

**OPTIMAL TEST CASE SELECTION FOR
MULTI-COMPONENT SOFTWARE SYSTEM**

A Thesis

**Submitted to the graduate committee of the
Louisiana State University and
Agricultural and Mechanical Engineering
In partial fulfillment of the
Requirement for the degree of
Master of Science in Industrial Engineering**

In

The Department of Industrial & Manufacturing Systems Engineering

**By
Praveen Babu Kysetti
B.S., Bangalore University, India, 2000
December, 2004**

ACKNOWLEDGEMENTS

This thesis is a product of three semesters of learning and dialogues with my Major Advisor Dr. Xiaoyue Jiang. I sincerely thank his innumerable suggestions and commend his patience.

I am very great full for my years at Industrial engineering at Louisiana State University. I am extremely indebted towards Dr. Thomas Ray and Dr. Charles McAllister for being a part of my thesis committee.

I would like to extend my gratitude to the department for awarding me research funds through ITAP (Information Technology Apprenticeship Program). I would like to thank my understanding employers at Emergent Technologies and LSU Center for Computation Technology for their co-operation.

There are many people who have helped and supported me and would take this opportunity to thank one and all.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS.....	ii
LIST OF FIGURES.....	v
ABSTRACT.....	vi
CHAPTER 1: INTRODUCTION.....	1
1.1 Software Development Process.....	2
1.2 Software Failure and Reliability.....	3
1.3 Software Reliability Growth Models.....	4
1.4 Modular Software Reliability models.....	5
1.5 Limitations of Existing Research.....	5
1.6 Research Goal	6
CHAPTER 2: LITERATURE SURVEY.....	8
2.1 Classification Schema.....	8
2.2 Single System Software.....	10
2.2.1 Failure Rate Models.....	10
2.2.1.a Jelinski-Moranda De-Eutrophication Model(1972).....	11
2.2.2 Failure Count Models.....	12
2.2.2.a Goel-Okumoto NHPP Model (1979).....	13
2.2.3 Static Models.....	14
2.2.4 Bayesian Models.....	15
2.2.4.a Littlewood and Verall Bayesian Model (1973).....	16
2.2.5 Markov Models.....	17
2.2.5.a Goel's Model (1985).....	18
2.3 Multi Component Models.....	19
2.3.1 Modular Software Architecture.....	19
2.3.2 State Based Models.....	21
2.3.2.a Littlewood Model(1979).....	22
2.3.2.b Laprie's Model(1984)	23
2.3.3 Path Based Models.....	25
2.3.4 Additive Models.....	25
2.4 Optimal Stopping Models.....	26
2.5 Testing Strategies.....	28
CHAPTER 3: MATHEMATICAL MODEL.....	30
3.1 Notation.....	30
3.2 Model Assumptions.....	31
3.3 Architecture.....	33
3.4 Component Failure Behavior.....	35
3.5 Solution Method.....	39

CHAPTER 4: OPTIMAL TEST CASE SELECTION.....	41
4.1 Stochastic Dynamic Programming.....	41
4.2 Value Equation of the System	41
4.3 Algorithm for Optimal Decision	45
4.4 Parallel Algorithm.....	51
CHAPTER 5: RESULTS AND SIMULATION.....	55
5.1 Pictorial Representation of the Algorithm for Optimal Test Case Selection.....	55
5.2 Policy Determination	58
5.3 Limitations.....	58
5.4 Results and Analysis.....	59
5.5 Conclusions, Extensions & Application.....	62
REFERENCES.....	65
VITA.....	69

LIST OF FIGURES

Fig 2.1	Classification Scheme.....	9
Fig 3.1	$g(t)$ v/s t (continuous)	36
Fig 3.2	$G(t)$ v/s t (continuous)	36
Fig 3.3	$g(t)$ v/s t (Discrete)	37
Fig 4.1	Backward recursion MDP representation	45
Fig 4.2	Markov Decision Process(MDP).....	46
Fig 4.3	Algorithm.....	47
Fig 4.4	Matlab Functions.....	50
Fig 4.5	Parallel Architecture.....	52
Fig 4.6	Parallel simulation.....	53
Fig 5.1	Value function v/s test stages (3 components 2 test cases)	59
Fig 5.2	Value function v/s test stages (3 components 2 test cases)	60
Fig 5.3	Value function v/s test stages (2 components 2 test cases)	61
Fig 5.4	Value function v/s test stages (3 components 3 test cases)	62

ABSTRACT

The omnipresence of software has forced upon the industry to produce efficient software in a short time. These requirements can be met by code reusability and software testing. Code reusability is achieved by developing software as components/modules rather than a single block. Software coding teams are becoming large to satiate the need of massive requirements. Large teams could work easily if software is developed in a modular fashion. It would be pointless to have software that would crash often. Testing makes the software more reliable. Modularity and reliability is the need of the day.

Testing is usually carried out using test cases that target a class of software faults or a specific module. Usage of different test cases has an idiosyncratic effect on the reliability of the software system. Proposed research develops a model to determine the optimal test case policy selection that considers a modular software system with specific test cases in a stipulated testing time.

The proposed model, models the failure behavior of each component using a conditional NHPP (Non-homogeneous Poisson process) and the interactions of the components by a CTMC (continuous time Markov chain). The initial number of bugs and the bug detection rate are known distributions. Dynamic programming is used as a tool in determining the optimal test case policy. The complete model is simulated using Matlab.

The Markov decision process is computationally intensive but the implementation of the algorithm is meticulously optimized to eliminate repeat calculations. This has saved roughly 25-40% in processing time for different variations of the problem.

Index Terms: Software reliability, Modular software, Dynamic programming, test case, Bayesian analysis

CHAPTER 1: INTRODUCTION

The twentieth century has been dubbed the information age. Information is synonymous to “empowerment”. Computers and software together have delivered the empowerment rather emphatically. This can be witnessed in every single nuance of the modern life. It has changed the way we live, trade, explore and enjoy life for the better.

Software is a program that provides instructions to silicon based processor to function, generating the desired result. Software is broadly classified as operating system and application software. The Operating system carries out the basic operations of a processor while application software works on a level higher than operating system providing special services.

Software plays a key role in the modern life. Software is a functioning element in home appliances, automobiles, space ships, banking, communications, manufacture etc. This has increased our dependence on machines and its reliability. The idea of unreliable software may be unimaginable and damaging. A malfunctioning pacemaker in a heart patient or a Mars path finder that has lost contact due to a software bug or a hacker taking advantage of a bug in financial system to siphon away cash electronically in the luxury of his home portrays the problem. Inadequate testing of the delivery system of Titan IV rocket lead to two Titan rockets being lost. Expensive military equipment necessary to the U.S. Governments defense program (namely early warning satellites) were unable to be deployed. The head of the N.R.O. (National Reconnaissance Office) has attributed this error to "a misplaced decimal point" in software, which controlled the rocket. This has lead to devising methods to make the software more reliable (<http://www.beanmeadowcroft.com/reports/systemfailure>).

The software is made reliable by following stringent coding and testing standards. It is generally a known fact in the industry that testing constitutes sizeable amount of cost and time.

The software creating teams would like to manage their resources better while not compromising on reliability. Modeling of testing and bug detection process helps achieve this goal.

Testing is the only effective way of conforming to high software reliability and one of the five important stages in software development

1.1 Software Development Process

The software development process consists of five phases, starting with analysis, design, coding, testing and finally operation (Pham 2003).The phases are described below.

In the analysis phase the software team communicates with the customer to understand their needs. The project is split into major subgroups.

In the design phase the work is further subdivided. A plan of action is charted out for the whole team. The design phase takes care of those aspects that are common to the whole development team. An improper design could ruin a whole project as a mistake is difficult to reverse in the later stages.

The coding phase is when each programmer is assigned a specific task. Code is developed and tested in a minimal scale. In this phase attention is given to future code maintenance and ease of debugging.

Testing involves an effort of designing and executing test cases in a systematic fashion. The bugs determined here are removed. Usually testing involves usage of data representative of the actual scenario.

Operation phase usually involves in installing, training and maintenance and occasionally debugging.

The main goal of software testing is to eliminate bugs and obtain reliable software. If the total number of bugs in software is known before hand then the number of bugs eliminated

would indicate the system reliability. But, in practice we don't know how many bugs are embedded into the software and it is required to quantify reliability.

“To measure is to know” (Lord Kelvin).

1.2 Software Failure and Reliability

Software has a reliability of 100% if it does not fail or 0% if it fails is not appropriate and is a rigid definition. This is true as software generates outputs for a varied number of input types or user profiles. Each of the user profiles utilizes different paths or functions to generate outputs. A certain user profile may function flawlessly while a different one may fail and we cannot conclude that software has zero reliability.

Reliability of software is not deterministic, as a faulty program can still give correct output sometimes. Therefore reliability is best measured probabilistically (Roger Cheung 1980). A more accurate definition of software reliability is if F is a class of faults, defined arbitrarily and T is a measure of relevant time, units of which are dictated by the application on hand. Then the reliability of software package with respect to class F and T is the probability that no fault of the class occurs during the execution of the program for a specified period of relevant time (Goel 1985).

Measuring software reliability alone does not solve the problem of achieving reliable software; it merely reflects the quality of software on hand. Testing is usually a lengthy process in the software industry accounting for 40-50% of the development process. The bugs detected as time elapses is used to update the reliability information of the software. This information could be translated into determining the testing time or resources required in order to meet various criteria of reliability or cost. The reliability of the software is usually estimated by devising mathematical models describing a typical behavior of a debugging process.

Software reliability models in the 1970's usually were directed at single unit software systems and were called software reliability growth models (SRGM) and later on models were devised to address multi component systems or modular systems.

In all the models the basic definitions are briefly described here

$R(t)$ = Probability that software will be functioning without failure under a given environmental condition during time $[0, t)$

T the random failure free time interval of system

$F(t)$ Cumulative distribution function of T , $f(t)$ its density which is

$$f(t) = F'(t) \tag{1.1}$$

$$R(t) = 1 - F(t), t \geq 0 \tag{1.2}$$

Failure rate (hazard rate, failure intensity) $r(t)$ of $F(t)$ is defined as

$$r(t) = \lim_{\Delta t \rightarrow 0} \frac{R(t) - R(\Delta t + t)}{\Delta t R(t)} = \frac{f(t)}{R(t)} = \frac{f(t)}{1 - F(t)} \tag{1.3}$$

Software reliability models are broadly classified based on the software structure. The two different structures of the software are single system and multi component software.

1.3 Software Reliability Growth Models

Software reliability models developed mostly were applied in the debugging phase of software development. The models considered software as a monolithic whole. The only data used is the failure data. The stochastic models called software reliability growth models (SRGM's) or black box models modeled the failure behavior based on cumulative failures or times between failures. The last decade has seen the development of modular software and its wide acceptability in the industry which has initiated a need in the reliability models for modular systems.

1.4 Modular Software Reliability Models

Dolbec and Shepard (1996) described the features of modular software as follows. The modular software structure predominantly is deciding what a component represents. They are usually components whose interactions with the system are the specified input and output. All other resources are never shared with any other part of the system. This kind of structure is implemented using two methods, Structure design (SD) and Object oriented design (OOD). OOD approach is the most famous and is implemented in languages like Java, C++, and Small-talk. The approach isolates errors, code reuse and maintainability is much easier.

The reliability models developed for modular software systems are called white box models which consider the internal structure of the software in reliability estimation. Most of the models of this category are utilization of SRGM to model individual component failure behavior and combining the results for the complete system.

1.5 Limitations of Existing Research

Jelinski & Moranda (1972), Goel & Okumoto (1979) NHPP model, Schik & Wolverton (1978) are some of the software reliability growth models, which consider software as a single unit ignoring the architecture of the system. Incorporating the architecture enhances the ability to design better test cases and ability to target more faults.

Modular software models like Littlewood (1979), Kubat (1989), and Cheung (1980) are the cornerstone models in this category. Krishnamurthy & Mathur (1997) and Rajgopal & Mazumdar (1999) described few other models. Most of the models basically discuss calculation of system reliability from component reliabilities. Some of the papers discuss the aspect of using techniques to evaluate the affect of component reliability on system reliability and hence target testing that module. Operation or user profile has also been considered. Reliability estimates are

defined as a function of different user profile. Each user profile uses different modules and hence different system reliabilities.

Optimal stopping models determine stopping times for software testing in software reliability models. Dallal & Mallows (1998), Kubat (1998), Zheng (1999) are some of the optimal stopping models. The different models consider stopping based on criteria like reliability or cost. Some of the models determine stopping times using multiple test cases. The stopping criterion is determined using Bayesian decision approach.

All the models fail to address the situation where a modular software system with multiple test cases exists and knowing the testing time how to determine which test cases would result in better reliability.

1.6 Research Goal

Software in the present day is developed as modules and integrated to perform a specific task. On integration the system is tested to detect faults and correct them. Testing is carried out systematically using various test cases. Each test case is designed to target a family of bugs or a specific module. The test cases used and the number of times it is executed affect the system reliability. Before the actual testing begins the knowledge of the sequence of testing cases to be used would help us obtain better reliability and managing testing resources more efficiently knowing the stopping time.

Briefly, the proposed research tries to determine the optimal test case policy for a modular software system in a stipulated testing time. The model considers modular software architecture which is more accurate than the considering the whole system as a single unit. The testing time is assumed to be known and the failure behavior of each component is modeled by a

conditional non-homogeneous Poisson process (NHPP). This work is unique and addresses the shortcomings of the previous research.

Testing and bug detection is modeled as a Markov process. Each Markov state being defined by the number of test cases used and bugs detected in each module. Test case policy is to be determined by the large number of state space combinations that arises out the model. The process is computationally intensive

The research further addresses the issue of keeping the optimal policy determination process dynamic by way of Bayesian updating. As discussed earlier this could help in having a general model based on the prior distributions assumed. The complete process is simulated using Matlab enabling to analyze the results of the model developed here.

In Chapter 2, we describe and elucidate the Classification of software reliability models, Chapter 3 describes proposed mathematical and the stochastic properties of the component and the system as a whole in the third chapter, further defining the reliability calculations. In Chapter 4, Dynamic Programming Decision making tool for “Test Case Selection” is shown. Chapter 5, we show the results obtained by simulating the complete model in “Matlab”.

CHAPTER 2: LITERATURE SURVEY

In the previous chapter we introduced software reliability and the direction of research that is presented in this paper. This chapter would give an overview of software reliability models and the underlying mathematical concepts that profoundly impacts the understanding of the present work.

2.1 Classification Schema

The software reliability models are broadly classified into black box (single system) and white box (multi component software) models.

The black box models are classified further into models based on Inter failure times, Failure count and static models. Markov assumptions are also made for some models and they are overlapping with the failure count and inter failure time modeling. Note the fact that all known models can be extended to be a Bayesian model. If the model is Bayesian then we are estimating the model parameters using Bayesian techniques.

White box models are those that model modular software systems considering the architecture of the system. Popstajanova and Trivedi (2001) presented a classification of software reliability models that classified multi component software systems. The white box models are broadly classified into State based, Path based and Additive models.

The state based models are further classified into continuous time Markov chains (CTMC), discrete time Markov chain (DTMC) and semi-Markov models. All these Markov models fall into either absorbing or irreducible class of Markov chains. Terminating software is represented by absorbing type and continuously operating system is represented by irreducible chain.

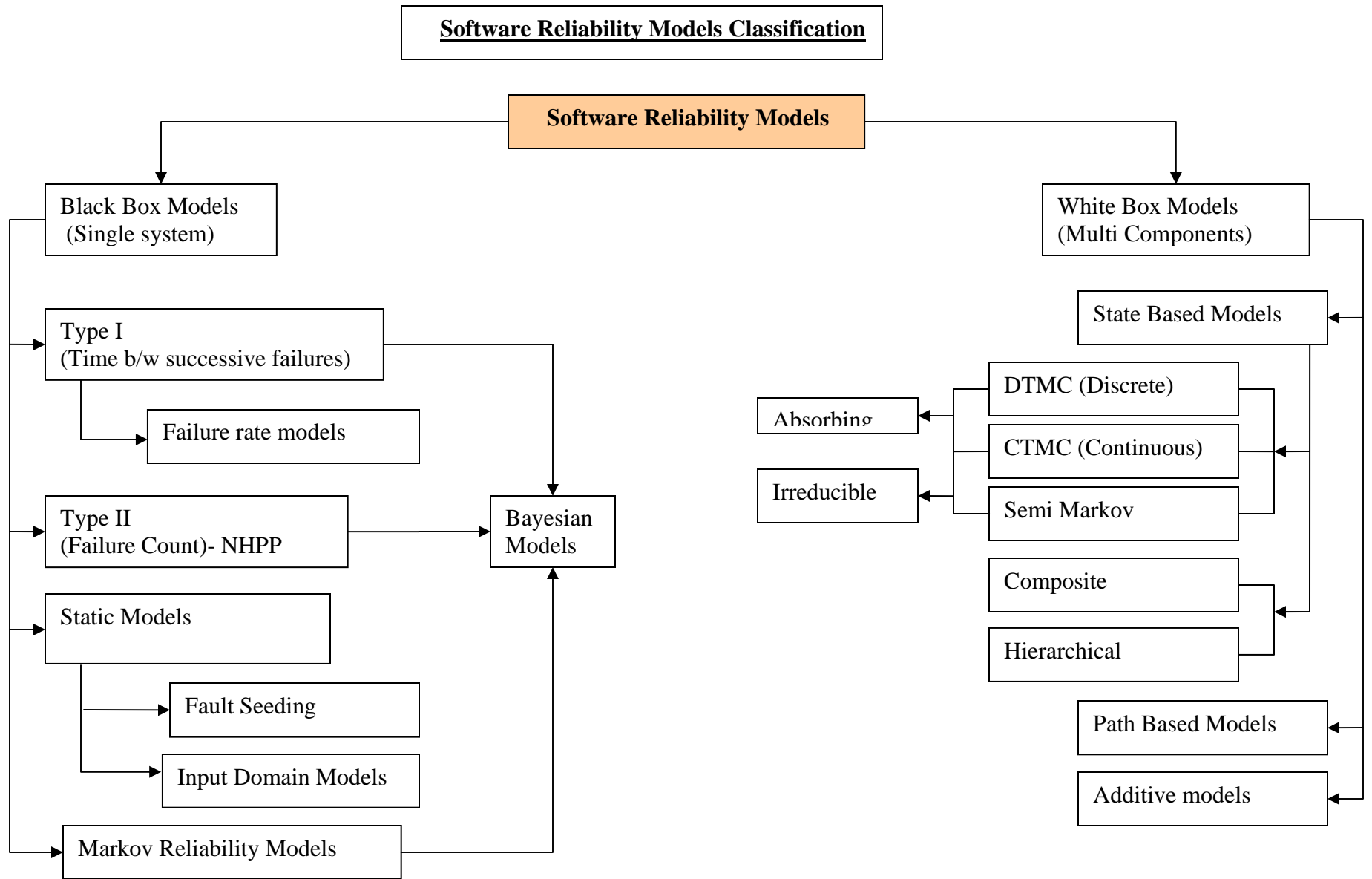


Figure 2.1 Classification Schema

Bayesian models are extensions of software reliability growth models. In Bayesian models the stochastic model reflects the system accurately at that instant based on the latest failure data. In Bayesian models the distributions representing the failure behavior is represented by a prior and posterior distribution. The prior distribution reflects the expert knowledge available on the specific failure process which is incorporated in the stochastic model. Bayesian models have wide acceptance due to these reasons in the software industry.

In the following sections we describe each class of software reliability models with a representative reliability model.

2.2 Single System Software

This model considers the whole software as a single monolithic system and the structure of the model is not considered in the process of reliability estimation of the software system. The reliability is solely estimated based on the failure history. Popular reliability estimation models include the Goel-Okumoto (1979) model, and the Jelinski-Moranda model (1972). Thus, for example, these models do not account for the structure of the software. Single system software failure behavior is usually modeled using software reliability growth model (SRGM) and classified as times between failure and failure count models.

2.2.1 Failure Rate Models

Inter failure times are the main modeling parameter for these models. Generally it is expected that the time for next failure increase as more bugs are detected and corrected. This may not be always correct as inter failure times are random variables and subjected to statistical fluctuations.

The failure rate is a modeling tool for type I models. The reliability function is a non decreasing function of mission time. This demonstrates an increase in the software credibility. Another method of modeling inter failure times is that each T_i is a random function of previous T_i .

2.2.1.a Jelinski Moranda De-Eutrophication model (1972)

▪ Assumptions

- 1.) Initial software faults are known with fixed costs.
- 2.) Debugging is perfect
- 3.) Time between failures are independent, exponentially distributed random quantities.
- 4.) All remaining faults contribute equally to failure intensity
- 5.) Inter failure times are exponentially distributed random variables with parameter

Initial number of faults: N_0

Initial failure intensity $N_0\phi$

Failure intensity contributed by each fault: ϕ

After k failures the failure intensity $(N_0 - k)\phi$

Time between (i-1) and i failure T_i

▪ Estimates

$$\lambda(i) = \phi[N_0 - i + 1] \quad i = 1, 2, \dots, N_0 \quad (2.1)$$

$$P(T_i < t_i) = \phi[N_0 - i + 1] \exp(-\phi(N_0 - (i - 1)t_i) \quad (2.2)$$

Parameters of the J-M model are estimated by the maximum likelihood estimation. Suppose

failure data $t = \{t_1, t_2, t_3, t_4, \dots, t_n, n > 0\}$

Parameter ϕ and N_0 in JM model

2.2.2 Failure Count Models

A counting process model for $N(t)$ the number of times a software fails in time interval $[0, t]$. All variations of Poisson distributions belong to this model. Each model differs only in the Poisson distribution parameters. Failure counts are assumed to follow a known stochastic process with a time dependent discrete or continuous failure rate.

A counting process $\{N(t), t > 0\}$ modeled by NHPP, $N(t)$ follows a Poisson distribution with parameter $m(t)$.

- **Non-homogeneous Poisson process**

NHPP are useful to describe failure processes having reliability growth or deterioration.

The cumulative number of faults $N(t)$ can be described by a NHPP.

Assumptions of the NHPP process are:

$$N(0) = 0$$

$\{N(t), t \geq 0\}$ has independent increments

$$P\{N(t+h) - N(t) = 1\} = \lambda(t) + o(h) \quad (2.3)$$

$$P\{N(t+h) - N(t) \geq 2\} = o(h) \quad (2.4)$$

$o(h)$ Denotes a quantity which tends to zero as h tends to zero.

Probability that $N(t)$ is a given integer is given by

$$P\{N(t) = n\} = \frac{[m(t)]^n * \exp[-m(t)]}{n!} \quad (2.5)$$

Intensity function is defined as the derivative of the mean value function of NHPP failure process where $m(t)$ is the mean function.

$$\lambda(t) = \frac{dm(t)}{dt} \quad (2.6)$$

Different $m(t)$ functions define different NHPP models. The simplest is if $\lambda(t)$ is constant we obtain a homogeneous Poisson process.

2.2.2.a Goel-Okumoto NHPP Model (1979)

Most of the models till G-O NHPP model assumed multiple Poisson process to model the parameters. In 1979, Goel and Okumoto presented an intuitive model that assumes that the cumulative failure process follows a NHPP with a simple mean value function $m(t)$.

▪ Model Assumptions

The cumulative number of bugs detected at time t follows a Poisson distribution. The faults are independent and have the same probability of being detected. All detected faults are removed immediately and perfect debugging is assumed. G-O model assumes the failure process to be modeled by a NHPP with a mean function

$$m(t) = a(1 - \exp(-bt)), \quad a > 0, b > 0 \quad (2.7)$$

▪ Estimates

$$P\{N(t) = n\} = \frac{[m(t)]^n * \exp[-m(t)]}{n!} \quad (2.8)$$

The expected number of remaining software faults at time t is defined as

$$\overline{E[N(t)]} = E[N(\infty) - N(t)] \quad (2.9)$$

This is calculated as.

$$\overline{E[N(t)]} = m(\infty) - m(t) = a - a(1 - \exp(-bt)) = a \exp(-bt) \quad (2.10)$$

The reliability function at time t_0 is exponential given by

$$R(t | t_0) = [m(t_0) - m(t)] \cdot k = 0,1,2... \quad (2.11)$$

2.2.3 Static Models

These models use statistical techniques to evaluate software reliability and can be evaluated only if complete failure data is available. These models were developed in the initial stage of reliability model evolution and are now seldom used as it cannot incorporate the structure of the software.

- **Fault seeding models and Input domain models**

In fault seeding models, a known number of bugs are seeded (planted) in the program. The software is then tested for faults. The bugs detected would have a combination of inherent faults and seeded faults. The number of inherent faults in the software is calculated based on the inherent and seeded faults detected using maximum likelihood estimation and combinatorics. The drawback of this approach is that the seeded faults and the inherent faults must have the same detection probability. This is difficult to achieve.

The basic approach in input domain based model is to generate a set of test cases from an input distribution. The reliability measure is calculated from the number of failures observed during symbolic or physical execution of sampled test cases. The test cases selected from the representative input space is executed recording the results. The probability of the success can be evaluated using statistical techniques.

It is generally difficult to estimate the input distribution (operational profile); generally the input distribution is obtained based on the different paths that exist in the software.

2.2.4 Bayesian Models

A main difficulty encountered in using Markov and NHPP models is parameter estimation. Maximum likelihood and least square methods. Estimates are very unstable and often do not reflect the data.

Usually it is difficult to estimate the posterior distribution. But if the priors are beta or gamma distribution the posteriors are easy to calculate. Bayesian models estimate model parameters accurately than maximum likelihood estimators.

Various models have been developed in this category which is basically extensions of single component software reliability models. Littlewood-Verall (1979) model is the cornerstone model in this category. Other important models in this category are Littlewood & Sofer (1987) (Bayesian modification of J-M (1972) model), Meinhold & Singpurwalla (1983) and Jewell (1985).

- **Unification of Software Reliability models**

The software industry is interested in a general model rather than different models. Unification can be achieved by Bayesian or Self exciting Poisson process (Singpurwalla & Wilson (1994)).

Langberg and Singpurwalla (1985) assumed prior distributions on parameters N and Λ of the model by Jelinski and Moranda (1972). Goel and Okumoto and Littlewood and Verall (1973) arise as special cases. This is the exact purpose of unification. Different models arise as special cases of general model.

- **Illustration**

Following is a general description of Bayesian model.

Bayesian models are generally used to combine previous knowledge and a present data of a process (failure behavior of software) to make accurate prediction and estimations.

Let,

θ -Required parameters on which has prior knowledge or data available

Ω -Total parameter space

$g(\theta)$ -Prior density

$\tilde{t} = \{t_1, t_2, t_3, \dots, t_n; n > 0\}$

By, Baye's theorem posterior density of θ given \tilde{t} is

$$h(\theta | \tilde{t}) = \frac{f(\tilde{t} | \theta)g(\theta)}{\int_{\Omega} f(\tilde{t} | \theta)g(\theta)d\theta}, \theta \in \Omega \quad (2.12)$$

$f(\tilde{t} | \theta)$ - is the likelihood of the data set \tilde{t} given θ .

2.2.4.a Littlewood and Verall Bayesian Model (1973)

This model is a Bayesian extension to reliability model based on inter failure times. The inter failure times are assumed to follow an exponential distribution with a random variable parameter. The distribution of the random variable is gamma distribution. The failure rate is assumed to be random rather than constant.

Assumptions:

Successive execution times between failures are independent exponential random variables with parameters $\xi_i, i = 1 \dots n$.

The ξ_i 's for a sequence of independent random variables, each with a gamma distribution of parameters α and $\psi(i)$.

The function $\psi(i)$ is an increasing function of i that describes the quality of the programmer and the difficulty of the task.

Littlewood and Verrall suggest linear and quadratic forms for the $\psi(i)$ function:

$$\psi(i) = \beta_0 + \beta_1 i \quad (2.13)$$

$$\psi(i) = \beta_0 + \beta_1 i^2 \quad (2.14)$$

Note that the Littlewood-Verrall model, like the Jelinski-Moranda model, attempts to predict the effects of debugging on the failure rate.

$$f(t_i | \lambda_i) = \lambda_i \exp(-\lambda_i t_i) \quad (2.15)$$

$$f(\lambda_i | \alpha, \psi(i)) = \frac{\psi(i)^\alpha \lambda_i^{(\alpha-1)} \exp(-\psi(i)\lambda)}{\Gamma(\alpha)} \quad (2.16)$$

Jelinski -Moranda models is often inaccurate for the reason that it uses maximum likelihood estimators to estimate the model parameters. Littlewood and Sofer (1987) model is a Bayesian extension to JM model.

2.2.5 Markov Models

In a Markov process $N(t)$ represents the number of detected faults in the software system. Markov models are very useful in studying the software fault removal process. The state of the process at time t is here the number of faults remaining at that time. If the fault removal process is perfect it is represented by a pure death Markov model. If the fault removal is imperfect, i.e. new faults could be introduced while debugging then the model is birth-death Markov process.

A Markov process is characterized by its state space together with the transition probabilities between these states. The Markov assumption implies the memory less property of

the process, which is a helpful simplification of many stochastic processes and is associated with the exponential property.

The Markov property essentially translates into the fact that the future behavior is independent of the past history of the process. CTMC is used to obtain system reliability based on component reliabilities. The initial condition of the process together with the transition probabilities completely determines the stochastic behavior of the Markov process.

Jelinski & Moranda (1972), Goel (1985), Schick & Wolverton (1978) and Shantikumar (1981) are examples of Markov models. JM (1972) model was the earliest in this category and basis of future Markov models.

The same Markov process can be used to model both component interactions in a multi component system and the bug detection process in software.

2.2.5.a Goel' Model(1985)

Goel's model is essentially a Markov based software reliability model for single system software. The model is a generalization of JM (1972) considering imperfect debugging.

Most of the assumptions are similar to Jelinski-Moranda model both with some changes which are described below.

1. A detected fault is removed with a probability p and the fault not being removed is $q = 1 - p$.
2. Counting process of the cumulative number of detected faults at time t is modeled as Markov counting process. The transition probabilities are dependent on perfect-imperfect debugging probability. Such a Markov model is called birth-death process.
3. Times between transitions are assumed with parameters dependent on remaining fault content.

Failure intensity between $(i - 1)$ and i failure is

$$\lambda(i) = \phi[N_0 - p(i + 1)] \quad i = 1, 2, \dots, N_0 \quad (2.17)$$

$$\lambda(i) = \phi.p \left[\left(\frac{N_0}{p} \right) - (i + 1) \right] \quad i = 1, 2, \dots, N_0 \quad (2.18)$$

Note the model is similar to JM model with the parameters

$$\phi \rightarrow \phi.p \quad \text{and} \quad N_0 \rightarrow \frac{N_0}{p}$$

The models described above are part of single component failure behavior models. As mentioned before modular software development has many advantages and hence need reliability models. In the coming sections Multi –component software systems and respective failure models is discussed.

2.3 Multi Component Models

White Box model or Multi component system considers the Software Architecture of the system, its interactions with the different modules. White box models are used to model component based software. Such models seek to explicitly incorporate the testing method used during the testing phase, as well as the structure of the software being tested. Littlewood (1979), Kubat (1989), Cheung (1980), Krishnamurthy & Mathur (1997), Kyle Siegriest (1988) and Rajgopal & Mazumdar (1999) are some the models in the modular software category.

Modular systems have a general framework in which the system reliability is calculated. All modular systems are grouped as state based, path based and additive models.

2.3.1 Modular Software Architecture

The framework in general can be defined as the foundation required for defining the mathematical model accurately. The framework for the component based architectural reliability

models follows three steps (Katerina & Trivedi 2001). We use this approach as it is a good framework for a modular software system.

- **Module Identification**

The core aspect on which the architectural modeling approach revolves is that of the module/component. A component is the fragment of code could be designed, implemented and tested as a standalone entity. In terms of programming language this could be considered as a function or method.

- **Software Architecture**

Software architecture could be defined as the mathematical model representing the interactions of the modules. The different aspects under consideration while in this phase is that of whether the system is

1. Termination type or continuous running type of system
2. Defining where the control of the system is at any point in the system whether in a module or in a process of transfer between modules.
3. Some the architectures used are series, parallel models, discrete time Markov models, continuous Markov models.

State Based Models: DTMC (Discrete Time Markov Chain), CTMC (Continuous Time Markov Chain), semi Markov process.

- **Failure Behavior**

Failure Behavior is defined and associated with architecture of the system. Failure typically occurs in these 3 modes:

1. Execution in a module
2. Control transfer

3. Combination of the above two modes

Failure behavior is defined mathematically in terms of reliabilities or failure rate that is in terms of (Constants or Time Dependent variables).

The interactions are defined as control transfers, essentially implying that the architecture is a control-flow graph where the nodes of the graph represent modules and its transitions represent transfer of control between the modules. The failure behavior for these modules (and the associated interfaces) is then specified in terms of failure rates or reliabilities (which are assumed to be known or are computed separately from SRGM's). The failure behavior is then combined with the architecture to estimate overall software reliability as a function of component reliabilities. Failure behavior is combined with the architecture using three methods namely state based, path based and additive models.

2.3.2 State Based Models

Architecture based models assume that components fail independently and that a component failure will ultimately lead to a system failure. Unlike hardware reliability every component is always in use software components need a utilization factor. In state based models. State based models are generally modeled using Markov models like CTMC, DTMC or semi Markov models. Models in this category are Littlewood (1979), Cheung (1980), Kubat (1989) and Laprie (1984). System reliability estimates are obtained using both architecture and failure model. This is achieved using two methods, Composite and hierarchical solution approach.

▪ Composite-Hierarchical Approach

The state based models are further classified into composite and hierarchical based on the solution approach to obtain the reliability of the system. Composite method combines the architecture model with the failure model and then solved for reliability prediction. If the

architecture model is first solved first and then superimposed on the failure behavior on the architecture model solution to predict reliability.

Littlewood (1973) and Laprie (1984) are state based models for multi component systems.

2.3.2.a Littlewood Model (1979)

The model is among the earliest in the white box model with state based approach. An approximation of the overall failure process is carried out assuming the failures occur rarely to obtain an analytically tractable solution. The remainder of the paper models a cost estimation based on software failures. Below we discuss the architecture, failure behavior and solution process of Littlewood model.

▪ Architecture

1. Continuously running applications are represented by irreducible semi Markov process, which describes the software architecture with continuous time Markov chain.
2. The program comprises a finite number of modules and the transfer of control between modules is described by the probability p_{ij} between the modules i and j .
3. Time spent in each module has a general distribution $F_{ij}(t)$

▪ Failure Behavior

1. There are two types of failures one within the module and the other while there is a transfer of control. Failure within the modules follows a Poisson distribution with parameter λ_i .
2. The transfer of control is subjected to a failure by probability v_{ij} when module i call module j .

- **Solution Method**

Composite method where in the architectural model is superimposed on failure model and then solved. Total number of failures in $(0,t]$ over all modules and the interfaces denoted by $N(t)$. The complete failure point process is possible to calculate but the exact solution is too complex to be of any practical use. The asymptotic process approximation for $N(t)$ is obtained under the assumption that the system has more up time than down time (fail state).

The failure rate of the Poisson process is given by

$$\lambda_s = \sum_i a_i \lambda_i + \sum_{i,j} b_{ij} \nu_{ij} \quad (2.19)$$

$$a_i = \frac{\pi_i \sum_j p_{ij} m_{ij}}{\sum_i \pi_i \sum_j p_{ij} m_{ij}} \text{ represents the proportion of time spent in module } i. \quad (2.20)$$

$$b_{ij} = \frac{\pi_i p_{ij}}{\sum_i \pi_i \sum_j p_{ij} m_{ij}} \text{ is the frequency of transfer of control.} \quad (2.21)$$

The variable a_i , b_i depends on only the parameters that characterize the software architecture.

The parameters are transition probability between two modules p_{ij} , steady state probability of embedded Markov chain π_i and mean execution time m_{ij} .

2.3.2.b Laprie's Model (1984)

Laprie's model is a special case of Littlewood model.

- **Architecture**

Software system is a multi component system with n modules. The control transfer b/w modules are represented by a CTMC. Parameters of the CTMC architecture are

$$\text{Mean execution time of component } i = \frac{1}{\mu_i}$$

Probability of transfer of control from module i to j p_{ij}

- **Failure Behavior:**

Each component fails with constant failure rate λ_i

- **Solution Method:**

The architecture is modeled as a CTMC where the system is up in state $i, 0 \leq i \leq n$ and $(n+1)$ is the absorbing or failure state.

Associated generator matrix between up states $B = [b_{ij}]$

$$b_{ii} = -(\mu_i + \lambda_i) \tag{2.22}$$

$$b_{ij} = p_{ij}\mu_i \text{ for } i \neq j \tag{2.23}$$

The matrix B is seen as the sum of two generator matrices such that execution process is governed by B' whose diagonal entries are equal to $-\mu_i$ and it's off diagonal entries to $p_{ij}\mu_i$. The failure process is governed by B'' whose diagonal entries are equal to $-\lambda_i$ and off diagonal entries are zero. It assumed that $\lambda_i \ll \mu_i$ many exchanges of control occur before a failure occurs. The execution process converges towards steady state before failure is likely to occur. Therefore execution process is asymptotic which allows us to adopt hierarchical solution approach.

System failure rate tends to

$$\lambda_s = \sum_{i=1}^n \pi_i \lambda_i \tag{2.24}$$

Steady state probabilities are $\pi = [\pi_i]$ is the solution for $\pi B = 0$. The physical interpretation of the parameters can be given as follows

π_i is the proportion of time spent in state i when no failures occur and

$\pi_i \lambda_i$ = Equivalent failure rate of component i.

2.3.3 Path Based Models

Similar to state based models, the path based models consider software architecture with components and interfaces. Initially the different paths in system are obtained either experimentally or algorithmically. Path reliability is the product of all component reliabilities along the path. The system reliability is average of all the path reliabilities. State based models analytically account for the infinite loops in a path but path based models terminate the loop to one or to an average execution time of the path.

Shooman model (1976) considers reliability of modular software introducing the path based approach by using the frequencies with which different paths are run. Krishnamurthy and Mathur (1997) developed a method to combine architecture and failure process by estimating the path reliabilities based on the sequence of components executed for a single test run and the average over all test runs to obtain the system reliability.

2.3.4 Additive Models

Additive models consider software testing phase and each component reliability is modeled by NHPP. This implies system failure rate is also NHPP with cumulative number of failures and failure intensity functions that are the sums of corresponding function of each component. Additive model do not consider architecture of the software. Xie & Wohlin (1995) developed an additive based architecture model.

Some other modular software models are that of Gokhale, Lyu and Trivedi (2001) and Wang, Wu, Chen (2001). Most of the existing analytical models to predict the reliability and performance of component based systems are based on Markovian assumptions [Roger Cheung (1980)]. Semi Markov and Markov regenerative models attempt to relax the Markovian

assumption of exponential failure and repair times description, in a restrictive manner. They are also exposed to the state space explosion problem. Discrete-event simulation on the other hand offers an attractive alternative to analytical models as it can capture a detailed system structure and facilitate the study of influence of various factors such as reliability growth, various repair policies. Wang, Wu, Chen (2001) developed an architecture based analytical modular software reliability model. According to the model system reliability is calculated based on reliability of each module, operational profile and architecture.

State based models are an important category of models for modular systems. All software development teams have to answer an obvious question of when to stop testing and release the software, whether the software is single component or multi-component.

2.4 Optimal Stopping Models

Okumoto and Goel (1980), Koch and Kubat (1983), Shantikumar and Tufekci (1983), Dallal and Mallows (1988) are some of the papers that describe optimal stopping time model for software testing.

Most of the models assume the parameters of these models to be known constants and optimization achieved is merely static in nature. Static optimizations are those that cannot handle the stochastic nature of the debugging process. Dynamic optimizations are those approaches where the optimality is achieved according to the testing process.

In Shantikumar and Tufekci (1983), the de-eutrophication model of Jelinski & Moranda (1972) is used as the reliability model. The model assumes that if more faults are found there are fewer faults left in the system and hence better reliability.

Singpurwalla (1991) developed an optimal stopping solution for a single stage software system. The decision is based on a utility function. Two utility functions were suggested, one

based on cost and the other being the realized reliability of the software. Obviously a single test case was used.

In Dallal and Mallows (1998), an optimal stopping rule for stopping the testing of software based on the tradeoff between continued cost of testing and expected losses due to a bug after release. The economic costs and bug fixing costs are known functions. The randomness is in the bug detection. Asymptotic criterion function and the cumulative bugs, a function of time are plotted. The point of intersection of the curves denotes the stopping time. The total number of faults N is a Poisson distributed random variable with parameter λ . λ being another random variable with gamma distribution. The model extends the J-M de-eutrophication model.

Singpurwalla (1991) and Dallal and Mallows (1998), both proposed a Bayesian decision theoretic approach. Dallal and Mallows provides an exact but a complicated solution, thereby an asymptotic solution. Singpurwalla addressed a two stage problem where the solution is complicated by pre-posterior analysis. Morali and Soyer (2002) addressed the problem of optimal stopping in development phase. Model formulates a Bayesian decision theoretic approach by formulating the optimal release problem as a sequential decision problem.

Kubat (1998) describes a stochastic model of modular software systems and derive the overall system failure rate. The optimal failure rate of each module is derived based on minimized cost function. This optimization ensures maximum system reliability. Model considers multiple components and multiple test cases. Markov processes representing the control transfer is a more exact representation. Reliability for a certain task (profile) is a function of probability of a bug in module and expected average number of visits to the module.

Shaohui Zheng (2000) proposes a solution for the optimal release problem of computer software. A conditional NHPP is used to describe the software reliability growth behavior. A

Markov Decision program model is described to minimize the total discounted cost. Optimal policy is threshold. That is the optimality is reached when a certain predefined value is reached. Model is further extended with a constraint on system reliability. The mean value of NHPP depends on a random variable X , representing the general software quality. X is adapted as testing progresses. This results into dynamic procedure for deciding the system release time.

Optimal stopping models determine the time to stop testing. Testing scenarios are complicated if multiple test cases are used in combination with multiple components. Test case selection and type of testing with regard to single component or system wide testing on integration all affect the final reliability at the end of testing.

2.5 Testing Strategies

Software testing for a modular system is broadly classified into component testing and integration testing. There is a third way of testing which uses a combination of the above two. Testing carried out for a component or system is designed to target a class of bugs or specific parts of code. The advantage of component testing is that the complete system need not be ready for testing. Component testing alone does not deliver the type of testing required. There may be some faults that would emerge after integration. There is another way of testing that uses only testing on integration. This would be the case if we are testing the software in the final stages before release. Generally papers have considered that errors could emerge from interfaces too.

Rajgopal, Mazumdar and Majety (1999) developed an optimal policy that obtains an optimal combination of the different testing strategies. The paper describes a two stage mathematical programming approach to develop a test policy that would prove effective in obtaining a more reliable software system.

In this chapter we have given a brief introduction to the present literature available in the related areas of the proposed research. The following chapters describe the mathematical model, solution approach and matlab simulation of the proposed optimal test case selection problem.

CHAPTER3: MATHEMATICAL MODEL

The software reliability model proposed is intended for a multi component system tested using multiple test cases. The model is defined broadly by software architecture, failure behavior and method to combine the failure behavior with the architecture. The mathematical notations need to be clearly defined before a thorough formulation.

3.1 Notation

- **Architecture:**

p_{ij} - Pr {Control transfer from module i to j}

$P[M \times M]$ -Probability Transition Matrix

M -Total number of modules

C -Total number of Test Cases

T_D -Total testing time or Deadline time

π_i – Steady state probability for the architecture

- **Failure Behavior**

$N(t)$ – Number of Bugs detected in the module is random and follows a NHPP with mean function and intensity function.

$X(t)$ – Average Number of Bugs found in a component is random with a Gamma Distribution

$G(t)$ –Time to detect a bug follows exponential distribution with mean $1/\lambda$

$$g(t) = G'(t) \text{ Density function of } g(t) \tag{3.1}$$

$$\Lambda(t) = x(1 - \exp(-\mu t)) \quad \text{Mean function} \tag{3.2}$$

$$\lambda(t) = x.g(t) = x.(\mu.\exp(-\mu t)) \quad \text{Intensity function} \tag{3.3}$$

The notation listed above is used in various stages of the model formulation starting with architecture, failure behavior and solution approach. Any mathematical model is defined starting with certain assumptions so that the model defines the capability and limitations clearly.

3.2 Model Assumptions

The assumptions relate the different notations described earlier and defining the process of testing, defining and obtaining tractable mathematical solution.

- 1.) Failure intensity decreases with test time
- 2.) Failure rate is proportional to time and $G(t)$
- 3.) Perfect debugging
- 4.) Debugging at end of test stages
- 5.) Testing is representative of operational usage
- 6.) One failure –One fault
- 7.) Number of faults detected during non-overlapping intervals is independent of each other
- 8.) Time as a basis for failure rate
- 9.) Control at any instant is in only one component

▪ Remarks

The implications of the above assumptions are described here

1.) Failure rate decreases with test time

This assumption implies that software gets better with testing in a statistical sense. This seems to be a reasonable assumption in most cases and can be justified as follows. As testing proceeds faults are detected. They are either removed before testing continues or they are not removed and testing is shifted to other parts of program. In the former case, the subsequent

failure rate decreases explicitly. In the later case, the failure rate decreases implicitly since a smaller portion of code is subjected to testing.

2.) Failure rate is proportional to the number of remaining faults and $G(t)$

All models discussed previously, faults are assumed to have equal probability of being detected. This is achieved by selecting test cases that test all paths of the software equal probability. In the present model the failure rate is assumed proportional to remaining faults and the cumulative distribution $g(t)$.

3.) Perfect debugging:

Perfect debugging essentially means no new bugs are introduced while debugging. This is a restrictive assumption as a bug removed in a single path after an occurrence of a bug may affect the functioning of another path. This assumption is practical if faults introduced are fairly minimal.

4.) Debugging at end of test stage:

Faults in general testing situations can be removed immediately or could be removed later. In later case the fault detection process can be assumed as if the fault is removed unless future testing uses the same path. In the proposed model each stage has a single test case and multiple runs. The faults detected are independent of the paths traversed by test cases.

5.) Testing is representative of operational usage

This assumption is necessary as the reliability estimated based on testing stage is projected onto operational usage. Most ideal testing situations are assumed mirror images of operational usage. This goal is usually difficult to achieve in practice.

6.) Single failure-Single fault

A fault in software could be due to a single or multiple bugs. In the present module we assume that each fault is a direct consequence of a single software bug. This assumption simplifies the mathematical model to a large extent.

7.) Number of faults detected during non-overlapping intervals is independent of each other

Faults or bugs detected in one stage cannot reappear in the succeeding stages which is true as the bugs are corrected after detection.

8.) Time as a basis for failure rate

Failure rate is determined on the basis of the testing time and most failure records are maintained based on testing time. There are models that calculate failure intensity based on other parameters like lines of code, number of test cases or functions tested.

9.) Control at any instant is in only one component

The execution of software occurs only in one component. This model is not suitable in parallel computing architectures where software runs in multiple places and multiple components.

The different assumptions discussed above makes the model robust and the solution conceivable. These assumptions would reflect itself in the various stages of the model formulation namely architecture, failure behavior and solution.

3.3 Architecture

Software architecture is described by an irreducible CTMC. The software contains finite number of modules and transfer of control between components is described by the probability $p_{ij} = \Pr \{ \text{Control transfer from module } i \text{ to } j \}$. p_{ij} is obtained as a result of algorithmic

analysis and are assumed to be known for all practical purposes. The control structure being defined by transfer rates and the probabilities are steady state probabilities.

- **Markov Assumption**

All state space models assume that the probability of a component being executed depends only on the previous component executed. This is first order Markov chain represented by n states. If the probability of a component being executed depends on previous two components then the process is represented by a second order Markov chain of n^2 . The ideal case would be that the probability of a component being executed in a particular path before reaching the component. This was considered in Ledoux (1999) which would make the solution unconceivable.

In our model we assume first order irreducible Markov chain. This would be ideal if a mechanism is introduced into the system such that a faulty output from any component is not allowed to continue and system is halted indicating a fault in the component.

- **Transition probability estimation**

Transition probabilities affect the system reliability estimations drastically; its estimation must be as accurate as possible. If interactions among components are minimal then the transition probability matrix would be sparsely populated. Estimations are carried out in two methods. One is based on operational profile and the other is based on execution counts during coverage testing.

CTMC is used to obtain system reliability based on component reliabilities. The initial condition of the process together with the transition probabilities completely determines the stochastic behavior of the Markov process. The next process before estimating the system reliability is to describe failure behavior.

3.4 Component Failure Behavior

Individual modules fail with a failure rate that is constant in each testing stage. The transfer of control between modules (interface) is not subject to failure. All failures are attributed to the modules. When module i interacts with module j there is a probability that failure could be in one of the modules p_{ij} .

▪ **Stochastic assumptions are as follows**

1. X is a random variable that represents the average fault content in the component. X is assumed known only through its distribution. X encompasses the prior knowledge of the component's failure behavior which would be updated based on the actual faults $N(t)$ detected of the component.
2. If $X = x$, $N(t)$ the number of faults detected up to time t , follows a NHPP with mean function $\Lambda(t)$ and intensity function $\lambda(t)$.
3. Time taken to find a bug is random variable that follows the density function $g(t)$ which decreases with increase in testing time t .

The G distribution is assumed known which simplifies the solution process. G is static, not reflecting the bugs detected till time t . G defined with a prior and posterior being calculated based on bugs detected. Dallal and Mallows (1988) assumed G to be nearly exponential as used by many other models like Goel and Okumoto (1979), J-M model (1972), Langberg and Singpurwalla (1985), and Musa (1975)

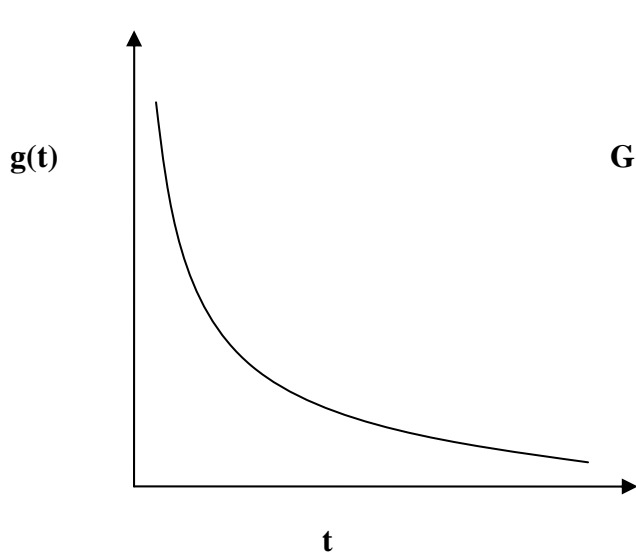


Figure 3.1 $g(t)$ v/s t (continuous)

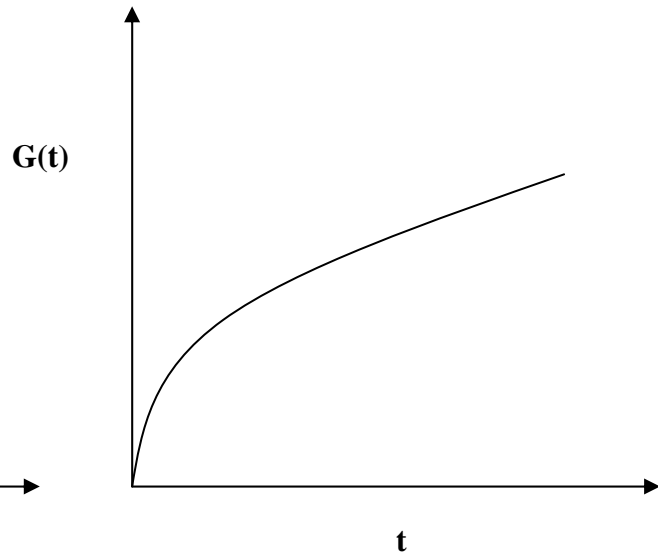


Figure 3.2 $G(t)$ v/s t (continuous)

The mathematical representation is as follows:

$$G(t) = P(T \leq t) = 1 - P(T > t) \quad (3.4)$$

$$G(t) = 1 - \exp(-\lambda t) \quad (3.5)$$

$$g(t) = G'(t) = \lambda \exp(-\lambda t) \quad (3.6)$$

Most models consider G as a continuous distribution but in the proposed model G is assumed to be discrete. The assumption of end of cycle debugging makes it necessary that $\lambda(t)$ should remain constant through each cycle which makes $g(t)$ a step case function.

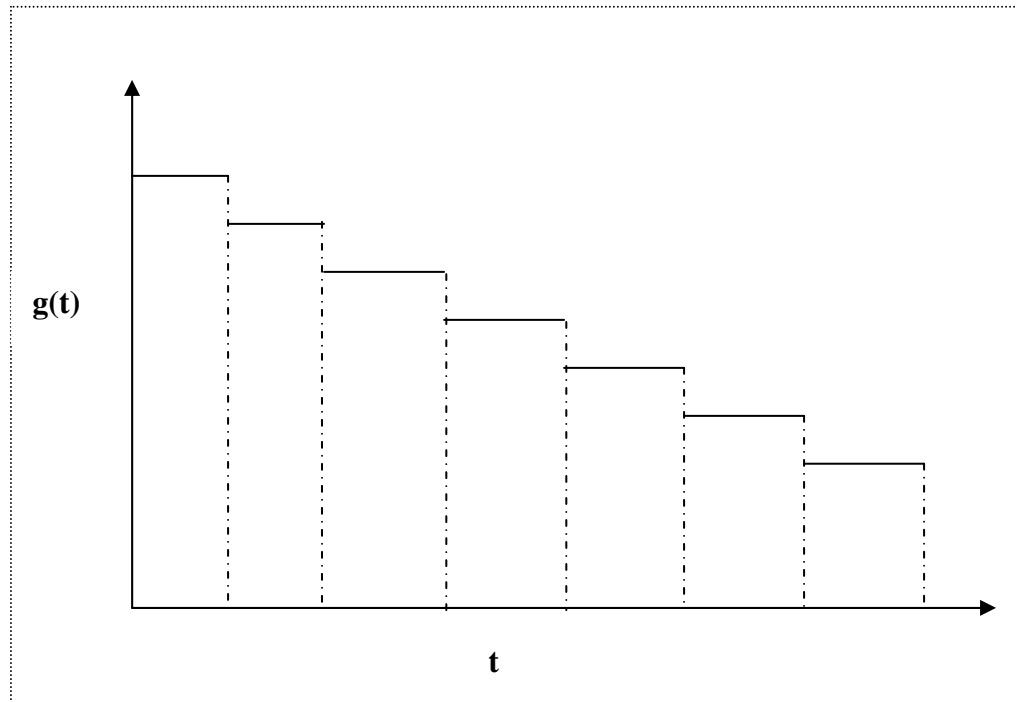


Figure 3.3 $g(t)$ v/s t for a single module (Discrete)

▪ **Remark:**

Here an illustration of the functions $g(t)$, $N(t)$, $X(t)$ and their parameters are described and their relationships highlighted.

Let N be the total number of bugs when testing begins in the system. N is Poisson distributed random variable with parameter X . X being another random variable N is conditionally Poisson distributed.

$$\Lambda(t) = x(1 - \exp(-\mu t)) \quad \text{Mean function} \quad (3.7)$$

$$\lambda(t) = x \cdot g(t) = x \cdot (\mu \cdot \exp(-\mu t)) \quad \text{Intensity function} \quad (3.8)$$

$$P[N(t) = n | X] = \frac{[\Lambda(t)]^n e^{-\Lambda(t)}}{n!} \quad (3.9)$$

$$P(X | N(t) = n) = P(X, N(t) = n) / P(N(t) = n)$$

$$= \frac{P(N(t) = n | X) * P(X)}{\int_{\Omega}^{\infty} P(N(t) = n | x) * P(x) dx} \quad (3.10)$$

the denominator is constant,

$$\propto \frac{\{[X * G(t)]^n * \exp[-X * G(t)]\} * P(X)}{n!} \quad (3.11)$$

Prior of X is Gamma Distribution:

$$P(X) = \Gamma(a, b) = b^a x^{a-1} \exp(-bx) / \Gamma(a) \quad (3.12)$$

▪ **Gamma Distribution with parameters a and b:**

Initially the parameters a, b represent the prior and at each succeeding stage a, b are updated based on the observed values of the number of bugs.

Posterior Distribution of X

$$P(X | N(t) = n) \propto \frac{\{X^n [G(t)]^n \exp[-XG(t)]\} * b^a x^{a-1} \exp(-bx)}{\Gamma(a)} \quad (3.13)$$

$$\propto X^{n+a-1} * \exp[-X(b + G(t))]$$

$$= \Gamma[a + n, b + G(t)] \quad (3.14)$$

Suppose the faults are independent and the time to detect faults is exponentially distributed with mean λ . Then we can easily show that $N(t)$ the number of faults detected up to time t , will follow conditional NHPP.

The next stage is where the failure behavior and the software architecture is combined to obtain the system reliability estimate.

3.5 Solution Method

As described earlier in the literature survey the solution for state based models could be either hierarchical or composite. The model we adopt is the hierarchical approach.

The type of software under consideration is continuously running. Littlewood (1975) considered moment generating function of number of failures $N(t)$ from the composite model and showed the result is analytically intractable. He simplified the model using asymptotic analysis which leads to the Poisson process with parameter

$$\lambda_s = \sum_i \pi_i \left[\lambda_i + \sum_{j \neq i} q_{ij} v_{ij} \right] \quad (3.15)$$

where $\pi = [\pi_i]$ is steady state vector of irreducible CTMC with transition rate matrix $Q = [q_{ij}]$, i.e. $\pi Q = 0$

$\left[\lambda_i + \sum_{j \neq i} q_{ij} v_{ij} \right]$ of equation (3.2) is combined failure rate of component and interface failure

rates. The above term in our model will reduce to $[\lambda_i]$ as we assume failure do not occur while interactions. $\lambda_s = \sum_{i=1}^m \lambda_i$

Steady state probability vector represents the average proportion of time spent in state i in the absence of any failure. Laprie (1984) presented a model that is a special case of Littlewood (1975) which considers only component reliabilities and the system reliability is calculated as shown.

$$\lambda_s = \sum_{i=1}^n \lambda_i = \sum_{i=1}^n \pi_i^j [E[X_i]G_i(t_i)] \quad (3.16)$$

System reliability or the probability that system does not fail until time t is given by

$$R(t) \approx e^{-\lambda t} = e^{-\sum_{i=1}^n \lambda_i t} \quad (3.17)$$

This is a special case of versatile Markov process introduced by Neuts (1979). The construction of versatile Markov point process is by assuming an $(n+1)$ CTMC with n transient states and one absorbing state. The infinitesimal generator Q obtained after deletion of absorbing $n+1$ state.

The reliability estimate is obtained for the modular software system. The next stage is to determine the test case policy so that maximum system reliability is achieved within deadline time.

CHAPTER 4: OPTIMAL TEST CASE SELECTION

An important contribution of the present research is the determination of test case policy for the testing of multi-component software in a known testing time. The solution for this purpose is obtained by formulating a stochastic dynamic programming or Markov decision programming to minimize the value function. The value function is the failure intensity of the software system weighted by the probability of finding the respective bugs in the system.

The background required for this approach is stochastic dynamic programming and later the algorithm is illustrated.

4.1 Stochastic Dynamic Programming

Dynamic programming invented by Richard Bellman has proved its robustness as a optimization technique .Dynamic programming converts a n dimensional optimization problem into n single dimensional problem.Dynamic offers true global maxima and minima rather than local solutions. Stochastic dynamic programming or Markov decision approach (MDP) can solve problems that are Markovian in nature of an n stage problem. Markovian essentially means that each stage solution is dependent only on the previous stage.

4.2 Value Equation of the System

The value function forms the heart of the dynamic program formulation.

$$R(\vec{n}, \vec{m}) = \min_{j \in \{1, 2, \dots, c\}} \left\{ \sum_{k^j=0}^{\infty} \left[R(\vec{n} + \vec{k}^j, \vec{m}_j + 1_j) * \prod_{i=1}^M \text{probability of } k^j \text{ bugs in component } i \right] \right\} \quad (4.5)$$

$$R(\vec{n}, \vec{m}) = \min_{j \in \{1, 2, \dots, c\}} \left\{ \sum_{k_i^j}^{\infty} R(\vec{n} + \vec{k}^j, \vec{m} + 1_j) * \prod_{i=1}^M \left[\frac{\{E[X_{n_i}(t_i)]g(t_i)\pi_i^j\}^{k_i^j} * \{\exp - [E[X_{n_i}(t_i)]g(t_i)\pi_i^j]\}}{k_i^j!} \right] \right\} \quad (4.6)$$

$k_1, k_2 \dots k_m = 0$ is the number of bugs in modules 1...m

Calculation of time spent in each module by all the test cases is given by:

$$t_i = \sum_{j=1}^M \pi_i^j m_j \quad (4.7)$$

Probability of finding k^j bugs in module i is given by

$$\begin{aligned} P(k^j) &= \frac{E[X_{n_i}(t_i)]g(t_i)\pi_i^j}{k_i^j!} * \exp - \left(\frac{E[X_{n_i}(t_i)]g(t_i)\pi_i^j}{k_i^j!} \right) \\ &= \lambda_i^{k_i^j} * \exp - (\lambda_i) / [k_i^j!] \end{aligned} \quad (4.8)$$

The Value equation defined above is the failure intensity of the system.

$$R(n_i, m_j) = \min_j \left\{ \sum_{k_j=0}^{\infty} \left[R(\vec{n} + \vec{k}^j, \vec{m} + 1_j) * \prod_{i=1}^M \frac{\lambda_i^{k_i^j} * \exp - (\lambda_i)}{k_i^j!} \right] \right\} \quad (4.9)$$

▪ **Terminating stage:**

In the final stage at time T_D no bugs are detected and the user profile is run and the value function is defined as

$$R(\vec{n}, \vec{m}) = \sum_{i=1}^M \lambda_i \quad (4.10)$$

Total number of test cases used at this stage is $\sum_{i=1}^c m_i = T_D$ (4.11)

Failure intensity the last stage is given by:

$$\lambda_i = E[X_{n_i}(t_i)]g_i(t_i)\pi_i \quad (4.12)$$

The value function for the last stage is simply the total system failure for the software system.

▪ **Analysis of Recursive Value function:**

The value function is defined for the cumulative bugs found represented by the vector $[\vec{n}]$ for all the modules and the total test cases used until the stage under consideration starting from start of testing ($t = 0$) represented by $[\vec{m}]$. $R(\vec{n}, \vec{m})$ is the recursive value function estimated for stages start from time $T_D - 1$ and successively for every stage till time $t = 0$.

The recursive value function at time t is a function of $R(\vec{n}, \vec{m})$ in time $(t + 1)$ stage. This recursive function is calculated in this fashion for all stages except at the stage where the time elapsed is T_D the deadline.

The recursive value function is dissected as follows. The recursive value function can be defined as the cumulative failure intensity of the software system based on the possible bugs in the state. For every calculation of value function, it is weighted by the probability of finding k_i^j in component i with test case j .

Probability of k_i^j in component i with test case j is given by

$$P(k^j) = \lambda_i^{k_i} * \exp - (\lambda_i) / [k_i!] \quad (4.13)$$

where λ_i is the expectation of average number of bugs in the component

$$\lambda_i = E[X_{n_i}(t_i)] = \left[\frac{a + N_i(t_i)}{b + G_i(t_i)} \right] \text{ where } X_{n_i}(t_i) \text{ follows gamma distribution.} \quad (4.14)$$

The minimization process tries to eliminate all those cases which have higher value functions. The goal is to determine those actions that would produce higher reliability. This process of minimization is carried from stages $T_D - 1$ to 0. For stage T_D the user profile is executed and no bugs are expected to surface as it is not a debugging stage. The value function for this stage is given by

▪ **State Space of the Dynamic Programming:**

$$R(\vec{n} + \vec{k}, \vec{m})_t$$

$$\vec{k} = [k_1 \ k_2 \ k_3] \quad \text{Vector of the number of bugs that may be found for modules 1, 2, 3}$$

Each k_i is predisposed by a distribution that does put a limitation on the number of bugs that could be found in each testing cycle based the testing case/profile. Say from 0 to 4.

The total number of combinations of k values is $5*5*5=125$.

$$\vec{m} = [m_1 \ m_2] \quad \text{Vector } m \text{ is the number of test cases used up till the testing stage}$$

There are two test cases. m_j is the sum of total number of j test cases used until the present stage

Representation of the state space for:

Total Test Cases: $c= 2$

Total Components: $m= 2$

Total Testing stages: $T=2$

Figure 4.1: State space and Bug space representation

4.3 Algorithm for Optimal Decision

- **Initial Parameters:**

Total number of modules – M

Total number of Test Cases – C

Total testing time - T_D

Probability Transition Matrix – $P[M \times M]$

P_{ij} = Probability of transition from module i to module j

Proportion of time spent in a module i for a test case j is given by the matrix $\pi[M \times C]$

π_{ij} = Proportion of time spent in module i using test case j .

$$\sum_{j=1}^M \pi_{ij} = 1 \quad \text{The time spent in test case } i \text{ for all modules is unity.}$$

- **Initial Distributions**

$X_i(t)$ – Gamma Distribution for Average Number of bugs in each module i

$G_i(t)$ – Discret Exponential Distribution for surfacing of bugs

$N_i(t)$ – NHPP for number of bugs in the system

Poisson distribution with meant time $m(t)$



Figure 4.1 Backward Recursion MDP representation

 : Progress of Single stage Optimality solutions

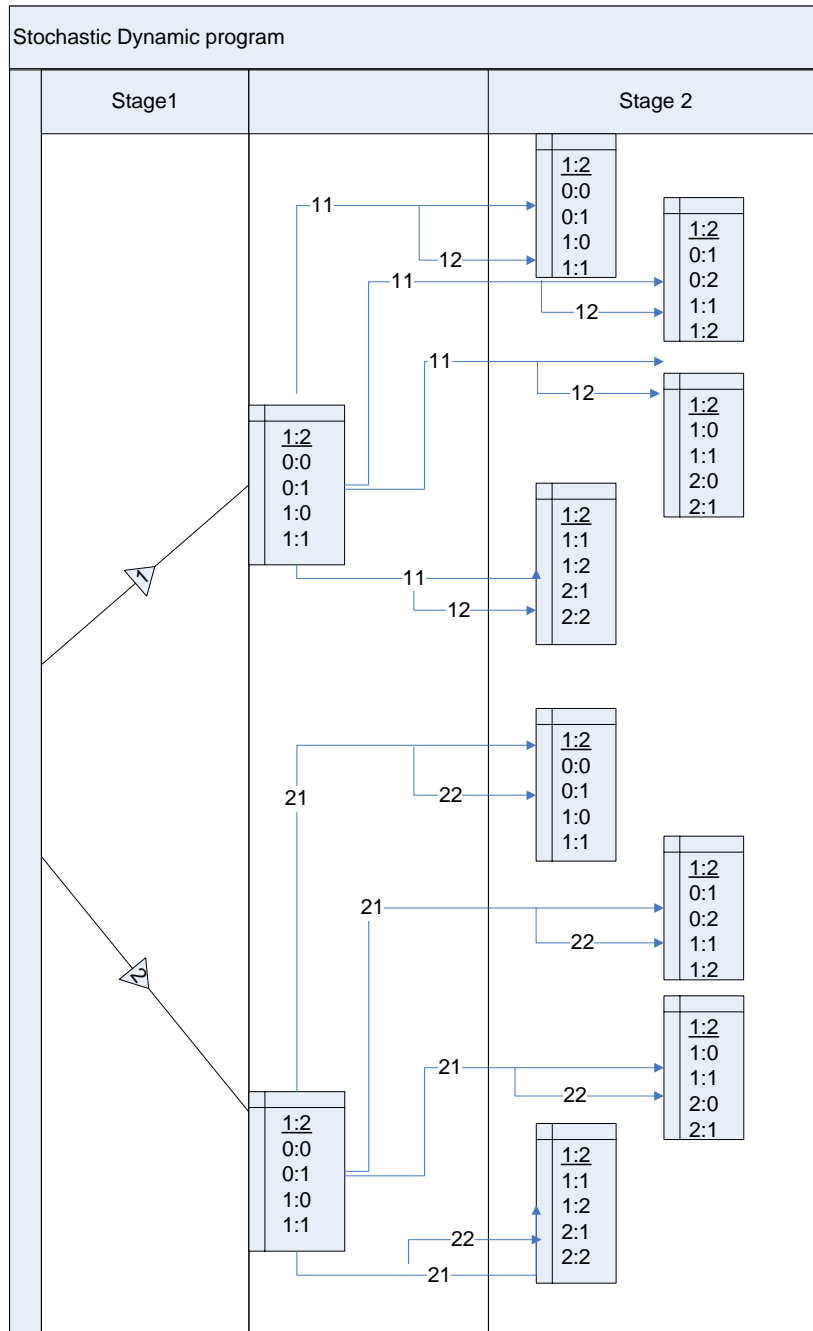


Figure 4.2 Markov Decision Process (MDP)

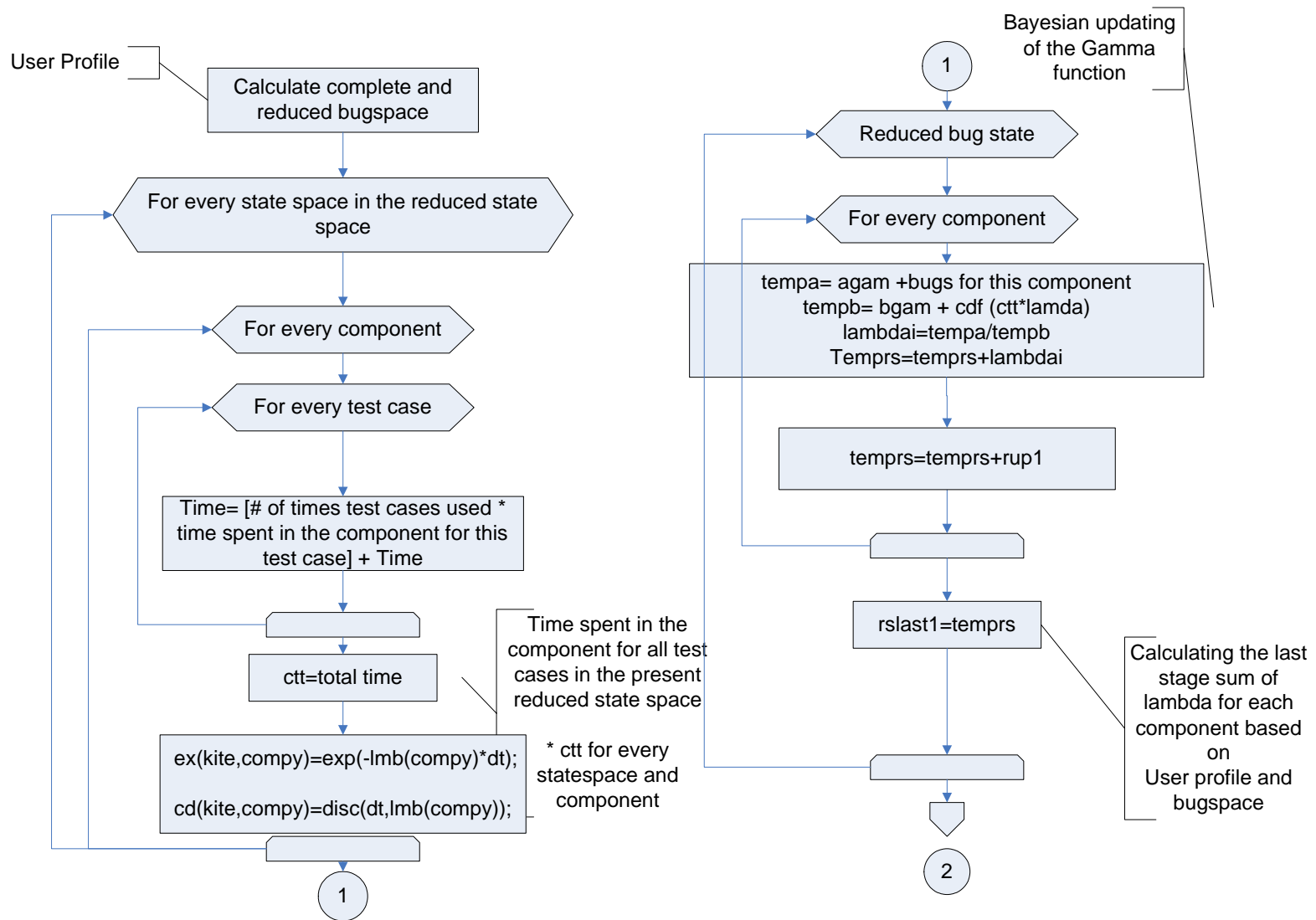


Figure 4.3 Algorithm (figure continued)

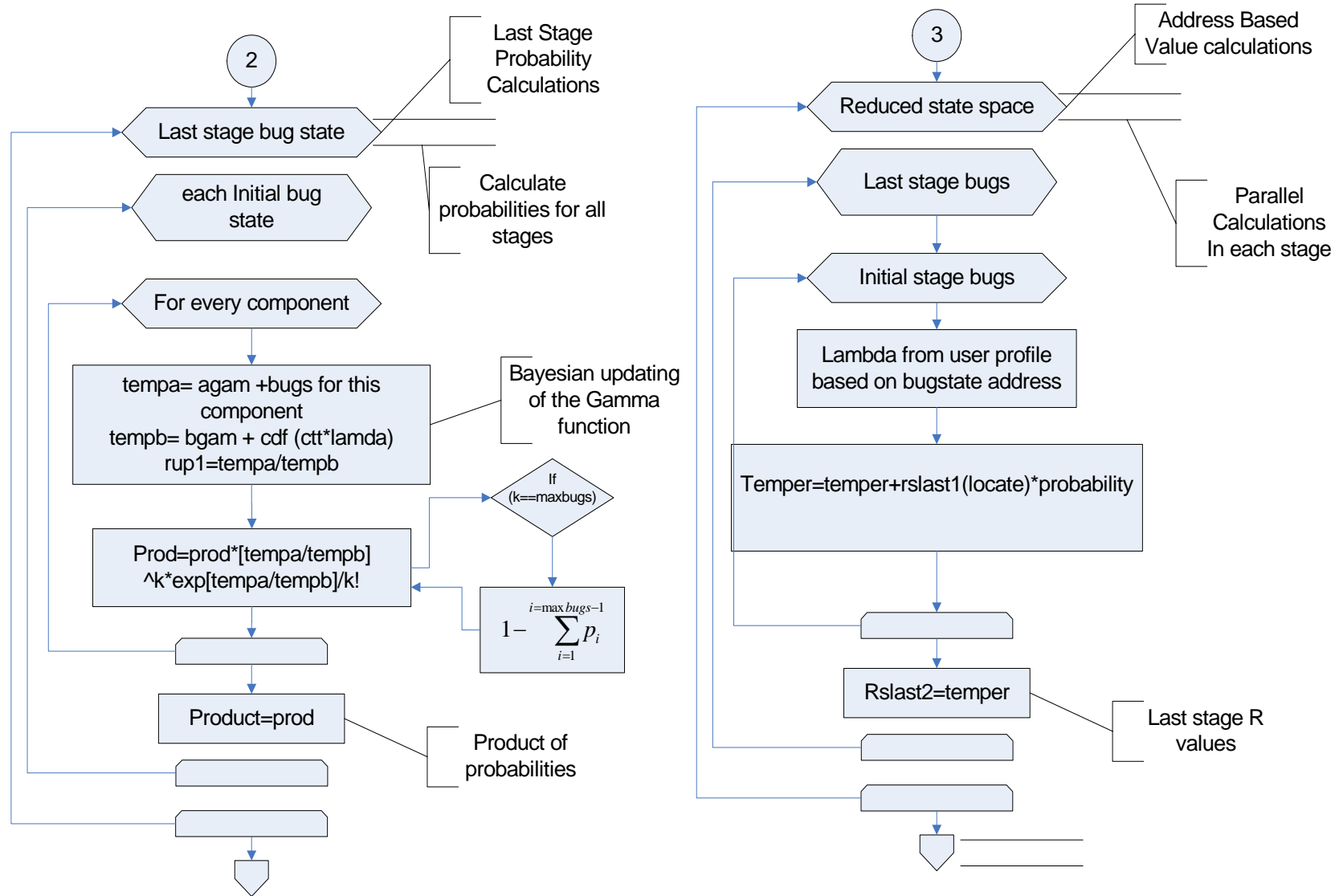


Figure 4.3 (figure continued)

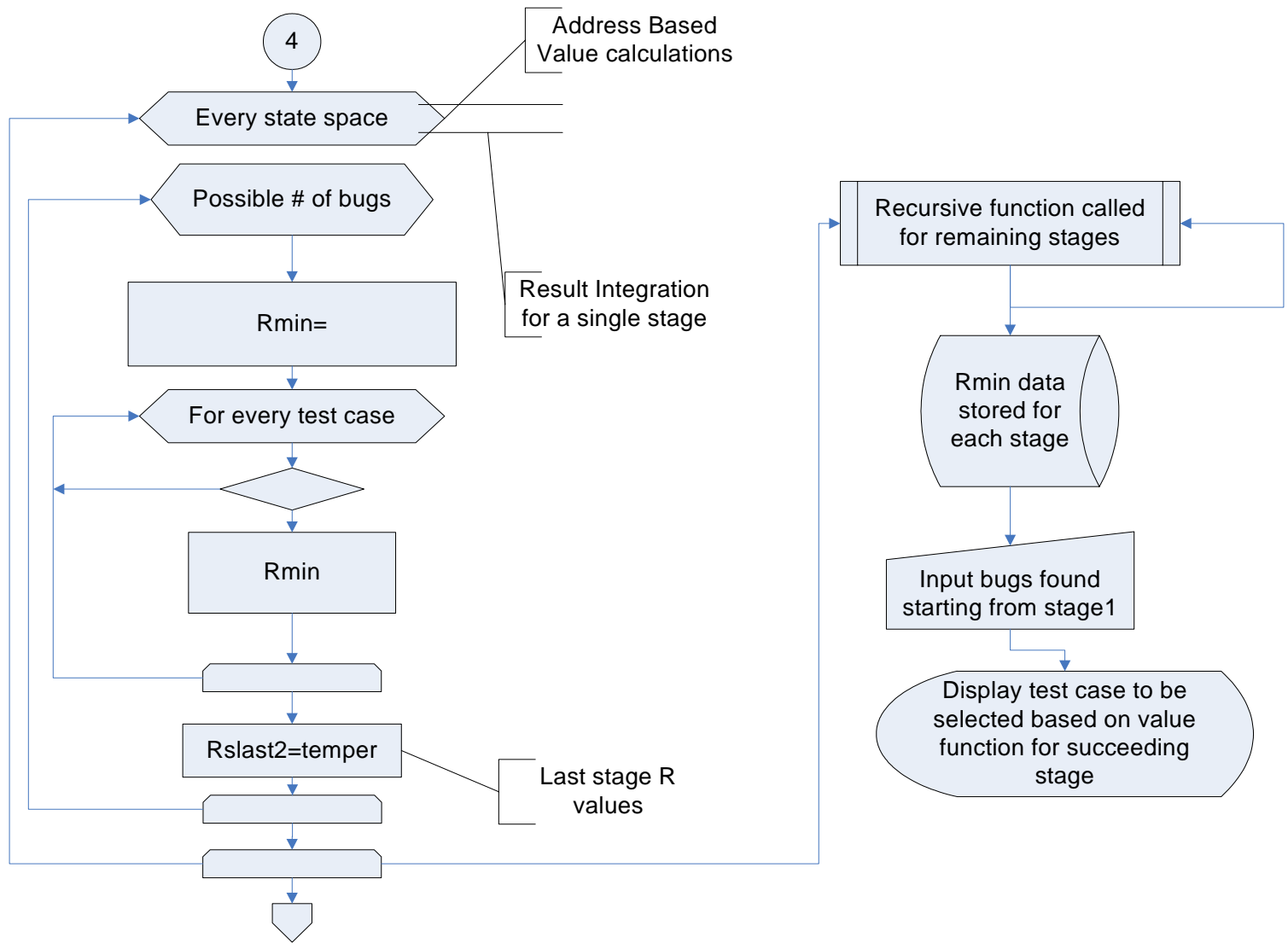


Figure 4.3

Optimal Test Case Selection	
Main	Integrates all the modules
Distributions	<ul style="list-style-type: none"> <input type="checkbox"/> parameter for the exponential distribution $G(t)$ of bug detection <input type="checkbox"/> (a, b) Gamma distribution parameters for each component <input type="checkbox"/> pts1 - proportion of time spent in module i test case j matrix <input type="checkbox"/> Noc : # of Components <input type="checkbox"/> Notcs: # of test cases <input type="checkbox"/> Maxbugs: # maximum bugs possible in each stage each module <input type="checkbox"/> Teststages: Maximum test stages in testing
Bug space	<div style="display: flex; justify-content: space-between;"> <div style="border: 1px solid black; padding: 5px; width: 45%;"> <p>Input: # components , # maximum bugs , stage of testing</p> <p>Output: total bug space and reduced bug space</p> </div> <div style="border: 1px solid black; padding: 5px; width: 45%; border-radius: 10px;"> <p>Bug space: Combination of the possible bugs in each component in a stage</p> </div> </div>
State space	<div style="display: flex; justify-content: space-between;"> <div style="border: 1px solid black; padding: 5px; width: 45%;"> <p>Input: # components , stage of testing</p> <p>Output: total state space and reduced state space Test case counter for each state space</p> </div> <div style="border: 1px solid black; padding: 5px; width: 45%; border-radius: 10px;"> <p>State space: Combination of the possible cumulative test cases used in each stage</p> </div> </div>
Recursive	<div style="border: 1px solid black; padding: 5px; width: 100%;"> <p>Input: Previous stage value function (Rmin)</p> <p>Output: Present stage value functions</p> </div>
tcc	<div style="border: 1px solid black; padding: 5px; width: 100%;"> <p>Input: State space value ,test case to be counted</p> <p>Output: Count value of test case</p> </div>
Case Counter	<div style="border: 1px solid black; padding: 5px; width: 100%;"> <p>Input: State space value ,# of test cases</p> <p>Output: Reduced format state space</p> </div>
Roundof f	<div style="border: 1px solid black; padding: 5px; width: 100%;"> <p>Input: Decimal number to be truncated</p> <p>Output: Truncated integer</p> </div>

Figure 4.4 Matlab Functions

4.4 Parallel Algorithm

It has been seen after simulations in Matlab using single processor CPU, the simulations may be very time consuming. To minimize the time requirement of the simulations parallel computing can be very effectively used. The simplest language that has parallel processing API is Fortran97. The algorithm for a single and multi CPU processing is not very different. In the following section the parallel algorithm is illustrated for Maxbugs=3, Componentes=3 and Testcases=3.

Each of the blocks represents the different combinations of bugs that could be detected in the testing stage. Each block is calculated with a unique test case which forms what is called the reduced state space. In a single processor system each of the blocks in each testing stage is calculated sequentially and moved one stage back. (Fig 4.5 and Fig 4.6)

In the parallel processing architecture each of the blocks in a single stage is spawned on to a different processor (computer) which is possible by a language like Fortran97. The calculation of probabilities can be executed in parallel for all stages which is the basis of stage wise parallel calculations. The results obtained out of each block are later sent back to a computer which manages all the processes. Once the results are combined the same process is repeated for all testing stages.

Though the stage wise calculations will be sequential but calculations within a single stage can utilize a parallel processing environment. If Matlab had the API for parallel computing the present code would become parallel by just adding a few lines of code.

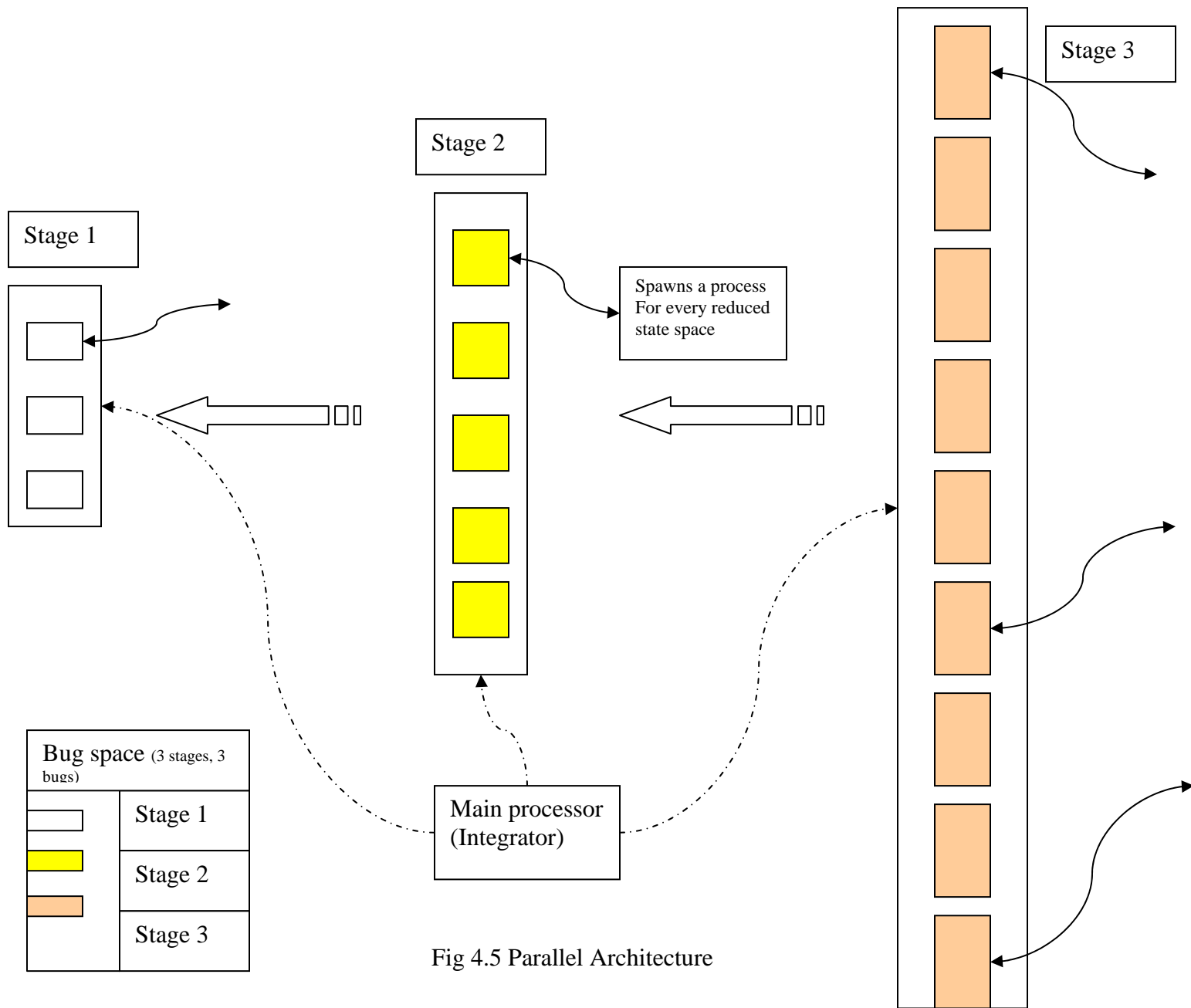


Fig 4.5 Parallel Architecture

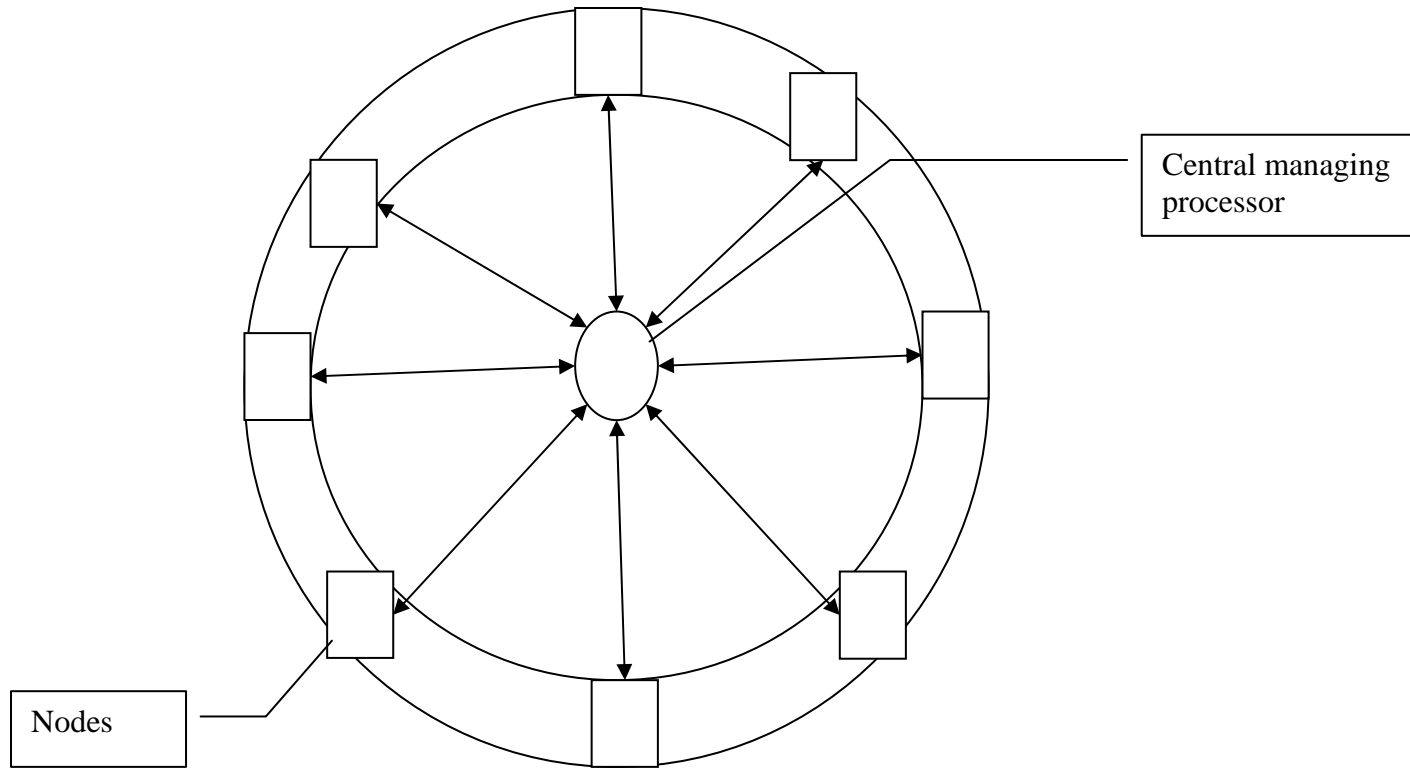


Fig 4.6 Parallel simulation

In this chapter we have described the recursive value function for the stochastic dynamic program formulation for optimal test cases solution and described the algorithm Next chapter we describe the brief contribution of the present research.

CHAPTER 5: RESULTS AND SIMULATION

5.1 Pictorial Representation of the Algorithm for Optimal Test Case Selection

Here we show the path taken by the algorithm to show the process of calculation. This must be correlated to the flowchart given in the previous chapter.

The stage wise bugs that can occur are 0, 1
 Number of Components
 Total number of testing stages is 2
 Number of Test Cases used

Max-bugs	1
Components	2
Test stages	2
Test cases	2

- **Programming representations**

State space is the array of all values that represent the test cases used till a stage of testing. Complete state space is array where there are repetitions of values. The test cases used may differ in the sequence of tests used but the number of times each test case used remains same. Reduced state space eliminates all such repetitions.

Complete state space for any number of test case systems is right justified. The test case used in the latest stage would appear in the last. If the testing stage is fourth then there would be four digits, each digit would be numbers representing the test case used and the place would determine the stage. Reduced state space has a maximum number of digits equal to the number of components. Last digit represents number of times the last test case is used. The complete state space is used only to know which values to be compared. It is similar to an addressing scheme.

Complete state space (ss) (stage 2)

ss	ssr
11	20
12	11
21	11
22	2

Reduced state space: (ssr)

ssr
2
11
20

Bug space is array of all possible combination of bugs in a particular testing stage. The number of columns is equal to the number of components with an additional column to represent the bugs in all the three components. The number of digits equals the number of components in the concatenated number. The digits are gain right justified with the last stage representing the number of bugs in the last component.

Complete bug space (16)

comp1	comp2	concat
0	0	0
0	1	1
1	0	10
1	1	11
0	1	1
0	2	2
1	1	11
1	2	12
1	0	10
1	1	11
2	0	20
2	1	21
1	1	11
1	2	12
2	1	21
2	2	22

All calculations are based on the reduced state space to reduce the number of calculations. The complete bug-space is used to obtain the test policy. There are again no redundant calculations here.

Time spent in the each component till the present stage based on all the test cases used (obtained from the state space value)

Component test time (ctt) based on reduced state space

comp1	comp2	ssr
0.4	1.6	2
0.9	1.1	11
1.4	0.6	20

- **User Profile Calculations**

Calculation of user profile does not involve calculation of any probabilities and the number of calculations is minimal involving reduced bug-space and reduced state space. Array `rslast1` consists of summation of lambda values for a combination of reduced state space and bug-space. In the present case it would be 27. Reduced bug-space has 9 and reduced state space has 3 distinct values.

- **Last Stage Calculations**

The array `rup1` has the lambda values for each component for a combination of state space and bug space. The array of `rslast1` is the summation of the lambda values for the values calculated. These calculations are based on user profile calculated in the previous stage.

Product: Matrix: (*) All unique probabilities are calculated before hand and stored in this array.

Calculations of `rslast2` and `Rmin` values for the present stage:

Array `rslast2` has the 'R' values before minimization takes place. The matrix is obtained based on the reduced state space, product matrix having the probabilities and the value functions obtained from the previous stage. The previous stage in the last stage is the lambda values obtained from the user profile calculations.

The matrix `Rmin` is obtained after executing a minimizing function on the R values that is `rslast2` matrix.

- **Recursive function**

The function calculates the value function in the similar fashion as above but for the calculation of `rslast2`. `rslast2` is nothing but the values from the previous stage `Rmin`. In the recursive function the probability function is used to weigh the recursive value function.

5.2 Policy Determination

Once the above calculations are run, the policy or the test case to be used at each stage while actual testing is to be determined based on the calculations. The policy is different based on the bugs surfacing in reality at each stage. The 'policy.m' Matlab is run and starting from second stage it asks for the bugs determined in the first stage and tests run till the last but one stage. The first test case run is a unique decision. The out put after every stage is the test case to be selected next and a prediction of the value function at the end of all testing stages.

▪ Graph

The policy determined is reflected in the graph of:

- 1.) The Value functions of the system as testing progresses.
 - System Value function (R) v/s test stage for the selected policy.
 - This graph represents the test case selected for each stage.

5.3 Limitations

The nature of the problem is that it has a very large state space. The algorithm is optimized to have a very low memory footprint. This helps to reduce the overall memory requirement of Matlab. Each of the parameters maximum bugs, components and test stages alter the size of the calculations profoundly.

The Poisson distribution is curtailed with respect to the number of bugs. The number of bugs for a Poisson distribution is infinite but for calculation purposes the number of bugs is a finite number. This is the reason we see that the probability of obtaining maximum number of bugs is high and hence the failure intensity is higher.

As the problem is computationally intensive in nature, a single processor computer may take a long time to obtain the results. Usually testing is a long term process covering many months and hence the tool developed will yield results that can be used for testing.

Examples below show the outcome of the simulation based on the bugs that is detected while actual testing. The parameters are varied for different examples and the results shown.

5.4 Results and Analysis

- **Example 1: Mcst-2332**

		Component	1	2	3
Max-bugs	2	G(t) parameters (bug detection)	lmb(1)=[0.2]	lmb(2)=[0.32]	lmb(3)=[0.4]
Components	3	Gamma Distribution	agam(1)=[0.2]	agam(2)=[0.2]	agam(3)=[.15]
Test stages	3		bgam(1)=[1.6]	bgam(2)=[1.6]	bgam(3)=[1.5]
Test cases	2				

Proportion of time spent in module per unit time: (pts) 3 components 2 test cases

Proportion of time spent for user profile: (ptsu)

$$pts = \begin{bmatrix} 0.4 & 0.4 & 0.2 \\ 0.3 & 0.2 & 0.5 \end{bmatrix} \quad ptsu = [0.3 \quad 0.4 \quad 0.3]$$

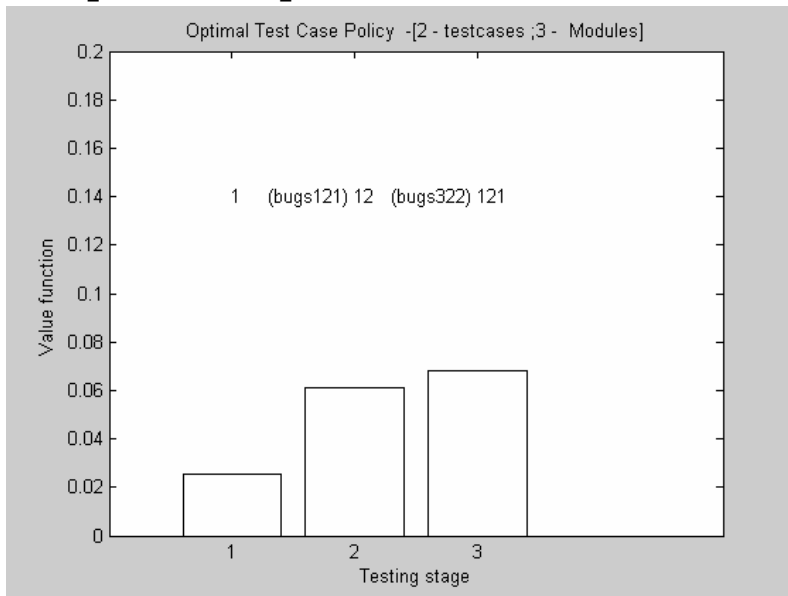


Figure 5.1 Value function v/s s test stages (3 components 2 test cases)

Analysis: Cumulative bugs found in stage 1 is 121 and in stage 2 is 322 which shows an increase in value function.

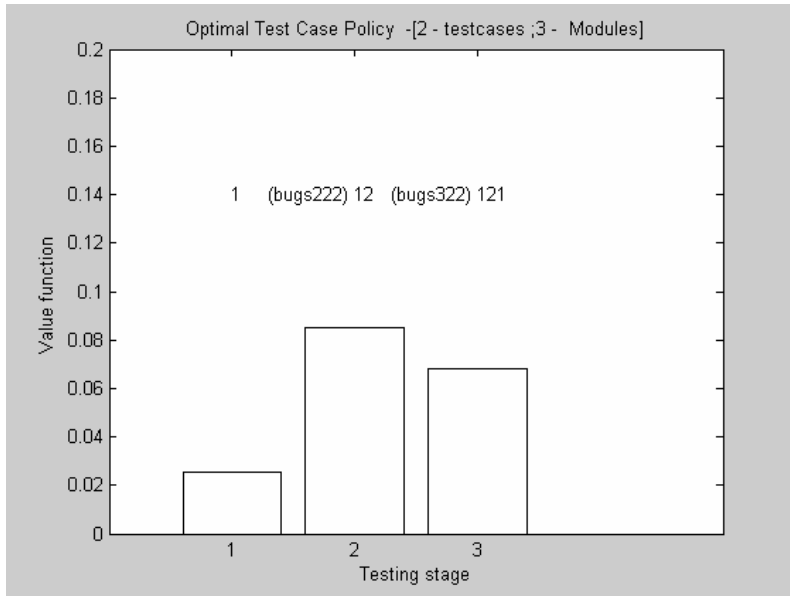


Figure 5.2 Value function v/s test stages (3 components 2 test cases)

Analysis: Cumulative bugs found in stage 1 is 222 and in stage 2 is 322 which shows a decrease in value function as the total number of bugs found in the second stage is 100 which is lower than in the first stage.

▪ **Example 2: Mcst-2232-test**

		Component	1	2
Max-bugs	2	G(t) parameters (bug detection)	lmb(1)=[0.25]	lmb(2)=[0.32]
Components	2	Gamma Distribution	agam(1)=[0.2]	agam(2)=[0.2]
Test stages	3		bgam(1)=[1.6]	bgam(2)=[1.6]
Test cases	2			

Proportion of time spent in module per unit time: (pts) 2 components 2 test cases

Proportion of time spent for user profile: (ptsu)

$$pts = \begin{bmatrix} 0.8 & 0.2 \\ 0.1 & 0.9 \end{bmatrix} \quad ptsu = [0.4 \quad 0.6]$$

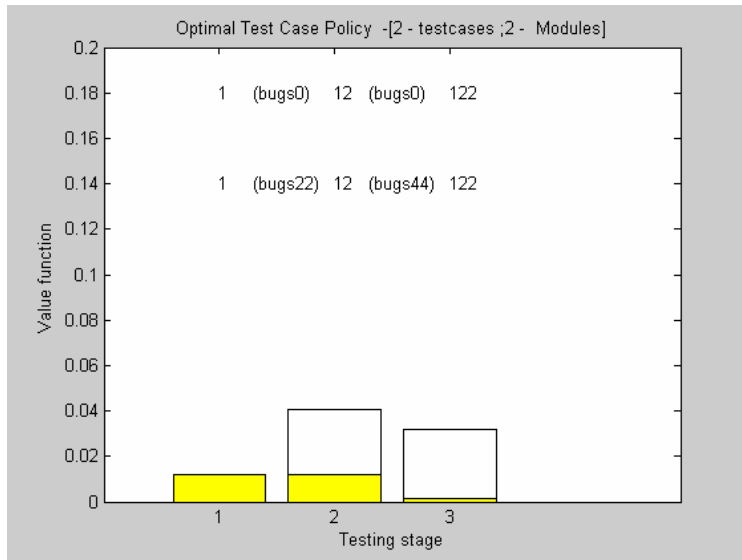


Figure 5.3 Value function v/s s test stages (2 components 2 test cases)

Analysis: Cumulative bugs found in stage 1 is 22 and in stage 2 is 44 with value function represented in white graph while the yellow graph represents the value function for 0 bugs found in all stages.

▪ **Example 3: Mcst-3333**

		Component	1	2	3
Max-bugs	3	G(t) parameters (bug detection)	lmb(1)=[0.2]	lmb(2)=[0.32]	lmb(3)=[0.4]
Components	3	Gamma Distribution	agam(1)=[0.2]	agam(2)=[0.2]	agam(3)=[0.4]
Test stages	3		bgam(1)=[1.6]	bgam(2)=[1.5]	bgam(3)=[1.3]
Test cases	3				

Proportion of time spent in module per unit time: (pts) 3 components 3 test cases

Proportion of time spent for user profile: (ptsu)

$$pts = \begin{bmatrix} 0.2 & 0.3 & 0.5 \\ 0.1 & 0.3 & 0.6 \\ 0.25 & 0.25 & 0.5 \end{bmatrix} \quad pts = [0.3 \quad 0.4 \quad 0.3]$$

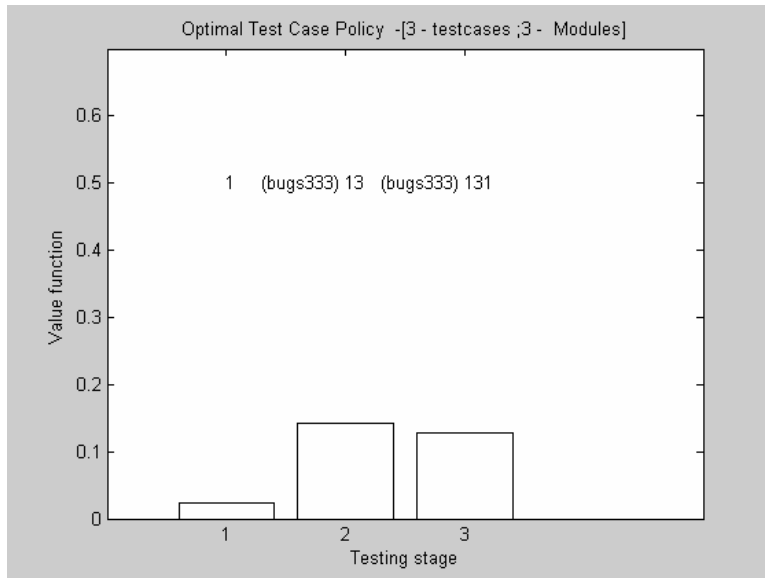


Figure 5.4 Value function v/s s test stages (3 components 3 test cases)

Analysis: Cumulative bugs found in stage 1 is 333 and in stage 2 is 333 which shows an decrease in value function. The decision policy is 131. Test case 1 in stage 1, test case 3 in stage 2 and 1 in stage3.

▪ **Analysis of Results**

- 1.) Different bugs results may result in different policies based on the parameters used to define the model.
- 2.) The model parameters affect the results profoundly and a sound knowledge of the parameters and its affect must be known.

5.5 Conclusion, Extensions & Application

The developed model for optimal test case selection for multi-component is implemented in Matlab. The important feature of decreasing failure intensity was noted in the graph patterns. The results obtained are probabilistic in nature.

The proposed research proposes a model to represent a multi component software system with each component's failure behavior being modeled by a conditional NHPP. Stochastic dynamic programming was used to obtain a test policy in a predetermined testing time. The model is simulated using Matlab. The methodology for utilizing multi processor architecture to carry out the simulation is briefly explained.

The model compared to present research would lead to the conclusion with the fore mentioned unique advantages.

1. Test case policy determination.
2. Policy determined is dynamic in nature. The bugs obtained at each stage of testing are a random variable and hence the policy would accordingly be different for different faults found at each stage.
3. A bug detected follows a distribution $g(t)$ for each component. Function $g(t)$ helps to incorporate expert knowledge of the bug detection. The selection of $g(t)$ would help in defining any pattern of bug detection.

- **Extensions**

- 1.) Implement the parallel algorithm described in the present research to handle a very large number of modules and possible number of bugs in languages like FORTRAN 97.
- 2.) Compare the policy outcome in terms of failure intensity for a live software testing.

- **Application**

Software development typically involves large teams working on diverse parts of software. These diverse modules is finally integrated and tested. Testing done at pre integration stages is insufficient to guarantee a reliable software system. Test cases are used to target bugs in different modules based on the usage of the software. The successful

application of the number of times different test cases affects the system reliability. The policy helps determine crucial decisions in a time stipulated testing environment.

The present research solves the problem of test case policy determination and predicts the average failure intensity the system would have based on the dynamic evolution of the debugging process.

REFERENCES

- Cheung, R.C., (1980), "A User Oriented Software Reliability Model," *IEEE Transactions on, July Software Engineering*, **6(2)**
- Dalal, S.R. and Mallows, C.L. (1988), "When Should One stop Testing Software?," *Journal of the American Statistical Association*, **83(403)**, pp. 872-879.
- Dolbec, J., (1995), "A Structure Based Software Reliability Model," *IBM Centre for Advanced Studies Conference, Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*, pp. 19-28.
- Franid, P.G., Hamlet, R.G., Hamlet, B., Littlewood, B. and Strigini, L., (1998), "Equating Testing Methods by Delivered Reliability," *IEEE Transactions on Software Engineering*, **24**, pp. 586-601
- Gluss, B., (1972), *An Elementary Introduction to Dynamic Programming*, Allyn and Bacon, Inc
- Goel, A.L., and Okumoto, K., (1980), "Optimum Release time for Software Systems based on Reliability and Cost Criteria," *The Journal of Systems and Software*, **1**, pp. 171-186
- Howard, R.A., (1960), *Dynamic Programming and Markov Process*, *Technology Press MIT and John Wiley and Sons*
- Jelinski, Z. and Moranda, P., (1972), "Software Reliability Research," *Statistical Computer Performance Evaluation*, W.Freiberger (ed), *Academic Press*, New York, pp. 465-484
- Jewell, W.S., (1985), "Bayesian Extensions to a Basic Model of Software Reliability," *IEEE Transactions on Software Engineering*, **11(12)**, pp. 1465-1471
- Popstojanova, K.G., and Trivedi, K.S., (2001), "Architecture-based approach to reliability assessment of software systems," *Preprint submitted to Elsevier 2001, Citseer*
- Krishnamurthy, S., and Mathur, A.P., (1997), "On the Estimation of Reliability of a Software System Using Reliabilities of its Components," *Proceedings 8th International Symposium of Software Reliability Engineering (ISSRE)*, pp. 146-155
- Kubat, P., (1989), "Assessing Reliability of Modular Software," *Operations Research Letters*, **8(1)**, February, pp. 35-41
- Laprie, J.C., (1984), "Dependability Evaluation of Software Systems in Operation," *IEEE Transactions on Software Engineering*, **10(6)**
- Littlewood, B., (1979), "Software Reliability Model for Modular Program Structure," *IEEE Transactions on Reliability*, **28(3)**, pp. 241-246

- Littlewood, B., (1975), "A Reliability model for Markov structured software," *Proceedings of the international conference on Reliable Software*, pp. 204-207
- Littlewood, B., and Verall, J.L., (1973), "A Bayesian Reliability Growth Model for Computer Software," *Journal of Royal Statistical Society Series C-Applied*, **22**(3), pp. 332-345
- Littlewood, B., and Sofer, A., (1987), "A Bayesian modification to the Jelinski-Moranda software reliability growth model," *Software Engineering Journal*, **2**, pp. 30-43
- Lynn, K., and Yang, T., (1995), "Bayesian Computation of Software Reliability," *Journal of Computational and Graphical Statistics*, **4**, pp. 65-82
- Meinhold, R.J., and Singapurwalla, N.D., (1983), "Bayesian Analysis of a Commonly Used Model for Describing Software Failures," *The Statistician*, **32**, 168-173
- Musa, J.D., (1986), "A theory of software reliability and its application," *IEEE Transactions on Software Engineering*, **1**(3), pp. 312-327
- Musa, J.D., (1975), "A logarithmic Poisson execution time model for software reliability measurement," Proceeding 7th International Conference on Software Engineering, Orlando, Florida, pp. 230-237
- Musa, J.D., Iannino, A. and Okumoto, K., (1987), "Software Reliability: Measurement, Prediction, Application," *McGraw Hill*, ISBN 0-07-044093-X.
- Neuts, M.F., (1979), "A versatile Markovian point process," *Journal of Applied Probability*, **16**, pp. 764-779
- Pham, H., (2003), "Software reliability and cost models: perspectives, comparison, and practice," *European Journal of Operation Research*, **149**(3), pp 475-489
- Rajgopal, J., and Mazumdar, M., (1996), "A System based Component Test Plan for a series system, with Type II censoring," *IEEE Transactions on Reliability*, pp. 375-378
- Rajgopal, J., and Mazumdar, M., and Majety, S.V., (1999), "Optimum Combined Test Plans for systems and Components," *IIE Transactions*, **31**, pp. 481-490
- Rajgopal, J., and Mazumdar, M., (1992), "Modular Test Plans for Certificate of Software Reliability," Citeseer
- Ramamoorthy, C.V. and Bastani, F.B., (1982), "Software reliability status and perspectives," *IEEE transactions of Software Engineering*, **8**, pp. 354-371
- Ross, S.M., (1989), Introduction to Probability models, Academic Press Inc

- Shantikumar, J.G., and Tufekci, S., (1983), "Application of a software reliability model to decide software release time," *Microelectronics and Reliability*, **23(1)**, pp. 41-59
- Shigeru, Yamada, and Shunji, Osaki, (1985), "Software Reliability Growth Modeling: Models and Applications," *IEEE Transactions on Software Engineering*, **11(12)**
- Siegrist, K., (1988), "Reliability of Systems with Markov Transfer of Control," *IEEE Transactions on Software Engineering*, **14(7)**, pp. 1049-1053
- Shooman, M.L., (1976), "Structural models for software reliability prediction," *Proceedings of second International Conference on Software Engineering*, pp. 268-280
- Shooman, M.L., "Software Engineering: Design, Reliability and Management," *McGraw Hill*.
- Singapurwalla, N.D., and Wilson, S.P., (1994), Statistical methods in software engineering: reliability and risk, *Springer-Verlag*, New-York, Inc.
- Smidts, C., and Sova, D., (1999), "An architectural Model for Software reliability Quantification: Sources of Data," *Reliability Engineering and System safety*, **64**, pp. 279-290
- Trivedi, K.S., Gokhale, S.S., and Lyu, M.R., (2001), "Reliability Simulation of Component-Based Software Systems," *Citeseer*
- Trivedi, A.K., and Shooman, M. L., (1975), "A many state Markov Model for the estimation and prediction of computer software performance parameters," *Proceedings of the international conference on Reliable software*, pp. 208-220
- Trivedi, K.S., and Gokhale, S.S., (1997), *Proceedings of Advanced Computing (ADCOMP)*, Chennai, India
- Wang, W.L., Wu, L., and Chen, M.H., (2000), "An Architecture-Based Software Reliability Model," *Citeseer*
- Whittaker, J.A., and Poore, J.H., (1993), "Markov Analysis of Software Specification," *ACM Transaction of Software Engineering and Methodology*, **2**, 93-106
- Xie, M., "Software Reliability Modeling", (1991), *World Scientific Publishing Company*,
- Xie, M., and Wohlin, C., (1995), "An additive reliability model for the analysis of modular software failure data," *Proceedings 6th International Symposium of Software Reliability Engineering*, pp. 188-194
- Yamada, S., and Osaki, S., (1985), "Software Reliability Growth Modeling: Models and Applications," *IEEE Transactions on Reliability*, **11(12)**, pp. 118-12

Zheng, S., (2002), "Dynamic Release policies for software systems with a reliability constraint,"
IIE Transactions, **34**, pp. 253-262

VITA

Praveen Babu, Kysetti was born in Bangalore, India. A city rich in heritage, culture and labeled as the silicon valley of the east in the present decade. He is the son of Mrs. Geetha Kysetti and Mr. Nityanandam Kysetti. He got his primary and secondary education in Bangalore. He received his Bachelor of Science in mechanical engineering from People's Education Society Institute of Technology (P.E.S.I.T), Bangalore University, in 2000. He worked as an interim in Mico Bosch (Bangalore). He later worked for a brief time after obtaining his bachelor's in TVS Sundram Fasteners (Hosur) as production engineer. Later, he joined the graduate program at Louisiana State University, Baton Rouge, in the spring of 2002. He worked in the areas of information technology and software reliability during his graduate study in LSU. He is a candidate for the degree of Master of Science in Industrial Engineering to be awarded in the commencement of December 2004.