to compute software quality measures, such as the reliability and mean time between failure at any stage of the testing process. Comparison of the two chains is achieved using an information theoretic discriminant function based on the ergodic properties of the stochastic processes. Among its uses this comparison yields an analytical stopping criterion for the testing process. The latter chain is updated based upon appropriate expected values to obtain a third chain which is used to predict future software quality. The model presented is a complete certification strategy encompassing usage modeling, statistical

testing, and reliability analysis.

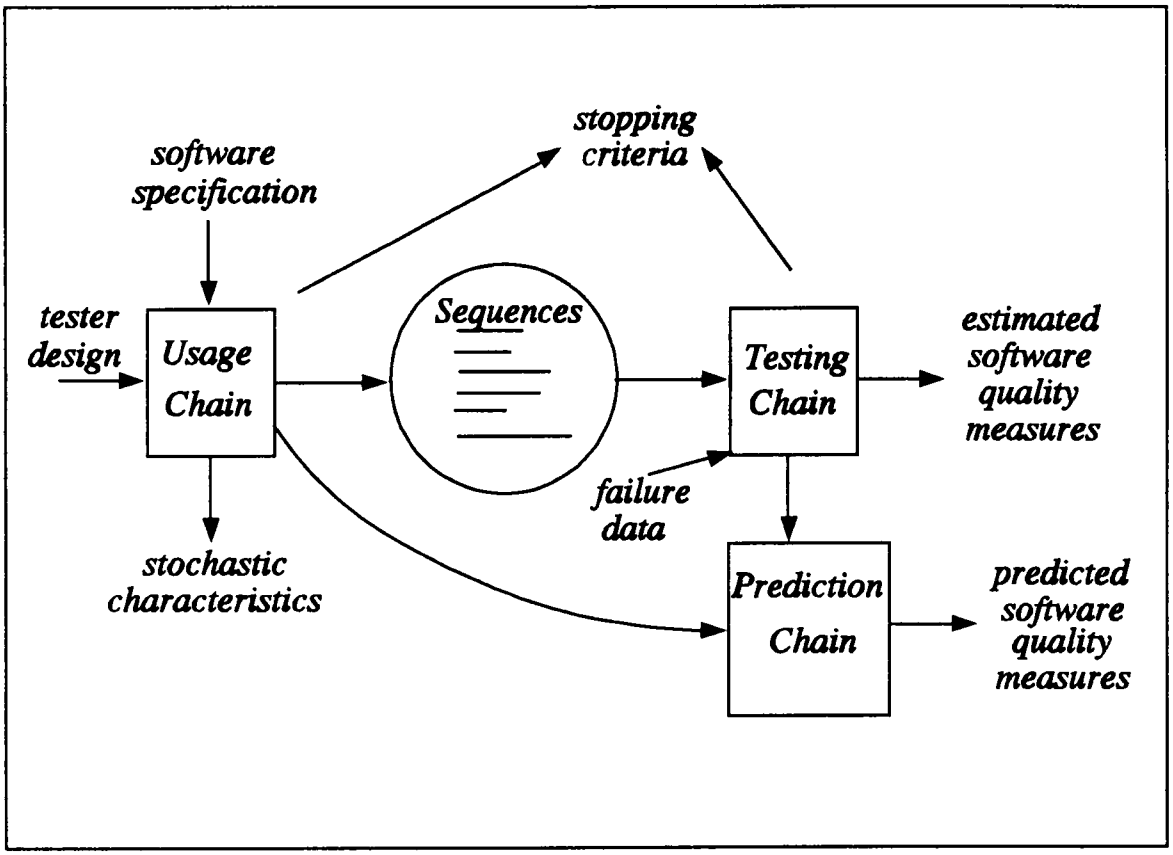


Figure 1.1 : Overview of the Testing Model

The document is organized as follows. Chapter 2 presents the mathematical background necessary to derive equations and prove results in later chapters. Although no new results are developed in this chapter, many concepts from various sources are gathered and organized. Specifically, the interpretation of elements of probability theory which have meaning in software testing are presented here. Chapter 3 develops a top down approach for constructing a Markov usage model. Quantities important to software engineers are defined and derived using the model. Chapter 4 shows how the usage model is used to conduct testing and how it evolves into a discrete reliability model. The statistical properties of the model are explored. Chapter 5 presents a case study employing the new techniques. The final chapter, Chapter 6, gives a synopsis of the work and suggests future research problems in statistical software testing.

Chapter 2

Mathematical Principles - Markov Chains

2.1 Introduction

This chapter presents concepts from the theory of Markov processes that are important in the development of the results of this research. For a complete discussion of Markov chains the reader is referred to Feller [22] and Kemeny and Snell [33].

2.2 Basic Concepts

A *stochastic process* is a description of a probabilistic experiment that takes place in stages and develops over time [49]. A *realization* of a stochastic process is a connected sequence of stages $X_0, X_1,...,X_n$. In order to define the probability law governing the stochastic process, the joint probability $p(X_0, X_1,...,X_n)$ must be known in its entirety. For an arbitrary stochastic process,

$$p(X_0,X_1,...,X_n) = p(X_0)\, p(X_1|X_0)\, p(X_2|X_0,X_1) \cdots p(X_n|X_0,...,X_{n-1}) .$$

In words, the outcome of a particular stage of the can depend upon the outcome of any or all of the previous stages. A *Markov chain* is a stochastic process in which the outcome of stage n depends only upon the outcome of stage n-1. So we have

$$p(X_0,X_1,\ldots,X_n) = p(X_0)\, p(X_1|X_0)\, p(X_2|X_1)\cdots p(X_n|X_{n-1}) ,$$

as the probability law for a Markov chain. This conditional independence is referred to as the Markov property. A process which obeys this Markov property is called a *first order* Markov chain. The Markov chain useful for the purposes of this research is defined as follows (the stages of the Markov chain are referred to as states). A *finite state, discrete parameter, time homogeneous, Markov chain* is a 3-tuple (S,δ,S_0) where S is a finite set of states, δ is a probabilistic transition relation in $(S \times [0,1]) \times S$ such that all transitions obey the Markov property and $S_0 \in S$ is the initial state. Furthermore, the transition relation δ remains fixed with time, which allows reference to the first state of a sequence as X_0 without loss of generality. δ is often expressed in matrix form with the states as indices and transition probabilities as entries. Thus the probability of a transition from the current state i to some next state j will be the matrix entry in the ith row and the jth column, denoted p_{ij}. The matrix is square since each state is both a row and column index. In addition, all rows of the matrix are non-negative and sum to one. This matrix, denoted M, is called the *transition matrix* of the Markov chain.

Markov chains are ergodic stochastic processes. A stochastic process is said to be *ergodic* if it has the property that sample averages formed from a sufficiently long realization approximate the actual ensemble averages of the process [49]. The approximation becomes increasingly more accurate (in the sense of weak or strong large number laws) as the length of the realization increases. The subject of such large number effects for Markov chains is discussed in detail in section 2.3.1.

Once a Markov chain is defined, the next step is to classify the states according to whether it is possible to reach one state from another. If $i, j \in S$, then j is said to be *accessible* from i if $p_{ij}(n) > 0$ for some $n \geq 1$. In this definition, n denotes the number of steps to get from

i to j. If $i, j \in S$ and each is accessible from the other, then i and j are said to *communicate*. If $i \in S$ and i does not communicate with any other state, then i is called a *nonreturn* state.

The communication relation partitions the states into classes (called communicating classes). Two states are in the same communicating class if and only if they communicate. Thus the Markov chain can be partitioned into non null communicating classes and a set of nonreturn states. A communicating class C is *closed* if for every $i \in C$ and $j \notin C$, $p_{ij}(n) = 0$ for every n. Once a closed communicating class is entered it cannot be left.

As an example, consider a Markov chain with state set $S = \{1, 2, 3, 4, 5\}$ where state 1 is the starting state. The transition relation, δ, is defined in matrix form as the transition matrix

$$M = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & \frac{1}{10} & \frac{9}{10} & 0 & 0 \\ 0 & \frac{3}{5} & 0 & \frac{2}{5} & 0 \\ 0 & 0 & 0 & \frac{1}{3} & \frac{2}{3} \\ 0 & 0 & 0 & \frac{3}{4} & \frac{1}{4} \end{bmatrix}.$$

In this chain, state 1 is a nonreturn state since it cannot communicate with any other state. States 2 and 3 form a communicating class, but it is not closed because state 4 is accessible from this class. However, states 4 and 5 form a closed communicating class, i.e., once the process enters this class it stays there.

An important result concerning the classification of states and their asymptotic behavior is the *zero-one* law. Intuitively, the law states that the probability of an infinite number of visits to a state k, given that the process starts in k, is equal to one only when the probability that the state will occur again is equal to one. Otherwise the probability of an infinite number of visits

to k is zero. The result is formalized in the following theorem. Let f_{kk} be the probability of *ever* returning to state k given that the process starts in state k and let g_{kk} be the probability of an *infinite* number of visits to k given that the process starts in k, then

$$g_{kk} = \lim_{n \to \infty} f_{kk}^{n} .$$

The proof of this theorem is found many places in the literature (for example Feller [22]). The idea is that a state with a probability of return less than one will only appear in the realizations a finite number of times. Such a state is called *transient.* A transient state is either a nonreturn state or part of a nonclosed communicating class. States for which the probability of return is equal to one are called *recurrent* and are members of a closed communicating class. Thus each state in a Markov chain is either transient or recurrent and its number of appearances in realizations of the Markov chain is governed by the zero-one law.

2.3 Irreducible Markov Chains

A Markov chain is called *irreducible*[1] when the state set S forms a single closed communicating class. That is, each state in the chain is recurrent. The knowledge that a Markov chain is irreducible gives considerable computational power to the developer. Such chains display asymptotic behavior that can be computed in terms of random variables' expected values and other moments. These properties allow for deep characterization and understanding of the process that the Markov chain represents.

[1]Irreducible chains are also called *ergodic* or *recurrent* by other authors. This document uses the terminology of Feller [22].

2.3.1 Asymptotic Behavior

The transition matrix for a Markov chain represents the one step transition probabilities between the states. Computation of multi-step transition probabilities for an irreducible chain can be accomplished by taking powers of the transition matrix. If M is the transition matrix of an irreducible Markov chain, then the matrix M^2 contains the two step transition probabilities. That is, $p_{ij}(2) \in M^2$ is the probability of a transition from state i to state j in exactly two steps. Higher powers of the transition matrix are similarly interpreted.

The powers of the transition matrix of an irreducible chain will either converge in the limit or oscillate according to some cycle. In the former case the chain is called *aperiodic*. An irreducible and aperiodic Markov chain has a *long run distribution* that can be computed by taking powers of the matrix until convergence as in equation 2.1.

$$M^{conv} = \lim_{n \to \infty} M^n \tag{2.1}$$

The matrix M^{conv} will be a valid transition matrix. Furthermore, all rows will be the same since the starting state has no impact on the long run statistics of an irreducible, aperiodic chain. Thus, the long run distribution of M is any row vector of M^{conv}.

If the powers of the matrix oscillate, then the chain is periodic. An irreducible Markov chain with state set S is *periodic* if a return to some state can be made only at fixed intervals of time greater than one. In order to compute the long run distribution of an irreducible, periodic chain, one must sum successive powers of the matrix to attain mean convergence as in equation 2.2.

$$M^{conv} = \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} M^i \tag{2.2}$$

It is often difficult to determine the periodicity of a Markov chain. However, this is a very important issue for users of Markov chains. The following property is frequently useful in the identification of periodicity in a Markov chain. An irreducible chain is aperiodic if $p_{kk} > 0$ for any state in S (see [22] for proof). Intuitively, when p_{kk} is greater than zero, a single transition loop exists at state k. This loop can be traversed *any number of times* destroying all cyclic behavior of the chain.

An irreducible Markov chain will also have a *stationary distribution.* The stationary distribution is the vector π computed by solving the matrix equation

$$\pi = \pi M \ . \tag{2.3}$$

It can be shown that for an irreducible Markov chain the stationary distribution is unique and furthermore, it is equal to the long run distribution.

The asymptotic behavior of irreducible Markov chains obeys both the weak and strong laws of large numbers. The weak law of large numbers [18,22,33] states:

$$\lim_{n\to\infty} \ prob\left(\left|\frac{o_j^{(n)}}{n} - \pi_j\right| > \varepsilon\right) \to 0 \ , \tag{2.4}$$

where $\varepsilon>0$ and $o_j^{(n)}$ is the random variable that counts the occurrences of state j in n steps through the Markov chain. The weak law describes a probability in the limit, i.e., as the length of a realization is run to infinity, the probability that the expected state occupancy, π_j, computed analytically, and the observed relative occupancy count, o_j/n, will not match within ε approaches zero.

The strong law of large numbers [18] lends even greater confidence in the long run statistics observed in an irreducible, aperiodic Markov chain. The law states:

$$prob \quad \lim_{n\to\infty} \left(\left| \frac{o_j^{(n)}}{n} - \pi_j \right| > \epsilon \right) \to 0 \; . \tag{2.5}$$

This strong law describes a limit with certainty, i.e., for sufficiently large n, the statistics associated with any realization of length n will match the statistics computed analytically from the transition matrix (except technically for realizations occurring with probability 0). Note that the strong law implies the weak law. These large number laws allow one to interpret the entry π_j in the stationary distribution as the portion of time spent in state j in the long run in any realization.

2.3.2 Further Computations on Irreducible Markov Chains

The structure of the Markov chain makes possible the computation of properties of random variables that may be of interest in certain applications. This section develops several equations that will be used later as computations on Markov chains that represent software usage. The first of these is the expected number of transitions until the first occurrence of state k given that the process started in state j. This is called the mean first passage, m_{jk}, and is computed by solving the system of equations represented by[3]

[3]The proofs of the propositions presented in this chapter are widely known and are not reproduced here. The interested reader is referred to Feller [22] for an algebraic treatment and to Kemeny and Snell [33] for a matrix oriented approach.

$$m_{jk} = 1 + \sum_{i \neq k} p_{ji}\, m_{ik} \,. \tag{2.6}$$

The second moment of the first passage is

$$m_{jk}^{(2)} = 2m_{jk} - 1 + \sum_{i \neq k} p_{ji} m_{ik}^{(2)} \,. \tag{2.7}$$

The variance is $m_{jk}^{(2)} - (m_{jk})^2$. Letting $j=k$ gives the mean first recurrence m_{kk}. This is the expected number of transitions until a state recurs. The calculation proves to be much easier by using the entry in the stationary distribution for state k. This value is calculated by

$$m_{kk} = \frac{1}{\pi_k} \,. \tag{2.8}$$

There is an important ratio property in the stationary distribution, namely, $\pi_j / \pi_i = m_{ii} / m_{jj}$ is the expected number of occurrences of state j between recurrences of state i.

2.3.3 Typical Sequences

The concept of entropy [4] for stochastic processes is important for any formulation involving Markov chains. Entropy is an information theoretic measure that quantifies uncertainty in a stochastic source. For an arbitrary stochastic source, X_n, the source entropy is computed using conditional probabilities as

$$H(X_n) = \lim_{k \to \infty} H(X_k \,|\, X_1, \cdots, X_{k-1}) \,, \tag{2.9}$$

which can be written as the average joint event entropy,

$$H(X_n) = \lim_{k\to\infty} \frac{1}{k} H(X_1,\cdots,X_k) \ . \tag{2.10}$$

This value is not analytically computable for an arbitrary stochastic process. However, when the stochastic source is an irreducible Markov chain with stationary distribution π and transition probabilities p_{ij}, the source entropy is computed as [4]

$$\begin{aligned} H(X_n) &= \lim_{k\to\infty} H(X_k|X_{k-1}) \\ &= -\sum_i \pi_i \sum_j p_{ij} \log p_{ij} \ . \end{aligned} \tag{2.11}$$

This single number is related, exponentially, to the number of sequences that are statistically typical realizations of the Markov chain. That is, a Markov chain has a set of *typical sequences* whose ensemble statistics closely match the statistics of the chain. The probability of a typical sequence for a Markov chain is (assuming natural logarithms) [1]

$$p(s_t) = e^{-NH} \ , \tag{2.12}$$

where N is the length of the sequence and is sufficiently large for large number effects to be true. Thus higher source entropy implies an exponentially greater number of typical sequences, i.e., more sequences exist because of the uncertainty present in the model.

The implication of higher (or lower) source entropy in applications where a Markov chain is used as a sequence generator is that a greater (or smaller) number of sequences must be generated in order to accurately describe the Markov source. One can see from equation 2.11 that low entropy is attributed to a biased stationary distribution or from uneven distributions over the exit arcs of the states of the chain.

2.4 Absorbing Markov Chains

An *absorbing* Markov chain consists of a nonempty set of transient states and a finite number of closed communicating classes (also called absorbing subsets). Typically, such a process goes through a transient phase and is then absorbed into one of the closed communicating classes. It can be shown that absorption occurs with probability one. One interesting result, therefore, is into which subset the process is absorbed. Let y_{iC} be the probability of absorption into the closed communicating class C, given starting in state i. Then

$$y_{iC} = \sum_{l \in C} p_{il} + \sum_{k \in \tau} p_{ik}\, y_{kC} , \tag{2.13}$$

where τ is the set of transient states. This can be simplified for applications in which the absorbing subset of interest consists of a single state. In this situation, the computation is reduced to the probability of absorption in the single state l. This is computed by

$$y_{il} = p_{il} + \sum_{k \in \tau} p_{ik}\, y_{kl} . \tag{2.14}$$

Another random variable of interest for absorbing Markov chains is the number of transitions until absorption given a starting state j. Its expectation is given by

$$m_j = 1 + \sum_{k \in \tau} p_{jk}\, m_k . \tag{2.15}$$

and its second moment is

$$m_j^{(2)} = 2m_j - 1 + \sum_{k \in \tau} p_{jk} m_k^{(2)} \ . \tag{2.16}$$

This allows us to compute the variance as $m_j^{(2)} - (m_j)^2$.

Besides counting transitions, one may wish to count the number of times a particular state occurs before absorption given a start state. The expectation is

$$m(j|i) = \sum_{k \in \tau} p_{ik} \, m(j|k) + \begin{cases} 1 & \text{if } i=j \\ 0 & \text{otherwise} \end{cases} . \tag{2.17}$$

In addition to the above computations involving absorbing chains, limited asymptotic results can be obtained. A common technique is to extract a closed communicating class from an absorbing chain. This class can be studied on its own as an irreducible Markov chain. It will have the asymptotic distributions according to its periodicity. Thus, one can calculate a conditional stationary distribution. That is, an absorbing Markov chain will have the stationary distribution π^C conditioned on entering the closed communicating class C. In this case all transient states as well as recurrent states in other closed communicating classes besides C will have zero entries in this distribution.

2.5 Additional Results

The results presented in this chapter are dependent upon the classification of the Markov chain. That is, certain computations apply for absorbing chains and others for irreducible chains. This section is concerned with converting an absorbing chain to an irreducible chain and vice versa. This conversion allows one to expand the computations possible by using one classification of chain to study the other. For example, it is possible to use absorbing chain theory to study an

irreducible chain by forcing absorption to occur in the original irreducible chain. If this technique is applied correctly, expected values are preserved and the legitimacy of the computation is not violated.

An absorbing Markov chain can be converted to an irreducible chain by redirecting the arc or arcs causing absorption to the starting state of the chain. The obvious advantage of such a conversion is that long run statistics for the chain can now be calculated. Intuitively, such statistics represent "many runs" of the original absorbing chain. Each time absorption occurs, the chain is restarted and the collection of long run statistics can be accomplished.

Conversion from an irreducible chain to an absorbing chain can also yield valuable computations. Such a transformation is useful when one is interested in a more detailed behavior of the process before entering a particular state i. The solution is to make i an absorbing state and then use the equations developed in section 2.4 to compute items of interest. These include the probability of absorption into i and the number of transitions before absorption into i. Thus a state can be isolated and the process studied in detail. These techniques are used throughout this research.

Chapter 3

The Usage Analysis

3.1 Introduction

The first step in building a statistical testing model for a software system is to define the underlying random experiment for the test. In this chapter a *usage analysis* is developed that uses a Markov chain to define the sample space and the event space of the software system. The Markov chain induces a usage distribution on the sample space that is available analytically from the chain. The chain also delivers a set of sequences that can be executed as test cases in order to determine the quality of the software. Using this model, it is possible to compute a statistical description of the test cases before any testing begins, indeed, before any coding begins. This description gives testers advance knowledge of which functions of the specification will get the most attention from the test cases and an insight into the amount of testing that will be required for certification.

The starting point for the usage analysis is the software specification document. In most cases, this document is a natural language description of the software's intended function. The specification enumerates the input domain of the software and the transition rules which determine the external behavior. The informal and often ambiguous nature of these descriptions require that the usage analysis be a design step as opposed to an algorithm. However, by following the top down method presented in this chapter to discover events and define relations one can organize

the information into a formal usage model. Specifically, the usage model is a first order Markov chain. This chain is called the *usage Markov chain* and is used to guide the software testing process. Technically, all probabilities, expected values, and other computations are conditioned on this usage chain.

As mentioned in Chapter 1, the Markov chain has many beneficial features that make it an attractive model. However, it is limited in its ability as a sequence generator by its classification as a formal language generator. The realizations of a Markov chain are probabilistically generated sequences and as such can be analyzed in terms of the formal language to which they belong. The Markov chain is the stochastic equivalent to a regular grammar [61]. As a language generator it is capable of defining sequences of the generic form uv^iw for $i \geq 0$ where u and w are strings of zero or more symbols, v is a string of at least one symbol, and v^i means i copies of v associated with looping through recurrent states. A probabilistic nonregular, context-free language (represented by a full multitype branching process [22]) is necessary to fully describe the probabilistic sequence structure of the more complex generic form uv^iwx^iy for $i \geq 0$ where u, w, and y are strings of zero or more symbols, v and x are strings of at least one symbol each, and looping through i copies of v exactly corresponds to subsequent looping through i copies of x. Thus nonregular, context-free models must be used to accurately represent sequences in which dual tracking of symbols is necessary [26]. It may be that for some classes of software a stochastic, nonregular model is necessary for a precise description of its formal language. However, such a language theoretic analysis of the underlying formal language of a software specification is not feasible until these documents become formal descriptions in discrete mathematical terms. Until this happens, each software system must be individually considered to determine the applicability of the Markov chain.

3.2 Constructing the Usage Model

The usage Markov chain has a two phase construction. In the *structural phase* the states and arcs of the chain are established and in the *statistical phase* the transition probabilities are assigned.

At the highest level, software usage can be described by the three state transition diagram depicted in Figure 3.1. The software is invoked, a cycle of usage ensues and eventually the software is terminated and control is returned to the operating system. Alternatively, "power on" invokes the software and usage occurs until "power off" causes termination as in real-time systems. The usage state can be expanded by consulting the specification for all possible actions that a user may pursue immediately following invocation. A new state is created for each action with an arc from the invocation state as its incoming arc. Exit arcs from the newly created states are established by determining (from the specification) which actions can be performed following the state in question. These arcs can be directed to pre-existing states or they can lead to the creation of new states in the same manner. This method of creating states yields a directed graph whose states are labeled with entries drawn from the input domain of the software. The arcs of the graph define an ordering which the inputs must obey.

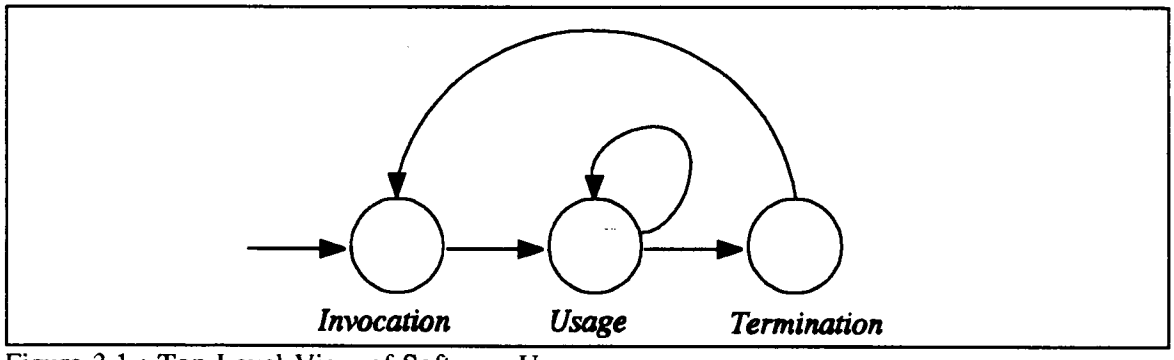


Figure 3.1 : Top Level View of Software Usage

As an example, consider the window pictured in Figure 3.2 with behavior specified in Table 3.1. Although this is a significantly abbreviated specification, it is sufficient to describe the function for the purpose of this example. Invoking the software causes the window to be displayed, thus a state labeled *Window* is created accessible from the invocation state. In order to determine the exit arcs from this new state, each possible action from the *Window* state is considered. These are: *Maximize* (▲), *Minimize* (▼), *Move*, *Size*, and *Close*. The creation of these states yields the transition diagram of Figure 3.3. Each of these states is analyzed in turn to determine the placement of their respective exit arcs. The state labeled *Maximize* simply returns the user to a fully functional window. Thus the single exit arc from this state is back to the pre-existing *Window* state. The state labeled *Minimize* requires another series of states to be created to model the icon behavior. Establishing the state *Icon* as a neighbor of *Minimize*, we note that only one action is specified from the *Icon* state, namely *Restore* (which simply returns the user to the window). Thus the series, on single transitions is: *Window*, *Minimize*, *Icon*, *Restore*, *Window*. *Move* and *Size* require mouse activity to be modeled. When either is selected, a directive to drag the mouse (*Drag Mouse*) is given followed by the direction (if one desires more detail, a distance could also be supplied). Since the user may choose to change directions of the mouse movement, the exit arcs from the direction states return to *Drag Mouse* in order to model this behavior accordingly. When all movement is complete, the window is restored (marked by a return to the *Window* state). Finally, the state labeled *Close* simply terminates the window. Thus its single exit arc is to the termination state. The complete transition diagram appears in Figure 3.4.

This example illustrates the structural phase of the usage Markov chain construction. The structural phase yields a Markov chain with states defined to be stimuli drawn from the input domain of the software. The arcs define an ordering upon the state set. However, some software

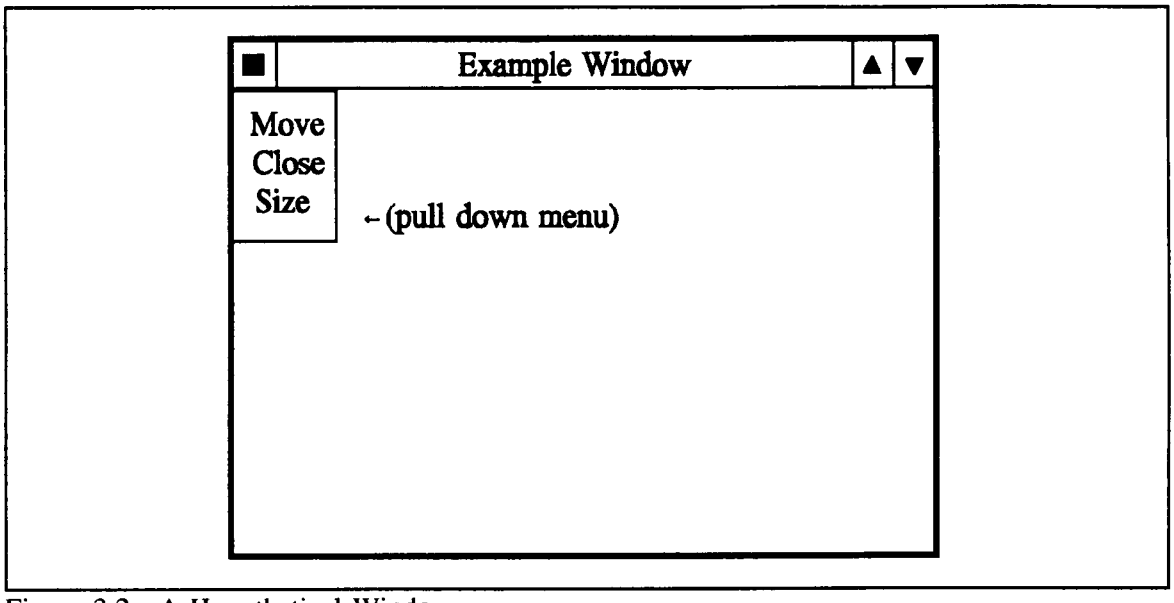


Figure 3.2 : A Hypothetical Window

Table 3.1 : Example Software Specification

Stimulus	Response
Invocation	Place the window of Figure 3.2 on the screen
Select ▲	Expand the window dimensions to cover the entire area of the screen
Select ▼	Remove the window and replace it with its corresponding icon
Select ■ and choose *Move* from the pull down menu	Move the window as directed by the mouse input (obeying screen boundaries)
Select ■ and choose *Size* from the pull down menu	Size the window as directed by the mouse input (obeying minimum and maximum limits)
Select ■ and choose *Close* from the pull down menu	Remove the window from the screen
Select the icon and release	Remove the icon from the screen and restore the window

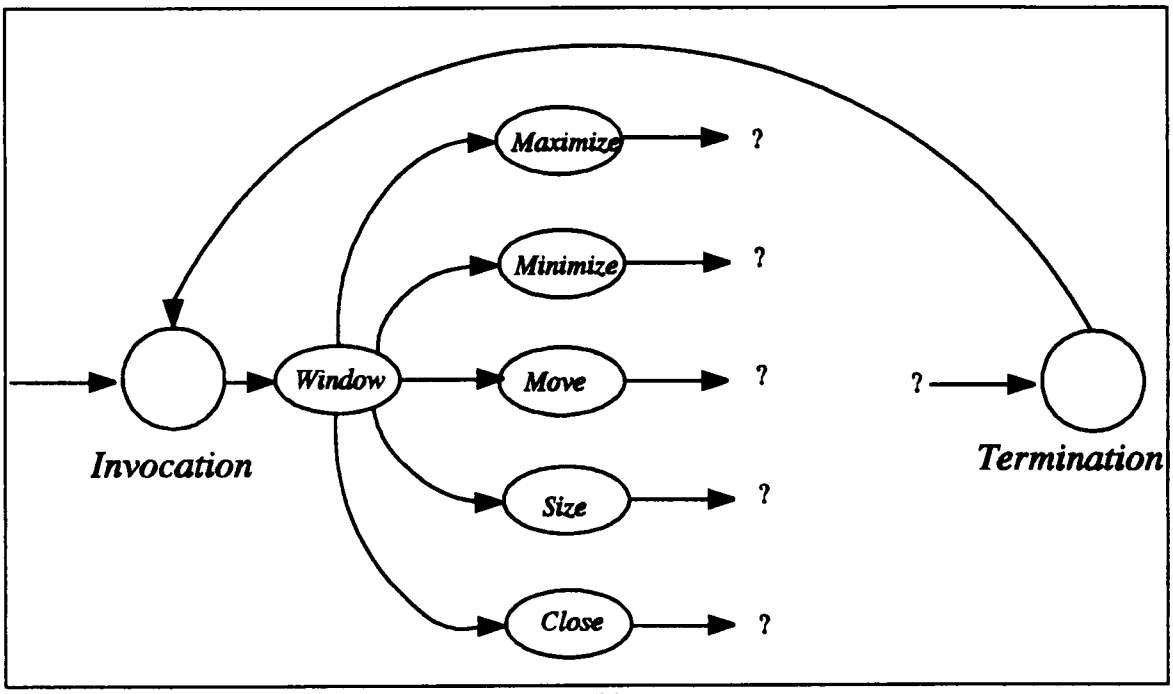


Figure 3.3 : Expansion of the Top Level Usage Diagram

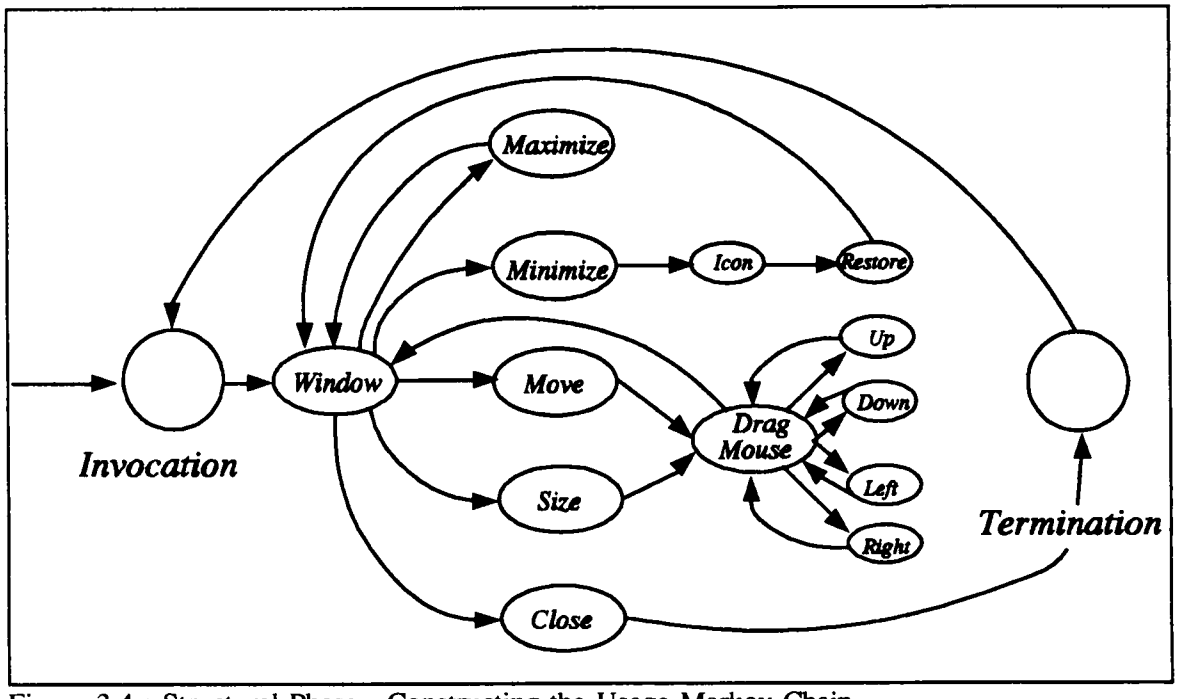


Figure 3.4 : Structural Phase - Constructing the Usage Markov Chain

systems are not as straightforward as this example. It is often the case that complex specifications contain a transition from state i to state j which depends upon the occurrence of one of several alternative previous states. This information dealing with relevant history of use is a violation of the Markov property. Technically, the specification describes a higher order Markov chain. Such a chain can be converted to a first order chain by a technique called *state splitting* [22]. State splitting is achieved by splitting state i into multiple states, one new state for each dependency noted at state i. Each of the states created in this manner may retain the same label or obtain an updated label that describes the dependency. A detailed example of state splitting is given in the case study of Chapter 5.

The structural phase is complete when usage (as defined by the specification) is completely modeled. At this stage the assignment of transition probabilities is the last step toward completion of the chain. This step constitutes the statistical phase of the usage model construction. There are three approaches to the statistical phase. The first is called the *uninformed* approach. It consists of assigning a uniform probability distribution across the exit arcs for each state. This approach, which maximizes the entropy across the exit arcs, is useful when no information is available to support a more informed choice. The second approach, called the *informed* approach, is used when some actual usage sequences are available. These sequences could be captured events from a prototype or prior version of the software. The events in the sequences have corresponding states in the usage chain. The event transitions can be counted and the resulting relative frequencies can be used to estimate the transition probabilities in the usage chain. The third approach, called the *intended* approach, is similar to the informed approach except the sequences are obtained by hypothesizing runs of the software by a specific type of user, e.g., a skilled and reasonable novice or an expert. Relative frequency estimates of the transition probabilities are computed from the symbol transition counts as in the informed approach. An

example of the counting technique appears in Table 3.2.

The transition probabilities can be stratified in various ways in order to emphasize special aspects of system behavior. For example, one may wish to raise the probability of states that model critical functions regardless of their frequency of use. This is achieved through manipulating the arc probabilities associated with paths from the initial state to the state (or states) in question. Changing transition probabilities yields a new chain with its own set of statistics and typical sequences. One may wish to perform multiple statistical tests against these various usage models. Such tests are always possible and often desirable.

The irreducible structure of the usage chain induces a probability distribution (the usage distribution) on the input domain of the software. That is, each state i has steady-state probability π_i which can be analytically computed and is a direct consequence of the structure and statistics of the usage Markov chain. Even the uninformed method of assigning transition probabilities induces, in general, a nonuniform usage distribution on the input domain. The usage distribution is given, along with several other analytical results, in the next section.

3.3 Analytical Results from the Usage Chain

The usage Markov chain is the source of test sequences for the software. A *statistical test case* is any connected state sequence of the usage chain that begins with the invocation state and ends with the first occurrence of the termination state. Thus, a test case is any realization of the usage chain of the form *Invocation, ..., Termination.* The process of generating statistical test cases is easily automated using a random number generator[4] and any high level language. One

[4]Care should be exercised in the selection and use of a random number generator.

Table 3.2 : Statistical Phase - Assigning the Transition Probabilities

From-State	To-State	Frequency	Probability
Invocation	*Window*	6	1
Window	*Maximize*	1	1/12
Window	*Minimize*	1	1/12
Window	*Move*	2	1/6
Window	*Size*	2	1/6
Window	*Close*	6	1/2
Maximize	*Window*	1	1
Minimize	*Icon*	1	1
Icon	*Restore*	1	1
Restore	*Window*	1	1
Move	*Drag Mouse*	2	1
Size	*Drag Mouse*	2	1
Drag Mouse	*Window*	4	4/15
Drag Mouse	*Up*	1	1/15
Drag Mouse	*Down*	5	1/3
Drag Mouse	*Left*	3	1/5
Drag Mouse	*Right*	2	2/15
Up	*Drag Mouse*	1	1
Down	*Drag Mouse*	5	1
Left	*Drag Mouse*	3	1
Right	*Drag Mouse*	2	1
Close	*Termination*	6	1
Termination	*Invocation*	-	1

Captured or hypothesized sequences:

1. *<Invocation><Window><Maximize><Window><Close><Termination>*
2. *<Invocation><Window><Minimize><Icon><Restore><Window><Close><Termination>*
3. *<Invocation><Window><Move><Drag Mouse><Down><Drag-Mouse><Right><Drag Mouse><Down><Drag Mouse><Window><Close><Termination>*
4. *<Invocation><Window><Size><Drag Mouse><Left><Drag-Mouse><Up><Drag Mouse><Left><Drag Mouse><Window><Close><Termination>*
5. *<Invocation><Window><Move><Drag Mouse><Down><Drag-Mouse><Left><Drag Mouse><Down><Drag Mouse><Window><Close><Termination>*
6. *<Invocation><Window><Size><Drag Mouse><Down><Drag-Mouse><Right><Drag Mouse><Window><Close><Termination>*

has simply to step through the states of the chain based on the transition probabilities. The sequence of states visited becomes the test case. Thus, once a usage model is constructed, any number of test cases can be obtained automatically from the model. Ideally, the usage model is ready in advance of the software to be analyzed by testers in order to determine whether it generates feasible sequences for testing the software. A major advantage of using the Markov chain as the stochastic model for software usage is that the statistical profile of the test cases can be computed analytically from the usage chain. This profile gives insight into the usage model and the test cases that will be used to certify the software. First, we analyze the usage chain in its irreducible form and then turn to absorbing chain theory to characterize further the usage model and the sequences it generates.

Building the usage chain in the manner described above yields an irreducible Markov chain. Each state communicates with every state in the chain, making the state set a single closed communicating class. As described in Chapter 2, the mean first passage times, m_{jk}, can be computed from such a chain using equation 2.6. These values are interpreted as the expected number of states visited from state j until the first occurrence of state k. In terms of test case generation, when state j appears in the sequences, one should anticipate m_{jk} states to appear on average before the first appearance of state k. This value gives insight into the relationship between states in the usage chain. The mean first passage values (and variances) from the *Invocation* state are enumerated in Table 3.3 for the example chain. Note that the mean first passage from *Invocation* to *Termination* is the expected test case length. We denote this value by

$$x = m_{Inv,Term} \; . \tag{3.1}$$

For the above example, x=9.67 states with variance equal to 123.44 states. An interesting use of

the mean first passage times is to compare m_{jk} to the expected test case length. If $m_{jk}>x$ we would anticipate the occurrence of state j followed by state k to require multiple test cases. Otherwise, if state j appears we would anticipate state k to appear before the test case ends. Table 3.3 shows other interesting data as well. For example, the probability of transition (*Invocation,Maximize*) is equal to the probability of transition (*Invocation,Minimize*) but $m_{Inv,Max}$=64 states and $m_{Inv,Min}$=62. This seeming discrepancy arises from the fact that the computation of the mean first passage uses the accessibility relation defined by the arcs of the chain. Thus *Minimize* and *Maximize* are reached from *Window* with the same transition probability but they are accessible through different paths accounting for the difference in the first passage times.

It is sometimes the case that the mean first passage values give testers reason to question and perhaps change the usage model. For example, if state j and state k are related in some manner (e.g., both states are necessary to perform a specific function of the software) but the transition probabilities assigned to the arcs of the chain have created an m_{jk} value that does not indicate such a close relationship, then it may be justifiable to adjust appropriate transition probabilities and recompute m_{jk} to determine the effect of the adjustment. This might represent a shift from uninformed probabilities to informed probabilities or possibly necessitate additional information gathering about potential use. Hopefully, a more acceptable value can be obtained. However, one must always be careful to reexamine *all* the analytical results after such a change to ensure that some other equally desirable attribute has not been destroyed.

The irreducible usage chain can also be analyzed to determine its asymptotic behavior. The *usage distribution*, π, is the limiting distribution for the states of the chain computed using equation 2.3. This vector represents the probability distribution induced on the input domain of the software by the Markov chain. The strong law of large numbers allows us to interpret each entry π_i as the amount of time the usage chain spends in state i in the long run. In terms of the

test cases, this is the expected appearance rate of state i asymptotically. Thus, if $\pi_i > \pi_j$, state i will appear a proportionally greater number of times than state j in the test cases in the long run. Subsets of states can be similarly compared. For example, the π values for states associated with a particular function of the software (e.g., states in the same window) can be summed to obtain a long run value for that subset of states. This value can be compared to other subsets (not necessarily disjoint) to obtain information about usage patterns of functions of the software.

The ratio properties mentioned in Chapter 2 have significance for software testers. The mean recurrence time, m_{ii}, for state i is computed by $1/\pi_i$. Because of the special structure of the usage chain we can verify the expected test case length using this quantity. For the example usage chain, $m_{Inv,Term} = m_{Inv,Inv} - 1$ because each occurrence of the termination state is followed by the invocation state with probability one. The m_{ii} values (and their corresponding variances) appear in Table 3.3 for the example usage model. The ratio π_j / π_i has a valuable interpretation for software testers. For example, since each occurrence of the invocation state marks a new test case, we can compute the expected number of sequence changes between occurrences of state i as π_{Inv} / π_i. A similar result is the expected number of occurrences of state i in a test case, π_i / π_{Inv}. These ratios are enumerated in Table 3.3 for the example usage model. Recall that the vector π is an asymptotic characteristic of the usage chain. Any computation directly involving an entry from π represents long run behavior and must be interpreted as such.

Absorbing chain theory also yields important quantities in the analysis of the usage model. An absorbing usage chain is achieved by replacing the arc (*Termination,Invocation*) with a new arc from the termination state back to itself. This creates an absorbing Markov chain with the *Termination* state as the single absorbing state. We can now use the computations developed in Section 2.4 to analyze this chain. We compute $m(i|Inv)$, the expected number of occurrences of each state before absorption starting with invocation (equation 2.17). These values along with

Table 3.3 : Analytical Results from the Irreducible Usage Chain

State i	$m_{Inv,i}$	$m_{Inv,i}^{(2)}-(m_{Inv,i})^2$	π_i	m_{ii}	$m_{ii}^{(2)}-(m_{ii})^2$	π_i/π_{Inv}	π_{Inv}/π_i
Invocation	10.67	123.44	0.093750	10.67	123.44	1	1
Window	1	0	0.187500	5.33	19.06	2	0.5
Maximize	64	4410	0.015625	64	4410	0.167	6
Minimize	62	4154	0.015625	64	4154	0.167	6
Icon	63	4154	0.015625	64	4154	0.167	6
Restore	64	4154	0.015625	64	4154	0.167	6
Move	25.5	723.75	0.031250	32	765	0.333	3
Size	25.5	723.75	0.031250	32	765	0.333	3
Drag Mouse	10.5	85.25	0.234375	4.27	36.86	2.5	0.4
Up	73.5	4751.25	0.015635	64	4666	0.167	6
Down	22.3	363.09	0.078125	12.8	277.84	0.833	1.2
Left	30.83	730.36	0.046875	21.33	645.11	0.5	2
Right	41.5	1394.25	0.031250	32	1309	0.333	3
Close	8.67	123.44	0.093759	10.67	123.44	1	1
Termination	9.67	123.44	0.093750	10.67	123.44	1	1

their corresponding variances are listed in Table 3.4. Note that they are the same values obtained from the usage distribution ratio π_i/π_{Inv}. One can often obtain identical results from separate calculations, indicating the preservation of properties of random variables in the conversion of the chain from irreducible to absorbing.

The probability that state i occurs exactly once in a test case, $y_{Inv,i}$, can be computed by forcing absorption at state i (in addition to absorption at *Termination*) and using equation 2.14 to compute the probability of absorption at i. Thus we obtain the probability that i occurs (absorption at i) as opposed to the probability that it does not occur (absorption at *Termination*). These values are also tabulated in Table 3.4 for our example chain. Note that *Window* and *Close*

occur with probability one because they are on every path from invocation to termination. Since making either of these states absorbing effectively blocks the occurrence of the *Termination* state, they act as the only absorbing state. From Chapter 2 we know that absorption occurs in such a state with probability one.

The final result computed from the absorbing chain is the expected number of sequences necessary until state i appears for the first time given invocation. We denote this quantity h_i and derive it by solving the equation

$$h_i \, y_{Inv,i} = 1$$

where the random variable y is the probability that state i occurs in a test case. Solving for h_i yields

$$h_i = \frac{1}{y_{Inv,i}} . \tag{3.2}$$

The largest value in the vector h is the expected number of sequences necessary until the least likely state (or states) appears at least once. This vector appears in Table 3.4 with the rest of the results from the absorbing usage chain.

The last result discussed in this chapter is the source entropy of the usage chain. This value, denoted H, is computed using equation 2.11. The source entropy serves as a comparative measure for usage chains with the same structure but different transition probabilities. For example, suppose one develops usage chains U_1 and U_2 using the same structural analysis but assigns U_1's transition probabilities using the uninformed statistical analysis and U_2's using the informed statistical analysis. Let H_1 and H_2 be the source entropies for U_1 and U_2, respectively. As explained in Chapter 2, when $H_1 > H_2$ one should expect to generate an exponentially greater number of sequences using U_1 to obtain asymptotic behavior than using U_2. The uninformed

Table 3.4 : Analytical Results from the Absorbing Usage Chain

State	$m(i\|Inv)$	$m(i\|Inv)^{(2)}$-$(m(i\|Inv))^2$	$y_{Inv,i}$	h_i
Invocation	1	0	1	1
Window	2	2	1	1
Maximize	0.167	0.194	0.143	7
Minimize	0.167	0.194	0.143	7
Icon	0.167	0.194	0.143	7
Restore	0.167	0.194	0.143	7
Move	0.333	0.444	0.25	4
Size	0.333	0.444	0.25	4
Drag Mouse	2.5	22.5	0.4	2.5
Up	0.167	0.278	0.118	8.5
Down	0.833	3.611	0.27	3.7
Left	0.5	1.5	0.222	4.5
Right	0.333	0.778	0.182	5.5
Close	1	0	1	1
Termination	1	0	1	1

model, U_1, often serves as a good basis for determining how much the informed approach has biased the usage chain. The values for the example are $H_1 = 1.0884$ bits for the uninformed chain and $H_2 = .8711$ bits for the informed chain. As described in Chapter 2, the U_2 source will require fewer sequences to reach asymptotics than U_1 because the sequences from U_2 are higher in information content. Note that it is possible for the informed probabilities to raise the entropy for a given structure. In such a case, usage of the given structure is uniform to the extent that the informed approach requires more sequences to convey its information content.

The source entropy also gives insight into the expected variability of the testing process

as a random experiment. As an experiment, the testing process evolves based on random sequences from the usage chain. The same usage chain could generate different sequences in separate, independent testing experiments (for example, multiple test sites). Higher source entropy implies greater variability in the sequences generated for such experiments.

The analytical results obtained from the usage chain give testers advance knowledge of the resources it will take to conduct suitable testing for the software in question. Such results can serve as an analysis tool for the feasibility of the usage model. For example, testers need to determine if the expected test case length is appropriate for the application. If it is not then adjusting the transition probabilities of the chain may be necessary to obtain an acceptable expected value. The mean first passage times may show that there is too low a probability of specific related events occurring within the same sequence. If this is the case, then one must justify different transition probabilities in the chain to determine an acceptable resolution. Of course, each analytical result will have a specific interpretation for certain applications and some may be more meaningful in one environment than another. However, using a Markov chain to model software usage makes all of the results available for testers to interpret and use as the application warrants. The results are computed and analyzed for an actual software specification in the case study of Chapter 5.

Chapter 4

The Reliability Analysis

4.1 Introduction

Classical reliability modeling proceeds as follows [59]. Engineers obtain test data in the form of failure times, repair reports, etc. This data is then plotted and the curve examined using statistical analysis, engineering judgment, or another mechanism to choose an appropriate failure rate model. The parameters of the model are estimated from the analysis of the graph. The failure rate model yields the failure density function, and the reliability is computed in terms of this function (oftentimes assuming some growth trend in the data). This research follows a similar strategy except that we use the Markov source of the test data to build a model that automatically produces the reliability analysis.

The usage model developed in Chapter 3 delivers a set of sequences that are applied to the software to obtain data that enables statistical certification of the quality of the software. This chapter proposes a method of incorporating the data into a separate Markov chain. This new chain, called the *testing Markov chain*, models the result of executing the sequences as test cases for the software. It is the basis for computing software quality measures and for determining when to stop testing.

4.2 Constructing the Testing Markov Chain

The usage chain is constructed by the test team in parallel with the development of the code that implements the specification by the design team. At some point in the development process, the code is completed by the design team and delivered to the test team for compilation and certification. Well in advance, the usage chain has generated the sequences that will be used in the certification process. The sequences generated by the usage chain serve a dual purpose. They are applied to the software as test cases and also used to construct a new testing Markov chain that evolves according to the outcome of the testing process. The testing Markov chain, denoted T, is created by initializing the frequency model F with the same states and arcs as the usage Markov chain, except that the arc labels are frequency counts instead of probabilities. When testing begins, these counts will all be zero. Each test sequence generated by the usage chain is incorporated into F by updating the arc labels when the sequence is executed as a test case. That is, whenever a transition is made in a test sequence, the corresponding arc count in F is incremented. At any time during testing the frequency counts in F can be converted to relative frequency probabilities in order to obtain T.

To illustrate, consider the example usage chain, U, of Figure 3.4 with the transition probabilities of Table 3.2. As described in Chapter 3, U is employed as a sequence generator. Suppose U generates the following sequences:

1. *Invocation, Window, Maximize, Window, Close, Termination*;
2. *Invocation, Window, Size, Drag Mouse, Down, Drag Mouse, Left, Drag Mouse, Window, Close, Termination*;

3. *Invocation, Window, Move, Drag Mouse, Down, Drag Mouse, Right, Drag Mouse, Up, Drag Mouse, Window, Maximize, Window, Close, Termination;*

4. *Invocation, Window, Close, Termination;*

5. *Invocation, Window, Size, Drag Mouse, Down, Drag Mouse, Window, Move, Drag- Mouse, Left, Drag Mouse, Up, Drag Mouse, Window, Close, Termination.*

The first sequence causes the arcs for *F(Invocation, Window)*, *F(Window,Maximize)*, *F(Maximize,Window)*, *F(Window,Close)*, and *F(Close,Termination)* to be incremented. These arcs correspond to the symbol transitions in the sequence. We also increment the arc *F(Termination,Invocation)* to balance the frequency counts at the *Termination* state. After the first sequence is incorporated, *F* appears as in Figure 4.1 (with the "0" arc labels omitted to improve readability).

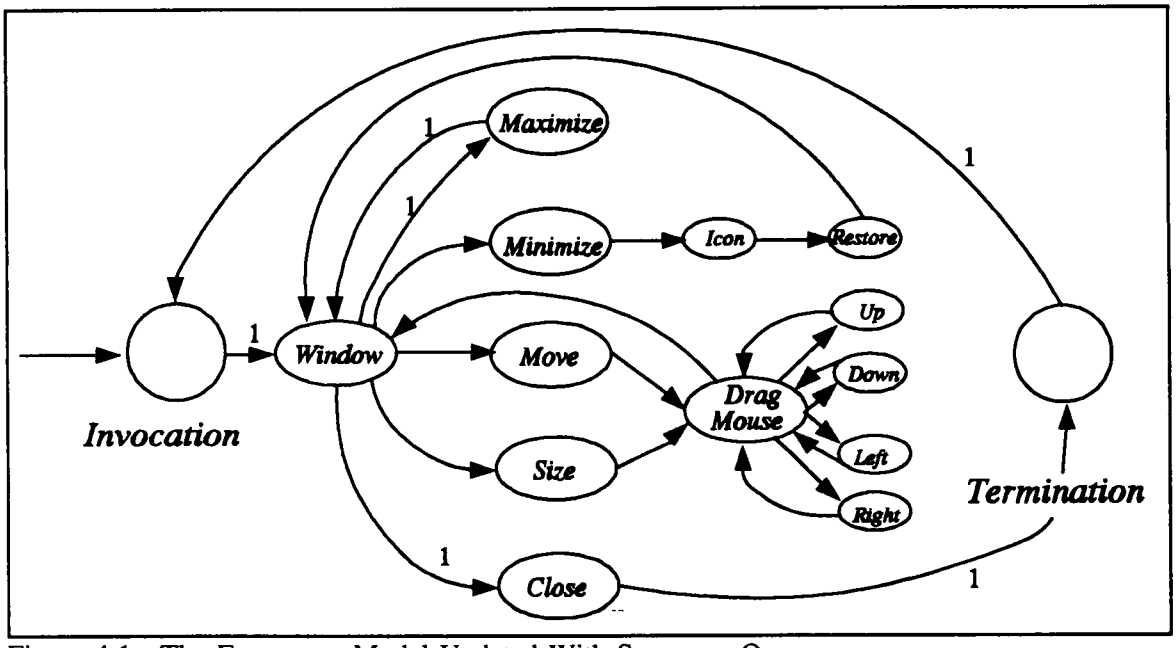


Figure 4.1 : The Frequency Model Updated With Sequence One

The second sequence further updates F as shown in Figure 4.2. After all five sequences, F appears as in Figure 4.3. Normalizing the entries over the respective row sums yields the testing Markov chain, T, at the fifth sequence.

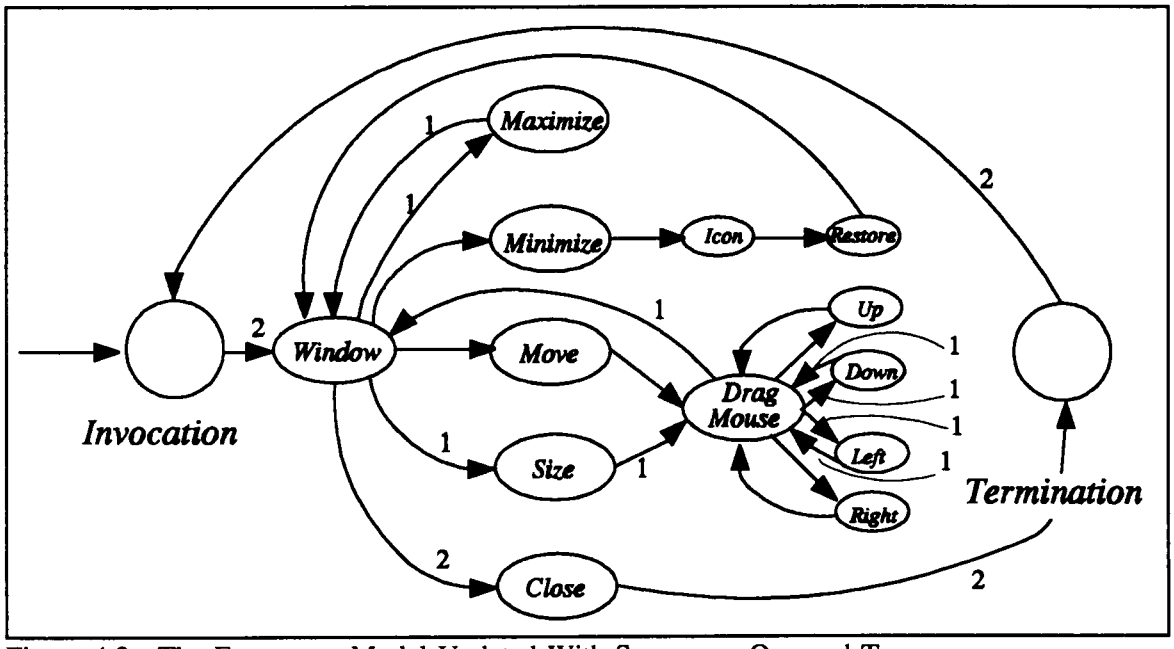


Figure 4.2 : The Frequency Model Updated With Sequences One and Two

4.2.1 Failure Data

Failure data is incorporated into F and T by creating states and arcs to model the failures. In order to systematically incorporate failure information into the testing chain, we assume that failures are unambiguously detected during testing. That is, an oracle exists that removes all uncertainty about when and where failures occur. If a failure is observed while executing state i in some test sequence, then a new state f is created with an arc from i to f labeled with frequency count 1. The exit arc from f follows the execution of the software after the failure has

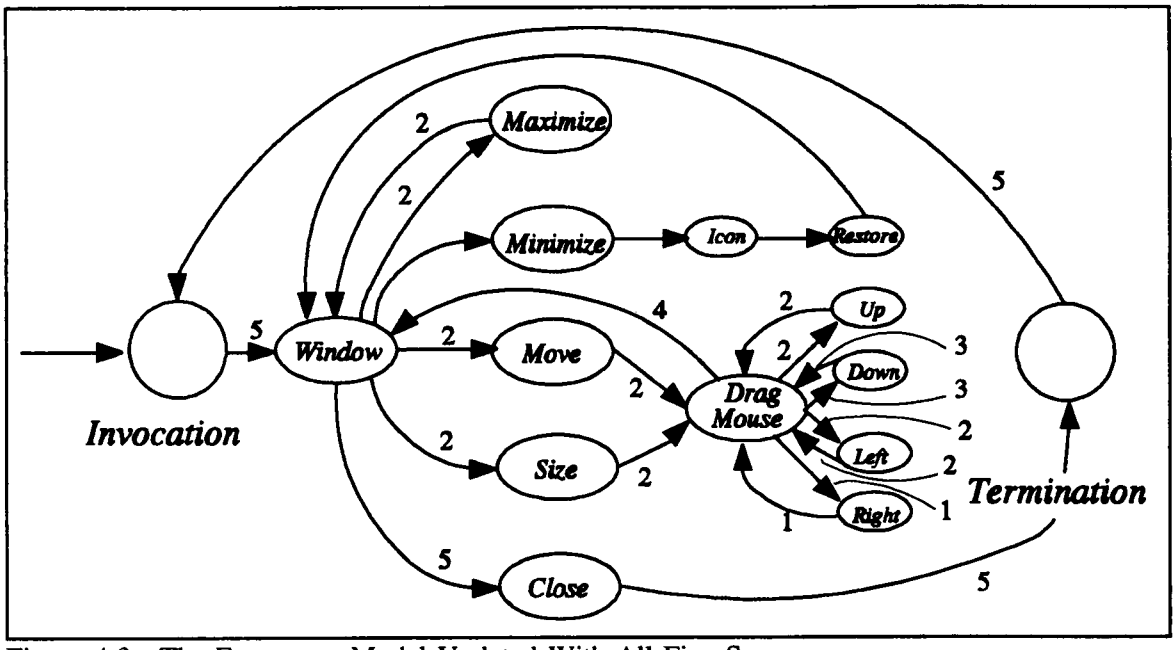


Figure 4.3 : The Frequency Model Updated With All Five Sequences

occurred. In the case that the failure is catastrophic and the software crashes, the arc will be from *f* to the termination state (which represents an exit to the operating system). If the failure is not so serious and it has been determined that testing should continue, then the arc will be from *f* to the appropriate next state in the sequence. In either case, the entry arcs into *f* and its exit arcs must connect *f* to non-failure states. The goal is to modify *F* in exact correspondence to what happened while executing the test cases. All failures must be incorporated into the chain and the execution must be mimicked accordingly. This is necessary to insure that the testing chain accurately represents the testing experience.

To illustrate the inference procedure, suppose that the next sequence generated by *U* is *Invocation*, *Window*, *Minimize*, *Icon*, *Restore*, *Window*, *Close*, *Termination*, and that a catastrophic failure is observed at the *Restore* state. *F* is updated as shown in Figure 4.4. Note that, during testing, the suffix of the sequence after the failure state is not executed. A decision must be made either to rerun the sequence in order to execute every state or to continue testing with a new

sequence. The latter strategy is recommended in order to ensure that the design team relies on verification methods (a major part of Cleanroom, see Linger, Mills, and Witt [38] for details on the technique) and not testing to ensure that the code is properly fixed.

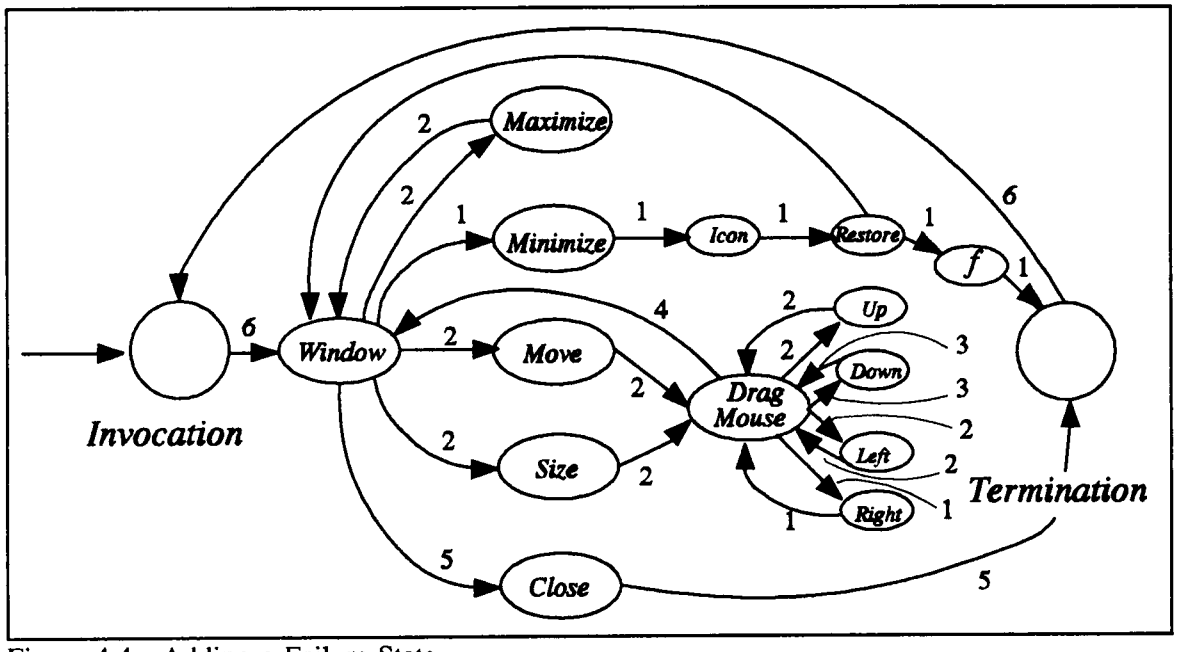


Figure 4.4 : Adding a Failure State

4.2.2 Properties of the Testing Markov Chain

The testing Markov chain, a model of the testing process built from the test sequences and any observed failure data, is useful for computing properties of random variables as shown in the next three sections. However, one might wish simply to obtain statistics directly from the sequences without building a model. This section compares these two methods to determine which properties are preserved when the testing chain is constructed.

PROPOSITION 4.1 The steady state probability of each state i in the testing chain is:

$$\pi_i = \frac{c_i}{\sum_j c_j},$$

where c_i is the relative frequency count of state i in the sequences.

Proof. In order for the proposition to hold, two conditions must be satisfied [49]:

$$(1) \quad \sum_i \left| \frac{c_i}{\sum_j c_j} \right| < \infty,$$

which holds because the term is normalized, and

$$(2) \quad \left[\frac{c_i}{\sum_j c_j} \right] = \left[\frac{c_i}{\sum_j c_j} \right] P$$

where brackets denote row vector notation. Condition (2) can be shown as follows:

$$\sum_k \left(\frac{c_k}{\sum_j c_j} \right) \left(\frac{c_{kl}}{c_k} \right) = \sum_k \frac{c_{kl}}{\sum_j c_j}$$

$$= \frac{1}{\sum_j c_j} \sum_k c_{kl}$$

$$= \frac{1}{\sum_j c_j} c_i$$

$$= \frac{c_i}{\sum_j c_j}.$$

Since these two conditions hold, the π values from the testing chain are in the same proportion as the relative frequencies in the sequences. Thus normalizing the relative frequency values yields

the analytical stationary probabilities. □

COROLLARY 4.1 The expected number of occurrences of state i in a test case, π_i/π_{Inv}, and the expected number of sequence changes between occurrences of state i, π_{Inv}/π_i, computed from the testing chain are equal to the respective averages from the set of sequences.

Proof. Since these two random variables are computed using the π values which are preserved in the inference, they are equivalently obtained from either the testing chain or the sequence set. □

It might seem from the above proposition that it is of little or no use to bother with building a model when many properties are preserved by the testing chain. However, important features are not preserved. One is the probability of a failure-free sequence, which is defined to be a software quality measure in section 4.4.

PROPOSITION 4.2 The probability of a failure free sequence, computed as the ratio of the number of sequences not containing a failure state to the total number of sequences, is not, in general, equal to the probability that a realization of the testing chain does not encounter a failure state.

Proof. (By counterexample) Given the testing chain of Figure 4.1 with all arcs labeled zero and the sequences:

1. *Invocation, Window, Move, Drag Mouse, Down, Drag Mouse, Down, Drag Mouse, Down, Drag Mouse, Window, Close, Termination;*

2. *Invocation, Window, Minimize, Icon, Restore, Window, Move, Drag Mouse, Window, Size, Drag Mouse, Right, Drag Mouse, Left, Drag Mouse, Down, Drag Mouse, Window, Close, Termination*;

3. *Invocation, Window, Maximize, Window, Size, Drag Mouse, Down, Drag Mouse, Up, FAILURE*;

with the failure state installed in the testing chain as a result of sequence 3. For these sequences we have 2/3 as the ratio of failure-free sequences, but the probability of not encountering a failure in the testing chain is .714 (using equation 2.14 with the *Termination* state and the *FAILURE* state made absorbing). This counterexample shows that for a finite sample set the probability of absorption (and thus the probability of a failure-free sequence) is not preserved. □

Proposition 4.2 shows that building a model can give different results than a finite set of samples in the computation of properties of random variables. The reason is that the testing Markov chain has additional information above what the sequence set contains, namely, the transition probabilities and the accessibility relation embodied in the arc structure. The transition probabilities enable the testing chain to use the probability of a sequence in the analytical results. If a failure occurs in a high probability sequence, it will have a different effect than if it occurs in a low probability sequence. Since this effect is desirable for a quality measure based on software usage and it is not available directly from the sequence set, we choose to use the testing chain as the source of software quality measurements. An example of the effect of the location of a failure state is given later in section 4.4. The reason that the π values are preserved in the testing chain is that they represent counts of states *relative* to the entire state set. The location of states is not important, only the rate of appearance. Thus these values are equivalently obtained from the testing chain or the sequence set.

4.3 A Stopping Criterion

Stopping criteria for the testing process can be as simple as choosing some target reliability and testing until the estimate of the reliability meets or exceeds the target. However, the model based approach to certification presented in this research allows an analytic stopping criterion directly from the statistical properties of the usage and testing models.

The generate-and-infer process developed above is repeated with each sequence applied to the software. As the sequences are incorporated, the testing Markov chain inherits its stochastic properties from the usage chain, but modified by any failures actually observed. In the event that no failures are found by testing, the testing chain will have exactly the same states and arcs as the usage chain. Furthermore, when enough sequences have been generated by the usage chain to obtain asymptotic statistics, then by the strong law of large numbers, the usage and testing chains will be indistinguishable within some tolerance. This means that both chains are equivalent when viewed as a stochastic source (i.e., they have the same state occupancy statistics and generate sequences with the same probabilities).

Failure states are introduced into the testing chain by actual observations of software failure during testing. Since the usage chain does not have these failure states, they have an implied long run probability of zero in U. In order to match the stochastic characteristics of the testing chain T in which the failure states may exist, enough sequences from U must end in correct termination to push the occupancy of all failure states in T close to zero. Thus, failures observed during testing tend to increase the number of sequences necessary to certify the software. Hopefully, failures will be few in number and occur relatively early in the testing process. If failures become too prevalent, it may be prudent to stop testing and redesign and reverify the software [14].

Whether failures are encountered or not, we are seeking to identify the steady state of the generate-and-infer process. We say that the *steady state of the testing process* has been reached when the stochastic properties of the usage chain and the testing chain are indistinguishable within some acceptable tolerance. The stochastic properties in this definition refer to long run state occupancy and state sequencing information. In order to measure the steady state, one could compute the stationary distribution of each chain and use a goodness of fit criterion (e.g., Chi-squared [16]) to measure their similarity. However, this approach only takes into account a single (albeit very important) attribute of the two models. If we want to measure the difference of the ensemble characteristics of each chain, then another approach is necessary. Consider each chain as an ergodic stochastic source. There exists a set of typical sequences for each source. If both chains have the same set of typical sequences, we may draw the conclusion that the two chains are indistinguishable as sequence generators. Stated differently, given sequence $X_0, X_1,...,X_n$ of sufficient length it should be extremely difficult, if not impossible, to determine whether the sequence was generated by U or T.

The *log likelihood ratio* [34] is a fundamental computation in measuring the evidence an observation provides for or against a hypothesis. The observation may be the outcome of a real or conceptual experiment. In the case that the hypothesis is "stochastic process U is equivalent to stochastic process T", an observation is a sequence generated by U. We define a measure for two stochastic processes as the expected value of the log likelihood ratio, called the *discriminant* [34]. This value is computed for two arbitrary ergodic stochastic processes λ_0 and λ_1 as [30]

$$D(\lambda_0,\lambda_1) = \lim_{n\to\infty} \frac{1}{n}[\log p(X_0,X_1,\cdots,X_n|\lambda_0) - \log p(X_0,X_1,\cdots,X_n|\lambda_1)] .$$

Just as the source entropy $H(X_n)$ cannot be computed analytically for arbitrary stochastic processes, neither can $D(\lambda_0,\lambda_1)$. As with $H(X_n)$, the Markov property allows $D(\lambda_0,\lambda_1)$ to be computed.

PROPOSITION 4.3 The *discrimination* of the testing chain from the usage chain, denoted $D(U,T)$, is computed by

$$D(U,T) = \sum_{i,j} \pi_i \, p_{ij} \log \frac{p_{ij}}{\hat{p}_{ij}} , \tag{4.1}$$

where π is the usage distribution, p_{ij} is the probability of a transition from i to j in the usage chain, and $\hat{p}_{ij}$ is the corresponding probability in the testing chain.

Proof. By manipulating the logarithms we obtain

$$D(U,T) = \lim_{n\to\infty} \frac{1}{n}\left[\log \frac{p(X_0,X_1,\cdots,X_n|U)}{p(X_0,X_1,\cdots,X_n|T)}\right].$$

Since U and T are ergodic Markov sources, the probability of the sequence X_0, X_1, ..., X_n is obtained from the transition probabilities and the long run probability of the start symbol. This yields

$$D(U,T) = \lim_{n\to\infty} \frac{1}{n}\left[\log \frac{\pi_0 p_{01} p_{12} \cdots p_{n-1,n}}{\hat{\pi}_0 \hat{p}_{01} \hat{p}_{12} \cdots \hat{p}_{n-1,n}}\right] .$$

Further manipulation yields,

$$D(U,T) = \lim_{n\to\infty} \frac{1}{n} \left[\log \frac{\pi_0}{\hat{\pi}_0} + \log \frac{p_{01}}{\hat{p}_{01}} + \log \frac{p_{12}}{\hat{p}_{12}} + \cdots + \log \frac{p_{n-1,n}}{\hat{p}_{n-1,n}}\right],$$

which can be written,

$$D(U,T) = \lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} \log \frac{p_{i,i+1}}{\hat{p}_{i,i+1}} + \lim_{n\to\infty} \frac{1}{n} \log \frac{\pi_0}{\hat{\pi}_0},$$

where the last term goes to zero in the limit. So

$$D(U,T) = \lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} \log \frac{p_{i,i+1}}{\hat{p}_{i,i+1}}.$$

This has the form of an average value in the limit. The ergodic nature of U and T assures that its convergence to the expected value is certain (i.e., convergence in probability) and allows the computation of the expectation by the appropriate steady state values. This yields

$$D(U,T) = \lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} \log \frac{p_{i,i+1}}{\hat{p}_{i,i+1}} = \sum_{i,j} \pi_i \, p_{ij} \log \frac{p_{ij}}{\hat{p}_{ij}}$$

where $\pi_i p_{ij} = p(X_n=i, X_{n+1}=j)$ as $n\to\infty$. □

The discrimination is a directional measure computed from the usage chain to the testing chain and is not generally symmetric.

PROPOSITION 4.4 The discrimination is non-negative and equal to zero if and only if $U=T$.

Proof. Using the fact that $\log p \geq 1 - \frac{1}{p}$ we have

$$D(U,T) = \sum_{ij} \pi_i p_{ij} \log \frac{p_{ij}}{\hat{p}_{ij}} \geq \sum_{ij} \pi_i p_{ij} \left(1 - \frac{\hat{p}_{ij}}{p_{ij}}\right)$$

$$= \sum_{ij} \pi_i p_{ij} - \sum_{ij} \pi_i \hat{p}_{ij}$$

which is equal to zero if and only if $p_{ij}=\hat{p}_{ij}$ for all i,j. □

The two sources U and T are likely to generate a similar set of typical sequences if and only if the value for $D(U,T)$ is very small. This has several implications for software testing. First, it ensures that each usage state is generated in the correct proportion as computed in the usage distribution and that the statistics of the generated sequences closely match the statistics of U. Second, it forces the occupancy of the failure states in T to be pushed nearly to zero. This means that confidence must be gained in every path through the chain before the steady state will be reached. Third, it recognizes the limitations of the usage chain for testing the software. Since the statistics of the testing chain and usage chain match, the usage chain is unlikely to generate a sequence that adds any additional information to the experiment.

It is now possible to measure the discrimination of the testing chain from the usage chain analytically to determine how closely the actual statistics (the test cases and failure data in T) match the expected statistics from U. To monitor the testing process, D can be computed with each sequence applied to the software. A downward trend in the values of D as testing progresses signifies growing similarity of the two models. Usage chain U never changes; however, D reflects the impact of each additional sequence on the stochastic characteristics of the testing chain. D can, for example, actually rise when no failures are observed if a sequence reinforces some low probability event. Of course, one would expect such a rise when a failure occurs. Testers must

take care in analyzing D with each sequence and to notice trends in its behavior. When the discrimination drops below some predefined threshold, or 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.

4.4 Analytical Results from the Testing Chain

The states of the testing Markov chain model usage of the software during correct functioning and during software failure. Thus it is possible to define random variables that characterize the relationship of failure states to non-failure states in order to describe the observed quality of the software during testing. Three common software quality measures are the reliability, the mean time between failure (*MTBF*), and the failure rate. Each of these values is obtained independently by analytical computation on the testing Markov chain.

The *reliability* of the software's usage is the probability that a realization, beginning with the *Invocation* state and ending with the first occurrence of the *Termination* state, of the testing chain will not contain a failure state. In order to compute the reliability, each failure state is converted to an absorbing state and the termination state is also made absorbing. The computation is thus reduced to computing the probability of absorption into the termination state. This value is (from equation 2.14)

$$R_{ik} = \hat{p}_{ik} + \sum_{j \in \tau} \hat{p}_{ij} R_{jk} . \qquad (4.2)$$

where i is the invocation state, k is the termination state, and τ is the set of non-absorbing usage states.

The *MTBF* in use of the software is the expected number of states encountered in the testing chain between occurrences of failure states.

PROPOSITION 4.5 The *MTBF* in use of the software is computed by

$$\mathbf{MTBF} = \sum_{i \in f_1,\ldots,f_m} v_i \left(\sum_{j \in s_1,\ldots,s_n} \hat{p}_{ij}(m_j + 1) \right), \tag{4.3}$$

where v_i is the renormalized long run probability for failure state f_i, m_j is the mean time until the first occurrence of any failure state from j, $s_1,\ldots,s_n$ is the set of usage states, and $f_1,\ldots,f_m$ is the set of failure states.

Proof. In order to compute the *MTBF*, the failure states are made absorbing and the termination state recurrent. The *MTBF* is the mean time to absorption from the set of failure states excluding the case of immediate absorption. The contribution of an arbitrary failure state f_i, to the *MTBF* is the probability of being in f_i, given that the process is in a failure state, times the expected number of states visited from f_i until the occurrence of the first failure state. The probability of being in f_i given that the process is in a failure state is:

$$v_i = \frac{\hat{\pi}_{f_i}}{\hat{\pi}_{f_1} + \hat{\pi}_{f_2} + \cdots + \hat{\pi}_{f_m}}.$$

Note that proposition 4.1 allows us to use the frequency counts of failure states to compute v as well because these normalized counts are equivalent to the π values for the failure states. We compute the expected number of states visited by making the failure states an absorbing subclass and computing the expected number of states visited from each state s_1, s_2, ..., s_n until absorption (excluding immediate absorption). In order to achieve this, we take one step away from the failure state f_i to a usage state s_j and then compute the expected number of states visited to

absorption from s_j, namely m_j (computed using equation 2.15). Thus absorption from f_i will take $p_{ij}(m_j+1)$ as the expected number of states visited. Summing over these mutually exclusive events completes the proof:

$$MTBF = \sum_{i \in f_1,\ldots,f_m} v_i \left(\sum_{j \in s_1,\ldots,s_n} \hat{p}_{ij}(m_j + 1) \right). \qquad \square$$

The *failure rate* is the expected number of failure states encountered in a realization of the testing chain, beginning with the *Invocation* state and ending with the first occurrence of the *Termination* state.

PROPOSITION 4.6 The failure rate, denoted Z, is computed by

$$Z = \sum_{f_i} \frac{\hat{\pi}_{f_i}}{\hat{\pi}_{TERM}}, \qquad (4.4)$$

where the sum is over the set of failure states of T.

Proof. In its original form, T is irreducible with a single exit arc from the *Termination* state to the *Invocation* state. Therefore, the expected number of occurrences of failure state f_i in a realization of T is

$$E(f_i) = \frac{\hat{\pi}_{f_i}}{\hat{\pi}_{TERM}}$$

using the ratio property described in Chapter 2. In other words, this is the expected number of occurrences of f_i between occurrences of the *Termination* state. Since each occurrence of the *Termination* state marks the end of a sequence, this is the expected number of occurrences of f_i

in a sequence. So the expected number of visits to all failures states is

$$Z = E(f_1 + f_2 + \ldots + f_m)$$
$$= E(f_1) + E(f_2) + \ldots + E(f_m)$$

which is

$$Z = \sum_{f_i} \frac{\hat{\pi}_{f_i}}{\hat{\pi}_{TERM}} . \qquad \square$$

The software quality measures computed for the testing chain account for test data across all versions of the software. A new version is obtained when the software is repaired by the developers after one or more failures are encountered. Thus the quality measurements described represent quality across all versions of the software. In addition, these measures have several beneficial features. First, they are based on actual occurrences of failures. In other models, assumptions about the distributions of interfail times are required in order to measure the reliability [14]. Second, each state in each sequence is accounted for in the computations. Each sequence contributes to the model in proportion to its length and probability of occurrence. Other models treat each sequence the same regardless of its attributes [14,43]. Third, each failure contributes to the model according to its location in the chain. Failures attached to relatively high probability paths will reduce the reliability more than failures attached to lower probability paths. Thus the testing chain delivers software quality measures which are based on the usage patterns described in the usage model.

As an example consider the usage chain of Figure 3.4 with the transition probabilities of Table 3.2. After 20 sequences are inferred, a failure is observed and accounted for at the state labeled *Maximize*. Using equation 4.2, we compute $R = .980$. On the other hand, the model gives $R = .981$ if the failure state is installed at the state labeled *Up*. Since *Maximize* is on a higher

probability path, a failure occurring at this state is more serious (with respect to this measure) than a failure on the lower probability path through *Up*. The difference is not dramatic in this small example but can be significant in larger chains. Examples of the reliability, *MTBF*, and the discrimination are given in the case study in the next chapter.

4.5 Predicting Future Software Behavior

The software quality measures computed in the previous section represent *estimations* based upon current data. That is, if testing has been carried out through time t, then the discrimination of equation 4.1, the reliability of equation 4.2, the *MTBF* of equation 4.3, and the failure rate of equation 4.4 are estimations at time t. This section is concerned with predicting the expected value of these random variables at some time $t+\alpha$. The units on the parameters t and α are sequences, i.e., the estimations of the previous section are made with t sequences incorporated into the testing chain. In order to make predictions about the same random variables, a value for α is supplied and the frequency model F is updated using appropriate expected values to obtain a prediction model P. The equations 4.1, 4.2, 4.3, and 4.4 can then be applied to P to compute the predicted analytical results. It is important to note that the method described in this section *does not attempt to predict the location of future failures*, only the pattern of behavior of properties of the above mentioned random variables.

One could take an empirical approach to this problem by generating α sequences from U and incorporating this information into the T in the usual manner. However, the analytical approach described in this section uses ergodic values to describe the α sequences that bypasses the expense of generating and inferring a (possibly large) number of sequences. The model can be constructed by assuming that no more failures occur in the α sequences, or assuming that the

number of failures in the α sequences is determined by the same rate as the actual t sequences.

The *prediction Markov chain*, P, is constructed under the first assumption (no more failures) by determining the expected makeup of the α sequences and incorporating this information into the chain along with the current information from T. The expectations of interest in this case are the expected number of occurrences of each usage state in the α sequences. This value for state i, denoted a, can be computed for each sequence as

$$\begin{aligned} a_i &= \alpha \left(\frac{1}{h_i}\right) \\ &= \frac{\alpha}{h_i}, \end{aligned} \tag{4.5}$$

where h_i is the expected number of sequences until state i appears (equation 3.2). The value a_i is interpreted as the expected number of occurrences of state i in the α sequences. Since the values in h are computed directly from the usage distribution, the values in a represent counts that are exactly typical of U.

In order to update the prediction chain with this information each arc in F is incremented by the appropriate value in the vector a weighted by the probability of that arc. That is, arc (i,j) has frequency $a_i\, p_{ij}$. Updating F in this manner yields

$$F(i,j) = F(i,j) + a_i\, p_{ij} \tag{4.6}$$

where p_{ij} is the probability of a transition from i to j in U. The prediction chain, P, achieved through normalizing F, now contains the actual sequences up to time t and the expected information from an additional α sequences assuming no failures occur in the α sequences and that they are typical in the sense of stationary probability. The prediction chain is treated in the same manner as the original testing chain and the equations for discrimination, reliability, *MTBF*, and failure rate are used to compute the predictions. Since this scheme reinforces the successful

paths through the chain, i.e., no new failure states are installed or old ones updated, the predictions will yield higher R and $MTBF$ values, and lower D and Z values. Furthermore, the a_i vector represents expected values and thus the α sequences are exactly typical in the occurrence rate of usage states. This will have the effect of substantially lowering the discrimination measure which essentially measures T's deviation from the typical behavior defined by U. The construction of the prediction chain in this manner is truly best case behavior. The states appear in exactly their expected proportions and no more failures occur.

One can also create the prediction chain under the assumption that the number of failures detected in the α sequences is determined by the current failure rate, Z. The α sequences will experience a failure rate of αZ. This information is incorporated into P, after it has been updated using equation 4.6, by creating a new state f_α to mark the predicted failure. Since we are not trying to predict the location of the new failure, we simply create an arc from each of the n usage states to f_α with frequency $\alpha Z/n$. Thus each usage state is equally likely to have uncovered the failure. This new frequency model can be normalized and used to compute the usual measures.

Creating the prediction chain in this manner yields results that vary depending upon the makeup of T at the time the prediction is requested. First, the discrimination will again behave optimistically because the usage state occurrence distribution is exactly typical. If prediction is made when T has already started to become typical, the result is likely to be accurate. However, if the prediction is made early in the testing process when T is still evolving rapidly the outcome will likely have large error because it is based on too little data.

The behavior of R, $MTBF$, and Z in the predictions is conditioned upon an assumed failure rate at t. When the failure rate is assumed to be zero, R and $MTBF$ will both increase because this has the effect of reinforcing failure-free paths of the chain. Z will likewise decrease because no new failures will contribute in its computation. When the failure rate is not zero, P is adapting

to non-failure states and a new failure state as well. The location of failures in T is now a factor. If these states lie on low probability paths, then the averaging effect of the predicted failure may cause the failure rate to increase, thus causing R and *MTBF* to drop slightly. On the other hand, high probability actual failures are likely to have a greater failure rate than the average predicted failure causing a subsequent rise in both R and *MTBF*. These predictions are computed in the case study of Chapter 5.

Chapter 5

A Case Study

5.1 Introduction

This chapter presents a case study employing the methods discussed in this research. The software specified is called the CleanRoom Environment (CRE). It is described in complete detail in [50-52]. The CRE is a software engineering environment containing a collection of tools to assist software development under the Cleanroom software engineering process, including development, certification and management. CRE observes the IBM SAA and CUA standards. It consists of a main control window (CreMain) which calls 3 subsystems: the Development Assistant, the Certification Assistant, and the Management Assistant. These subsystems contain tools for assisting engineers in various aspects of the Cleanroom software life cycle. This section specifies the behavior of the main control window (CreMain). The following terms and notation are defined to facilitate the description.

Input messages consist of {hwnd, msg, mp1, mp2} and result from keyboard and mouse stimuli. A message will represent a user operation (e.g. selection, open, drag) on a graphic object or on an action (e.g. icon, menu item, entry field, button) inside the window using keystrokes or mouse clicks.

An *object* is anything that users can manipulate as a single unit and on which the users' attention is focused. *Action* is a general term for ways that users can modify, manipulate, or

create objects.

A *selection* is performed by moving the mouse pointer over an object and clicking on the left mouse button once.

An *open* action is performed by placing the mouse pointer over the desired object and double clicking the left mouse button.

When a window is *activated*, it is superimposed over the other images on the screen and gains the focus of attention. If the window is in a minimized state, it is restored to its original size before being activated.

A unique error message box is used throughout the system to report errors. It has a title bar containing "ERROR". It contains the error message, an exclamation point icon, and an OK button.

The notation <MenuName>.<ItemName> is used to refer to a menu item. For example, FileMenu.Open refers to item Open on menu File.

The next section contains the definition of the top-level black box (in the sense of the Box Structure method [41]) of the main control window.

5.2 The Specification

```
define_BB MRESULT EXPENTRY CreMainProc as
        Stimulus List
                HWND   hwnd,
                USHORT msg,      /* PM messages */
                MPARAM mp1,
                MPARAM mp2
                /* Stimuli of Subsystems */
        Response List
                /*  window CreMain
                 *  window MsgBox
                 *  Responses of Subsystems */
```

Transition Rule

/*

An open on the CRE icon in will invoke the system. The main window CreMain will be created and activated. The CreMain window has a title bar, system menu icon (Restore, Move, Size, Minimize, Maximize, Close, Switch to), minimize icon, maximize icon, file menu (Open, Exit), view menu (Text on, Text off), help menu (Extended help), and a client window in which a set of bit maps are displayed. These bit maps represent subsystems that can be invoked inside the control window. The CreMain window is shown in Figure 5.1.

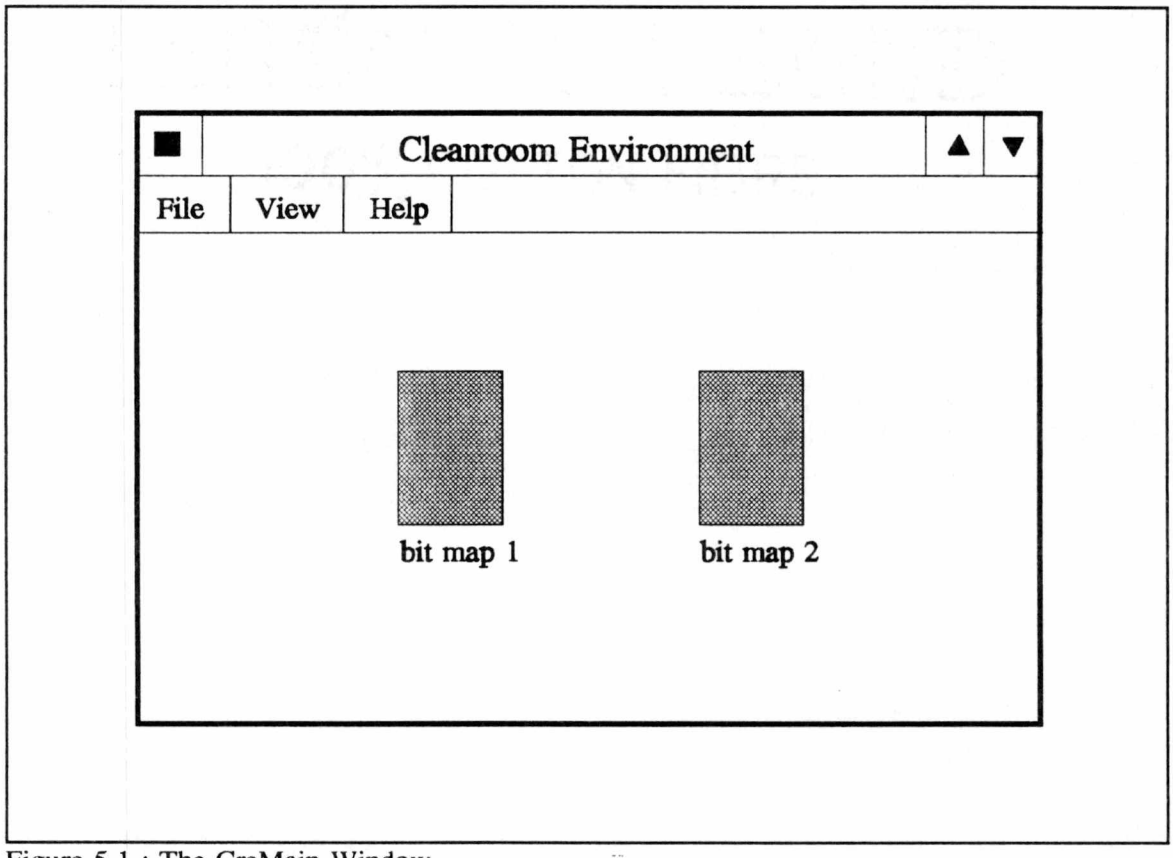


Figure 5.1 : The CreMain Window

The behavior of the CreMain window is defined as follows:

Action	Response
Open SystemMenuIcon	The CreMain window is closed. Control returns to the operating system.
Select SystemMenuIcon	Displays a pull down menu with the options: *Close*, *Restore*, *Move*, *Size*, *Minimize*, and *Maximize*.
Select SystemMenu.Close	Same as an open on SystemMenuIcon. See the transition above.
Select SystemMenu.Restore	Restore the window to its original size. This option can be used after the window has been minimized or maximized.
Select SystemMenu.Move	Move the window in the direction of the movement of the mouse until the left button is clicked.
Select SystemMenu.Size	Make the window larger or smaller. Inside the client area, the size of text and bit maps will not change. Distances and positions of text and bit maps will change proportionally.
Select SystemMenu.Minimize	Make the window the size of an icon.
Select SystemMenu.Maximize	Make the window the size of the whole screen. Inside the client area, the size of text and bit maps will not change. Distances and positions of text and bit maps will change proportionally.
Select TitleBar	Move the window in the direction of the movement of the mouse until the left button is released.
Select the window edge	Size the window, smaller or larger, in the direction of the mouse movement.
Select MinimizeIcon	Same as a selection on SystemMenu.Minimize, see the transition above.
Select MaximizeIcon	Same as a selection on SystemMenu.Maximize, see the transition above.
Open Icon	Same as a selection on SystemMenu.Restore, see the transition above.
Select Icon	Displays a pull down menu with the options: *Close*,

	Restore, *Move*, and *Maximize*.
Select FileMenuIcon	Displays a pull down menu with the options: *Open* and *Exit*.
Select FileMenu.Open	If the corresponding subsystem is not active, the corresponding control window is created and given the focus. Otherwise, the corresponding control window is given the focus.
Select FileMenu.Exit	Same as an open on SystemMenuIcon, see the transition above.
Select ViewMenuIcon	Displays a pull down menu with the options: *Text On* and *Text Off*.
Select ViewMenu.TextOn	Identifying text will appear below the bit maps. ViewMenu.TextOn will be disabled and ViewMenu.TextOff will be enabled.
Select ViewMenu.TextOff	The text will disappear. The bit maps will move to the top left of the client area. ViewMenu.TextOff will be disabled and ViewMenu.TextOn will be enabled.
Select HelpMenuIcon	Displays a pull down menu with the option: *ExtendedHelp*.
Select HelpMenu.ExtHelp	The message box, containing a title, an OK button and the message "Help will be implemented in a future release.", will be displayed in the center of the screen.
Select a bit map	The bit map will be highlighted by a wide blue border around the bit map. The previously selected bit map inside the same control window, if any, will be de-highlighted. The control window will be activated.
Select open area	The previously selected bit map inside the same control window, if any, will be de-highlighted. The control window will be activated.
Open a bit map	Same as a select FileMenu.Open, see the transition above.

```
        */
end_BB
```

5.3 The Usage Analysis

The usage model is organized as follows: a state is identified by both a state label and a state number. This way it is possible to have different states with the same label as state splitting occurs. We begin by defining the arc for invocation of the software and the display of the CreMain window.

1 Invocation 2 CreMain

Next, the exit arcs from CreMain are defined by enumerating all possible stimuli from the window. Note that, for simplicity, we do not define states for illegal actions. However, this can be achieved by including a superstate that encapsulates all illegal moves. This superstate could then be expanded for all possible illegal stimuli. In this model, illegal sequences of stimuli are possible.

2 CreMain	3 OpenSystemMenu
	4 SelectSystemMenu
	5 SelectFileMenu
	6 SelectViewMenu
	7 SelectHelpMenu
	8 Minimize
	9 Maximize
	10 Move
	11 Size
	12 SelectBitMap1
	13 SelectBitMap2
	14 SelectOpenArea
	15 OpenBitMap1
	16 OpenBitMap2

Continuing in this manner creates more states and arcs.

3 OpenSystemMenu	70 Termination
4 SelectSystemMenu	17 Close
	18 Restore
	19 Move
	20 Size

		21	Minimize
		22	Maximize
5	SelectFileMenu	23	Open
		24	Exit
6	SelectViewMenu	25	Text On
		26	Text Off
7	SelectHelpMenu	27	Extended Help
8	Minimize	28	CreIcon

The next state to define is "Select MaximizeIcon". We note at this point that the maximized window does not have the same functionality as the minimized window. For example, window sizing is no longer possible and the restore option is now enabled. Technically, the transition from these states are dependent upon the occurrence of a previous state (maximize). Thus we use the state splitting technique to create a new set of states for the maximized window.

9	Maximize	29	CreMain (Max)

Here state 29 represents the maximized window. When we resolve the exit arcs from this state they will go to new states which represent actions on the maximized window, i.e. split from their counterparts on the initial window. We continue with the initial window actions.

10	Move	30	Drag Mouse
11	Size	30	Drag Mouse

We assume two running subsystems in the client area of the CreMain window.

12	SelectBitMap1	2	CreMain
13	SelectBitMap2	2	CreMain
14	SelectOpenArea	2	CreMain
15	OpenBitMap1	31	Subsystem1
16	OpenBitMap2	32	Subsystem2
17	Close	70	Termination

18 Restore	2 CreMain
19 Move	30 Drag Mouse
20 Size	30 Drag Mouse
21 Minimize	28 CreIcon
22 Maximize	29 CreMain (Max)
23 Open	31 Subsystem1 32 Subsystem2
24 Exit	70 Termination
25 Text On	2 CreMain
26 Text Off	2 CreMain
27 Extended Help	2 CreMain
28 CreIcon	33 Close 34 Restore 35 Move 36 Maximize
29 CreMain (Max)	37 OpenSystemMenu (Max) 38 SelectSystemMenu (Max) 39 SelectFileMenu (Max) 40 SelectViewMenu (Max) 41 SelectHelpMenu (Max) 42 Minimize (Max) 43 Restore (Max) 44 Move (Max) 45 SelectBitMap1 (Max) 46 SelectBitMap2 (Max) 47 SelectOpenArea (Max) 48 OpenBitMap1 (Max) 49 OpenBitMap2 (Max)
30 Drag Mouse	50 Up 51 Down 52 Left 53 Right 2 CreMain

The subsystems will not be expanded here. Separate Markov chains may be built for these

systems and included as a part of the CreMain usage chain as necessary.

31	Subsystem1	2	CreMain
32	Subsystem2	2	CreMain
33	Close	70	Termination
34	Restore	2	CreMain
35	Move	28	CreIcon
36	Maximize	29	CREmain (Max)

At this point the establishment of the states that model the maximized window begins. Note that the structure of the sub-chain that represents the maximized window is very similar to the structure of the initial window sub-chain because the functionality of most of the states has not changed.

37	OpenSystemMenu (Max)	70	Termination
38	SelectSystemMenu (Max)	54	Close (Max)
		55	Restore (Max)
		56	Move (Max)
		57	Minimize (Max)
39	SelectFileMenu (Max)	58	Open (Max)
		59	Exit (Max)
40	SelectViewMenu (Max)	60	Text On (Max)
		61	Text Off (Max)
41	SelectHelpMenu (Max)	62	Extended Help (Max)

The one exception to this is the CreIcon sub-chain. Since the icon must eventually either go to termination or back to the initial CreMain state (2), whether it is reached from a maximized window or not, the Markov property is not violated and their is no need to create new states. Thus we retain the number 28 for the CreIcon state.

42	Minimize (Max)	28	CreIcon

43 Restore (Max)	2 CreMain
44 Move (Max)	63 Drag Mouse (Max)
45 SelectBitMap1 (Max)	29 CreMain (Max)
46 SelectBitMap2 (Max)	29 CreMain (Max)
47 SelectOpenArea (Max)	29 CreMain (Max)
48 OpenBitMap1 (Max)	64 Subsystem1 (Max)
49 OpenBitMap2 (Max)	65 Subsystem2 (Max)
50 Up	30 Drag Mouse
51 Down	30 Drag Mouse
52 Left	30 Drag Mouse
53 Right	30 Drag Mouse
54 Close (Max)	70 Termination
55 Restore (Max)	2 CreMain
56 Move (Max)	63 Drag Mouse (Max)
57 Minimize (Max)	28 CreIcon (Max)
58 Open (Max)	64 Subsystem1 (Max) 65 Subsystem2 (Max)
59 Exit (Max)	70 Termination
60 Text On (Max)	29 CreMain (Max)
61 Text Off (Max)	29 CreMain (Max)
62 Extended Help (Max)	29 CreMain (Max)
63 Drag Mouse (Max)	66 Up (Max)

67 Down (Max)
68 Left (Max)
69 Right (Max)
29 CreMain (Max)

64 Subsystem1 (Max) 29 CreMain (Max)

65 Subsystem2 (Max) 29 CreMain (Max)

66 Up (Max) 63 Drag Mouse (Max)

67 Down (Max) 63 Drag Mouse (Max)

68 Left (Max) 63 Drag Mouse (Max)

69 Right (Max) 63 Drag Mouse (Max)

Finally, the arc creating an irreducible chain is installed completing the structural phase.

70 Termination 1 Invocation

Since there is no data available with which to establish an informed set of transition probabilities, the uninformed approach is used. The analytical results can now be computed from this usage chain. The expected values for the random variables discussed in Chapter 3 for the CRE usage chain in its irreducible form are listed below.

#	State	$m_{Inv,i}$	π	m_{ii}	π_i/π_{Inv}	π_{Inv}/π_i
1	Invocation	28.48	0.035111	28.48	1	1
2	CreMain	1	0.167490	5.97	4.7	0.209
3	OpenSystemMenu	81.59	0.011964	83.59	0.3407	2.935
4	SelectSystemMenu	75.72	0.011964	83.59	0.3407	2.935
5	SelectFileMenu	81.09	0.011964	83.59	0.3407	2.935
6	SelectViewMenu	82.59	0.011964	83.59	0.3407	2.935
7	SelectHelpMenu	82.59	0.011964	83.59	0.3407	2.935
8	Minimize	75.78	0.011964	83.59	0.3407	2.935
9	Maximize	70.17	0.011964	83.59	0.3407	2.935

#	State	$m_{Inv,i}$	π	m_{ii}	π_i/π_{Inv}	π_{Inv}/π_i
10	Move	74.59	0.011964	83.59	0.3407	2.935
11	Size	74.59	0.011964	83.59	0.3407	2.935
12	SelectBitMap1	83.59	0.011964	83.59	0.3407	2.935
13	SelectBitMap2	83.59	0.011964	83.59	0.3407	2.935
14	SelectOpenArea	83.59	0.011964	83.59	0.3407	2.935
15	OpenBitMap1	82.59	0.011964	83.59	0.3407	2.935
16	OpenBitMap2	82.59	0.011964	83.59	0.3407	2.935
17	Close	499.52	0.001994	501.52	0.0568	17.609
18	Restore	501.52	0.001994	501.52	0.0568	17.609
19	Move	492.52	0.001994	501.52	0.0568	17.609
20	Size	492.52	0.001994	501.52	0.0568	17.609
21	Minimize	493.72	0.001994	501.52	0.0568	17.609
22	Maximize	488.11	0.001994	501.52	0.0568	17.609
23	Open	166.17	0.005982	167.17	0.1704	5.869
24	Exit	165.17	0.005982	167.17	0.1704	5.869
25	Text On	167.17	0.005982	167.17	0.1704	5.869
26	Text Off	167.17	0.005982	167.17	0.1704	5.869
27	Extended Help	83.59	0.011964	83.59	0.3407	2.935
28	CreIcon	48.56	0.026701	37.45	0.7605	1.315
29	CreMain	41.32	0.063112	15.84	1.7975	0.556
30	Drag Mouse	27.82	0.139575	7.16	3.9753	0.252
31	Subsystem1	66.87	0.014954	66.87	0.4259	2.348
32	Subsystem2	66.87	0.014954	66.87	0.4259	2.348
33	Close	147.81	0.006675	149.81	0.1901	5.259
34	Restore	149.81	0.006675	149.81	0.1901	5.259
35	Move	197.36	0.006675	149.81	0.1901	5.259
36	Maximize	152.68	0.006675	149.81	0.1901	5.259

#	State	$m_{Inv,i}$	π	m_{ii}	π_i/π_{Inv}	π_{Inv}/π_i
37	OpenSystemMenu (Max)	203.98	0.004855	205.98	0.1383	7.232
38	SelectSystemMenu (Max)	214.84	0.004855	205.98	0.1383	7.232
39	SelectFileMenu (Max)	223.64	0.004855	205.98	0.1383	7.232
40	SelectViewMenu (Max)	245.30	0.004855	205.98	0.1383	7.232
41	SelectHelpMenu (Max)	245.30	0.004855	205.98	0.1383	7.232
42	Minimize (Max)	216.09	0.004855	205.98	0.1383	7.232
43	Restore (Max)	205.98	0.004855	205.98	0.1383	7.232
44	Move (Max)	237.30	0.004855	205.98	0.1383	7.232
45	SelectBitMap1 (Max)	246.30	0.004855	205.98	0.1383	7.232
46	SelectBitMap2 (Max)	246.30	0.004855	205.98	0.1383	7.232
47	SelectOpenArea (Max)	246.30	0.004855	205.98	0.1383	7.232
48	OpenBitMap1 (Max)	245.30	0.004855	205.98	0.1383	7.232
49	OpenBitMap2 (Max)	245.30	0.004855	205.98	0.1383	7.232
50	Up	62.65	0.027915	35.82	0.7951	1.258
51	Down	62.65	0.027915	35.82	0.7951	1.258
52	Left	62.65	0.027915	35.82	0.7951	1.258
53	Right	62.65	0.027915	35.82	0.7951	1.258
54	Close (Max)	821.93	0.001214	823.93	0.0345	28.929
55	Restore (Max)	823.93	0.001214	823.93	0.0345	28.929
56	Move (Max)	855.25	0.001214	823.93	0.0345	28.929
57	Minimize (Max)	834.04	0.001214	823.93	0.0345	28.929
58	Open (Max)	451.29	0.002427	411.96	0.0691	14.464

#	State	$m_{Inv,i}$	π	m_{ii}	π_i/π_{Inv}	π_{Inv}/π_i
59	Exit (Max)	409.96	0.002427	411.96	0.0691	14.464
60	Text On (Max)	452.29	0.002427	411.96	0.0691	14.464
61	Text Off (Max)	452.29	0.002427	411.96	0.0691	14.464
62	ExtendedHelp (Max)	246.30	0.004855	205.98	0.1383	7.232
63	Drag Mouse (Max)	197.11	0.030342	32.96	0.8641	1.157
64	Subsystem1 (Max)	205.11	0.006068	164.79	0.1728	5.786
65	Subsystem2 (Max)	205.11	0.006068	164.79	0.1728	5.786
66	Up (Max)	360.89	0.006068	164.79	0.1728	5.786
67	Down (Max)	360.89	0.006068	164.79	0.1728	5.786
68	Left (Max)	360.89	0.006068	164.79	0.1728	5.786
69	Right (Max)	360.89	0.006068	164.79	0.1728	5.786
70	Termination	27.48	0.035111	28.48	1	1

It can be concluded from this data that the expected test case length is 27.48 states. The entropy of the usage model is 1.401858146.

From this data we can isolate certain states and analyze their behavior. For example, the state *Subsystem1* is expected to occur 66.87 states (after invocation of CRE) from the CreMain window and 205.11 states from the maximized window. The steady state value for *Subsystem1* is 0.014954+0.006068=0.021022 for states 31 and 64. The expected number of occurrences of *Subsystem1* in a single test case is the expected number of occurrences of states 31 and 64 between occurrences of the *Invocation* state. This expectation is 0.4259+0.1728=0.5987. The expected number of occurrences of either subsystem is 1.198. Thus with each test case we expect a visit to a subsystem.

As explained in Chapter 3, the π entries may be summed to obtain the steady state

probability associated with a subset of states. For example, the value .20026 represents the probability of all the actions from a maximized window. Thus according to the uninformed transition probabilities, test sequences are expected to spend approximately 20% of their activity in the maximized window.

The analytical results derived in Chapter 3 for the CRE usage chain in its absorbing form are listed below.

#	State	$m(i\|Inv)$	$y_{Inv,i}$	h_i
1	Invocation	1	1	1
2	CreMain	4.7704	1	1
3	OpcnSystemMenu	0.3407	0.3407	2.93
4	SelectSystemMenu	0.3407	0.2788	3.59
5	SelectFileMenu	0.3407	0.2911	3.43
6	SelectViewMenu	0.3407	0.2541	3.93
7	SelectHelpMenu	0.3407	0.2541	3.93
8	Minimize	0.3407	0.2929	3.41
9	Maximize	0.3407	0.2967	3.37
10	Move	0.3407	0.2541	3.93
11	Size	0.3407	0.2541	3.93
12	SelectBitMap1	0.3407	0.2541	3.93
13	SelectBitMap2	0.3407	0.2541	3.93
14	SelectOpenArea	0.3407	0.2541	3.93
15	OpenBitMap1	0.3407	0.2541	3.93
16	OpenBitMap2	0.0568	0.2541	3.93
17	Close	0.0568	0.0568	17.61
18	Restore	0.0568	0.0537	18.61
19	Move	0.0568	0.0537	18.61
20	Size	0.0568	0.0537	18.61

#	State	$m(i\|Inv)$	$y_{Inv,i}$	h_i
21	Minimize	0.0568	0.0553	18.09
22	Maximize	0.0568	0.0554	18.04
23	Open	0.1704	0.1456	6.86
24	Exit	0.1704	0.1704	5.86
25	Text On	0.1704	0.1456	6.86
26	Text Off	0.1704	0.1456	6.86
27	Extended Help	0.3407	0.2541	3.93
28	CreIcon	0.7605	0.4129	2.42
29	CreMain	1.7975	0.4308	2.32
30	Drag Mouse	3.9753	0.4429	2.26
31	Subsystem1	0.4259	0.2987	3.35
32	Subsystem2	0.4259	0.2987	3.35
33	Close	0.1901	0.1901	5.26
34	Restore	0.1901	0.1598	6.26
35	Move	0.1901	0.1302	7.68
36	Maximize	0.1901	0.1596	6.27
37	OpenSystemMenu (Max)	0.1383	0.1383	7.23
38	SelectSystemMenu (Max)	0.1383	0.1199	8.34
39	SelectFileMenu (Max)	0.1383	0.1191	8.39
40	SelectViewMenu (Max)	0.1383	0.1047	9.55
41	SelectHelpMenu (Max)	0.1383	0.1047	9.55
42	Minimize (Max)	0.1383	0.1199	8.34
43	Restore (Max)	0.1383	0.1215	8.23
44	Move (Max)	0.1383	0.1047	9.55

#	State	$m(i\|Inv)$	$y_{Inv,i}$	h_i
45	SelectBitMap1 (Max)	0.1383	0.1047	9.55
46	SelectBitMap2 (Max)	0.1383	0.1047	9.55
47	SelectOpenArea (Max)	0.1383	0.1047	9.55
48	OpenBitMap1 (Max)	0.1383	0.1047	9.55
49	OpenBitMap2 (Max)	0.1383	0.1047	9.55
50	Up	0.7951	0.2845	3.52
51	Down	0.7951	0.2845	3.52
52	Left	0.7951	0.2845	3.52
53	Right	0.7951	0.2845	3.52
54	Close (Max)	0.0346	0.0346	28.93
55	Restore (Max)	0.0346	0.0334	29.93
56	Move (Max)	0.0346	0.0320	31.25
57	Minimize (Max)	0.0346	0.0332	30.04
58	Open (Max)	0.0691	0.0596	16.79
59	Exit (Max)	0.0691	0.0691	14.46
60	Text On (Max)	0.0691	0.0596	16.79
61	Text Off (Max)	0.0691	0.0596	16.79
62	ExtendedHelp (Max)	0.1383	0.1047	9.55
63	Drag Mouse (Max)	0.8642	0.1233	8.11
64	Subsystem1 (Max)	0.1728	0.1233	8.11
65	Subsystem2 (Max)	0.1728	0.1233	8.11
66	Up (Max)	0.1728	0.0719	13.89
67	Down (Max)	0.1728	0.0719	13.89
68	Left (Max)	0.1728	0.0719	13.89

#	State	$m(i\|Inv)$	$y_{Inv,i}$	h_i
69	Right (Max)	0.1728	0.0719	13.89
70	Termination	1	1	1

From this data we observe that 31.25 test cases are expected to be necessary before each usage state occurs at least once (see h_i, state 56). From the values listed for h, it is clear that the states for the main window will appear more frequently than the states from the maximized window. In fact, only 18.61 test cases are expected to cover the main window states (the lowest frequency states from the main window are states 18-20).

5.4 The Reliability Analysis

The usage chain of the previous section is used to generate sequences that represent the test cases for the CreMain window and to infer the testing model. When the code that implements the specification is ready for testing, the sequences are applied to the software and the analytical results are tabulated. The following listing shows the results with five failures included. The units on the *MTBF* are states.

Sequence Number	Failure Number	Estimated *R*	Estimated *MTBF*	Estimated *Z*	Estimated *D*
1		1	∞	0	∞
2		1	∞	0	∞
3		1	∞	0	∞
4	1	0.8421052632	86	0.25	∞
5		0.8571428571	287.4818182	0.2	∞

Sequence Number	Failure Number	Estimated *R*	Estimated *MTBF*	Estimated *Z*	Estimated *D*
6		0.875	306.3333333	0.166667	∞
7		0.8888888889	323.7619048	0.142857	∞
8		0.9	364.75	0.125	∞
9		0.9075630252	401.2727273	0.111111	∞
10	2	0.8244274809	205.5231884	0.2	∞
11		0.8391608392	227.8994730	0.181818	∞
12		0.8516129032	231.1642951	0.166667	∞
13		0.8622754491	242.0374012	0.153846	∞
14		0.8711340206	246.4017822	0.142857	∞
15		0.8792270531	249.68	0.133333	∞
16		0.8860759494	259.087037	0.125	∞
17		0.8921933086	271.8604127	0.117647	∞
18		0.8978873239	273.7664676	0.111111	∞
19		0.9026548673	306.4696970	0.105263	∞
20	3	0.8659517426	144.5222781	0.15	∞
21		0.8716707022	147.7773467	0.142857	∞
22		0.8770301624	151.8778391	0.136363	∞
23		0.8819599109	153.6575211	0.130434	∞
24		0.8865096360	159.0165996	0.125	∞
25		0.8907216495	161.3066514	0.12	∞
26		0.8946322068	163.7920678	0.115385	∞
27		0.8982725528	166.2445129	0.111111	∞
28		0.9016697588	177.4025157	0.107142	∞
29		0.9048473968	179.6666667	0.103448	∞
30		0.9076682316	193.0225989	0.1	∞
31	4	0.8877597219	59.0538221	0.129032	∞
32	5	0.8617882568	49.1742678	0.15625	∞

Sequence Number	Failure Number	Estimated *R*	Estimated *MTBF*	Estimated *Z*	Estimated *D*
33		0.8655620448	79.6611320	0.151515	∞
34		0.8692707086	80.2519866	0.147058	∞
35		0.8726560313	85.4540092	0.142857	∞
36		0.8758475143	120.3853298	0.138888	∞
37		0.8790063807	121.5905823	0.135135	∞
38		0.8819688465	123.7227570	0.131578	∞
39		0.8848264204	124.2709405	0.128205	∞
40		0.8874816610	127.0085316	0.125	∞
41		0.8900504884	128.3270387	0.121951	∞
42		0.8924557759	129.1309027	0.119047	∞
43		0.8948127863	129.7956116	0.116279	∞
44		0.8969643838	134.4271145	0.113636	∞
45		0.8990923123	139.2981510	0.111111	∞
46		0.9011816947	139.7413576	0.108695	∞
47		0.9031863074	140.8493740	0.106382	∞
48		0.9050480007	148.9935716	0.104166	∞
49		0.9062738291	158.0843471	0.10204	∞
50		0.9080574891	160.4895617	0.1	∞
51		0.9097724599	161.5085375	0.098039	∞
52		0.9114104779	167.1519359	0.096153	∞
53		0.9130152036	168.4782347	0.094339	∞
54		0.9145515377	296.6	0.092592	0.02698257716
55		0.9160103849	303.4	0.090909	0.02631995961
56		0.9174500692	304.2	0.089285	0.02609375776
57		0.9186413710	310.6	0.087719	0.02423935534
58		0.9199898602	312.6	0.086206	0.02306371947
59		0.9212943768	313.6	0.084745	0.02326726412

Sequence Number	Failure Number	Estimated *R*	Estimated *MTBF*	Estimated *Z*	Estimated *D*
60		0.9225419827	319.2	0.083333	0.02332765590
61		0.9237650471	320	0.081967	0.02322265892
62		0.9249500874	322.4	0.080645	0.02366854092
63		0.9260988498	323.2	0.079365	0.02361452333
64		0.9270621580	336.6	0.078125	0.02004349215
65		0.9281374575	339.2	0.076923	0.01996410930
66		0.9291621624	351.8	0.075757	0.01846702614
67		0.9301599236	354.2	0.074626	0.01873203837
68		0.9311464199	357.2	0.073529	0.01898029270
69		0.9320956464	367.8	0.072463	0.01723648899
70		0.9330228696	378.8	0.071428	0.01740195990
71		0.9338149650	402.6	0.070422	0.01507444369
72		0.9347048164	403.6	0.069444	0.01528825606
73		0.9355710574	404.4	0.068493	0.01517126641
74		0.9364146152	406	0.067567	0.01442032401
75		0.9372296056	410.6	0.066666	0.01453178489
76		0.9380303504	411.8	0.065789	0.01457367859
77		0.9387973873	417	0.064935	0.01435830830
78		0.9395509296	420	0.064102	0.01501623358
79		0.9402936843	421.2	0.063291	0.01509023050
80		0.9410161594	423.2	0.0625	0.01522379373
81		0.9416367132	425.2	0.061728	0.01474125442
82		0.9423085568	441.6	0.060975	0.01416992762
83		0.9429704802	444.4	0.06024	0.01455050746
84		0.9436281819	447.2	0.059523	0.01479764783
85		0.9442570282	472.8	0.058823	0.01487701363
86		0.9448722535	477.2	0.058139	0.01486267053

Sequence Number	Failure Number	Estimated *R*	Estimated *MTBF*	Estimated *Z*	Estimated *D*
87		0.9454816765	486.6	0.057471	0.01433592899
88		0.9460847113	487.4	0.056818	0.01428418091
89		0.9466639441	489.8	0.056179	0.01437797115
90		0.9471296669	509.4	0.055555	0.01451713703
91		0.9476936609	518	0.054945	0.01470903551
92		0.9481945802	531.6	0.054347	0.01529660480
93		0.9487358400	533.2	0.053763	0.01543670435
94		0.9492657025	536.8	0.053191	0.01534444132
95		0.9497864897	539	0.052631	0.01516199431
96		0.9502949541	543.2	0.052083	0.01503732349
97		0.9507912633	552.2	0.051546	0.01488938329
98		0.9512812881	553	0.05102	0.01485656311
99		0.9517569539	556	0.050505	0.01476804303
100		0.9522265008	558	0.05	0.01491410294

Key features of the reliability model are explicitly illustrated in this data. Notice that three sequences are applied to the software before a failure is observed. Thus the testing chain for the first three sequences has only a single absorbing state, i.e., the termination state. The probability of absorption into this state (and thus *R*) is one. Similarly, the *MTBF* is infinite until a failure is observed. When the first failure is observed at sequence 4, *R* drops to .8421 and the *MTBF* is 86 states. As sequences continue to be applied, both of these quantities grow until the next failure is observed at sequence 10. This failure causes both *R* and the *MTBF* to drop considerably since it is early in the testing process and failures have a relatively large impact upon the statistics. The third observed failure at sequence 20 does not affect *R* as much as it does the *MTBF*. This is because the computation of the *MTBF* is much more sensitive to the statistics of

the testing chain. Note that as new arcs are installed in the developing testing chain, many more paths are available between failure states. These paths may be short in length causing a large drop in the *MTBF*. However, the reliability is not as susceptible to these changes since the computation involves multiplying small probabilities (which tend to zero) instead of counting transitions. Graphs of the reliability and *MTBF* data appear in Figures 5.2 and 5.3, respectively.

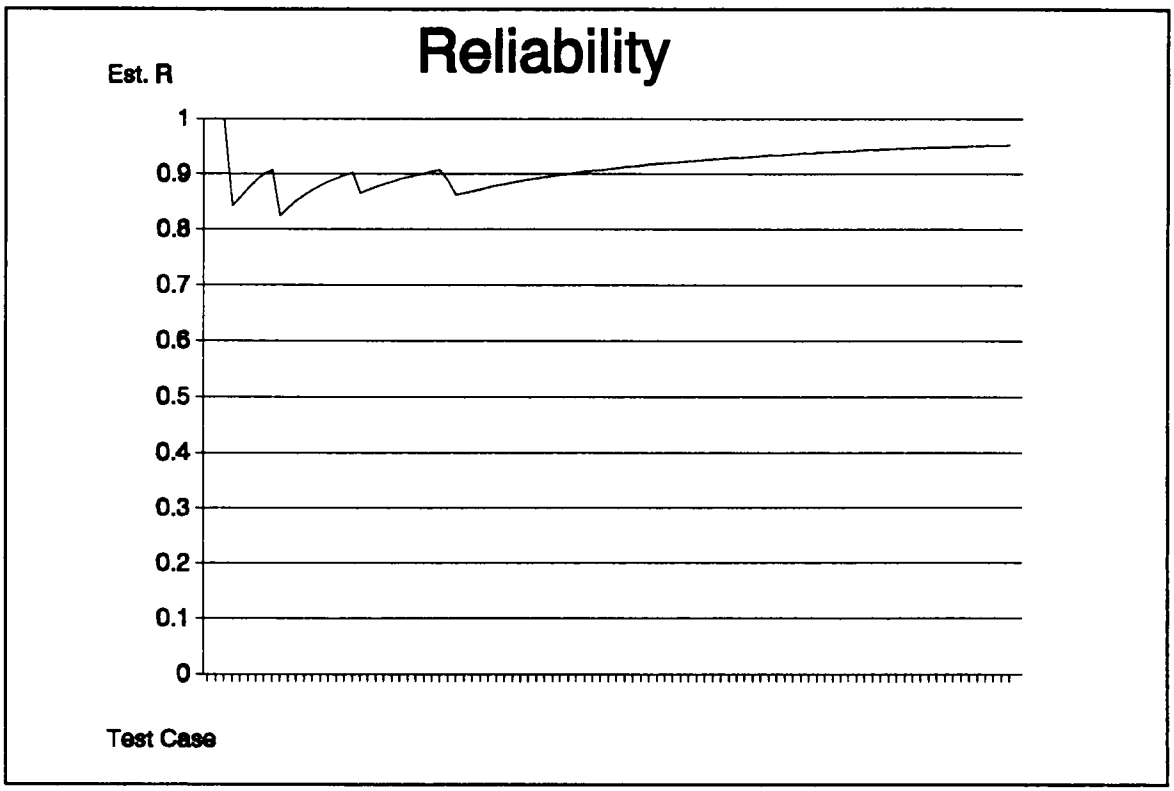


Figure 5.2 : Plot of *R* Data for Test Cases 1-100

The estimated discrimination measure is infinite until sequence 54 when all of the arcs of the testing chain have been traversed at least once. The discrimination measure is also very sensitive to the statistics of the two chains involved. Since atypical sequences often occur before the process settles into long run behavior, it often rises when no failures are observed. However, the overall trend for the discrimination is downward. The graph for the discrimination data for test cases 50-100 appears in Figure 5.4.

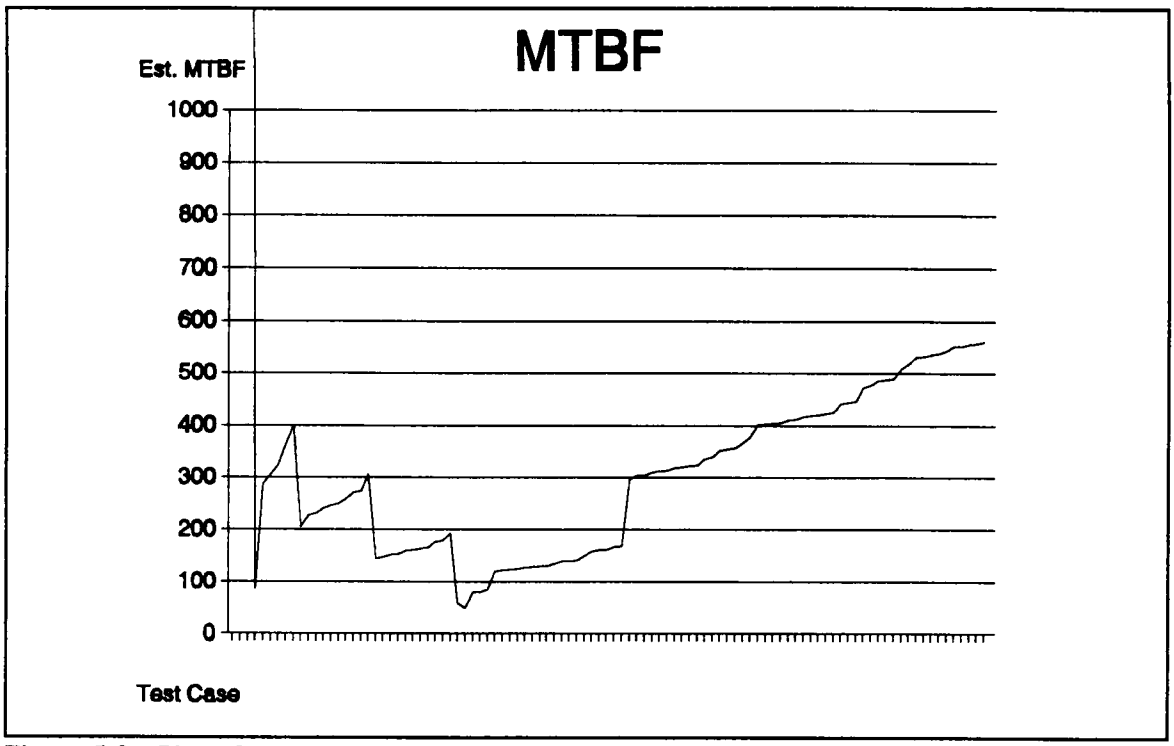


Figure 5.3 : Plot of *MTBF* Data for Test Cases 1-100

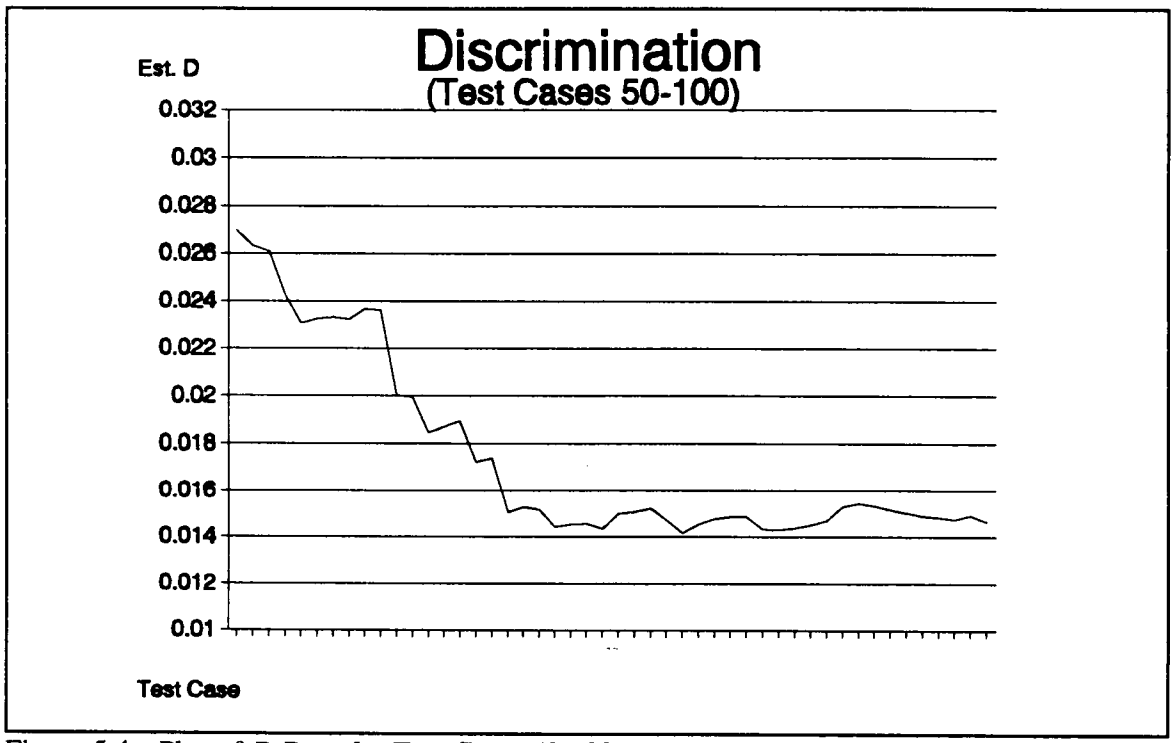


Figure 5.4 : Plot of *D* Data for Test Cases 50-100

The computations from the prediction Markov chain are shown in the following listing.

Future Sequences (α)	Expected Number of Failures	Predicted *R*	Predicted *MTBF*	Predicted *D*
+50		0.967666347	842.814815	0.0067605337
+50	2.5	0.952286406	562.128841	0.0076341007
+100		0.975565300	1127.629630	0.0040558166
+100	5	0.952329613	563.8815	0.0053581622
+500		0.991730147	3406.148148	0.0007260999
+500	25	0.952448783	566.944989	0.0028718404
+1000		0.995473487	6254.296296	0.0003206765
+1000	150	0.952481436	567.571576	0.0026600717

One sees from this data that in order to reach .99 reliability, another 500 sequences are required with no additional failures according to this prediction model. Note the optimistic behavior of *D* due to the typical nature of the additional sequences.

Chapter 6

Concluding Remarks

6.1 Synopsis

This research presents new ideas for statistical software testing and reliability modeling. The main theme is that measuring software quality requires accurate usage modeling and a comprehensive testing strategy as well as a reliability model. The model itself should be data driven and able to use all of the statistics of the test data to compute software quality measures. This work employed Markov chain techniques to achieve these goals.

The usage Markov chain is capable of describing software usage at varying degrees of detail and can be developed using only the specification before any code is written. No knowledge of potential use is necessary to construct the model but such data can be incorporated into the chain when it is available. From the usage chain testers obtain a suite of analytical results that, when properly interpreted, describe software usage characteristics. The usage chain becomes the source of statistical test cases, which are available automatically, when the code is ready to test.

As test cases are generated and applied to the software, failures may be observed and recorded. This information is organized as a second Markov chain, called the testing chain. The testing chain is capable of computing software quality estimates including the reliability, *MTBF*, and failure rate with each test case applied to the software. Thus testers gain immediate feedback

on the progress of testing (in the form of software quality measures) with each test case executed. The reliability model derived makes none of the usual assumptions about failure distribution(s) to compute these measures.

The progress of testing can be monitored with each test case applied to the software. Testers may wish to stop testing when some target reliability is reached or they can use the discriminant available analytically from the testing chain. This single number is computed after each test case and discriminates between what has occurred (the testing chain) and what should occur (the usage chain). When these two events are sufficiently similar, further testing will not likely yield significantly new information and testing can stop.

At any time during the test, testers can analyze the process of applying test cases and finding failures to gain insight into what quality and discrimination to expect in future sequences without actually generating the sequences. The model allows testers to look some number of test cases into the future and predict what the expected quality and progress will be at that time. Predicted values include the reliability, *MTBF*, failure rate, and discrimination.

It is possible to extend the model from the testing environment to the actual usage environment as well. In order to accomplish this transfer, the software must be built to capture appropriate usage history and maintain the testing chain in some manner. Of course, failure data must be incorporated by an engineer when it is reported. This will enable software to track its own quality after release of the code to the field.

The algorithms developed in this work have been implemented with encouraging results at the University of Tennessee Software Quality Laboratory. In [55] a Markov usage model was constructed and used to generate test cases for a box structure editor. The Markov chain for this project was found useful in describing this highly interactive system. The second use of Markov chains in statistical testing was a STARS case study [53]. During this project, an equation for

expected test case length was derived and several other analytical results were computed. The case study of Chapter 5 gives the flavor of the ongoing CRE project at the University of Tennessee [50-52]. The CRE is a tool set for the Cleanroom software development process. This project includes a programming language preprocessor, several file processing systems, graphical user interfaces, and subsystems of a mathematical nature. Included in the CRE are early increments of code that implement many of the ideas in this research. The Markov chain analysis of CRE included the usage analysis and the reliability analysis described here. The Markov model has performed well on our software projects and we believe that the field of software engineering will benefit from the contribution of these formal and mathematical methods.

6.2 Future Work

During the course of this research, many questions and problems arose that were ruled outside the scope of this dissertation. A major extension of immediate consequence is test optimization schemes that seek to minimize the number of sequences generated by the usage chain to reach asymptotics. One could simply generate a sequence, evaluate its contribution to the statistics of the testing chain via the discrimination measure and make a decision to execute it as a test case or discard it. However it should be possible to bias the transition probabilities of the usage chain based on the current makeup of the testing chain in order to significantly raise the probability of generating an informative sequence. Such a scheme would allow testers to achieve a high reliability in a shorter period of time and at a lower cost.

The prediction model in chapter four is based on the computed failure rate and the expected profile of future sequences. It does not attempt to predict the location of new failures, simply to model expected future behavior based on past trends. More complex prediction models

could be designed that track other pertinent data as well. This is a topic that deserves attention as further data is obtained on the accuracy of the model.

Another extension of this work is to develop similar algorithms and computations using other stochastic models. For example, a full multitype branching process (of which the finite state Markov chain is a special case) would require dealing with tree structured models. This additional power necessitates derivation of pertinent random variables yet could enable further detail to be included in the model. It may be that insight will be gained in determining characteristics of software that violate the Markov property. If this these characteristics are established, a full branching process will be necessary to accurately represent the usage of such software. An aspect of this work could be Markov chains as approximations that perform acceptably.

The analytical results computed in this research are from models which represent every version of the software during testing. Another formulation of this work is to build a separate model for each version of the software. Such an analysis would enable a certification result for just the last version of the code and be able to track the reliability growth as a result of engineering changes.

An in depth study of the role of entropy could prove extremely interesting and beneficial to software engineers. The role of "typical sequences" has been discussed in this work. However, a detailed analysis of quantifying the uncertainty of the statistical testing model will further characterize the model as a stochastic source. Such research could uncover additional stopping criteria and alternative sequence generation methods. The application of information theory in general and specifically entropy poses an extremely promising extension of this work.

The method of extracting a usage model has the potential for automation if an appropriately formal specification language is adopted. Research into formal specification techniques may make this possible at a future date. Another possibility for automating testing

exists in applying the sequences to the software. If a standard user interface mechanism is used, then the usage model may be substituted for this mechanism during testing. Sequences can be automatically applied and inferred making testing a completely automatic process. However, in order for this to be accomplished, the software engineers involved must make a commitment to formal and unambiguous specification and a standard form of accepting input from users.

Testing software through formal statistical methods has great potential as a software testing paradigm. This research has addressed only a small portion of the problems associated with statistical testing. Hopefully, research in this area will continue as its popularity grows and new problems are discovered and old methods are revised.

Bibliography

Bibliography

[1] N. Abramson, *Information Theory and Coding*, McGraw-Hill, New York, 1963.

[2] E. N. Adams, "Minimizing Cost Impact of Software Defects", *IBM Res. Division*, Rep. RC8228 (35669), April 11, 1980.

[3] T. W. Anderson and L. A. Goodman, "Statistical Inference About Markov Chains", *Annals of Mathematical Statistics*, Vol. 28, pp. 89-109 1957.

[4] R. Ash, *Information Theory*, John Wiley and Sons, New York, 1966.

[5] R. E. Blahut, *Principles and Practice of Information Theory*, Addison-Wesley, Reading, 1987.

[6] R. P. Bogott and M. A. Franklin, "Evaluation of Markov Program Models in Virtual Memory Systems", *Software Practice and Experience*, Vol. 5, pp. 337-346, 1975.

[7] T. L. Booth, "Estimation, Prediction, and Smoothing in Discrete Parameter Systems", *IEEE Transactions on Computers*, Vol. C-19, No. 12, pp. 1193-1203, December 1970.

[8] T. A. Budd and D. Angluin, "Two Notions of Correctness and Their Relation to Testing", *Acta Informatica*, No. 18, pp. 31-45, 1982.

[9] A. Celentano, S. C. Reghizzi, P.D. Vigna, C Ghezzi, G. Granata, and F. Savoretti, "Compiler Testing Using a Sentence Generator", *Software Practice and Experience*, Vol. 10, pp. 897-918, 1980.

[10] R. C. Cheung, "A User-Oriented Software Reliability Model", *IEEE Transactions on Software Engineering*, Vol. SE-6, pp. 118-125, March 1980.

[11] T. S. Chow, "Testing Software Design Modeled by Finite-State Machines", *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 1, pp. 178-187, May, 1978.

[12] A. B. Clarke and R. L. Disney, *Probability and Random Processes: A First Course with Applications*, John Wiley and Sons, New York, 1985.

[13] D. R. Cox and H. D. Miller, *The Theory of Stochastic Process*, Chapman and Hall, New York, 1965.

[14] P. A. Currit, M. Dyer, and H. D. Mills, "Certifying the Reliability of Software", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1, pp. 3-11, January, 1986.

[15] B. Curtis, "Measurement and Experimentation in Software Engineering", *Proceedings of the IEEE*, Vol. 68, No. 9, pp. 1144-1157, September, 1980.

[16] H. Cramer, *The Elements of Probability Theory*, Robert E. Krieger, Huntington, NY, 1955.

[17] S. R. Dalal and C. L. Mallows, "When Should One Stop Testing Software?", *Journal of the American Statistical Association*, Vol. 83, No. 403, pp. 872-879, September, 1988.

[18] J. L. Doob, *Stochastic Processes*, John Wiley and Sons, New York, 1953.

[19] J. Dodson, "The Suitability of APL for Automating Statistical Certification Testing", *Technical Report*, Department of Computer Science, University of Tennessee, May, 1991.

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

[21] J. W. Duran and J. J. Wiorkowski, "Quantifying Software Validity by Sampling", *IEEE Transactions on Reliability*, Vol. R-29, No. 2, pp. 141-144, June, 1980.

[22] W. Feller, *An Introduction to Probability Theory and its Applications*, Vol. 1, John Wiley and Sons, New York, 1950.

[23] L. Gillman, *APL: An Interactive Approach*, John Wiley and Sons, New York, 1984.

[24] D. Hamlet and R. Taylor, "Partition Testing Does Not Inspire Confidence", *IEEE Transactions on Software Engineering*, Vol. SE-16, No. 12, pp.1402-1411, December, 1990.

[25] W. Hetzel (Ed.), *Program Test Methods*, Prentice-Hall, Englewood Cliffs, NJ, 1972.

[26] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, PA, 1979.

[27] W. E. Howden, "Reliability of the Path Analysis Testing Strategy", *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, pp. 208-215, September 1976.

[28] G. R. Hudson, "Program Errors as a Birth and Death Process", Systems Development Corp., Rep. SP-3011, Dec. 4, 1967.

[29] A. Ianinio, B. Littlewood, J. D. Musa, and K. Okumoto, Criteria for Software Reliability Model Comparisons", *IEEE Transactions on Software Engineering*, Vol SE-10, pp. 687-691, November 1984.

[30] B. H. Juang and L. R. Rabiner, "A Probabilistic Distance Measure for Hidden Markov Models", *AT&T Technical Journal*, Vol. 64, No.2, pp. 391-408, February, 1985.

[31] Z. Jelinski and P. B. Moranda, "Software Reliability Research", in *Statistical Computer Performance Evaluation*, W. Friedberger, Ed., Academic Press, New York, 1972.

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

[33] J. G. Kemeny and J. L. Snell, *Finite Markov Chains*, Springer-Verlag, New York, 1976.

[34] S. Kullback, *Information Theory and Statistics*, John Wiley and Sons, New York, 1958.

[35] N. G. Leveson, "Software Safety: Why, What, and How", *Computing Surveys*, Vol. 18, No. 2, pp. 125-163, June 1986.

[36] S. E. Levinson, L. R. Rabiner, and M. M. Sondhi, "An Introduction to the Application of the Theory of Probabilistic Functions of a Markov Process to Automatic Speech Recognition", Bell Systems Technical Journal, 62, No. 4, pp. 1035-74, April, 1983.

[37] R. C. Linger and H. D. Mills, " A Case Study in Cleanroom Software Engineering: The IBM Cobol Restructuring Facility" in *Proceedings of COMPSAC '88*, IEEE, 1988.

[38] R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, MA, 1979.

[39] B. Littlewood, "How to Measure Software Reliability and How Not to", *Proceedings of the 3rd International Conference on Software Engineering*, May 1978.

[40] H. D. Mills, M. Dyer, and R. C. Linger, "Cleanroom Software Engineering", *IEEE Software*, pp. 19-24, September 1987.

[41] H. D. Mills, R. C. Linger and A. R. Hevner, *Principles of Information Systems Analysis and Design*, New York, Harcourt, Brace, Jovanovich, 1986.

[42] H. D. Mills and J. H. Poore, "Bringing Software Under Statistical Quality Control", *Quality Progress*, pp. 52-55, November, 1988.

[43] J. D. Musa, "A Theory of Software Reliability and its Application", *IEEE Transactions on Software Engineering*, Vol. SE-1, pp. 312-321, August, 1975.

[44] J. D. Musa, "The Measurement and Management of Software Reliability", *Proceedings of the IEEE*, Vol. 68, No. 9, pp. 1131-1143, September, 1980.

[45] G. J. Myers, *The Art of Software Testing*, John Wiley and Sons, New York, 1979.

[46] K. Narendra and M. Thathachar, *Learning Automata An Introduction*, Prentice Hall, Englewood Cliffs, NJ, 1989.

[47] E. Nelson, "Estimating Software Reliability from Test Data", *Microelectronics and Reliability*, Vol. 17, pp. 67-74, 1978.

[48] D. L. Parnas, "An Evaluation of Safety-Critical Software", *Communications of the ACM*, Vol. 23, No. 6, pp. 636-648, June, 1990.

[49] E. Parzen, *Stochastic Processes*, Holden-Day, San Francisco, 1962.

[50] J. H. Poore, M. Duitz, D. Fuhrer, H. Mao, C. Trammell, and J. A. Whittaker, "A Preprocessor for a Box Structured Design Language for C (Cleanroom Case Study)", *Technical Report*, Department of Computer Science, University of Tennessee, March 1991.

[51] J. H. Poore, J. Dodson, M. Duitz, D. Fuhrer, H. Mao, and J. A. Whittaker, "Certification Model (Cleanroom Case Study)", *Technical Report*, Department of Computer Science, University of Tennessee, March 1991.

[52] J. H. Poore, J. Dodson, M. Duitz, D. Fuhrer, H. Mao, and J. A. Whittaker, "OS/2 Cleanroom Environment (Cleanroom Case Study)", *Technical Report*, Department of Computer Science, University of Tennessee, March 1991.

[53] J. H. Poore, H. D. Mills, S. L. Hopkins and J. A. Whittaker, "Cleanroom Reliability Manager: A Case Study Using Cleanroom with Box Structures ADL", *Technical Report*, Software Engineering Technology, Inc., IBM-SID, STARS CRDL 1880, May 1990.

[54] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen, "On the Automated Generation of Program Test Data", *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, pp. 293-300, December, 1976.

[55] C. Senn, "Box Structure BDL Generator: A Clerical Tool to Support Software Development", Master's Thesis, Department of Computer Science, University of Tennessee, 1989.

[56] B. C. Sexton, "Statistical Testing of Software", Master's Thesis, Department of Computer Science, University of Tennessee, 1988.

[57] G. J. Schick and R. W. Wolverton, "An Analysis of Competing Software Reliability Models", *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 2, pp. 104-120, March 1978.

[58] M. L. Shooman, "Probabilistic Models for Software Reliability Prediction", in *Statistical Computer Performance Evaluation*, W. Friedberger, Ed., Academic Press, New York, 1972.

[59] M. L. Shooman, *Software Engineering -- Design, Reliability, and Management*, McGraw-Hill, New York, 1983.

[60] K. Siegrist, "Reliability of Systems with Markov Transfer of Control", *IEEE Transactions on Software Engineering*, Vol. 14, No. 7, pp. 1049-1053, July, 1988.

[61] M. G. Thomason, "Generating Functions for Stochastic Context-Free Grammars", *International Journal of Pattern Recognition and Artificial Intelligence*, Vol 4., No. 4, pp. 553-572, April 1990.

[62] M. G. Thomason and E. Granum, "Dynamic Programming Inference of Markov Networks from Finite Sets of Sample Strings", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-8, No. 4, pp. 491-501, July 1986.

[63] A. K. Trivedi and M. L. Shooman, "A Markov Model for the Evaluation of Computer Software Performance", Technical Report PI-EE/EP 74-011-EER110, Polytechnic Institute of New York, May 1974.

[64] J. A. Whittaker and J. H. Poore, "Markov Analysis of Software Specifications", in review, July, 1990.

[65] J. A. Whittaker and J. H. Poore, "Statistical Testing for Cleanroom Software Engineering", *Proceedings of the 25th Hawaii International Conference on Systems Science, IEEE*, January 1992.

[66] J. A. Whittaker and M. G. Thomason, "Markov Chain Representation of Sequences in Knowledge Based Systems", in review, January, 1991.

Vita

James A. Whittaker was born in Louisville, Kentucky on August 24, 1965. He received the Bachelor of Arts with a major in computer science from Bellarmine College, Louisville, in 1987. After graduation, he joined the Federal Bureau of Investigation as a systems analyst for one year. In 1988 he moved to Knoxville to attend the University of Tennessee as a graduate teaching assistant. He earned the Master of Science in 1990 and the Doctor of Philosophy in 1992, both in computer science. While at UT, he served as a research consultant to Software Engineering Technology, Inc.

The author is a member of the IEEE Computer Society and Sigma Xi.