Resource Allocation and Optimal Release Time in Software Systems

Arash Zaryabi Langaroudi

A Thesis

in

The Concordia Institute

for

Information Systems Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science (Quality Systems Engineering) at

Concordia University

Montréal, Québec, Canada

April 2009

Your file Votre référence
ISBN: 978-0-494-63324-3
Our file Notre référence
ISBN: 978-0-494-63324-3

Abstract

Resource Allocation and Optimal Release Time in Software Systems

Arash Zaryabi Langaroudi

Software quality is directly correlated with the number of defects in software systems. As the complexity of software increases, manual inspection of software becomes prohibitively expensive. Thus, defect prediction is of paramount importance to project managers in allocating the limited resources effectively as well as providing many advantages such as the accurate estimation of project costs and schedules. This thesis addresses the issues of statistical fault prediction modeling, software resource allocation, and optimal software release and maintenance policy.

A software defect prediction model using operating characteristic curves is presented. The main idea behind this predictor is to use geometric insight in helping construct an efficient prediction method to reliably predict the cumulative number of defects during the software development process. Motivated by the widely used concept of queue models in communication systems and information processing systems, a resource allocation model which answers managerial questions related to project status and scheduling is then introduced. Using the proposed allocation model, managers will be more certain about making resource allocation decisions as well as measuring the system reliability and the quality of service provided to customers in terms of the expected response time. Finally, a novel stochastic model is proposed to describe the cost behavior of the operation, and estimate the optimal time by minimizing a cost function via artificial neural networks. Further, a detailed analysis of software release time and maintenance decision is also presented.

The performance of the proposed approaches is validated on real data from actual SAP projects, and the experimental results demonstrate a compelling motivation for improved software quality.

Acknowledgements

In the first place I would like to record my gratitude to Dr. A. Ben Hamza for his supervision, advice, and guidance from the very early stage of this research.

I gratefully acknowledge Mr. Torsten Bergander for his advice, and crucial contribution, which made him a backbone of this research. I also would like to thank SAP Inc. for sponsoring the High-Profile Research Alliance Project with Concordia University, and for the financial support during my one-year internship at SAP Labs Canada. I was very happy and proud to be part of Mr. Bergander team and be given the chance to work on a challenging real-world industrial research project. Next, I would like to thank the SAP Research Lab manager Ms. Nolwen Mahé for her help throughout my internship and for supporting my work in many aspects.

Thanks to the Lab role model for hard workers Farshad Ghaderpanah, I am proud to record that I had the opportunity to work with him. Also, it is a pleasure to express my gratitude whole-heartedly to my cousin, Mohammad Hazemi,and his family for their kind hospitality and support. My parents deserve special mention for their inseparable support. My Father, Mohammadrasoul, in the first place is the person who put the fundament my learning character, showing me the joy of intellectual pursuit ever since I was a child. My Mother, Solmaz, is the one who sincerely raised me with her caring and gently love. Mina, thanks for being supportive and caring sister.

Words fail me to express my appreciation to Vian whose dedication, love and persistent confidence in me, has taken the load off my shoulder. I owe her for being unselfishly let her intelligence, passions, and ambitions collide with mine.

Finally, I would like to thank everybody who was important to the successful realization of thesis, as well as expressing my apology that I could not mention personally one by one.

Table of Contents

List of Figures

CHAPTER 1

Introduction

Testing a software product during its development cycle yields a wealth of information that can be used to support the decision processes involved to finally bring the product to the customer [1–3]. In more detail, the testing process generates messages that identify potential software defects [4–6]. These messages are archived, and software companies have a wealth of historical records about them.

Software quality is directly correlated with the number of defects in software systems. As the size and complexity of software increases, manual inspection of software becomes prohibitively expensive. Thus, defect prediction is of paramount importance to project managers in allocating the limited resources effectively, and it also provides many advantages such as the accurate estimation of project costs and schedules as well as improving product and process qualities. Selecting an appropriate defect predictor is a key practical issue [7] because many modeling approaches have been proposed in the literature including reliability growth models [8–11], Bayesian models [5], and artificial neural networks. Most of these models are built using historical defect data and are expected to generalize the statistical patterns for unseen projects. Thus, collecting defect data from past projects is the key challenge for constructing such predictors.

Chapter 1. Introduction

Software systems are mostly developed by different teams under different environments. One of the biggest problems in software development process is to apply optimal strategies on the process to have reliable and cost effective software. There is a wide spread disagreement among software engineers about the appropriate effort to be devoted to software testing before release. Such conflict is attributed to resource constraints and time-to-market considerations. It is well-known that more pre-release development and testing on systems can reduce future development costs and result in higher software quality. On the other hand, the pressure to deliver an operational product quickly can frequently affect the resource allocation among development phases or within one of the phases. Unfortunately, nowadays all these decisions are made intuitionally. However, human's brain is not able to take into account all the effecting parameters at the same time. Besides, human judgements are biased. Hence, there is a high demand for a strategic, mathematically proven approach for these decisions.

In order to have a reliable and cost effective software knowledge about the number of expected failures in a software at any stage is a very valuable asset. It provides essential information for decision making in many software development activities, such as cost analysis, resource allocation, and release and maintenance time decision. It is also useful to obtain a software reliability measure. In addition, having the optimal decisions will result in software quality increase.

The major part of this thesis is devoted to methods for optimal policies in software development processes. The first problem addressed in this thesis is software defect prediction using operating characteristic curves and Laplace trend statistic. The main idea behind our proposed technique is to use geometric insight in helping construct an efficient and fast prediction method to accurately predict the cumulative number of defects at any given stage during the software development process. On the other hand, using queuing theory and predictive models we introduce a resource allocation

model which answers managerial questions related to project status and scheduling. Using the proposed model, managers will be more certain in making resource allocation decisions from one project to the other. This model can also be used to measure the system reliability and quality of service provided to customers in terms of expected response time. Then, we introduce a novel stochastic model for optimal software testing and maintenance policy. We develop a discrete-time stochastic model in discrete operation condition, where the software testing environment and the operational environment are characterized by a environmental factor. In addition, we present a systematic study of fault detection and correction processes. In our model, we consider the fault correction time to estimate the optimal software release and maintenance time which takes into account the environmental factor and imperfect fault removal. More precisely, the total expected cost is formulated via the discrete type of software reliability models based on the difference between operational environments, imperfect fault removal, and fault correction process to remove a fault. To the best of our knowledge, there is no available method in the literature to find the optimal release and maintenance time of a software product by taking into account these assumptions.

Real data from actual SAP projects is used to illustrate the effectiveness and the much improved performance of the proposed methods in comparison with existing approaches. Although additional research efforts might provide a more detailed analysis of the predicted defects, the results presented in this thesis provide a compelling motivation for improved software quality.

1.1 Framework and Motivation

Software Quality Assurance (SQA) is defined as a planned and systematic approach to the evaluation of the quality of and adherence to software product standards, processes, and procedures.

SQA includes the process of assuring that standards and procedures are established and are followed throughout the software acquisition life cycle. Compliance with agreed-upon standards and procedures is evaluated through process monitoring, product evaluation, and audits. Software development and control processes should include quality assurance approval points, where an SQA evaluation of the product may be done in relation to the applicable standards.

One of the many challenges faced when attempting to build a business case for software process improvement is the relative lack of credible measurement data. If a company does not have the data to build the business case, then it does not have the improvement project to get the data. It is the classical chicken-and-egg dilemma. The motivation behind this thesis is to implement statistical models for predicting software defects using available defect data and use this data to find the optimal strategies in software production. The practitioners collect software defect data during software development processes but the decision support power of the collected data is wasted in most of the organizations. These defect data combined with the data of other features become a well-suited repository for using Bayesian statistics and machine learning techniques to predict future defects. Furthermore, these prediction models can be used to systematically define the best possible strategies in software production.

1.1.1 What are software defects?

A software engineer's job is to deliver quality products for their planned costs, and on their committed schedules. Software products must also meet the user's functional needs and reliably and consistently do the user's job. While the software functions are most important to the program's users, these functions are not usable unless the software runs. To get the software to run reliably, however, engineers must remove almost all its defects. Thus, while there are many aspects to

software quality, the first quality concern must necessarily be with its defects.

The reason defects are so important is main reason of customer dissatisfaction. Defects are inevitable because people make a lot of mistakes. In fact, even experienced programmers typically make a mistake for every seven to ten lines of code they develop. While they generally find and correct most of these defects when they compile and test their programs, they often still have a lot of defects in the finished product.

Some people mistakenly refer to software defects (faults) as bugs. When programs are widely used and are applied in ways that their designers did not anticipate, seemingly trivial mistakes can have unforeseeable consequences. As widely used software systems are enhanced to meet new needs, latent problems can be exposed and a trivial-seeming defect can truly become dangerous. While the vast majority of trivial defects have trivial consequences, a small percentage of seemingly silly mistakes can cause serious problems. Since there is no way to know which of these simple mistakes will have serious consequences, we must treat them all as potentially serious defects, not as trivial-seeming "bugs".

The term defect or fault refers to something that is wrong with a program. It could be a misspelling, a punctuation mistake, or an incorrect program statement. Defects can be in programs, in designs, or even in the requirements, specifications, or other documentation. Defects can be redundant or extra statements, incorrect statements, or omitted program sections. A defect, in fact, is anything that detracts from the program's ability to completely and effectively meet the user's needs. A defect is thus an objective thing. It is something you can identify, describe, and count. Failure, is when a defect becomes active or in other words we face that defect.

Simple coding mistakes can produce very destructive or hard-to-find defects. Conversely, many sophisticated design defects are often easy to find. The sophistication of the design mistake and the

impact of the resulting defect are thus largely independent. Even trivial implementation errors can cause serious system problems. This is particularly important since the source of most software defects is simple programmer oversights and mistakes. While design issues are always important, initially developed programs typically have few design defects compared to the number of simple oversights, typos, and goofs. To improve program quality, it is thus essential that engineers learn to manage all the defects they inject in their programs.

1.1.2 Software reliability growth models

Achieving highly reliable software from the customers perspective is a demanding job for all software engineers and reliability engineers. [12] summarizes the following four technical areas which are applicable to achieving reliable software systems, and they can also be regarded as four fault lifecycle techniques:

1. Fault prevention: to avoid, by construction, fault occurrences.
2. Fault removal: to detect, by verification and validation, the existence of faults and eliminate them.
3. Fault tolerance: to provide, by redundancy, service complying with the specification in spite of faults having occurred or occurring.
4. Fault/failure forecasting: to estimate, by evaluation, the presence of faults and the occurrences and consequences of failures. This has been the main focus of software reliability modeling.

Fault prevention is the initial defensive mechanism against unreliability. A fault which is never created costs nothing to fix. Fault prevention is therefore the inherent objective of every software

engineering methodology. Fault prevention mechanisms cannot guarantee avoidance of all software faults. When faults are injected into the software, fault removal is the next protective means. Two practical approaches for fault removal are software inspection and software testing, both of which have become standard industry practices in quality assurance.

When inherent faults remain undetected through the inspection and testing processes, they will stay with the software when it is released into the field. Fault tolerance is the last defending line in preventing faults from manifesting themselves as system failures. Fault tolerance is the survival attribute of software systems in terms of their ability to deliver continuous service to the customers. Software fault tolerance techniques enable software systems to (1) prevent dormant software faults from becoming active, such as defensive programming to check for input and output conditions and forbid illegal operations; (2) contain the manifested software errors within a confined boundary without further propagation, such as exception handling routines to treat unsuccessful operations; (3) recover software operations from erroneous conditions, such as checkpointing and rollback mechanisms; and (4) tolerate system-level faults methodically, such as employing design diversity in the software development. Finally if software failures are destined to occur, it is critical to estimate and predict them. Fault/failure forecasting involves formulation of the fault/failure relationship, an understanding of the operational environment, the establishment of software reliability models, developing procedures and mechanisms for software reliability measurement, and analyzing and evaluating the measurement results. The ability to determine software reliability not only gives us guidance about software quality and when to stop testing, but also provides information for software maintenance needs.

Software reliability may be the most important quality attribute of software, due to the fact that it quantifies software failures during the software development process. Software reliability models

usually make a number of common assumptions, as follows. (1) The operation environment where the reliability is to be measured is the same as the testing environment in which the reliability model has been parameterized. (2) Once a failure occurs, the fault which causes the failure is immediately removed. (3) The fault removal process will not introduce new faults. (4) The number of faults inherent in the software and the way these faults manifest themselves to cause failures follow, at least in a statistical sense, certain mathematical formulae.

There are essentially two types of software reliability models:

- those that attempt to predict software reliability from design parameters
- those that attempt to predict software reliability from test data

The first type of models are usually called "defect density" models and use code characteristics such as lines of code, nesting of loops, external references, input/outputs, and so forth to estimate the number of defects in the software. The second type of models are often called software reliability growth models (SRGMs) since the number of faults (as well as the failure rate) of the software system reduces when the testing progresses, resulting in growth of reliability. These models attempt to statistically correlate defect detection data with known functions such as an exponential function.

Each software defect encountered entails a significant cost for software companies. Hence the knowledge of the number of defects in a software product during its lifecycle is a very valuable asset. Being able to estimate the number of defects will substantially improve the decision processes in software lifecycle like time to release and maintenance time. In addition, the production process of the software can be considerably improved by employing a prediction model that reliably the number of defects.

During the development process of software, many defects may be introduced and often lead to critical problems and complicated breakdowns of computer systems [1]. Thus there is a high demand for controlling the quality and reliability of software development process. As an evaluation for software reliability, number of defects can be used. In the traditional software development environment, software reliability evaluation provides useful guidance in balancing reliability, time to market and development cost [2]. Therefore, there is a greater than ever demand for prediction the quality and reliability of software.

Among all SRGMs, a large family of stochastic reliability models are based on a non homogeneous Poisson process, which is known as NHPP reliability models, has been widely used to track reliability improvement during software testing. These models enable software developers to evaluate software reliability in a quantitative manner. They have also been successfully used to provide guidance in making decisions such as when to terminate testing the software or how to allocate available recourses. However, software development is a very complex process and there are still issues that have not yet been addressed.

Software fault and failure reports are available in three basic forms:

1. Sequence of ordered failure times $0 < t_1 < t_2 < \ldots < t_n$
2. Sequence of failure times τ_i where $\tau_i = t_i - t_{i-1}, i = 1, \ldots, n$
3. Cumulative number of faults.

The general NHPP software reliability growth model is formulated based on the following assumptions:

- The occurrence of software faults follows an NHPP with mean value function $m(t)$ and failure intensity function $\lambda(t)$.

- The software fault intensity rate at any time is proportional to the number of remaining faults in the software at that time.
- When a software fault is detected, a debugging effort takes place immediately.

Let $\{N(t), t \geq 0\}$ denote a counting process representing the cumulative number of faults detected by the time t, and $m(t) = E[N(t)]$ denote its expectation. The failure intensity $\lambda(t)$ and the mean value functions of the software at time t are related as follows

$$m(t) = \int_0^t \lambda(s)ds$$

and

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

The cumulative number of faults detected at time t follows a Poisson distribution with time-dependent mean value function as follows

$$P\{N(t) = n\} = \frac{m(t)^n}{n!} e^{-m(t)}, \quad n = 0, 1, 2, \ldots, \infty$$

The software reliability, i.e., the probability that no failures occur in $(s, s + t)$ given that the last failure occurred at testing time s is

$$R(t|s) = \exp[-(m(t + s) - m(t))]$$

The mean value function $m(t)$ is nondecreasing with respect to testing time t under the bounded condition $m(\infty) = a$, where a is the expected total number of faults to be eventually detected. Knowing its value can help us to determine whether the software is ready to be released to the customers and how much more testing resources are required. It can also provide an estimate of the number of failures that will eventually be encountered by the customers. The mean value

Model name	$m(t)$	$\lambda(t)$
Log-linear	$\frac{\exp(\alpha+\beta t)}{\beta}$	$\exp(\alpha+\beta t)$
Exponential (Goel-Okumoto)	$\alpha[1-\exp(-\beta t)]$	$\alpha\beta\exp(-\beta t)$
Weibull (Generalized Goel-Okumoto)	$\alpha[1-\exp(-\beta t^{\gamma})]$	$\alpha\beta\gamma t^{\gamma-1}\exp(-\beta t^{\gamma})$
Power law	$\left(\frac{t}{\alpha}\right)^{\beta}$	$\frac{\beta}{\alpha}\left(\frac{t}{\alpha}\right)^{\beta-1}$
S-shaped	$\alpha[1-(1+\beta t)\exp(-\beta t)]$	$\alpha\beta^2 t\exp(-\beta t)$

Table 1.1: NHPP models.

function can be expressed as follows

$$m(t) = aF(t),$$

where $F(t)$ is the cumulative distribution function. Hence,

$$\lambda(t) = aF'(t) = [a - m(t)]\frac{F'(t)}{1 - F(t)} = [a - m(t)]\rho(t),$$

where $\rho(t)$ is the failure occurrence rate per fault of the software, or the rate at which the individual faults manifest themselves as failures during testing. The quantity $[a - m(t)]$ denotes the expected number of faults remaining. The failure occurrence rate per fault (also known as *hazard* function)

$$\rho(t) = \frac{\lambda(t)}{m(\infty) - m(t)}$$

can be a constant, increasing, decreasing, or increasing/decreasing.

Table 2.1 and Figure 1.1 show examples of NHPP models with different failure intensity functions $\lambda(t;\boldsymbol{\theta})$, where $\boldsymbol{\theta} = (\alpha, \beta)$.

1.1.3 Operating characteristic curves

A statistical test provides a mechanism for making quantitative decisions about a process or processes [13]. The intent is to determine whether there is enough evidence to "reject" a conjecture or

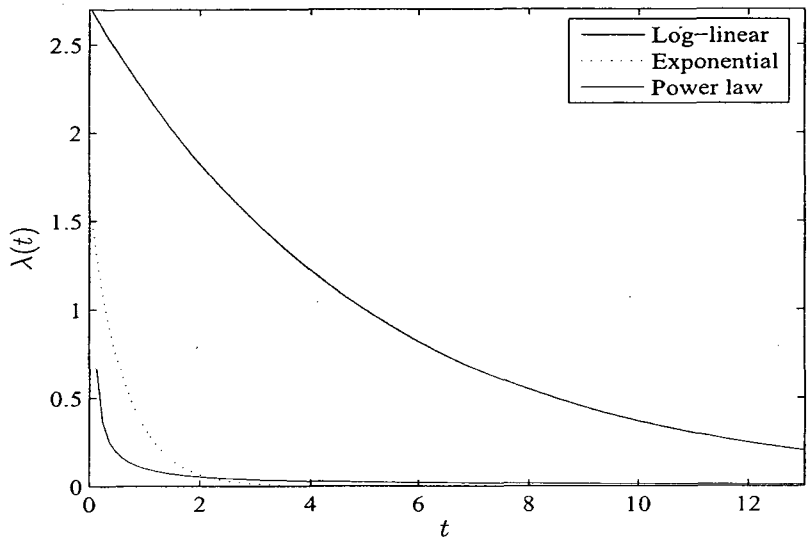

Figure 1.1: Illustration of failure intensity functions.

hypothesis about the process. The conjecture is called the null hypothesis. Not rejecting may be a good result if we want to continue to act as if we "believe" the null hypothesis is true. Or it may be a disappointing result, possibly indicating we may not yet have enough data to "prove" something by rejecting the null hypothesis. A classic use of a statistical test occurs in process control studies, and it requires a pair of hypotheses:

$$H_0 : \text{a null hypothesis}$$

$$H_1 : \text{an alternative hypothesis}$$

The null hypothesis is a statement about a belief. We may doubt that the null hypothesis is true, which might be why we are "testing" it. The alternative hypothesis might, in fact, be what we believe to be true. The test procedure is constructed so that the risk of rejecting the null hypothesis, when it is in fact true, is small. This risk, α, is often referred to as the *significance level* of the test. By having a test with a small value of α, we feel that we have actually "proved" something when

we reject the null hypothesis. The risk of failing to reject the null hypothesis when it is in fact false is not chosen by the user but is determined, as one might expect, by the magnitude of the real discrepancy. This risk, β, is usually referred to as the *error of the second kind.* Large discrepancies between reality and the null hypothesis are easier to detect and lead to small errors of the second kind; while small discrepancies are more difficult to detect and lead to large errors of the second kind. Also the risk β increases as the risk α decreases. The risks of errors of the second kind are usually summarized by an *operating characteristic curve* (OC) for the test [13].

1.1.4 Bayesian statistics

Bayesian inference is statistical inference in which evidence or observations are used to update or to newly infer the probability that a hypothesis may be true. The name "Bayesian" comes from the frequent use of Bayes' theorem in the inference process [14, 15]. Bayesian inference uses aspects of the scientific method, which involves collecting evidence that is meant to be consistent or inconsistent with a given hypothesis. As evidence accumulates, the degree of belief in a hypothesis changes. With enough evidence, it will often become very high or very low. Thus, proponents of Bayesian inference say that it can be used to discriminate between conflicting hypotheses: hypotheses with a very high degree of belief should be accepted as true and those with a very low degree of belief should be rejected as false. However, detractors say that this inference method may be biased due to initial beliefs that one needs to hold before any evidence is ever collected.

Bayesian inference uses a numerical estimate of the degree of belief in a hypothesis before evidence has been observed and calculates a numerical estimate of the degree of belief in the hypothesis after evidence has been observed. Bayesian inference usually relies on degrees of belief, or subjective probabilities, in the induction process and does not necessarily claim to provide

an objective method of induction. Nonetheless, some Bayesian statisticians believe probabilities can have an objective value and therefore Bayesian inference can provide an objective method of induction. Bayes' theorem adjusts probabilities given new evidence in the following way:

$$P(H_0|E) = \frac{P(E|H_0)P(H_0)}{P(E)},$$

where

- H_0 represents the null hypothesis that was inferred before new evidence, E, became available.
- $P(H_0)$ is called the prior probability of H_0.
- $P(E|H_0)$ is called the conditional probability of seeing the evidence E given that the hypothesis H_0 is true. It is also called the likelihood function when it is expressed as a function of H_0 given E.
- $P(E)$ is called the marginal probability of E: the probability of witnessing the new evidence E under all mutually exclusive hypotheses. It can be calculated as the sum of the product of all probabilities of mutually exclusive hypotheses and corresponding conditional probabilities: $\sum P(E|H_i)P(H_i)$.
- $P(H_0|E)$ is called the posterior probability of H_0 given E.

The factor $P(E|H_0)/P(E)$ represents the impact that the evidence has on the belief in the hypothesis. If it is likely that the evidence will be observed when the hypothesis under consideration is true, then this factor will be large. Multiplying the prior probability of the hypothesis by this factor would result in a large posterior probability of the hypothesis given the evidence. Under Bayesian inference, Bayes theorem therefore measures how much new evidence should alter a belief in a hypothesis. Bayesian methods aim at assigning prior distributions to the parameters in the model in order to incorporate whatever *a priori* quantitative or qualitative knowledge we have available,

and then to update these priors in the light of the data, yielding a posterior distribution via Bayes Theorem. The ability to include prior information in the model is not only an attractive pragmatic feature of the Bayesian approach, but it is also theoretically vital for guaranteeing coherent inferences.

1.1.5 Neural networks in software reliability growth modeling

Neural networks are composed of simple elements operating in parallel [16, 17]. These elements are inspired by biological nervous systems. As in nature, the network function is determined largely by the connections between elements. We can train a neural network to perform a particular function by adjusting the values of the connections (weights) between elements. Commonly neural networks are adjusted, or trained, so that a particular input leads to a specific target output. The network is adjusted, based on a comparison of the output and the target, until the network output matches the target. Typically many such input/target pairs are used, in this supervised learning, to train a network. Neural networks have been trained to perform complex functions in various fields of application including pattern recognition, identification, classification, speech, vision and control systems. Neural networks are learning mechanisms that can approximate any non-linear continuous functions based on the given data. The goal of using neural network is to approximate a non-linear function that can receive a vector $\boldsymbol{X} = (x_1, ..., x_n)$ in $\mathbb{R}^n$ and has a output vector $\boldsymbol{Y} = (y_1, ..., y_m)$ in $\mathbb{R}^m$. Hence, we define the network as follows:

$$\boldsymbol{Y} = F(\boldsymbol{X}) \tag{1}$$

The elements of $\boldsymbol{Y}(y_k)$ are given by

$$y_k = g\left(b_k + \sum_{j=1}^{H} w_{jk}^0 h_j\right), \quad k = 1, \ldots, M \tag{2}$$

where w_{jk}^0 is the output weight from the hidden layer node j to the output layer node k, h_j is the output of the hidden layer j, b_k is the bias of the output node k, and g is the activation function in output layers. The hidden layer values are given by

$$h_j = f\left(b_j + \sum_{i=1}^{N} w_{ij}^1 x_i\right), \quad j = 1, ..., H \tag{3}$$

where w_{ij}^1 is the input weight from the input layer node i to the hidden layer node j, x_i is the value at input note i, b_j is the bias of node j, and f is the activation function in the hidden layer.

1.1.6 Software resource allocation

Modern complex software systems are mostly developed incrementally with different components by different teams under different environments [18]. In such a situation estimating the cost of a product is not an easy task. Another problem in software engineering is how to quantitatively measure the quality of the software. Most of the software quality measurements are based on counting the defects found in a software systems. These approaches are typically developer-oriented. Optimal software resource allocation is one of the most important applications of defect prediction and SRGMs. Several research efforts have been conducted in this field and most of them are based on SRGMs and defect prediction [19–23].

1.1.7 Optimal software release time

In order to make the business a success, developing high quality products is of paramount importance. In a software development project, it is important to know when to stop software testing and deliver the software to the market [24].

1.2 Thesis Overview and Contributions

The organization of this thesis is as follows:

- The first Chapter contains a brief review of essential concepts and definitions which we will refer to throughout the thesis, and presents a short summary of material relevant to software defect prediction methods, Bayesian statistics, operating characteristic curves, neural networks, resource allocation, and optimal software release time.

- In Chapter 2, we present a software defect prediction model using operating characteristic curves and Laplace trend statistic [25]. The main idea behind our proposed technique is to use geometric insight in helping construct an efficient and fast prediction method to accurately predict the cumulative number of defects during the software development process. Experimental results illustrate the effectiveness and the much improved performance of the proposed method in comparison with the Bayesian prediction approaches.

- In Chapter 3, we introduce a new resource allocation model that answers managerial questions related to project status and scheduling [26]. Using the proposed model, managers will be more certain about making resource allocation decisions. This model can also be used to measure the system reliability and the quality of service provided to customers in terms of the expected response time. Experimental results illustrate the effectiveness of the proposed method in the software development process.

- In Chapter 4, we develop a discrete-time stochastic model for optimal software testing and maintenance policy, where the software testing environment and the operational environment is characterized by an environmental factor [27]. We present a systematic study of defect

detection and correction processes. In our model, we consider the defect correction time to estimate the optimal software release and maintenance time which takes into account the environmental factor and the imperfect fault removal. More precisely, the total expected cost is formulated via a discrete-type software reliability model based on the difference between operational environments, imperfect defect removal, and defect correction process.

- In the **Conclusions** Chapter, we summarize the contributions of this thesis, and we propose several future research directions that are directly or indirectly related to the work performed in this thesis.

CHAPTER 2

Predictive Operating Characteristic Curves

In this chapter, we introduce a software defect prediction model based on the concept of operating characteristic curve and Laplace trend statistic. The idea is to use operating characteristic curves in statistical quality control and a geometric approach to construct an efficient, fast, and accurate prediction method to estimate the cumulative number of software defects during the software development process. The experimental results demonstrate the effectiveness and the improved performance of the proposed method in comparison with the Bayesian prediction approaches.

2.1 Introduction

Knowledge about the number of expected defects in a software product at any stage provide essential information for decision making in many software development activities, such as cost analysis, resource allocation in testing and release decision time. The aim of software reliability growth modelling (SRGM) is to explain the behavior of software testing process caused by faults. Most existing SRGMs only model fault detection processes with unrealistic assumptions such as perfect debugging. In this report, we use an improved SRGM with more accuracy and realistic

assumptions.

During the development process of computer software systems, many software defects may be introduced and often lead to critical problems and complicated breakdowns of computer systems [1]. Hence, there is an increasing demand for controlling the software development process in terms of quality and reliability. Software reliability can be evaluated by the number of detected faults. A software failure is defined as an unacceptable departure of program operation caused by a software fault remaining in the software system [2, 3]. In the traditional software development environment, software reliability evaluation, which shorten development intervals and reduce development costs, provides useful guidance in balancing reliability, time-to-market and development cost [6]. Hence, there is an increasing demand for prediction the quality and reliability of software.

Several software reliability prediction models have been proposed in the literature for estimating system reliability, but all these kinds of models make unrealistic assumptions to ensure solvability [2, 8–11, 19, 28, 29]. These unreasonable assumptions have limited the applications of these models [5, 7].

Bayesian statistics provide a framework for combining observed data with prior assumptions in order to model stochastic systems. Bayesian methods aim at assigning prior distributions to the parameters in the model in order to incorporate whatever *a priori* quantitative or qualitative knowledge we have available, and then to update these priors in the light of the data, yielding a posterior distribution via Bayes's Theorem [15]. The ability to include prior information in the model is not only an attractive pragmatic feature of the Bayesian approach, but it is also theoretically vital for guaranteeing coherent inferences.

Motivated by the widely used concept of operating characteristic (OC) curves in statistical quality control to select the sample size at the outset of an experiment [13], we propose in this

chapter a software defect prediction technique using OC curves in order to predict the cumulative number of failures at any given time. The core idea behind our proposed methodology is to use geometric insight in helping construct an efficient and fast prediction method to accurately predict the cumulative number of failures at any given time.

The layout of this chapter is organized as follows. In the next Section, a problem formulation is stated. In Section 2.3, we briefly review some Bayesian prediction models that will be used for comparison with our proposed approach. In Section 2.4, we propose a new prediction algorithm based on OC curves. In Section 2.5, we present experimental results to demonstrate the much improved performance of the proposed approach in the prediction of software defects. Finally, some conclusions are included in Section 2.6.

2.2 Problem Formulation

Usually the fault reports are available in three basic forms:

1. in the form of a sequence of ordered time of occurrences

   $0 < t_1 < t_2 < \ldots < t_n$

2. in the form of a sequence of interfailure times τ_i where $\tau_i = t_i - t_{i-1}$ for $i = 1, \ldots, n$

3. in the form of cumulative number of failures detected by a time $N(t_i)$.

Failure(t_i) and interfailure ($\tau_(j)$)times are related by

$$t_i = \sum_{j=1}^{i} \tau_j,$$

The cumulative number of failures defines a non homogeneous Poisson process (NHPP) with failure intensity or rate function $\lambda(t_i)$ which is a function of time. The mean value function $m(t_i) =$

$E(N(t_i))$ of the process is given by $m(t_i) = \int_0^{t_i} \lambda(u)du$. Moreover, the probability of having κ failures in an interval is:

$$P(N(t_j) - N(t_i) = \kappa) = \frac{(m(t_j) - m(t_i))^\kappa}{\kappa!} \exp(-(m(t_j) - m(t_i))).$$

This is equal to say $N(t+s) - N(t)$ is a Poisson distributed with expected value

$$\int_{t_i}^{t_j} \lambda(u)du = m(t_j) - m(t_i).$$

where $\lambda(t)$ is the time dependant intensity. Hence, the number of failures in any interval $[t_i, t_j)$ defines a NHPP.

According to ANSI, Software Reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment [57]. Although Software Reliability is defined as a probabilistic function, and comes with the notion of time, we must note that, different from traditional Hardware Reliability, Software Reliability is not a direct function of time. Electronic and mechanical parts may become "old" and wear out with time and usage, but software will not rust or wear-out during its life cycle. Software will not change over time unless intentionally changed or upgraded. Software reliability $R(t_j|t_i)$ is defined as the probability that no software failure is detected in the time interval $(t_i, t_i + t_j)$, given that the last failure occurred at testing time t_i, and it is given by

$$R(t_j|t_i) = \exp\Big(-\big(m(t_i + t_j) - m(t_i)\big)\Big).$$

It is worth pointing out that if the failure intensity function is time-independent, then the cumulative number of failures $N(t_i)$ defines a homogeneous Poisson process (HPP).

Model name	$m(t)$	$\lambda(t)$
Log-linear	$\frac{\exp(a+bt)}{b}$	$\exp(a+bt)$
Exponential	$a(1-\exp(-bt))$	$ab\exp(-bt)$
Power law	$\left(\frac{t}{a}\right)^b$	$\frac{b}{a}\left(\frac{t}{a}\right)^{b-1}$

Table 2.1: NHPP models.

Note that the interfailure times may have non-exponential distributions, and hence the cumulative number of failures $N(t_i)$ would define a general renewal process.

The problem addressed in this section may now be concisely described as follows: Given the historical failure times data $\mathcal{D} = \{t_1, \ldots, t_n\}$ and its corresponding cumulative number of failures data $\mathcal{N} = \{N(t_1), \ldots, N(t_n)\}$, find the predicted cumulative number of failures at any given time t.

2.3 Prediction using Bayesian Statistics

Scientific experimental or observational results generally consist of (possibly many) sets of data. Bayesian statistics uses both prior and sample information. Usually something is known about possible parameter values before the experiment is performed.

We model the failure times using an NHPP with a parameterized failure intensity function $\lambda(t;\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is a vector of unknown parameters which can be obtained by historical data. Table 2.1 shows examples of NHPP models with different failure intensity functions $\lambda(t;\boldsymbol{\theta})$, where $\boldsymbol{\theta} = (a, b)$.

Bayesian methods aim at assigning prior distributions to the parameters $\boldsymbol{\theta}$ is the model in order to incorporate whatever *a priori* quantitative or qualitative knowledge we have available, and then to update these priors in the light of the data, yielding a posterior distribution via Bayes's Theorem.

The ability to include prior information in the model is not only an attractive pragmatic feature of the Bayesian approach, but it is also theoretically vital for guaranteeing coherent inferences.

2.3.1 Predictive density

Consider the problem of making prediction for a new failure time t without any measurements on the predictors for any of the individuals so that the dataset is just given by $\mathcal{D} = \{t_1, \ldots, t_n\}$. That is, we want to determine $p(t|\mathcal{D})$, the probability density function of the new failure time conditioned on the observed failure times. The function $p(t|\mathcal{D})$ is referred to as *predictive density* of a new failure time and may be written in integral form as

$$p(t|\mathcal{D}) = \int p(t|\mathcal{D}, \boldsymbol{\theta}) p(\boldsymbol{\theta}|\mathcal{D}) d\boldsymbol{\theta},$$

where $p(\boldsymbol{\theta}|\mathcal{D})$ is the posterior distribution of $\boldsymbol{\theta}$ given by

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathcal{D})} = \frac{\{\prod_{i=1}^{n} p(t_i|\boldsymbol{\theta})\}p(\boldsymbol{\theta})}{\int \{\prod_{i=1}^{n} p(t_i|\boldsymbol{\theta})\}p(\boldsymbol{\theta}) d\boldsymbol{\theta}}$$

and $p(\boldsymbol{\theta})$ is the prior distribution which represents information available about the unknown parameters. The prior estimate provides a means of combining exogenous information with observed data in order to estimate parameters of a probability distribution. It is convenient to choose simple forms of prior distributions which result in computationally tractable posterior distributions. Hence, the posterior distribution is found by combining the prior distribution $p(\boldsymbol{\theta})$ with the probability $p(\mathcal{D}|\boldsymbol{\theta})$ of observing the data given the parameters. The probability $p(\mathcal{D}|\boldsymbol{\theta})$ is also called the likelihood function of the data and it is given by

$$p(\mathcal{D}|\boldsymbol{\theta}) = \prod_{i=1}^{n} p(t_i|\boldsymbol{\theta}),$$

where

$$p(t_i|\boldsymbol{\theta}) = \lambda(t_i;\boldsymbol{\theta}) \exp\left(-\int_0^{t_i} \lambda(u;\boldsymbol{\theta})du\right)$$

assuming that the failure times data are independent and identically distributed (iid). The likelihood function is the probability of observing the given data as a function of $\boldsymbol{\theta}$.

Hence, the Bayesian approach consists of three main steps:

1. Assign prior distributions to all the unknown parameters.

2. Determine the likelihood of the data given the parameters.

3. Determine the posterior distribution of the parameters given the data.

Maximum Likelihood is a statistical estimator that can be used to estimate a models unknown parameters values from data. The maximum likelihood estimate (MLE) of $\boldsymbol{\theta}$ is that value of $\boldsymbol{\theta}$ that maximizes the likelihood function $p(\mathcal{D}|\boldsymbol{\theta})$ or equivalently that maximizes the log-likelihood function:

$$\log(p(\mathcal{D}|\boldsymbol{\theta}))$$

and it is the value that makes the observed data the most "probable".

2.3.2 Bayesian prediction

The Bayesian prediction approach proposed in [4] is based on the power law model shown in Table 2.1. The parameter b of the power law model may be estimated as follows

$$\widehat{b} = \frac{t_n}{\sum_{t=t_1}^{t_n} \log[N(t_n)/N(t)]},$$

and the predicted cumulative number of defects $N(t)$ at time t is given by

$$N(t) = N(t_n)\left(\frac{t}{t_n}F(2t, 2t_n; \gamma)\right)^{1/\hat{b}}, \tag{1}$$

where $\gamma = P\{\chi_n^2 \leq \chi_{\gamma,n}^2\}$, and $F(2t, 2t_n; \gamma)$ denotes the γ percentage point of the F-distribution with $2t$ and $2t_n$ degrees of freedom.

2.3.3 Bayesian prediction using MCMC

Markov chain Monte Carlo (MCMC) methods (which include random walk Monte Carlo methods), are a class of algorithms for sampling from probability distributions based on constructing a Markov chain that has the desired distribution as its equilibrium distribution. The state of the chain after a large number of steps is then used as a sample from the desired distribution. The quality of the sample improves as a function of the number of steps.

If we draw samples $\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(N)}$ from the posterior distribution $p(\boldsymbol{\theta}|\mathcal{D})$, then the predictive density may be approximated as follows

$$p(t|\mathcal{D}) \approx \sum_{i=1}^{N} p(t|\mathcal{D}, \boldsymbol{\theta}^{(i)})p(\boldsymbol{\theta}^{(i)}|\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N} p(t|\mathcal{D}, \boldsymbol{\theta}^{(i)}).$$

The samples $\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(N)}$ are draws from the posterior distribution of $\boldsymbol{\theta}$, and may be obtained using Markov chain Monte Carlo (MCMC) simulation algorithms [14, 56].

For the Bayesian prediction approach using MCMC, the predicted cumulative number of defects $N(t)$ at time t is also given by Eq. (5) where $\hat{b}$ is estimated using the MCMC algorithm [14].

The algorithm of MCMC estimate parameters $\widehat{b}$ consists of the following steps:

1. Using MCMC to simulate each parameter distribution.

2. Estimate the maximal likely value of parameter distribution which gives us the value of expected parameter.

2.4 Proposed Method

2.4.1 POC curve

Consider the two-sided hypothesis

$$H_0 : \ t = t_k$$

$$H_1 : \ t \neq t_k$$

where H_0 and H_1 are the null and the alternative hypotheses respectively.

Define $\chi^2_{\alpha,k}$ as the percentage value of the chi-square distribution with k degrees of freedom such that the probability that the chi-square distribution χ^2_n exceeds this value is α, that is

$$P\{\chi^2_k \geq \chi^2_{\alpha,k}\} = \alpha = P\{\text{reject } H_0 | H_0 \text{ is true}\},$$

where $\alpha \in (0, 1)$ is the probability of type I error (also referred to as the significance level). In other words we can be $100(1-\alpha)\%$ confident about the result.

Note that in probability theory and statistics, the chi-square distribution are k independent, normally distributed random variables.

Suppose $t = t_k + \delta$, where $\delta > 0$ (we have the same result for $\delta < 0$) then H_0 is false and H_1 is true. Hence, the distribution of the test statistic

$$Z = \frac{\chi^2_t - t_k}{\sqrt{2k}}$$

has a mean value equal to $\delta/\sqrt{2k}$, and a type II error will be made only if $-\chi^2_{\alpha/2} \leq Z \leq \chi^2_{\alpha/2}$. That is, the probability of type II error $\beta = P\{\text{accept } H_0 | H_0 \text{ is false}\}$ may be expressed as

$$\beta = \Phi\left(\chi^2_{\frac{\alpha}{2},t} - \frac{\delta}{\sqrt{2k}}\right) - \Phi\left(-\chi^2_{\frac{\alpha}{2},t} - \frac{\delta}{\sqrt{2k}}\right),$$

where Φ is the cumulative distribution function of χ^2_t.

The function $\beta(t)$ is evaluated by finding the probability that the test statistic Z falls in the acceptance region given a particular value of t.

An operating Characteristic (OC) curve is a graph used to determine the probability of accepting lots as a function of the lots or processes quality level when using various sampling plans. In other words the operating characteristic (OC) curve of a test is the plot of $\beta(t)$ against t. Note that given the OC curve parameters β, α, k, and δ, we can derive the predicted cumulative number of defects at time t as follows

$$N(t) = \left(\frac{\sqrt{2k}}{\delta}\right)^2 \left(\chi^2_{\alpha,\delta} + \chi^2_{\beta,\delta}\right)^2. \tag{2}$$

Figure 2.1 depicts a plot of the cumulative number of defects using OC curves.

The above method does not take into account the historical data to predict. To overcome this limitation, we propose a predictive operating characteristic (POC) curve where the predicted cumulative number of defects at time t is calculated as follows

$$N(t) = \left(\frac{\sqrt{2p}}{\delta}\right)^2 \left(\chi^2_{\alpha,\delta} + \chi^2_{\beta,\delta}\right)^2, \tag{3}$$

and the parameter p is given by (see Figure 2.2)

$$p = \begin{cases} N(t), & \text{if } t \leq t_n \\ N(t_n), & \text{if } t_n < t \leq T. \end{cases}$$

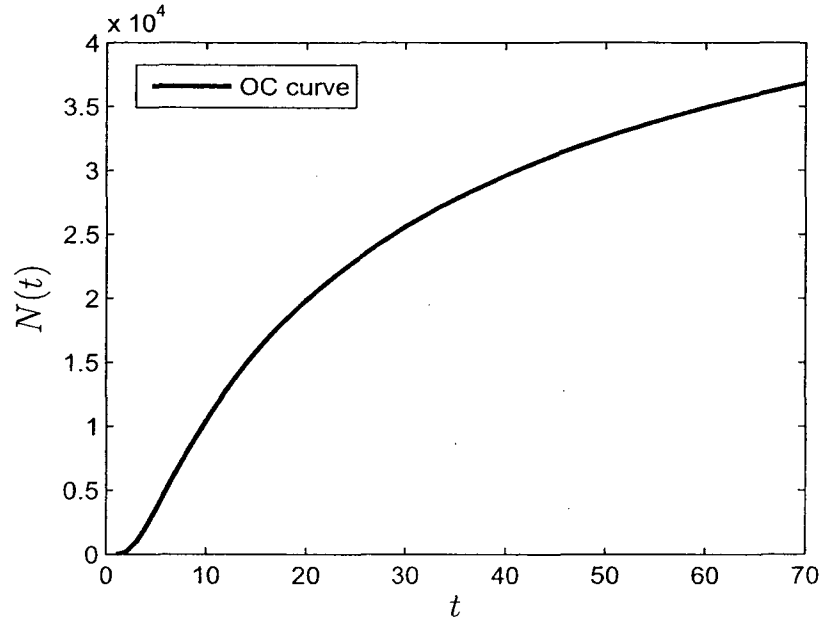

Figure 2.1: Illustration of cumulative number of defects using OC curves.

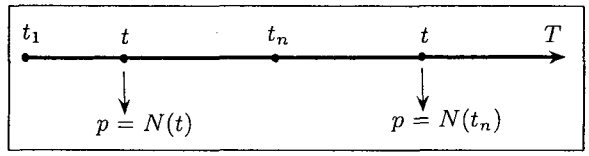

Figure 2.2: Illustration of the p parameter in the POC curve.

2.4.2 Laplace trend analysis

One of the drawbacks of POC prediction method is its inability to predict accurately the cumulative number of defects when the software is not stable, that is when the software does not have a reliability growth yet. To circumvent this limitation, we used a weighted Laplace trend to validate the reliability and stability of the software before using POC for defect prediction [40].

Suppose we wish to test the following hypotheses:

$$H_0: \quad HPP$$

$$H_1: \quad NHPP$$

where H_0 and H_1 are the null and the alternative hypotheses respectively.

Under the null hypothesis, we define the Laplace trend as:

$$U = \frac{\mathcal{L}(\theta_i)'}{E(-\mathcal{L}(\theta_i)'')}$$

where θ_i is a component of the vector θ such that its value makes the intensity function $\lambda(t;\theta)$ time independent.

With type I error probability α, we have the following interpretation of the Laplace trend value:

- $U < -z_\alpha$: reliability growth (stable system behavior).
- $U > z_\alpha$: reliability deterioration (non-stable system).
- $-z_\alpha < U < z_\alpha$: stable reliability (in control behavior).

where z_α is the upper α percentage of the standard normal distribution Z such that $P\{Z \geq z_\alpha\} = \alpha$. If H_0 is true, the distribution of the Laplace test statistic approximately follows standard normal distribution $N(0,1)$.

Note that Laplace trend analysis is used to determine whether the pattern of defects is significantly changing with respect time or not. To have a better analysis we may also use a weighted Laplace test statistic as discussed in [41]. However, for simplicity we focus on the standard Laplace test statistic.

Now we try to find a "Laplace trend stopping increase" point ($t = t_s$) as shown in Figure 2.3. We can start using the POC curve when Laplace trend starts to decrease ($t = t_s, ..., T$) because at this point the behavior of the system becomes stable and therefore we have reliability growth.

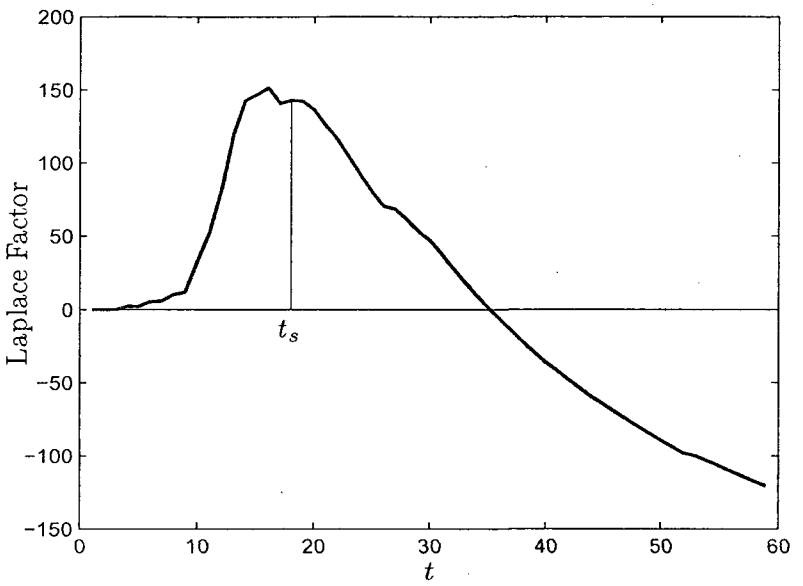

Figure 2.3: Laplace Factor vs. Defect Time.

2.4.3 Improved POC curve

In a real software project, removal of one defect might cause other defects in the system. In addition, the defect causing the failure cannot be removed immediately. In the improved POC curve approach, we can incorporate these assumptions to be able to predict the behavior of the software in a better way.

To overcome the problem of imperfect debugging, we assume that when a defect occurs and the correction process has been performed the defect is repaired with a probability p, in which case the defect rate is reduced by $\lambda(t)$. Otherwise the number of defects in the software and the defect rate remains the same. Therefore, the total number of expected occurrence of a defect in the system

is $1/p$. Hence, the predicted cumulative number of defects in the system at time t becomes

$$N(t) = \frac{1}{p}\left(\frac{\sqrt{2p}}{\delta}\right)^2 \left(\chi^2_{\alpha,\delta} + \chi^2_{\beta,\delta}\right)^2, \tag{4}$$

Moreover, if the information of defect detection process and defect correction process is available, we can model the defect detection process separately from the defect correction process. On the other hand, due to the fact that a defect can be removed only after its detection; it is more appropriate if the defect correction process to be delayed defect detection process. For simplicity we can assume for each detected defect takes the same amount of time Δ. Hence, given the defect rate $\lambda(t)$, the intensity of defect correction is given by

$$\lambda_c(t) = \begin{cases} 0 & t < \Delta \\ \lambda(t-\Delta) & t \geq \Delta \end{cases}$$

Hence, the predicted cumulative number of corrected defects in the system at time t is given by

$$N_c(t) = \frac{1}{p} N(t-\Delta) \tag{5}$$

With these improvements, we can now describe and predict the software defect behavior in its life cycle.

2.5 Experimental Results

We tested our proposed method on real software datasets (DS I and DS II) that were taken from SAP development systems. These datasets contains monthly software defects that were recorded for a period of 60 and 59 months as shown in Table 2.2 and Table 2.3 respectively.

In all the experiments, we use a probability of type I error $\alpha = 0.01$. The value of γ was set to 1-α. Figure 2.4 and Figure 2.5 depict the cumulative number of defects versus defect time (month) during a software life cycle.

Month (t_i)	$N(t_i)$	Month (t_i)	$N(t_i)$
1	17	31	2,217
2	39	32	2,430
3	53	33	2,586
4	87	34	3,884
5	106	35	4,099
6	140	36	4,385
7	165	37	5,104
8	286	38	8,074
9	359	39	10,120
10	412	40	12,618
11	461	41	16,715
12	555	42	21,606
13	654	43	24,592
14	747	44	27,789
15	836	45	29,739
16	926	46	30,843
17	989	47	32,011
18	1,049	48	32,599
19	1,103	49	33,010
20	1,152	50	33,707
21	1,182	51	34,103
22	1,213	52	34,426
23	1,225	53	34,736
24	1,266	54	34,903
25	1,306	55	35,110
26	1,331	56	35,261
27	1,363	57	35,440
28	1,443	58	35,614
29	1,495	59	35,763
30	1,737	60	35,876

Table 2.2: Software defect data (DS I).

Figure 2.6 and Figure 2.7 displays Laplace factor vs. Defect Time, and it clearly illustrates after the 45^{th} month for DS I and after the 15^{th} month for DS I, the Laplace trend starts to decrease.

2.5.1 Qualitative evaluation of the proposed method

In this subsection, we present simulation results where the Bayesian prediction method [4] and the POC curve algorithm are applied to the software failure dataset (DS I) and also to the truncated

Month (t_i)	$N(t_i)$	Month (t_i)	$N(t_i)$
1	3	31	34,909
2	5	32	35,055
3	19	33	35,129
4	30	34	35,198
5	74	35	35,269
6	115	36	35,339
7	543	37	35,421
8	1,379	38	35,556
9	3,372	39	35,617
10	7,272	40	35,664
11	11,434	41	35,707
12	14,291	42	35,789
13	17,429	43	35,852
14	18,806	44	35,922
15	21,625	45	35,951
16	24,201	46	35,974
17	26,096	47	36,004
18	27,221	48	36,032
19	28,395	49	36,047
20	29,105	50	36,292
21	29,553	51	36,374
22	30,133	52	36,448
23	30,712	53	36,469
24	32,111	54	36,510
25	32,894	55	36,521
26	33,476	56	36,574
27	34,209	57	36,606
28	34,499	58	36,617
29	34,658	59	36,631
30	34,781		

Table 2.3: Software defect data (DS II).

software failure data (DS II). Laplace trend starts to decrease, meaning that software reliability starts to grow. Based on our extensive experimentation, we decided to start applying the model from this point. Figure 2.8 through Figure 2.11 show the prediction results of the proposed POC curve in comparison with the Bayesian approaches for both datasets DS I and DS II. These results indicate that our method outperforms the Bayesian techniques used for comparison. Moreover, the proposed method is simple and easy to implement.

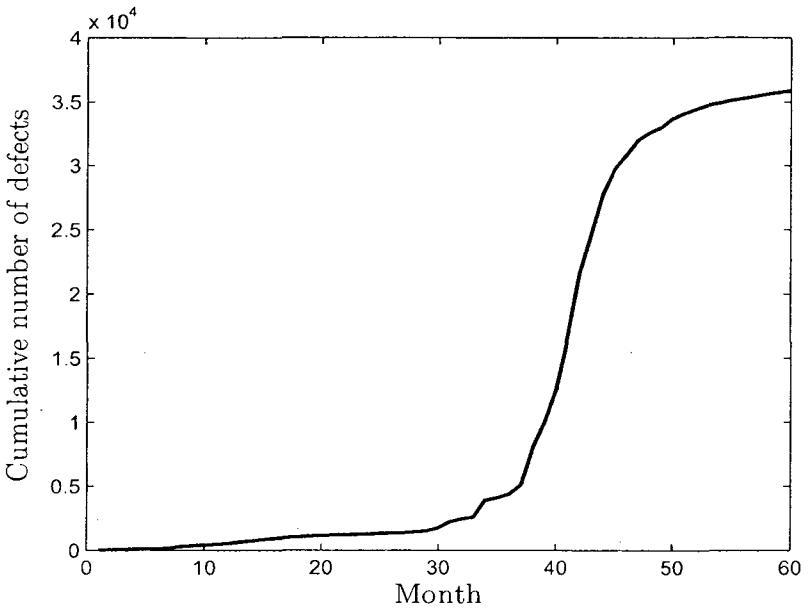

Figure 2.4: Cumulative Number of Defects vs. Defect Time (DS I)

2.5.2 Quantitative evaluation of the proposed method

Denote by $N_o(t)$ and $N_p(t)$ the observed and the predictive cumulative number of failures respectively. To quantify the better performance of the proposed predictive method in comparison with the Bayesian approaches, we computed three goodness-of-fit measures: the skill score, the Nash-Sutcliffe model efficiency coefficient, and the relative error between the observed $T_o \times 2$ data matrix

$$\mathcal{D}_o = \{(t, N_o(t)) : t = 1, \ldots T_o\},$$

and the predicted $T_p \times 2$ data matrix

$$\mathcal{D}_p = \{(t, N_p(t)) : t = 1, \ldots T_p\}.$$

Note that the size of observed data matrix $\mathcal{D}_o$ may not be equal to the size of the predicted data

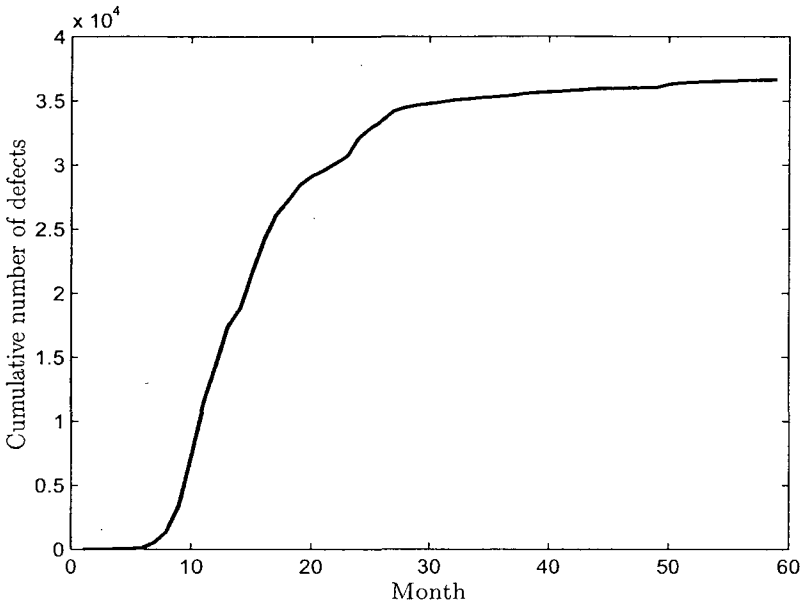

Figure 2.5: Cumulative Number of Defects vs. Defect Time (DS II)

matrix $\mathcal{D}_p$, and hence an intersection step is necessary to pair up the observed data to the predicted data. This intersection function is setup to pair up the first column in the observed data matrix and the first column in the predicted data matrix. Data values are located in the second column of both matrices. More precisely, we create a subset of matched data $\mathcal{D}_m = \{t, N_o(t), N_o(t) : t = 1, \ldots T_m\}$ that would be used to compute the following goodness-of-fit measures:

1. **Skill Score**: it is a error statistic that is used to quantify the accuracy of prediction models, and it defined as follows

$$SS = 1 - \frac{\text{RMSE}}{\sigma_{N_o}},$$

where RMSE is the root mean square error between the observed and the predicted data, and σ_{N_o} is the sample standard deviation of the observed data.

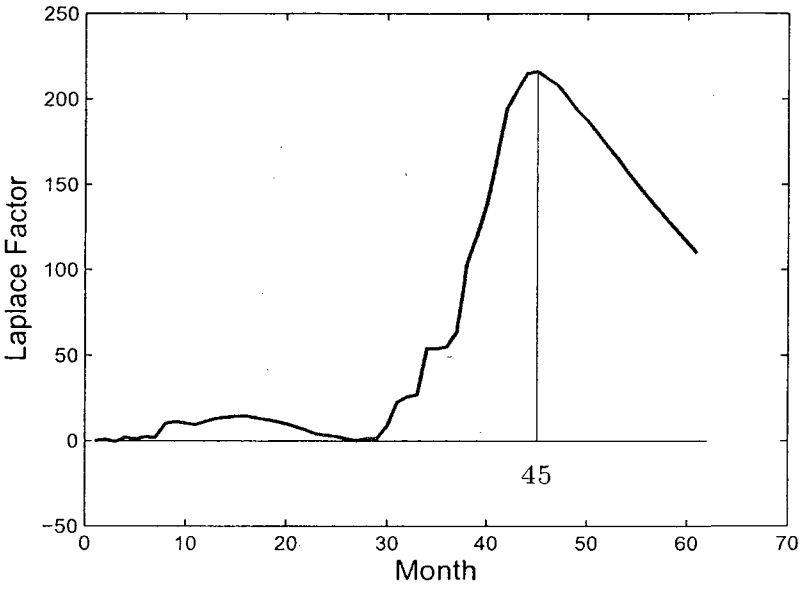

Figure 2.6: Laplace Factor vs. Defect Time (DS I).

$$SS = 1 - \frac{\sqrt{\frac{1}{T_m}\sum_{t=1}^{T_m}\left(N_o(t) - N_p(t)\right)^2}}{\sqrt{\frac{1}{T_m - 1}\sum_{t=1}^{T_m}\left(N_o(t) - \overline{N}_o\right)^2}},$$

The model prediction is better, when the value of the skill score SS is closer to one. When SS is less than zero, the model predictions are poor and the model errors are greater than observed data variability.

2. **Nash-Sutcliffe model efficiency coefficient**: is an indicator of the model's ability to predict about the 1:1 line between the observed and the predicted data, and it is defined as follows

$$E = 1 - \frac{\sum_{t=1}^{T_m}\left(N_o(t) - N_p(t)\right)^2}{\sum_{t=1}^{T_m}\left(N_o(t) - \overline{N}_o\right)^2}.$$

The Nash-Sutcliffe model efficiency coefficient is a statistic similar to the skill score in that the closer to one the better the model prediction. A value of $E = 1$ indicates that the model

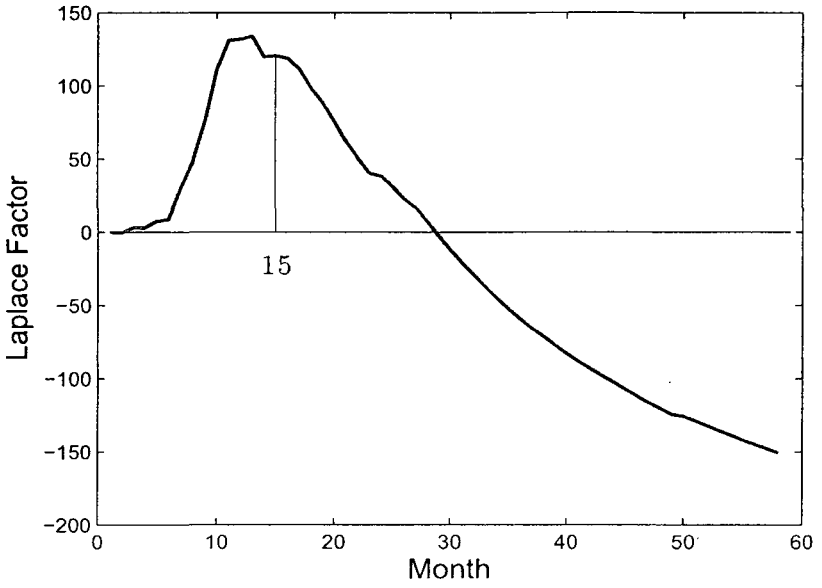

Figure 2.7: Laplace Factor vs. Defect Time (DS II).

prediction is perfect, and if the value of E is equal to or less than zero, then the model prediction is considered poor.

3. **Relative error**: it measures how close a model is estimated with respect to the actual data. The relative error(RE) is defined as

$$RE = \frac{N_p(t) - N_o(t)}{N_o(t)}, \quad t = 1, \ldots, T_m$$

The values of the three goodness-of-fit measures for all the experiments are depicted in Figure 2.12 through Figure 2.17, which clearly show that the proposed method gives the best results indicating the consistency with the subjective comparison.

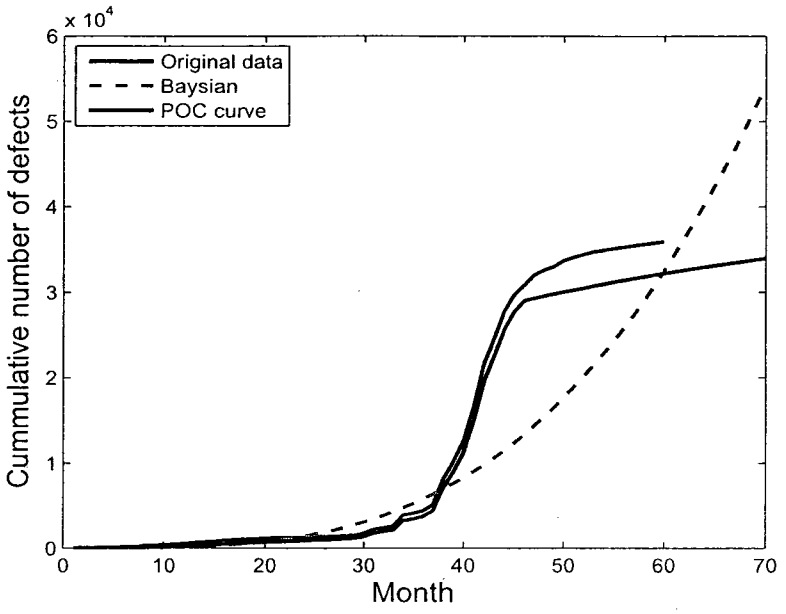

Figure 2.8: Comparison of the prediction results for known 46 months history DS I.

Skill Score	DS I	DS II
Bayesian	0.3964	0.4031
Bayesian MCMC	0.5426	0.628
OC curve	0.9377	0.7877

Table 2.4: Skill score results.

2.6 Conclusions

In this chapter, we introduced a new method for software defects prediction using operating characteristic curves and Laplace trend statistic. The core idea behind our proposed technique is to reliably predict the cumulative number of defects during the software development process. The

Nash-Sutcliffe	DS I	DS II
Bayesian	0.6295	0.6259
Bayesian MCMC	0.7872	0.8547
OC curve	0.9961	0.9527

Table 2.5: Nash-Sutcliffe score results.

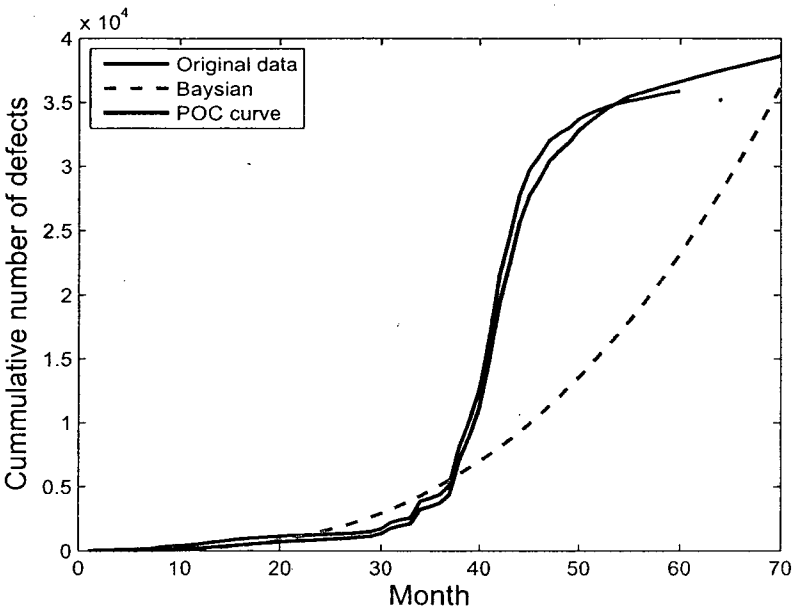

Figure 2.9: Comparison of the prediction results for known 55 months history DS I.

prediction accuracy of the proposed approach is validated on a real software failure data using several goodness-of-fit measures. The experimental results clearly show a much improved performance of the proposed approach in comparison with the Bayesian prediction methods.

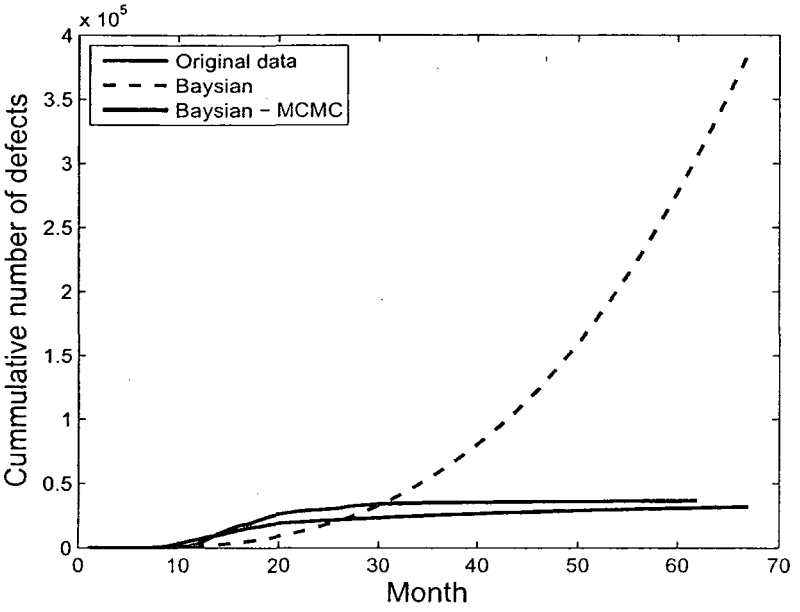

Figure 2.10: Comparison of the prediction results for known 20 months history DS II.

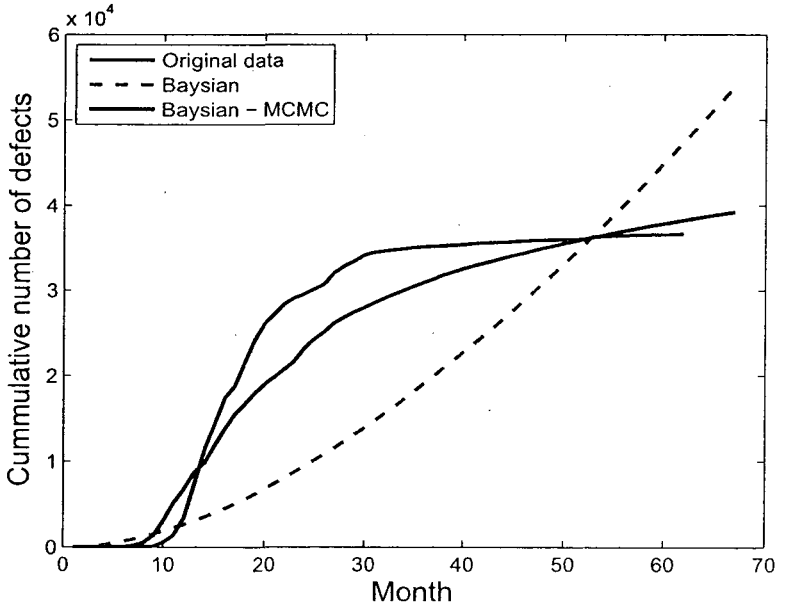

Figure 2.11: Comparison of the prediction results for known 40 months history DS II.

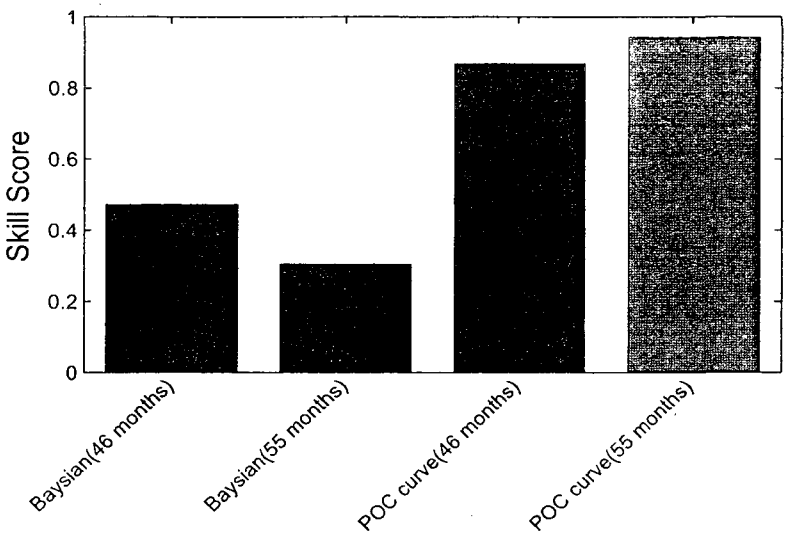

Figure 2.12: Skill score results for DS I.

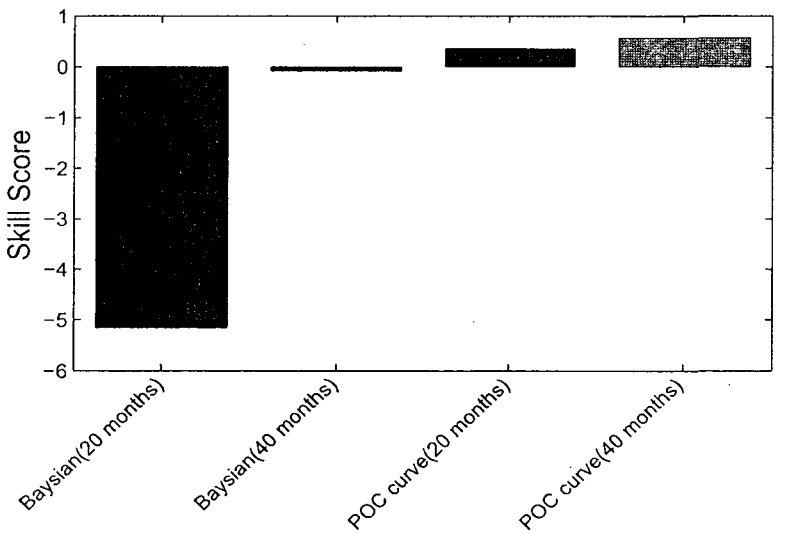

Figure 2.13: Skill score results for DS II.

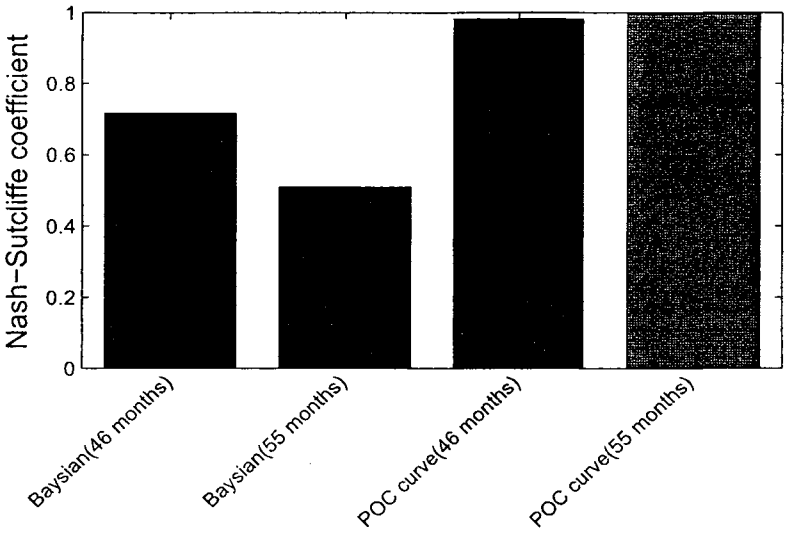

Figure 2.14: Nash-Sutcliffe model efficiency coefficient results for DS I.

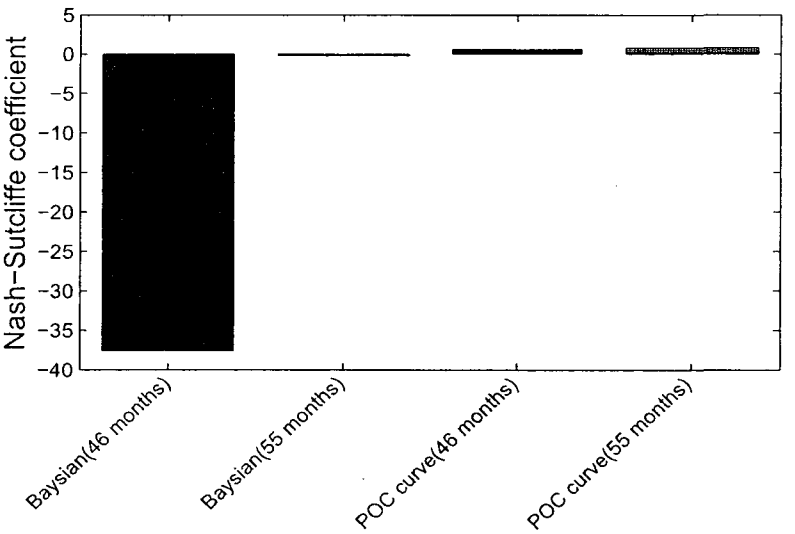

Figure 2.15: Nash-Sutcliffe model efficiency coefficient results for DS II.

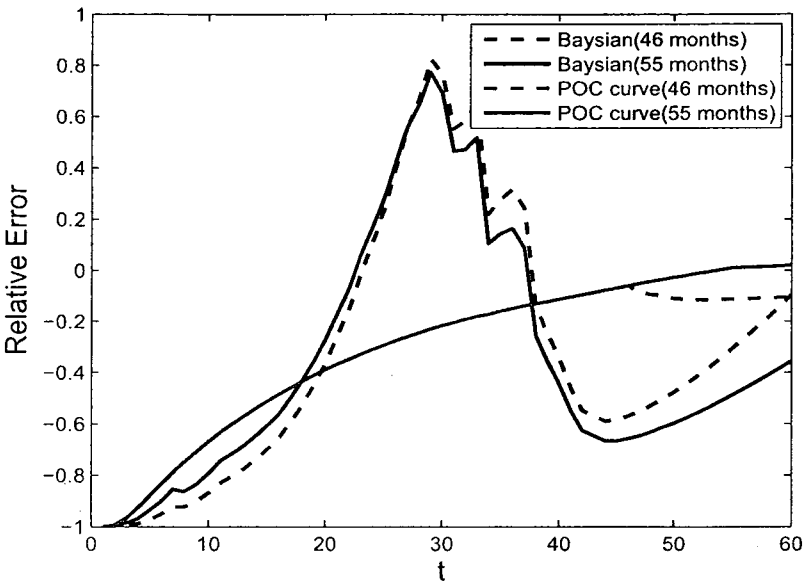

Figure 2.16: Relative error results for DS I.

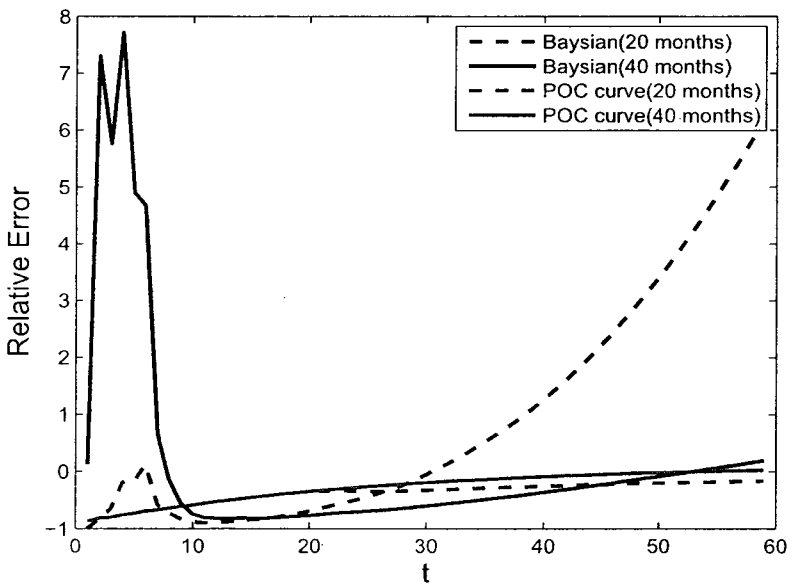

Figure 2.17: Relative error results for DS II.

CHAPTER 3

Resource Allocation using Queuing Theory

Most software quality measurement techniques are based on counting defects found in a software system, resulting in their impracticability to estimate the human-resource cost of maintenance and/or predict the reliability of a future product. In this chapter, we propose a queuing theory-based model for resource allocation in software development. The main objective is to model software management and maintenance during the system test, alpha test, and the beta test phases of a software system. The proposed model answers managerial questions related to project status and scheduling, and also provides a quantitative measure of the software.

3.1 Introduction

There is a wide spread disagreement among software engineers regarding the appropriate effort to be devoted to software testing before release. Such conflict is attributed to resource constraints and time-to-market considerations. It is well-known that more pre-release development and testing on systems can reduce future development costs and result in higher software quality. On the other hand, the pressure to deliver an operational product quickly can frequently affect the resource

allocation among development phases or within one of the phases.

Software systems are mostly developed by different teams under different environments. One of the problems in such environments is quantitative measurement of the software quality. Modern complex software systems are mostly developed incrementally with different components by different teams under different environments, making the estimation of the cost of a product a difficult task. In particular, component-based software engineering [42, 43] has drawn remarkable attention in developing cost-effective and reliable applications to meet short time-to-market requirements. Furthermore, Lyu *et al.* [44] formulated the system-testing problem as a combinatorial optimization problem with known attributes of the system components in multiple-application environment. Optimized resource allocation problems have been widely studied using dynamic programming [45], integer programming [46], non-linear optimization [33], and heuristic techniques [35, 47]. These methods are mostly series-parallel redundancy allocation problems, where either reliability is maximized or the total system testing effort/cost is minimized.

Another problem in software engineering is how to quantitatively measure the quality of the software. Most software quality measurements are based on counting the defects found in a software program. A defect is defined as an unacceptable departure of program operation caused by a software defect remaining in the software system [48]. Recently, Luo *et al.* [49] introduced a weighted Laplace test statistic for software reliability growth modelling which not only takes into account the activity in the system but also the proportion of reliability growth within the defect prediction model. These approaches are, however, developer-oriented, and do not provide useful estimates of the human-resource cost of maintenance or predictions of the reliability of a future product.

Motivated by the widely used concept of queueing theory in communication and information

processing systems, to design the system in terms of layout, capacities and control, we introduce in this chapter a resource allocation model which answers managerial questions related to project status and scheduling. Using the proposed model, managers will be more certain in making resource allocation decisions from one project to the other. This model can also be used to measure the system reliability and the quality of service provided to customers in terms of the expected response time.

The remainder of this chapter is organized as follows. In section 3.2, a problem formulation is stated followed by a brief review of queueing models. In Section 3.3, we describe the proposed resource allocation model and analyze the results. In section 3.4, we present experimental results to demonstrate the effectiveness of the proposed method in resource allocation. And finally, we conclude in section 3.5.

3.2 Problem Formulation

We assume that customers and quality assurance (QA) defect reports are in the queue, and that the developers take the defects from the queue according to the queue principle in order to fix them. The queue represents the defect report database, and the service facility for the maintenance team or the QA team. Roughly speaking, a queuing model may be defined in terms of three characteristics: the input process, the service mechanism, and the queue discipline [16, 17].

3.2.1 Queueing models

We assume that the inter-arrival time of defects has an exponential distribution with the defect rate of a monotonically decreasing function of time. To model the system's behavior over a time period

(e.g. weeks or months), we can use the average defect rate that is derived from the data collected during that period instead of the instantaneous defect rate. We assume that fixing a defect also has an exponential distribution with a constant fixing rate.

It is worth pointing out that according to the characteristics of the queue, these assumptions may vary, but for simplicity we assume a Poisson-distributed arrival time and an exponentially-distributed fixing rate.

$M/M/1$ queues

One of the classical queue models is the $M/M/1$ queue with exponential inter-arrival times with mean $1/\lambda$, exponential service times with mean $1/\mu$ and a single server [16, 17]. Defects are served on a first-come first served basis. The following condition should be satisfied

$$\rho = \frac{1/\mu}{1/\lambda} = \lambda/\mu < 1, \tag{1}$$

otherwise, the queue length will be overloaded (i.e. more defects present in the queue). The quantity ρ is the fraction of time the server (developer team) is busy which is also referred to as the utilization factor. This can be a key factor in management of development resources. The model provides the following attributes of the system:

$$p_n = (1-\rho)^n \rho^n, \; L = \frac{\lambda}{\mu - \lambda}, \text{ and } L_q = \frac{\lambda^2}{\mu(\mu - \lambda)} \tag{2}$$

where p_n is the probability of having n defect reports in the system, L is the expected number of defect reports in the system (system length), and L_q is the expected number of reports in the queue (queue length). Using the identity $L = \lambda T$, where T is the expected response time (i.e. waiting time + service time) we obtain

$$T = \frac{1}{\mu - \lambda}, \tag{3}$$

On the other hand, the time that a defect report waits in the queue is given by

$$T_q = \frac{L_q}{\lambda} = \frac{\lambda}{\mu(\mu - \lambda)}. \tag{4}$$

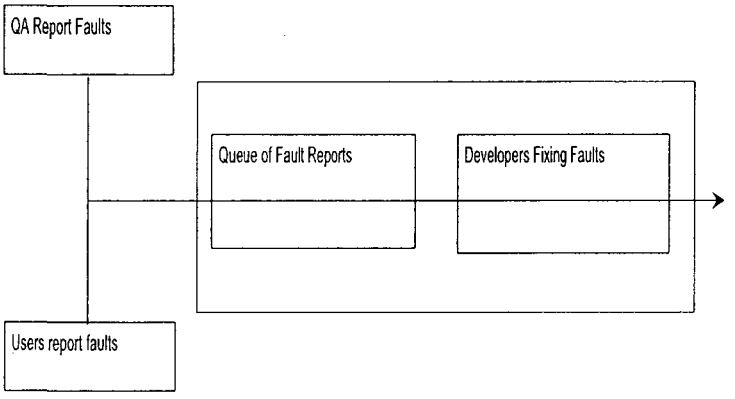

Figure 3.1: $M/M/1$ Resource allocation model.

$M/M/c$ queues

In the case of a $M/M/c$ queue, we have c parallel identical servers (developers) [16, 17], and defects are also served on a first-come first served basis. Similarly, the following condition should be satisfied

$$\rho = \frac{1/c\mu}{1/\lambda} = \frac{\lambda}{c\mu} < 1. \tag{5}$$

The $M/M/c$ model assumes the support of several developers, where c is the number of developers in a team. In most cases $M/M/c$ is more realistic than $M/M/1$ due to the fact that usually more than one developer are fixing the defects, resulting in a much better performance compared to the $M/M/1$ model. Furthermore, the $M/M/c$ model provides the following attributes of the system

$$p_n = \left(\sum_{n=0}^{c-1} \frac{r^n}{n!} + \frac{r^c}{c!(1-\rho)} \right)^{-1} \tag{6}$$

$$L = L_q + r, \quad \text{and} \quad L_q = \frac{r^c \rho}{c!(1-\rho)^2} p_0 \tag{7}$$

where the parameter r is given by $r = \lambda/\mu$.

Similar to the utilization factor in $M/M/1$, the parameter ρ is a key factor for management of resources in the system. Also, we can easily find the response time and the waiting time, which provide good measures for the quality of service.

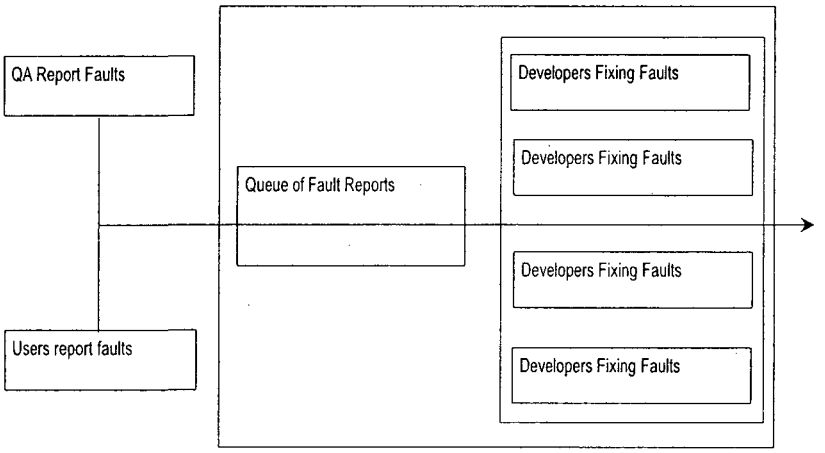

Figure 3.2: $M/M/c$ Resource allocation model.

3.3 Proposed Approach

3.3.1 Priorities of defect reports

In software systems, we usually have different types of defects which depend on their priorities. Defect arrival rates and defects fixing rates of different types of defects are not identical. In addition, they will be fixed in a different order.

Assume that we have a single server $M/M/1$ with r types of defects, the type i defects arrive

according to a Poisson distributed stream with the rate λ_i, $i = 1 \ldots r$. Note that we can easily extend it to the $M/M/c$ model [16, 17]. The service time and residual service of a type i defect is denoted by B_i and R_i respectively [16, 17]. Type I defects have the highest priority; type II defects have the second highest priority and so on.

There are basically two kinds of priorities:

1. Non-preemptive priority, in which higher priority defects may not interrupt the service time of low priority defect and will remain in the queue until the service time of the lower priority defects has been completed.

2. Preemptive-resume priority, in which interruptions are allowed. After serving the higher priority defects, serving the interrupted defect will be resumed at the point it was interrupted.

Let T_q^i be the mean waiting time of a type i defect in the queue. Then, we have

- For non-preemptive priority:

$$T_q^i = \frac{\sum_{j=1}^{r} \rho_j E(R_j)}{(1 - (\rho_1 + \ldots + \rho_i))(1 - (\rho_1 + \ldots + \rho_{i-1}))}$$

and

$$T^i = T_q^i + E(B_i)$$

where T^i is the mean service time of a type i defect in the queue, and $\rho_i = \lambda_i E(B_i)$, $i = 1, \ldots, r$.

- For preemptive-resume priority:

$$T_q^i = \frac{\sum_{j=1}^{i} \rho_j E(R_j)}{(1 - (\rho_1 + \ldots + \rho_i))(1 - (\rho_1 + \ldots + \rho_{i-1}))}$$

and

$$T^i = T_q^i + \frac{E(B_i)}{1 - (\rho_1 + \ldots + \rho_{i-1})}.$$

Note that the parameter $\rho = \sum_{i=1}^{r} \rho_i$, which shows the utilization of the resources, should be less than one.

3.3.2 Defect report rate estimation

For simplicity, we assume that the number of defect reports per month follows a straight-line regression model given by $y = \beta_0 + \beta_1 x + \varepsilon$, where β_0 and β_1 are unknown parameters referred to as intercept and slope respectively, and ε is a random error with mean zero and variance σ^2. The $\{\varepsilon\}$ are also assumed to be independent and identically distributed (i.i.d.) random variables.

Suppose that we have n pairs of observations (y_i, x_i), where $i = 1, \ldots, n$. These observations may be used to estimate the unknown parameters β_0 and β_1 of the linear regression model using the method of least square. The estimates of these unknown parameters are obtained by solving the least-square normal equations, and are given by

$$\hat{\beta}_1 = \frac{S_{xy}}{S_{xx}} \quad \text{and} \quad \hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x} \tag{8}$$

where

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i, \quad \bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i \tag{9}$$

$$S_{xx} = \sum_{i=1}^{n} (x_i - \bar{x})^2, \quad \text{and} \quad S_{xy} = \sum_{i=1}^{n} y_i (x_i - \bar{x}). \tag{10}$$

These estimates need no prior knowledge about the initial number of defects in the system when the system test starts. The average defect rate within the interval $[t_1, t_2]$ may be estimated as follows

$$\hat{\lambda} = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} (\hat{\beta}_1 t + \hat{\beta}_0) dt \tag{11}$$

The use of linear regression to estimate the defect report rate has the advantage that the estimate of the initial defect rate is not needed. However, this estimation is not robust.

Using queueing theory for resource allocation in software systems relies strictly on defect prediction. The better prediction of defect rates the more robust the model works. Unlike linear regression, expectation-maximization (EM) algorithms are more robust in defect prediction [38]. Also, the inter-arrival time in the EM model can be any type of distribution. Hence, according to the environment of the system we can use the defect prediction model introduced in [39], which robustly fits the environment.

3.3.3 Defect fixing rate estimation

Assume that the time of fixing a defect has an exponential distribution with fixing rate μ. Each developer's defect fixing rate is based on his/her capability which we refer to as productivity of a developer. This productivity can be measured either by the project manager (team leader) or from the historical data of work (e.g. amount of time required to fix defects over total number of defects fixed by a developer).

Let μ_i and p_i be the full rate of defect fixing and the proportional ratio of time that the i-th developer spends in fixing the defects respectively. The fixing rate of the i-th developer is $\mu_i p_i$. So if the team has c developers, then the fixing defect rate is given by

$$\mu = \sum_{i=1}^{c} \mu_i p_i. \tag{12}$$

Note that adding up these fixing rates is not very realistic because the summation process does not follow the linear model. For example, if two developers are working on the same error, then their joint productivity is less than adding their individual productivities when they are working

separately. To overcome this limitation, we assume that there is only one developer at a time who is serving a defect report.

3.3.4 Analyzing the utilization factor

The utilization factor ρ shows the fraction of time developers are busy fixing defects. Hence, a high value of ρ indicates that all the resources are busy fixing defects before release time. If $\rho > 1$ then the rate of defect reports are higher than the capacity of resources (developers). A project manager should not try to make the utilization factor too high ($\rho = 1$ or even close to 1 e.g. 0.98). This will make the process risky. In other words, a minor change like adding new requirements may delay the release schedule.

By applying the queue model and simulating the system, the question of how much personnel resources should be reallocated to other projects can be answered.

3.3.5 Bottleneck of personnel resource allocation

By assuming that each defect report can be served by a developer at a time, the rate of fixing defects can be improved by increasing the number of resources using Eq. (12), which helps find the bottleneck of personnel resource allocation and which can also be used to make decisions about the changing of personnel resource among teams. However, experience shows that adding new developers to ongoing projects will not necessarily reduce the development time.

3.3.6 Evaluating the quality of service

The quality of service is usually judged in terms of customer satisfaction. Therefore, there is a need to compute the team's expected response time to the customers' defect reports. As mentioned earlier, the response time is a good measure for quality of service.

Note that the overall fixing rate is assumed to be equal to μ and that the service is provided by the principe of the queue.

3.3.7 Improving the quality of service

Having a high utilization factor (e.g. $\rho = 0.99$) adds risks to the release schedule and results in a long response time for customers. Hence, we need to add more resources or extend the project schedule. Extending the project schedule does not improve the response time if ρ does not change. However, adding more resources improves the expected response time and makes the release schedule less risky and more precise. Consequently, if the system is stable (ρ is unchanged), we should make sure that there are less beta testers and less defect reports in order to avoid an overloaded defect report. We can increase the number of testers as thus increase the defect report rate λ as soon as the system becomes more stable.

3.4 Experimental Results

To test the performance of the proposed resource allocation model, we used a real software defect dataset that was taken from a SAP development system. This data contains monthly software defects that were recorded in the period of 60 months as shown Figure 3.3 and Figure 4.5.

The defects and fixes per month are chosen due to the fact that the variances of them in a day

are much bigger than the variances in a month. According to our data the average defect rate and the standard deviation of the sample are equal to $\lambda = 597.93$ and 1074.2 respectively. Also the average fixing rate is $\mu = 600.83$ with a standard deviation of 1053.

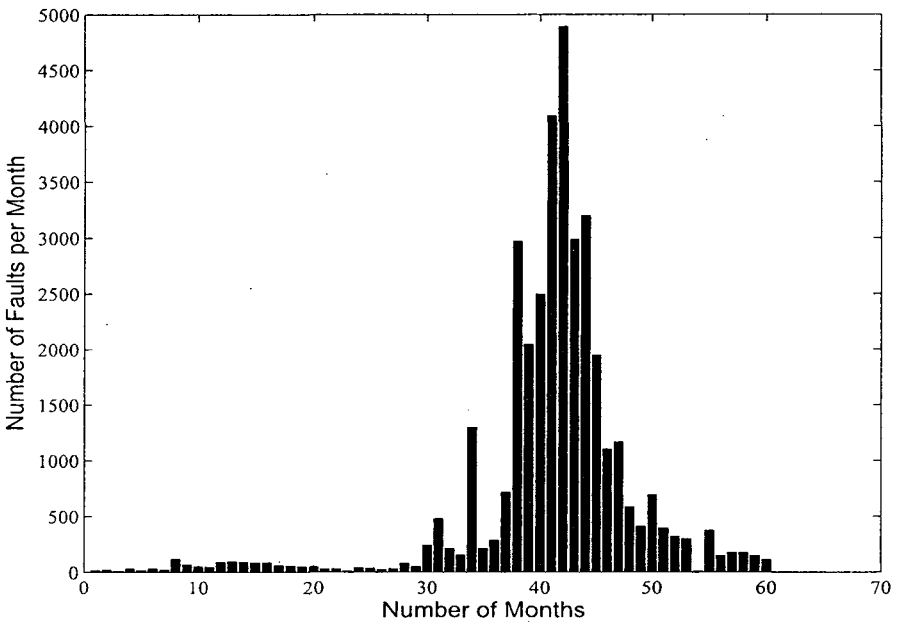

Figure 3.3: Number of defects per week.

As can be seen in Figure 3.3, the number of defects is volatile. Also note that it will not be possible to fix more defects per period if we take into consideration the engineer capabilities of the developers.

3.4.1 Analysis

The average defect rate and the standard deviation are 597.93 and 1074.2, and the fixing defect fixing rate is 600.83 with a standard deviation of 1053, in a period of 60 months. So in this case $\lambda = 597.93$ and $\mu = 600.83$. Hence the fraction of time that resources are busy (utilization factor)

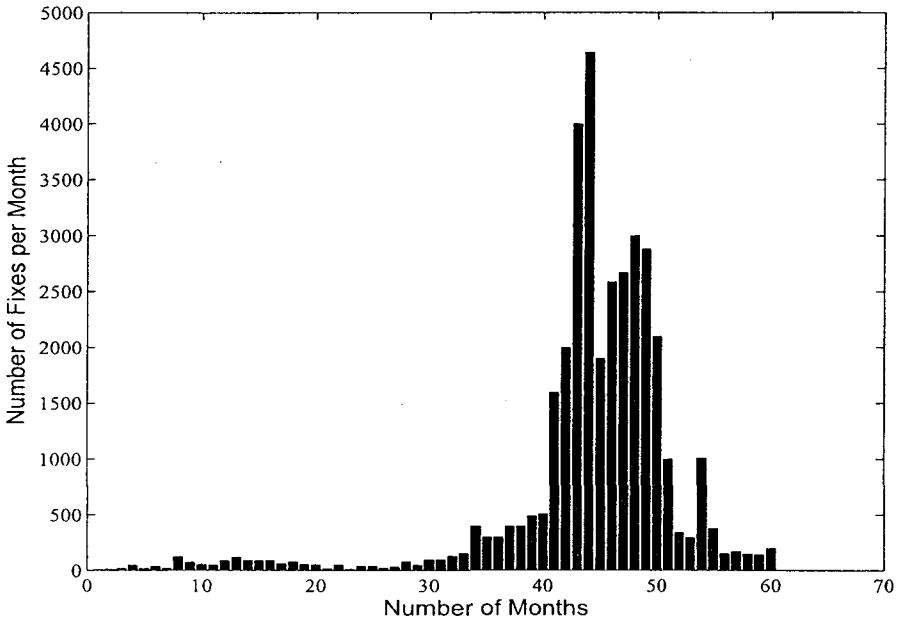

Figure 3.4: Number of fixes per week.

is $\rho \approx 0.99$ which shows that the manager used 99% of software team resources. Thus it is risky, that is only a minor change like adding new requirements will delay the release schedule. Note that in our case the release date is at the end of the 60-th month. From Figure 3.3 and Figure 4.5, we can see that the number of defect reports after the 60-th month is approximately zero. For example, if the release time is in the 60-th month, then 4.06% of our resources will be free as illustrated in Figure 3.5, and therefore we can assign them to other projects. Further, Figure 3.5 shows the amount of personnel resources that can be reallocated to other projects if the release time is after the 60-th month. This is mainly due to the fact that the defect report rate is decreasing, whereas the fixing rate remains approximately at the same level.

Suppose that we have 12 teams of developers, and according to our data each one can fix $\mu/c = 50.071$ defects per month. Note that in our case we cannot define the fixing rate of each

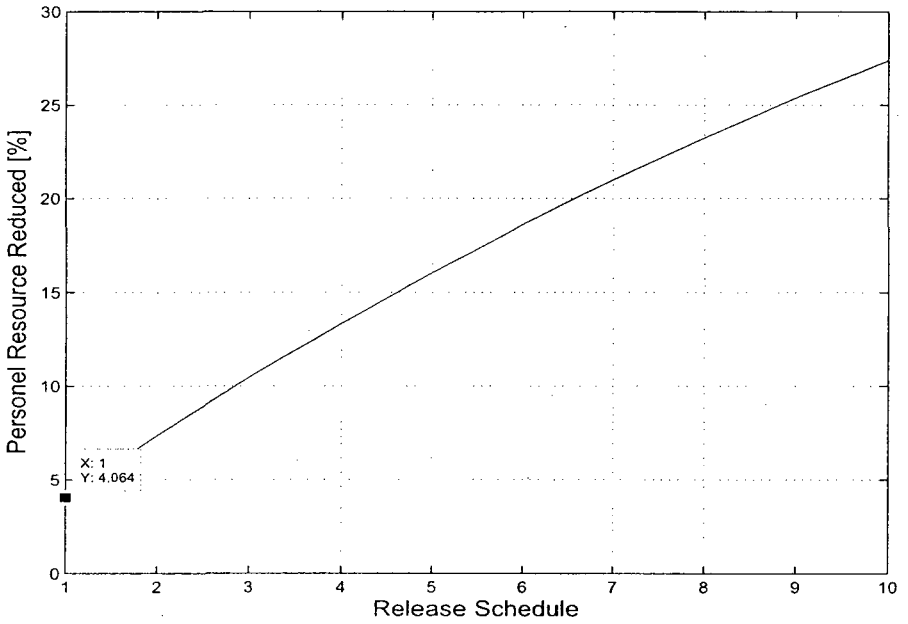

Figure 3.5: Release schedule and reduced resources.

developer or team separately but if we had enough data about the work of each developer or team we would find its fixing rate μ_i and the condition will be

$$\frac{\lambda}{\sum_{i=1}^{c} \mu_i}. \tag{13}$$

Selecting the appropriate model is usually based on the way the developers are assigned and work on the defect report. According to our assumption that each defect report is getting served by one developer at a time, the $M/M/c$ model is more realistic because developers are working on different defects at the same time. Note that the expected response time has not improved whereas the waiting time has, simply because the mean serving time for each defect has increased. Figure 3.6 shows that the number of defects in the queue is decreasing, whereas the number of defects in the system is increasing. And in Figure 3.7 we can see how of the response time and the mean waiting time are changing.

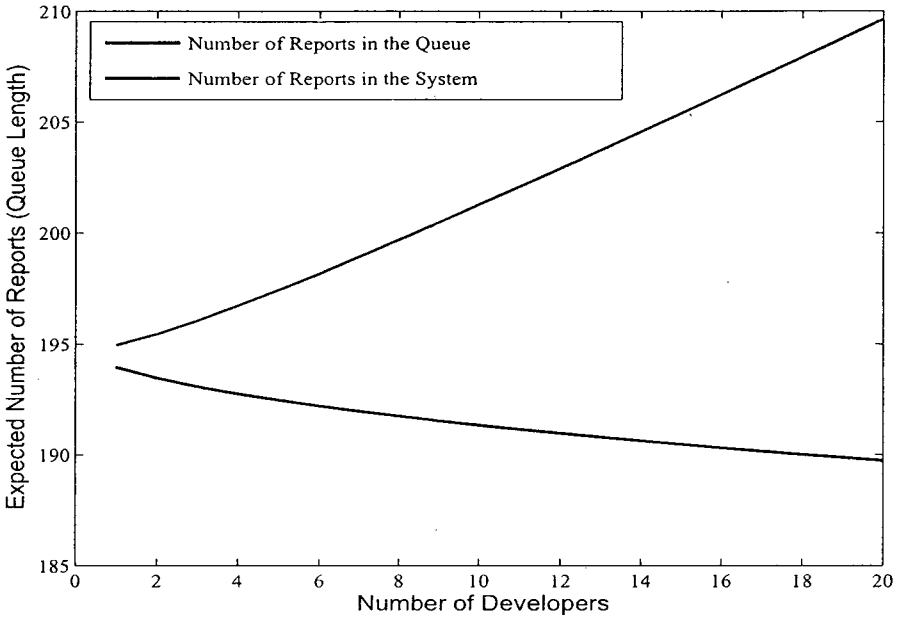

Figure 3.6: Mean queue and mean system lengths versus number of developers working on the system.

Note that number of developers in Figure 3.6 and Figure 3.7 is equal the number of servers with overall fixing rate of μ, meaning that we are not adding more developers.

3.5 Conclusions

The proposed resource allocation methodology in this chapter helps estimate the need for new developers and resources for future projects. Applying queuing theory to model software management and maintenance helps verify the progress of the testing phase and estimate its cost. Also, decisions can be made about changes in the employees early rather than letting the product miss the schedule deadline. The selection of a queuing model is organization-based. In other words we need to analyze an organization and find which model fits the best. For example if an organization

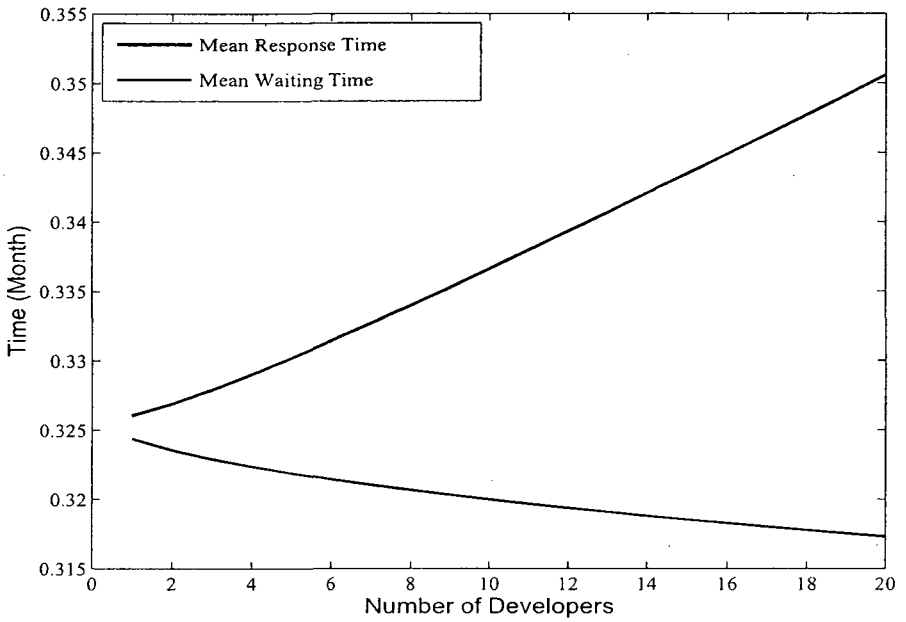

Figure 3.7: Mean response and mean waiting times versus number of developers working on the system.

considers developers as one team, then the $M/M/1$ model would fit the organization. Also note that we need to have a good estimation of defects in order to make better decisions. If we find the defect report rate, the fixing rate, and also if we decide which model is suitable, then we can use a simulation program to analyze our queue.

CHAPTER 4

Optimal Release and Maintenance Policy

In this chapter, we propose a novel stochastic discrete model to describe the cost behavior of the operation, under the assumptions of difference between the software testing environment and the operational environment, imperfect debugging and non-instantaneous fault removal. Then, we estimate the optimal time, which minimizes the relevant cost criterion, via artificial neural networks. Further, we present some fault correction models in software development which are based on general fault detection models, followed by introducing our cost function, which incorporates both fault-correction and detection models. Also, we provide a detailed analysis of the software release and maintenance decision. This procedure is simple and very useful in practical applications. In the experimental results, we demonstrate how to find numerically the joint optimal testing and maintenance policy, combined with the testing period and the planned maintenance limit.

4.1 Introduction

There is a wide spread disagreement among software engineers about the appropriate effort to be devoted to software testing before release and/or how much time should be spent on maintenance.

It is well-known that more pre-release development and testing on systems can reduce future development costs and therefore result in higher software quality. On the other hand, the pressure to deliver an operational product quickly can make the stakeholder more satisfied.

The determination of optimum software release time may be formulated as an optimization problem. To analyze this optimization problem, different criteria should be identified. If the requirement is a fault-free software or any other reliability goal, then the problem is to determine the minimal testing time in order to reach the reliability requirement. However, if the cost of software is considered then the optimal release time should be determined through an appropriate cost function to be used for minimizing the total expected software cost.

Given that testing all executable paths in a general program is not practically possible, it is therefore difficult to detect and remove all faults remaining in a software during the testing phase. Hence, the software failure may occur in the operational phase. It is common to provide maintenance service during the period of fixing software faults that are causing failures. In order to perform the maintenance phase, the management cost should be reduced as much as possible, but at the same time the human resources should be utilized effectively. The problem of determining the maintenance period is of paramount importance from the practical point of view. However, this problem has received less attention and only a few solutions have been proposed [42].

Software reliability growth models (SRGM) provide essential information for decision making in many software development activities, such as cost analysis, resource allocation in testing and release decision time. The aim of software reliability growth modelling (SRGM) is to explain the behavior of software testing process caused by faults. Most existing SRGMs only model fault detection processes with unrealistic assumptions such as perfect debugging and immediate fault correction. In this report, we use an improved SRGM by taking into account more realistic

assumptions in the model.

In this section, we focus on the optimal software testing and maintenance policy motivated by K. Rinsaka *et al.* [42]. We develop a discrete-time stochastic model in discrete operation condition, where the software testing environment and the operational environment is characterized by a environmental factor [43]. In addition, we present a systematic study of fault detection and correction processes. In our model, we consider the fault correction time to estimate the optimal software release and maintenance time which takes into account the environmental factor and imperfect fault removal. More precisely, the total expected cost is formulated via the discrete type of software reliability models [43, 44] based on the difference between operational environments, imperfect fault removal, and fault correction process to remove a fault. To the best of our knowledge, there is no available method to find the optimal release and maintenance time of a software product by taking into account these assumptions.

The rest of this chapter is organized as follows. In Section 4.2, we describe our assumptions and problem formulation. In Section 4.3, the proposed method is introduced. Section 4.4 evaluates the proposed method and presents experimental results using a real SAP dataset. Finally, we conclude in Section 4.5.

4.2 Problem Formulation

4.2.1 Assumptions

In the sequel, we make the following assumptions:

- If a failure occurs, then the fault causing the failure cannot be removed immediately.
- Fault correction is imperfect, and when a fault is discovered then it is perfectly repaired

with a certain probability p.

- Failure rates during testing and operational phases are different according to the environmental factor.
- The times to detect each software fault are independent.

4.2.2 Defect detection and correction model

In nonhomogeneous poisson process (NHPP)-based fault detection models, the cumulative number of failures $N(t)$ defines a NHPP with rate function (also called intensity function) $\lambda(t)$ which is time dependant, and with a mean value function $m(t) = E(N(t))$ given by

$$m(t) = \int_0^t \lambda(x)dx. \tag{1}$$

In general, different fault detection models may be obtained by using different non-decreasing mean value functions $m(t)$. The selection of a fault detection model is based on the goodness-of-fit of the model to the underlyting software failure data.

SRGMs are widely used to assess the fault related behavior of software systems. However, since most SRGMs embed certain restrictions or assumptions, selecting an appropriate model based on the characteristics of the software projects is often challenging. In order to choose a suitable model, two approaches may be adapted. The first one is to design a guideline, which could suggest fitting models for software projects. The second one is to select a model with the highest confidence after various assessments. The decision-making processes would therefore be a huge overhead while the software projects are huge and complicated.

In order to reduce such an overhead, neural networks have been previously employed as an alternative approach that can adapt the characteristics of failure processes from the actual data set.

The results show that the neural network approach is good at identifying software failures [54].

Neural networks are composed of simple elements operating in parallel. These elements are inspired by biological nervous systems. As in nature, the network function is determined largely by the connections between elements. We can train a neural network to perform a particular function by adjusting the values of the connections (weights) between elements. Commonly neural networks are adjusted, or trained, so that a particular input leads to a specific target output. The network is adjusted, based on a comparison of the output and the target, until the network output matches the target. Typically many such input/target pairs are used, in this supervised learning, to train a network. Neural networks have been trained to perform complex functions in various fields of application including pattern recognition, identification, classification, speech, vision and control systems.

Neural networks are learning mechanisms that can approximate any non-linear continuous functions based on the given data. In general, neural networks consist of three components as follow:

1. *Neurons*: each neuron can receive signal, process the signals and finally produce an output signal. Figure 4.1 depicts a neuron, where f is the activation function that processes the input signals and produces an output of the neuron, x are the outputs of the neurons in the previous layer, and w are the weights connected to the neurons of the previous layer.

2. *Network architecture*: the most common type of neural network architecture is called feed-forward network as shown in Figure 4.2. This architecture is composed of three distinct layers: an input layer, a hidden layer, and an output layer. Note that the circles are represented as neurons and the connection of neurons across layers is called the connecting weight.

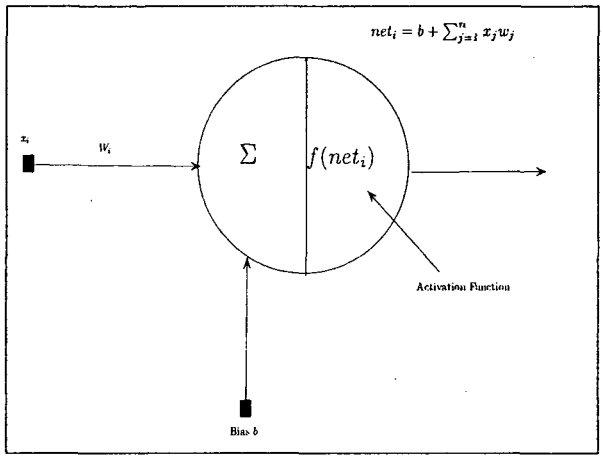

Figure 4.1: Illustration of a neuron.

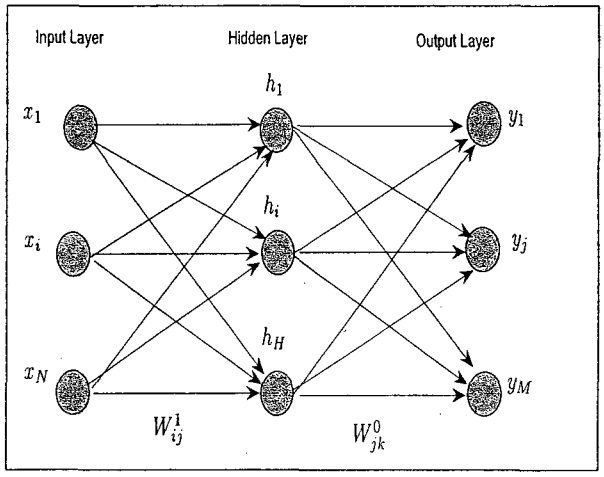

Figure 4.2: Feed-forward network.

3. *Learning algorithm*: this algorithm describes a process of adjusting the weights. During the learning processes, the weights of a network are adjusted to reduce the errors of the network outputs as compared to the standard answers. The back-propagation algorithm is the most widely employed one. In a back-propagation algorithm, the weights of the network are iteratively trained with the errors propagated back from the output layer.

The goal of using neural networks is to approximate a nonlinear function that can receive a vector $X = (x_1, \ldots, x_n)$ in $\mathbb{R}^n$ and has an output vector $Y = (y_1, \ldots, y_m)$ in $\mathbb{R}^m$. Hence, we define the network as $Y = F(X)$. The components of Y are given by

$$y_k = g\left(b_k + \sum_{j=1}^{H} w_{jk}^0 h_j\right), \quad k = 1, \ldots, M \tag{2}$$

where w_{jk}^0 is the output weight from the hidden layer node j to the output layer node k, h_j is the output of the hidden layer j, b_k is the bias of the output node k, and g is the activation function in output layers. The hidden layer values are given by

$$h_j = f\left(b_j + \sum_{i=1}^{N} w_{ij}^1 x_i\right), \quad j = 1, \ldots, H \tag{3}$$

where w_{ij}^1 is the input weight from the input layer node i to the hidden layer node j, x_i is the value at input note i, b_j is the bias of the hidden node j, and f is the activation function in the hidden layer.

Due to the fact that the neural network approximated function can be considered as a nested function such as $f(g(x))$, it can be applied to software reliability modelling since software reliability modelling is intended to build a model that explains the software failure behavior. In other words, if we derive a form of compound functions from a usual SRGM, then we can build a neural network-based model for software reliability. In this paper, we consider the logistic growth curve model [55]. The mean value function of this model is given by

$$m(t) = \frac{a}{1 + ke^{-bt}}, \tag{4}$$

where the parameters a, b, and k are positive real numbers.

We can simply find the following compound functions form of the mean value function by

replacing k with e^{-c} as follows

$$m(t) = \frac{a}{1 + ke^{-bt}} = \frac{a}{1 + e^{-(bt+c)}}. \tag{5}$$

It is worth pointing that the mean value function is the composition of the following functions:

$$g(x) = bx + c, \quad f(x) = \frac{1}{1 + e^{-x}}, \quad \text{and} \quad k(x) = ax.$$

Hence, we have

$$m(t) = k(f(g(t))) = \frac{a}{1 + e^{-(bt+c)}}. \tag{6}$$

Therefore, the mean value function is composed of g, f, and k. Now we derive the compound functions from the viewpoint of neural networks by using the basic feed-forward network as shown in Figure 4.2. However, the network has only one neuron in each layer as illustrated in Figure 4.3.

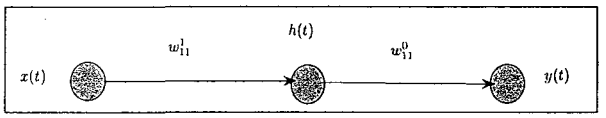

Figure 4.3: Feed-forward neural network with a single neuron at each layer.

The hidden layer input and output are given by

$$h_{in}(t) = w_{11}^1 t + b_1 \tag{7}$$

and

$$h(t) = f(h_{in}(t)) \tag{8}$$

respectively, where $f(x)$ is the activation function in the hidden layer. The input and output of the output layer are $y_{in}(t) = w_{11}^0 h(t) + b_0$ and $y(t) = g(y_{in}(t))$ respectively.

Now, if we assume the activation functions as

$$f(x) = \frac{1}{1 + e^{-x}} \tag{9}$$

$$g(x) = x \tag{10}$$

and remove the bias in the output layer. Then, we obtain

$$y(t) = y_{in}(t) = \frac{w_{11}^{0}}{1 + e^{-(w_{11}^{1}t + b_1)}}. \tag{11}$$

Hence, we derived the neural network via a logistic growth curve model. Note that this approach could be used for any other SRGM. To use the neural network-based models, the following steps are required:

1. Select a model which meets the assumptions of the software project.
2. Construct the neural network of selected models (defining bias and activation function which is differentiable everywhere).
3. Using the time interval data and cumulative number of faults in the system, we train the network using the back-propagation algorithm.
4. After training the network, we feed the future testing time to the network and the output is the forecasting number of faults in the future given time.

4.2.3 Defect correction models

If the information of fault detection processes and fault correction processes is available, we model fault detection process separately from the fault correction. On the other hand, due to the fact that a fault can be removed only after its detection, it is more appropriate if the fault correction process is considered to be related to the fault detection process. The fault correction process is assumed

to be a delayed fault detection process. Different models have been proposed in a variety forms for the time delay between these processes. This idea was proposed in [45], where a fault detection is modelled by a NHPP and a fault correction is assumed to have a constant delay from the fault detection process. This approach can be developed using different NHPP models or any other fault detection models like operating characteristic curves [39] for different fault detection processes. The difference between fault detection and correction is the time delay, which is the time spent to correct the detected fault. We denote this time delay by $\Delta(t)$. This delay can be modelled as deterministic or random variable, or it can be time-dependant [46]. We model the fault correction process by a mean value function $m_c(t)$ which can be derived from $m(t)$ and $\Delta(t)$.

Next we introduce some correction time models.

Constant correction time:

We assume that each detected fault takes the same amount of time to be corrected, that is $\Delta(t) = \Delta$. Hence, given the fault detection rate $\lambda(t)$, the intensity function of fault correction is given by

$$\lambda_c(t) = \begin{cases} \lambda(t) & t < \Delta \\ \lambda(t-\Delta) & t \geq \Delta \end{cases} \tag{12}$$

Hence, the mean value function for the fault correction process is given by

$$m_c(t) = m(t-\Delta), \quad t \geq \Delta. \tag{13}$$

Time dependant correction time:

The time lag between fault detection and correction can be time-dependant [33]. When the detected faults become increasingly difficult to correct, the time needed to correct them becomes longer. In

this case, we assume a time delay as follows

$$\Delta(t) = \frac{\log(1 + c\,t - c/b)}{b}, \quad c < b \tag{14}$$

Therefore, the correction rate and mean value functions are as follows

$$\lambda_c(t) = \lambda(t - \Delta(t)) \tag{15}$$

$$m_c(t) = \int_{\Delta}^{t} \lambda(x - \Delta(x))dx. \tag{16}$$

Exponentially distributed correction time:

Usually a deterministic correction time is not realistic in practice. The software fault correction is closely related to humans, who are considered as an uncertainty factor. In addition, the detected faults are different and their appearance sequence is random in system testing. Hence, it is more realistic if we model the correction time with a random variable.

The correction time is known to approximately follow an exponential distribution [33]. As a result, we assume that the correction time for each detected fault is a random variable with exponential distribution $\Delta(t) \sim \exp(\mu)$. Therefore, the correction rate and mean value functions are given by

$$\lambda_c(t) = E[\lambda(t - \Delta(t))] = \int_0^{\infty} \lambda(t - x)\mu \exp^{-\mu x}\, dx \tag{17}$$

$$m_c(t) = \int_0^{t} \lambda_c(x)dx. \tag{18}$$

4.2.4 Software cost

In any SRGM, the modelling is not the ultimate goal. The extracted information from the analysis could help management make decisions regarding a software development project. Our main focus is on decisions of when to release the software and when to stop maintaining the software after release. The cost of developing the software, as the most important aspect in software business, is used to make such decisions.

4.3 Proposed Method

4.3.1 Total expected software cost

We formulate the total expected software cost, which can occur in both testing and operational phases, as an optimization problem. In the operational phase, we consider two cost factors, namely the maintenance cost due to the software failure and the operational cost to keep the maintenance team. It should be noted that the operational environment after the release may differ from the debugging environment in the testing phase. We introduce an environmental factor r such that $r > 0$, which represents the relative severity in the operational environment, and assume that the times in testing phase and maintenance phase have a proportional relationship. Okamura *et al.* [43] introduced this approach to model the operational phase of the software. Also, when a fault is detected and the correction process is performed on the fault we assume it is repaired with probability p, in which case the failure rate is reduced by $\lambda(t)$. Otherwise, the number of faults in the software and the failure rate remains the same. Hence, the total number of expected occurrence of a fault is $1/p$.

Let T and M be the total expected cost function testing time and the maintenance time, respectively. Then, we define our total expected cost of a software system as follows

$$\begin{aligned} C(T,M) = & \frac{1}{p} c_1 m_c(T) + \frac{1}{p} c_2 r\Big[m_c(M) - m_c(T)\Big] \\ & + c_3 r\Big[m_c(\infty) - r\big(m_c(M) - m_c(T)\big) - m_c(T)\Big] \\ & + c_4 T + c_5 M \end{aligned}$$

where the parameters are defined as:

- $c_1 > 0$ is the cost for dealing with a fault in the testing phase.
- $c_2 > 0$ is the cost for dealing with a fault in the maintenance phase.
- $c_3 > 0$ is the cost for dealing with a fault after the maintenance phase.
- $c_4 > 0$ is the cost per unit of time in the testing phase.
- c_5 is the cost per unit of time in the maintenance phase.

Note that, the values of these parameters may be obtained from historical data of similar projects.

4.3.2 Optimal release and maintenance times

Now the objective is to find the joint optimal testing period T^* and the optimal planned maintenance limit M^* which minimize the total expected software cost $C(T, M)$.

We assume that the software has a limited life cycle $L > 0$, which a constant assumed to be known *a priori* and larger than the testing period T. In other words, the life cycle is measured from the point of release time T. Hence, in our proposed cost function we use $T + L$ instead of ∞.

Since the cost function $C(T, M)$ is a discrete function of T and M, and due to a limited life cycle we can therefore easily find the minimum points by searching in the range of this function. So, we construct a matrix, where the elements of each of its columns show the maintenance times

and the elements of each of its rows show the release times. We the find the minimum cost and consequently the optimal times by searching only through the column, the row and the diagonals crossing each matrix element. If the life cycle is large, then we can use generic search from an arbitrary starting point in the constructed matrix.

4.4 Experimental Results

4.4.1 Software dataset

In this section we present a statistical analysis using a real software defect dataset that was taken from a SAP development system. This dataset contains monthly software defects that were recorded for a period of 60 months as shown in Figure 4.4. The number of fixed faults is also shown in Figure 4.5. However, there is no tag indicating a certain fault is corrected or any other information and we have only grouped data, which correspond to the number of faults per month.

4.4.2 Parameter estimation

To use different fault detection and correction models, we need to estimate the parameters of the model based on our observed data using maximum likelihood estimation [38, 47]. In Table 1, the results of the estimation with the corresponding goodness-of-fit measure for all models are listed. As can be seen in this table, the model composed of neural network fault detection and fault correction process with constant correction time, provides the best fit for our dataset.

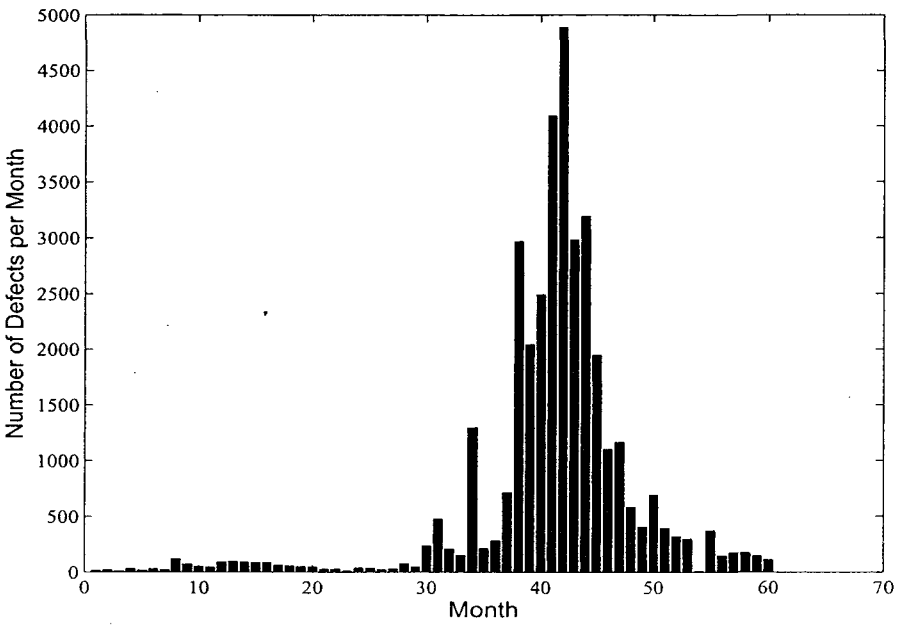

Figure 4.4: Number of defects per month in the period of 60 month.

4.4.3 Model selection

As mentioned earlier, we need to select a appropriate software fault prediction model. In our experiments, we have used different SRGMs to predict our software failure data. Qualitatively, we compared the neural network model with other SRGMs as shown in Figure 4.6.

Denote by $N_o(t)$ and $N_p(t)$ the observed and the predictive cumulative number of failures respectively. To compare the methods quantitatively, we use the so-called skill-score goodness of fit measure between the observed $T_o \times 2$ data matrix

$$\mathcal{D}_o = \{(t, N_o(t)) : t = 1, \ldots T_o\},$$

and the predicted $T_p \times 2$ data matrix

$$\mathcal{D}_p = \{(t, N_p(t)) : t = 1, \ldots T_p\}.$$

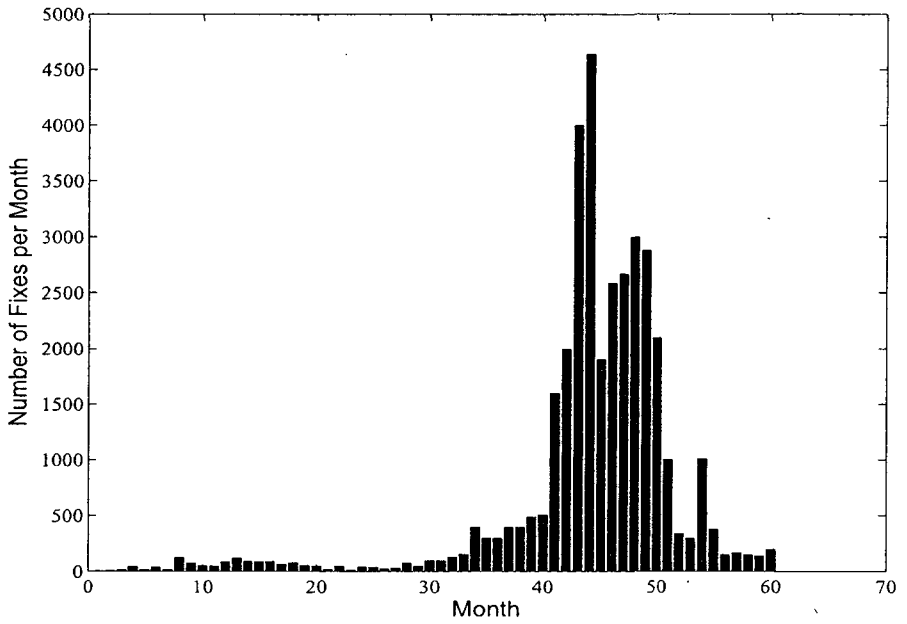

Figure 4.5: Number of fixes per month.

Note that the size of observed data matrix $\mathcal{D}_o$ may not be equal to the size of the predicted data matrix $\mathcal{D}_p$, and hence an intersection step is necessary to pair up the observed data to the predicted data. This intersection function is setup to pair up the first column in the observed data matrix and the first column in the predicted data matrix. Data values are located in the second column of both matrices. More precisely, we create a subset of matched data $\mathcal{D}_m = 1 - \{t, N_o(t), N_o(t) : t = 1, \ldots T_m\}$ that would be used to compute the skill score, which is defined as follows:

$$SS = \frac{\sqrt{\frac{1}{T_m}\sum_{t=1}^{T_m}\left(N_o(t) - N_p(t)\right)^2}}{\sqrt{\frac{1}{T_m - 1}\sum_{t=1}^{T_m}\left(N_o(t) - \overline{N}_o\right)^2}}. \tag{19}$$

This goodness-of-fit measure interprets model predictability using residual error and observed variability in your data. A skill score of 1 means a perfect fit. A skill score equal to or less than zero

Table 4.1: Summary of paired model estimates.

Model	Estimates
Constant Correction	$\hat{a} = 36608.68$
	$\hat{b} = 0.27$
	$\hat{c} = -80.90$
$\Delta(t) = \Delta$	$\hat{\Delta} = 339.23$
Time Dependant	$\hat{a} = 212.89$
	$\hat{b} = 0.02$
	$\hat{c} = -0.09$
$\Delta(t) = \frac{\log(1+c't-c'/b')}{b'}$	$\hat{b'} = 1.16$
	$\hat{c'} = 1.31$
Exponential	$\hat{a} = 1559.08$
	$\hat{b} = 0.95$
	$\hat{c} = 32.97$
$\Delta(t) \sim \exp(\mu)$	$\hat{\mu} = 0.12$

means that the model error is larger than the variability in the data, and should not be used any further without re-evaluating the model design. As can seen in Figure 4.7, the neural network model gives a better goodness-of-fit than the other detection models. Hence, we use the neural network approach to predict the faults. Moreover, as shown in Figure 4.8 the constant correction time fits better to our data. Therefore, we use constant correction time with a neural network fault detection model.

4.4.4 Optimal release and maintenance

Now we calculate numerically the optimal testing period T^* and the optimal planned maintenance limit $M^\star$ based on our software fault detection and correction data. We assume that the known parameters in the software reliability models are estimated using maximum likelihood estimation. By searching through the cost matrix $C(T, M)$, will find the minimum values for T and M that optimize the cost function.

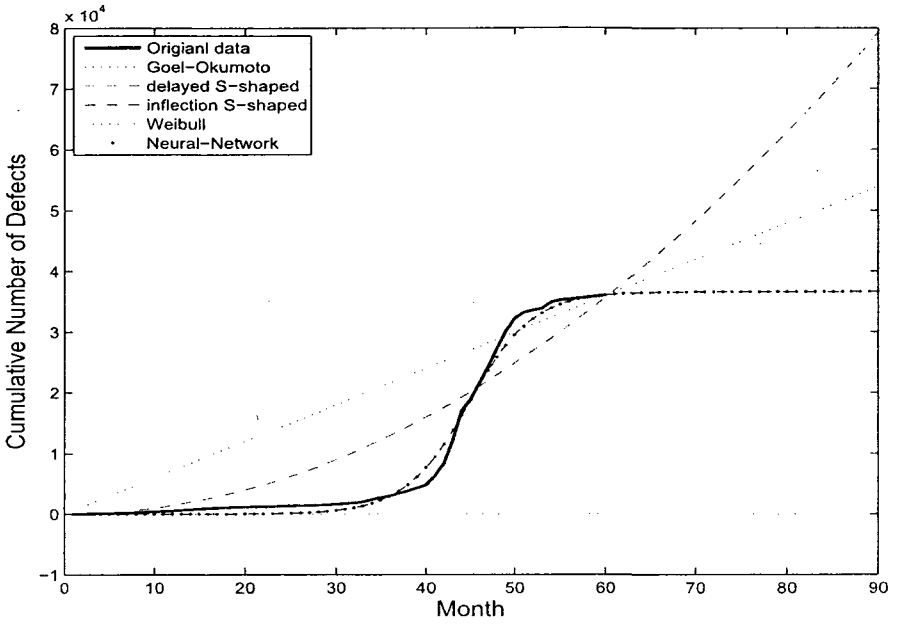

Figure 4.6: Cumulative Number of Defects vs. Defect Time.

As mentioned earlier, we use neural networks for the fault detection process with different delay functions. With our dataset, these models are applied to fit against the real data. Then, the skill score is calculated numerically. The results of the corresponding goodness-of-fit measure for all models are shown in Table 4.1. The 3D plot of the software cost matrix for constant correction time is shown in Figure 4.9, displaying the behavior of the total expected software cost based on different release and maintenance policies. The optimal release/maintenance policy is obtained by finding the minimum value in the matrix as depicted in Figure 4.9.

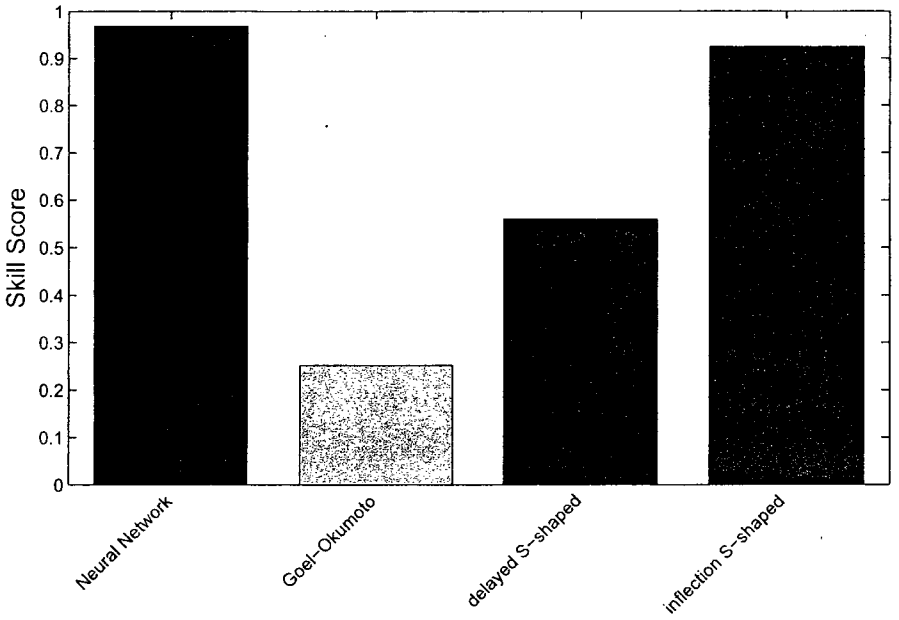

Figure 4.7: Skill Score.

4.5 Conclusions

In this chapter, the problem of optimal software release time and maintenance period based on fault correction process has been investigated. By considering a diversity of software projects, different SRGM can be used in the same way the delay could be constant, time dependant or random. Different models have also been compared using our data set. In the numerical experiments with real software fault-detection time data, we showed that the predictive performance of the optimal software release time using neural networks performs better than using the existing parametric SRGMs. Also, in our model we used more realistic assumptions such as a different severity in different environments and an imperfect defect removal.

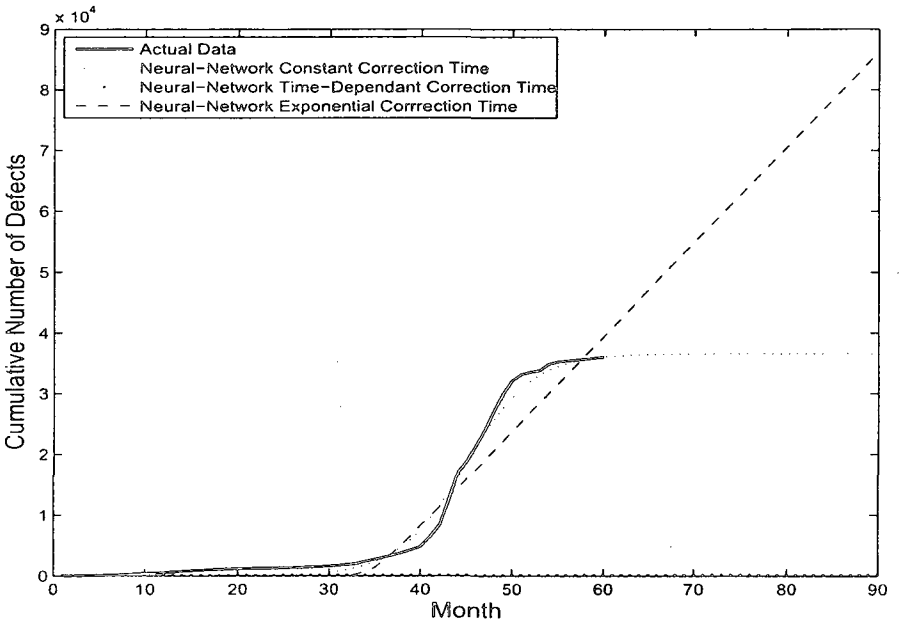

Figure 4.8: Cumulative Number of Defects vs. Defect Time.

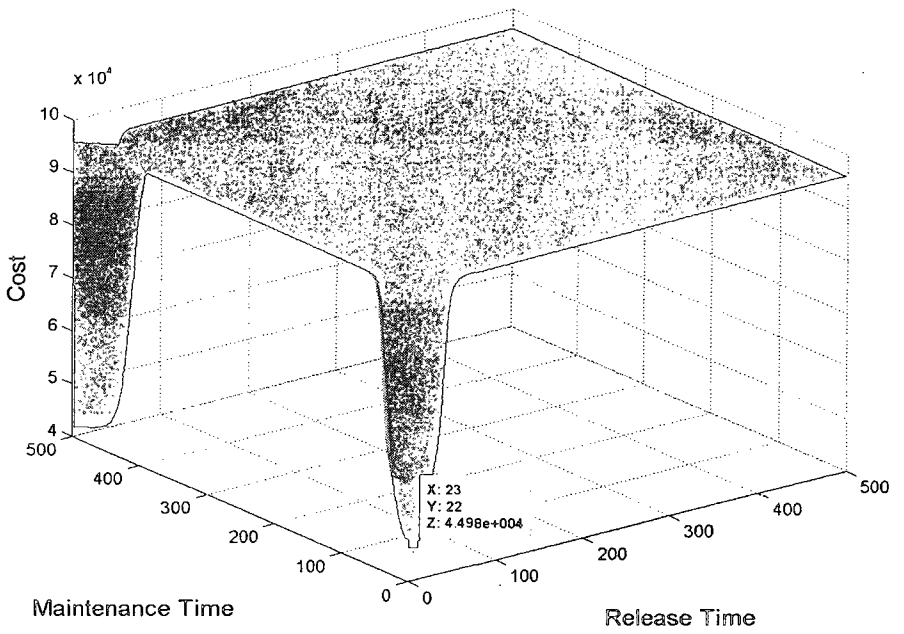

Figure 4.9: Behavior of Total Expected Cost and Optimal Times.

CHAPTER 5

Conclusions and Future Work

A defect prediction solution provides a guideline to the sources of defects that might be caused due to programmers inability, failure in requirements collection or design mistakes. Thus, a defect prediction model with source identification can give important ideas regarding the erroneous bottlenecks in the software development cycle. Especially, efficiency focused software development units can benefit using defect cause information. They can take necessary precautions in a proactive manner. In other words, a defect focused prediction solution can also help to change the development methods. Such a solution or systematic approach can affect in a positive manner to produce less defected software.

An important aspect of a defect prediction solution is that such a solution becomes necessary when there is a trade-off between to deliver earlier and to deliver with fewer defects. In today's software development industry, all companies and software development houses are in a severe competition that minimizing development time decreases the overall project cost [59, 60]. On the other hand, less development and testing time also increases the defect density ratio in the final product. So, with this fact the executive management of the software company should require a quantitative indicator to find the correct point in this balance. Therefore a defect prediction solution

may provide the required quantitative metric to make a decision on the product delivery. The senior management of the software development company would be able to decide launching the product if the defect density level is below a certain threshold.

This thesis has presented statistical tools to predict the cumulative number of software defects, optimized resource allocation in software production life cycle, and optimal release time and maintenance policy that can have a huge positive impact on making the software quality assurance easier. We have demonstrated the performance of the proposed algorithms on a variety of software defect datasets, and we compared our proposed techniques with existing methods.

In the next Section, the contributions made in each of the previous chapters and the concluding results drawn from the associated research work are presented. Suggestions for future research directions related to this thesis are provided in Section 5.2.

5.1 Contributions of the Thesis

5.1.1 Predictive operating characteristic curves for software defects

We introduced a software defect prediction model based on the concept of operating characteristic curve. The idea is to use Operating Characteristic (OC) curves in statistical quality control and a geometric approach to construct an efficient, fast, and accurate prediction method to estimate the number of software failures at anytime during the software development process. Our model is getting the information from past and present failure data to be more effective. In the experimental results, we demonstrate the effectiveness and the improved performance of the proposed method in comparison with the Bayesian prediction approaches.

5.1.2 Software development resource allocation using queuing theory

We proposed a resource allocation model using queuing theory. The goal is to model software management and maintenance during the system test, alpha test, and the beta test phases of a software system. The proposed model answers managerial questions related to project status and scheduling, and also provides a quantitative measure of the software. Using the proposed model, managers will be more certain in making resource allocation decisions from one project to the other. This model can also be used to measure the system reliability and the quality of service provided to customers in terms of the expected response time.

5.1.3 Software optimal testing and maintenance policy

We introduce a new method to define the optimal policy in software release time and maintenance. We focused on the optimal software testing and maintenance policy motivated by the approach proposed in [42]. We also developed a discrete-time stochastic model in discrete operation condition, where the software testing environment and the operational environment is characterized by a environmental factor. In addition, we present a systematic study of defect detection and correction processes. In our model, we consider the defect correction time to estimate the optimal software release and maintenance time which takes into account the environmental factor and imperfect defect removal. More precisely, the total expected cost is formulated via the discrete type of software reliability models based on the difference between operational environments, imperfect defect removal, and defect correction process.

5.2 Future Research Directions

Several interesting research directions motivated by this thesis are discussed next. In addition to designing robust statistical models for software defect prediction, we intend to accomplish the following projects in the near future:

5.2.1 Optimal release time using game theory

Our proposed model to find the optimal policies on software production does not take into account rivalry. In other words, we hope to continue our future work by investigating a model which permits competition between rival producers. This can potentially be done by using our model in the framework of a two-person non-zero sum game of timing. Through the series of preliminary results, it is shown that an optimal release policy exists as a Nash equilibrium point in the space of mixed strategies. Although in this model we used a classical neural network, an effort to improve the forecasting ability will be needed for future work. For instance, if the other environmental data for software testing, e.g. structural factors such as the numbers of codes, functions and modules, testing effort or cost such can be observed, the neural networks may carry more realistic information processing by taking into account these factors. In addition, our model does not take into account rivalry. In other words, we hope to explore a new model that would permit competition between rival producers. This can potentially be done by using a two-person game of timing.

5.2.2 Machine learning approaches

As a broad subfield of artificial intelligence (AI), machine learning is concerned with the design and development of algorithms and techniques that allow computers to "learn". As regards machines, one might say, very broadly, that a machine learns whenever it changes its structure, program, or data (based on its inputs or in response to external information) in such a manner that its expected future performance improves. The major focus of Machine learning research is to extract information from data automatically by computational and statistical methods, hence, machine learning is closely related to data mining and statistics but also theoretical computer science.

Machine learning usually refers to the changes in systems that perform tasks associated with AI. Such tasks involve recognition, diagnosis, planning, prediction, etc. The "changes" might be either enhancements to already performing systems or synthesis of new systems. One might ask "Why should machines have to learn? Why not design machines to perform as desired in the first place?" There are several reasons why machine learning is important. Some of these are:

- Some tasks cannot be defined well except by example, that is, we might be able to specify input/output pairs but not a concise relationship between inputs and desired outputs. We would like machines to be able to adjust internal structure to produce correct outputs for a large number of sample inputs and thus suitably constrain their input/output function to approximate the relationship implicit in the examples.
- It is possible that hidden among large piles of data are important relationships.
- Human designers often produce machines that do not work as well as desired in the environments in which they are used. In fact, certain characteristics of the working environment might not be completely known at design time. Machine learning methods can be used for on-the-job improvement of existing machines designs.

- The amount of knowledge available about certain tasks might be too large for explicit encoding by humans. Machines that learn this knowledge gradually might be able to capture more of it that humans would want to write down.
- Environments change over time. Machines can adopt to a changing environment would reduce the need for constant redesign.
- New knowledge about tasks is constantly being discovered by humans. Continuing redesign of AI systems to conform to new knowledge is impractical, but machine learning methods might be able to track much of it.

There are two major settings in which we wish to learn a function f: *supervised* and *unsupervised.* In supervised learning, we know the values of f for the m samples in the training set $\mathbb{S}$. We assume that if we can find a hypothesis h that closely agrees with f for the members of $\mathbb{S}$, then this hypothesis will be a good guess for f, especially if $\mathbb{S}$ is large. Curve fitting is a simple example of supervised learning of a function. In unsupervised learning, we simply have a training set of vectors without function values of them. The problem in this case, typically, is to partition the training set into subsets $\mathbb{S}_1, \ldots, \mathbb{S}_k$ in some appropriate way.

Our future efforts will be focused on evaluating various machine learning models to develop robust prediction approaches. The performance of each prediction method will be evaluated regarding their precision, recall, robustness and sensitivity using confusion matrices and simulations. A model's precision is defined as the ratio of the number of modules correctly predicted as defective, or true positive (t_p), to the total number of modules predicted as defective in the set ($t_p + f_p$). A model's recall is defined as the ratio of the number of modules predicted correctly as defective (t_p) to the total number of defective modules in the set ($t_p + f_n$). To perform well, a model must achieve both high precision and high recall.

List of References

[1] J.D. Musa, "A theory of software reliability and its application," *IEEE Transactions on Software Engineering*, vol. 1, no. 1, pp. 312-327, 1975.

[2] A.L. Goel and K. Okumoto, "Time-dependent error detection rate models for software reliability and other performance measures," *IEEE Transactions on Reliability*, vol. 28, no. 3, pp. 206-211, 1979.

[3] J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill Book Company, 1987.

[4] J.W. Yu, G.L. Tian, and M.L. Tang, "Predictive analyses for nonhomogeneous Poisson processes with power law using Bayesian approach," *Computational Statistics & Data Analysis*, 2007.

[5] C.G. Bai, "Bayesian network based software reliability prediction with an operational profile," *Journal of Systems and Software*, vol. 77, no. 2, pp. 103-112, 2004.

[6] X. Zhang and H. Pham, "Software field failure rate prediction before software deployment," *Journal of Systems and Software*, vol. 79, pp. 291-300, 2006.

References

[7] N.E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol.5, no. 5, pp. 675-689, 1999.

[8] S. Yamada, M. Ohba, and S. Osaki, "S-shaped reliability growth modeling for software error detection," *IEEE Transactions on Reliability*, vol. 32, no. 5, pp. 475-485, 1983.

[9] A.L. Goel, "Software reliability models: assumptions, limitations and applicability," *IEEE Transactions on Software Engineering*, vol. 11, no. 12, pp. 1411-1423, 1985.

[10] M.R. Bastos Martini, K. Kanoun, and J. Moreira de Souza, "Software-reliability evaluation of the TROPICO-R switching system," *IEEE Transactions on Reliability*, vol. 39, no. 3, pp. 369-379, 1990.

[11] K. Kanoun and J.C. Laprie, "Software reliability trend analysis from theoretical to practical considerations," *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 525-532, 1992.

[12] M.R. Lyu, *Handbook of Software Reliability Engineering*, IEEE Computer Society Press and McGraw-Hill, 1996.

[13] D.C. Montgomery, *Introduction to Statistical Quality Control*, John Wiley & Sons, 2005.

[14] C. Robert, *Bayesian Choice*, 2nd Edition, Springer Verlag, NY, 2001.

[15] W.M. Bolstad, *Introduction to Bayesian Statistics*, John Wiley, 2004.

[16] R.B. Cooper, *Introduction to queuing theory*, Second edition, Elsevier North Holland Inc, 1981.

References

[17] I. Adan and J. Resing, *Queuing Theory*, http://www.cs.duke.edu/ fishhai/misc/queue.pdf, 2001.

[18] S. Yamada, T. Ichimori, and M. Nishiwaki, "Optimal allocation policies for testing-resource based on a software reliability growth model," *Mathematical and Computer Modelling*, vol. 22, pp. 295-301, 1995.

[19] J.H. Lo and C.Y. Huang, "An integration of fault detection and correction processes in software reliability analysis," *Journal of Systems and Software*, vol. 79, no. 9, pp. 1312-1323, 2006.

[20] B.W. Boehm, C. Abts, A.W. Brown, S. Chulani, B.K. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece, *Software Cost Estimation with Cocomo II*, Prentice Hall PTR, 2000.

[21] X. Cai, M.R. Lyu, K-F. Wong, and R. Ko, "Component-based software engineering: Technologies, Development frameworks, and quality assurance schemes," *Proc. Asia-Pacific Software Engineering Conferenece*, pp.372-379, 2000.

[22] W. Kozacznski and G. Booch, "Component-based software engineering," *IEEE Software*, vol. 155, pp. 34-36, Sep./Oct. 1998.

[23] M.R. Lyu, S. Rangarajan, and A.P.A. van Moorsel, "Optimal allocation of test resources for software reliabilitygrowth modeling in software development," *IEEE Transactions on Reliability*, vol. 51, no. 2, pp. 183-192, 2002.

[24] C.Y. Huang and M.R. Lyu, "Optimal release time for software systems considering cost, testing-effort, and test efficiency," *IEEE Trans. on Reliability*, vol. 54, pp. 583-591, 2005.

References

[25] A. Zaryabi, A. Ben Hamza, T. Bergander, and N. Mahe, "Software Fault Prediction Modeling," *SAP Research Conference*, Palo alto, CA, USA, 2008.

[26] A. Zaryabi, T. Bergander, A. Ben Hamza, and N. Mahe, "Queing-Thoretic Approach to Software Resource Allocation," *Proc. 17th International Conference on Software Engineering and Data Engineering*, LA, California, USA, 2008.

[27] A. Zaryabi, A. Ben Hamza, T. Bergander, and N. Mahe, "Optimal software release and maintenance policy via neural networks," *to be submitted*, 2009.

[28] O. Gauodin, "Optimal properties of the Laplace trend test for software-reliability models," *IEEE Transactions on Reliability*, vol. 20, no. 9, pp. 740-747, 1992.

[29] H.E. Ascher and C.K.Hansen, "Spurious exponentiality observed when incorrectly fitting a distribution to nonstationary data," *IEEE Transactions on Reliability*, vol. 47, no. 4, pp. 451-45, 1998.

[30] W.R. Gilks, S. Richardson, and D. Spiegelhalter, *Markov chain Monte Carlo in Practice*, Chapman & Hall/CRC, 1995.

[31] Y. Nakagava and S. Myazaki, "Surrogate constraints algorithm for reliability optimization problems with two constraints," *IEEE Transactions on Reliability*, vol. 30, no. 2, pp. 175-180, 1981.

[32] K.B. Misra and U. Sharma, "An efficient algorithm to solve integer programming problems arising in system reliability design," *IEEE Transactions on Reliability*, vol. 40, no. 1, pp. 81-91, 1991.

[33] F.A. Tillman, C-L. Kwang, and W. Kuo, "Determining component reliability and redundancy for optimum system reliability," *IEEE Transactions on Reliability*, vol. 26, pp. 162-165, 1977.

[34] L. Painton and J. Campbell, "Genetic algorithms in optimization of system reliability," *IEEE Transactions on Reliability*, vol. 44, no. 2, pp. 172-178, 1995.

[35] D.W. Coit and A.E. Smith, "Reliability optimization of series-parallel system using a genetic algorithm," *IEEE Transactions on Reliability*, vol. 45, no. 2, pp. 254-260, 1996.

[36] B. Luong and D-B. Liu, "Resource allocation model in software development," *Proc. IEEE Annual Reliability and Maintainability Symposium*, Philadelphia, USA, 2001.

[37] M.A. Marsan, S. Donatelli, and F. Neri, "GSPN models of multiserver multiqueue systems," *Proc. International Workshop on Petri Nets and Performance Models*, pp. 19-13, 1989.

[38] H. Okamura, Y. Watanabe, and T. Dohi, "An iterative scheme for maximum likelihood estimation in software reliability modeling," *Proc. International Symposium on Software Reliability Engineering*, pp. 246-256, 2003.

[39] T. Bergander, Y. Luo, and A. Ben Hamza, "Software defects prediction using operating characteristic curves," *Proc. IEEE International Conference on Information Reuse and Integration*, Las Vegas, USA, 2007.

[40] Y. Luo, T. Bergander, and A. Ben Hamza, "Anisotropic Laplace trend to enhance software reliability growth modelling," *Proc. International Conference on Modelling and Simulation*, Montréal, Canada, May 2006.

[41] Y. Luo, T. Bergander, and A. Ben Hamza, "Software reliability growth modelling using a

weighted Laplace test statistic," *Proc. IEEE International Computer Software and Applications Conference*, Beijing, China, July 2007.

[42] K. Rinsaka, and D. Tadashi, "Discrete optimal testing/maintenance policy in a software develepement project," *Asia Pacific Management Review*, pp. 225-232, 2005.

[43] O. Hiroyuki, D. Tadashi, and O. Shunji, "A Reliability Assessment Method for Software Products in Operational Phase-Proposal of an Accelerated Life Testing Model," *Trans. Institute of Electronics, Information and Communication Engineers.*, vol. 83-A, no. 3, pp. 294-301, 2000.

[44] T. Kitaoka, S. Yamada, and S. Osaki, "A discrete non-homogeneous error detection rate model for software reliability," *IECE Trans.*, vol. E69, no.8, pp. 859865, 1986.

[45] N.F. Schneidewind, "Analysis of error processes in computer software," *Proc. International Conference on Reliable Software*, IEEE Computer Society Press: Los Alamitos, CA, pp. 337346, 1975.

[46] M. Xie, Q.P. Hu, Y.P. Wu, and S.H. Ng, "A study of the modeling and analysis of software fault-detection and fault-correction processes," *Qual. Reliab. Engng. Int.*, vol. 23, pp. 459-470, 2007.

[47] I.J. Myung, "Tutorial on maximum likelihood estimation," *Journal of Mathematical Psychology*, vol. 47, pp. 90-100, 2003.

[48] Q.P. Hu, M. Xie, S.H. Ng, and G. Levitin, "Robust recurrent neural network modeling for software fault detection and correction prediction," *Reliability Engineering & System Safety*, vol. 92, no. 3, pp. 332-340, 2007.

References

[49] Y.S. Su, C.Y. Huang, Y.S. Chen, and J.X. Chen, "An artificial neural network-based approach to software reliability assessment," *Proc. IEEE TENCON Conference*, pp. 1-6, 2005.

[50] M. Kimura, T. Toyota, and S. Yamada, "Economic analysis of software release problems with warranty cost and reliability requirement," *Reliability Engineering & System Safety*, vol. 66, pp. 49-55, 1999.

[51] G. Levitin and M. Xie, "Performance distribution of a fault-tolerant system in the presence of failure correlation," *IIE Transactions*, vol. 38, no. 6, pp. 499-509, 2006.

[52] M.C.K. Yang and A. Chao, "Reliability-estimation and stopping-rules for software testing, based on repeated appearances of bugs," *IEEE Trans. on Reliability*, vol. 44, pp. 315-321, 1995.

[53] P.K. Kapur, P.C. Jha, and A.K. Bardhan, "Optimal allocation of testing resource for a modular software," *AsiaPacific Journal of Operational Research*, vol. 21, pp. 333-354, 2004.

[54] T.M. Khoshgoftaar, R.M. Szabo, and P.J. Guasti, "Exploring the behavior of neural network software quality models," *Software Engineering Journal*, vol. 10, no. 3, pp. 89-96, 1995.

[55] M. Xie, *Software Reliability Modeling*, World Scientific Publishing, 1991.

[56] W.R. Gilks, S. Richardson, and D. Spiegelhalter, *Markov chain Monte Carlo in Practice*, Chapman & Hall/CRC, 1995.

[57] ANSI/IEEE, Standard Glossary of Software Engineering Terminology, STD-729-1991, *ANSI/IEEE*, 1991.

References

[58] B. Littlewood and L. Strigini, "Software reliability and dependability: a roadmap," *Proc. 22nd International Conference on Software Engineering*, Limerick, pp. 177-188, 2000.

[59] P.C. Pendharkar, G.H. Subramanian, and J. Rodger, "A probabilistic model for predicting software development effort," *IEEE Transactions on Software Engineering*, vol. 31, no.7, pp. 615-624, 2005.

[60] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," *Proc. ACM ICSE conference*, 2005.