SOFTWARE RELIABILITY MODELING AND ITS APPLICATIONS CONSIDERING FAULT DEPENDENCY AND ENVIRONMENTAL FACTORS

by

MENGMENG ZHU

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Industrial and Systems Engineering

Written under the direction of

Hoang Pham

And approved by

_____

_____

_____

_____

_____

New Brunswick, New Jersey

OCTOBER 2018

ABSTRACT OF THE DISSERTATION

SOFTWARE RELIABILITY MODELING AND ITS APPLICATIONS CONSIDERING FAULT DEPENDENCY AND ENVIRONMENTAL FACTORS

By MENGMENG ZHU

Dissertation Director:

Hoang Pham

The increasing dependence of our modern society on software systems has driven the development of software product to be more competitive and time-consuming. At the same time, large-scale software development is still considered as a complex, effort consuming, and expensive activity, given the influence of the transitions in software development, which are the adoption of software product lines, software development globalization, and the adoption of software ecosystems. Hence, the consequence of software failures becomes costly, and even dangerous. Therefore, in this dissertation, we have not only integrated software practitioners' opinions from a wide variety of industries, but also developed software reliability models by addressing different practical problems observed in software development practices.

We first revisit 32 environmental factors affecting software reliability in single-release software development and compare with the findings 15 years ago [27, 28]. Later, we investigate the environmental factors affecting software reliability in multi-release software development and compare the impact of environmental factors between the

development of multi-release and single-release software to provide a comprehensive analysis for software development practices.

Software faults are classified into two groups, Type I (independent) faults and Type II (dependent) faults. Two phases software debugging process are introduced according to different types of faults. Firstly, a one-phase software reliability model is proposed with the assumption that there is only Type II faults exist in the program given Type I faults have been removed in the preliminary testing phase. Later, a two-phase software reliability model is developed in consideration of Type I and Type II faults, fault dependency, and imperfect fault removal.

Given software multiple releases are commonly adopted in industry, a software reliability model for multi-release software product is proposed. The remaining faults from previous release, and the newly introduced faults, generated from the newly added features, are both considered into the model development. In addition, the detection of the new fault in the development of the next release depends on the remaining faults from previous release and the newly introduced faults from the newly added features.

Finally, given the environment factors studies in the early stage of this dissertation, the single-environmental-factor software reliability model under the Martingale framework in consideration of environmental factor, *Percentage of Reused Modules*, and the randomness caused by this environmental factor is developed. Later, we propose a generalized multiple-environmental-factors model framework incorporating multiple environmental factors and

the randomness caused by these environmental factors. We further propose two specific multiple-environmental-factors models considering two environmental factors, gamma-distributed *Percentage of Reused Modules*, and gamma-distributed or beta-distributed *Frequency of Program Specification Change*.

In sum, this dissertation firstly investigates 32 environmental factors affecting software reliability in the development of single-release and multiple-release software and further compares the findings of these two studies regarding environmental factors and development phase. Software reliability models are developed in each chapter in consideration of different problems/applications in practices, such as software fault dependency, imperfect fault removal, software multiple releases, and the impact of environmental factors on software reliability during the development process.

ACKNOWLEDGEMENT

Throughout the years of my graduate study at Rutgers University, I have received enormous help and support from advisor, mentors, colleagues, and friends. I would like to express my sincere gratitude to all of them, without whom this dissertation would not have been accomplished.

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Hoang Pham, for his guidance, encouragement, inspiration, patience, and great support during my Ph.D. study. He has set a role model of excellence as a researcher, professor, and life mentor, and led me on the track to discover the unknown in the academic world. I am so grateful to Dr. Hoang Pham for being my Ph.D. advisor and will always bear his advice in mind.

My sincere gratitude goes to Dr. Elsayed A. Elsayed, Dr. Myong K. Jeong, Dr. Honggang Wang, and Dr. Xuemei Zhang for serving on my dissertation committee and providing me sound and constructive suggestions to complete the work.

My special gratitude goes to Dr. Susan L. Albin. I am grateful for her training on the scientific presentation and research problem explanation, which was a great learning experience for me. Through her example, I started understanding what research is all about and how to explain my research to the audience in an effective way.

I truly appreciate many inspirations that I have received from the research papers and dissertations written by Dr. Hongzhou Wang, Dr. Xuemei Zhang, and Dr. Xiaolin Teng, respectively. Dr. Hongzhou Wang, who brought me into Reliability area in the class of Reliability Engineering II, with his great expertise and knowledge. Dr. Xuemei Zhang's research has provided a very solid foundation for Chapter 4 in this dissertation. One of the papers written by Dr. Xiaolin Teng has provided the great inspiration for solving the problems in Chapter 7.

My appreciation also goes to the faculty members, Ms. Cindy Ielmini, and other graduate fellows in the Department of Industrial and Systems Engineering at Rutgers.

Lastly, I would like to thank Drs. Wenke Liu, Xiaoshi Su, Nawei Sun, and Yisi Zhang, for their friendship, companion, along with the great cooking skill and healthy sarcasm.

DEDICATION

To my parents, and grandmother, for their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

1.1 Importance of Reliable Software

In today's technological world, almost everyone is directly or indirectly in contact with computer software. Computers have been rapidly expanding to a wide array of complex machinery and equipment applied in our everyday safety, security, infrastructure, transportation system, financial management, and so on. Since software product is extensively involved in various industries and service-based applications, the increasing dependence of our modern society on software-driven systems has led the development of software product to be very competitive and time-consuming [1]. Unlike hardware system, software cannot break down or wear out during its life cycle, but can fail or malfunction under certain configuration within specific condition [2]. Hence, the development, measurement and qualifying of software are challenging yet critical in such a fast-growing technological society.

In 1980, Lehman [3] summarized the *Laws of Program Evolution*. The first law, *Continuing Change*, expressed the universally observed fact that large programs are never completed. They just continue to evolve until the more cost-effective updated version to replace the systems. The second law, *Increasing Complexity*, could also be viewed as an instance of the second law of thermodynamics. As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases as well, unless the

mission is done, or maintenance is needed. The third law, *The Fundamental Law of Program Evolution*, subjects to dynamics which make the programming process, measures system attributes and collaborative projects, and self-regulating with statistically-proven trends and invariances. The fourth law, *Conservation of Organizational Stability (Invariant Work Rate)*, and the fifth law, *Conservation of familiarity*, both lead to the third law. The fourth law more focuses on the steadiness of multiloop self-stabilizing systems. A well-established organization is good at avoiding the dramatic change and particularly discontinuities in an increasing growth of an organization. Especially in the past two decades, the complexity of the task that software system performs has grown dramatically, faster than hardware due to the fast-paced high technology development [4].

A modern software product is prone to include a large number of modules, system components and Lines of Code (LOC) [4 – 5, 7 - 9]. The size of software product is no longer measured in terms of thousands of lines of code, but millions of lines of code. The latest investigation states that more than 10 Microsoft commercial software products could have more than 600 million LOC [6]. In view of such a great amount of LOC, complexity of software product, domain knowledge of programmer/tester, testing methodologies, testing coverage, and testing environment should be all carefully taken into account in software development.

Since the inception of electronic computing in the late 1940s, the development race of computer industry has led to unprecedented process [10]. Powerful, inexpensive computer workstation replaced the drafting boards of circuit and computer designers. Moreover, an

increasing number of design steps was automated. Computer and communication industries have grown into the largest, amongst the new rising industries in twenty centuries [11]. Hardware advances have allowed software programmers to create wonderful coding and develop new features and functionality [11]. However, there is an existing uneven progress between software and hardware in the computer revolution in the past few decades. Based on the latest technology review, hardware is leaving software behind. As a matter of fact, software is relied on less firm foundation, at the same time, carries a larger burden than hardware in operation. Given the current technology on manufacturing and electrical engineering, software has more potentials to allow designers to contemplate more ambitious systems in consideration of a broader multidisciplinary scope [12, 13].

The nonperformance and failures of software system are inconvenient, sometimes can lead to severe consequence especially in aerospace engineering and national defense systems. In March 2015 [14], a software glitch carried in software package of Lockheed Martin F-35 Joint Strike Fighter aircraft had made the aircraft could not correctly detect the target. The sensor on the plane cannot distinguish the difference between a singular and multiple threats. Additionally, different F-35 aircraft provides different detection information even they are aiming at the same threat, which depends on the angles they are aiming at and what their sensors have received. The delivery date had to be postponed as well because of this issue.

Software failures can also cause serious consequence in automobile. Toyota had to recall almost 2 million Prius hybrid vehicles, in order to fix a software glitch along with its engine control units (ECUs) in February 2014 [14]. A malfunction within the car's hybrid drive system caused by software glitch could, in certain circumstances, cut the system's power and cause the car to an unscheduled halt. A software glitch affecting the ECUs controlling the motor/generator and hybrid system could put extra thermal stress on certain transistors under certain condition. The same software issue recurred in July 2015, which has resulted in the recall of 625,000 Prius cars globally.

Software failures have affected healthcare system as well. Emergency services were unavailable for around six hours across seven U.S. states in April 2014 [14]. The incident had a major impact on 81 call centers, meaning about 6,000 people made 911 calls that were unable to connect in these seven states. There is a study announced by Federal Communications Commission found that the cause of service unavailable was an entirely preventable software error.

The nonperformance and failures of software are expensive. A study carried by National Institute of Standards & Technology in 2002 found that inadequate infrastructures for fixing software bugs cost the U.S economy $59.5 billion every year. What about the global cost on fixing software bugs every year? This study also estimated that more than a third of software bugs could be eliminated by improving software testing scheduling and methodology [15].

Hence, developing reliable software is a major challenge to software industry, technology industry, and other related industries, and this leads to the fact that *Software Reliability Engineering* is popular in both academia and industry.

1.2 Software Reliability Engineering

1.2.1 Trends in Software Development

Large-scale software development is very complex, effort consuming and expensive activity. Even though many innovations and improvements have been proposed on software architecting system and development approach, the large-scale software development is still largely unpredictable and error-prone [51].

Recently, Bosch-Sijtsema [51] discussed three trends in software development, which further accelerate the complexity of large-scale software development. The first trend is the increasing adoption of software product lines. A software product line consists of a software platform shared by a group of products. Each software product can select and configure components in the platform and extend the platform with desirable functionality. At the same time, the platform consists of many components with associated team. Each team takes charge of one product or several products. Software development is taken place within many teams in the organization. During the development cycle, interactions and communications between teams are much than traditional software development teams. Some research identified the adoption of software product lines allows 50 – 75% of development expenses reduction and decreases the defect density if the adoption is

successful [52, 53]. However, the adoption of software product lines also brings new level of dependency in the organization in software development.

The second trend is software development globalization. Recent years, many companies have multiple software development sites globally, or partnered with other remote companies especially located in India and China. There are many advantages in terms of software development, e.g., cycle time reduction, travel cost reduction, less communication issues about user experiences, faster response to customers [54]. At the same time, software development globalization brings challenges given the culture difference, time zone, software engineering maturity in every country, and technical skills between different countries.

The third trend is the adoption of software ecosystems. A software ecosystem is defined as a set of businesses functioning as a unit and interacting with a shared market for software and services. There are relationships amongst those units which are supported by a technological platform, operating through the exchange of information, resources, and artifacts [55, 56]. Software ecosystem takes external developer, domain experts and users, hence, the community-centric collaboration and coordination are very important, which are similar to the adoption of software product lines, as discussed in the first trend of software development. Thus, the dependencies between components will increase and the complexity of software development will increase accordingly [51].

1.2.2 Software Reliability Model

As discussed in Section 1.2.1, given large-scale software development is an increasing-complexity, effort-consuming, and expensive activity, how can we assure software quality is one of the challenging problems in industry. One of the fundamental quality characteristics is reliability. It is generally accepted that reliability is the key factor in software quality since it quantifies failures and misbehaviors of the product. As recognized in both industry and academia, reliability is an essential measurement metric for developing a robust and high-quality software product [1, 12, 13]. According to the definition given by ANSI, software reliability is defined as the probability that a software system can perform its designed function without failure during a specified time on a given set of inputs under defined environments [12].

On the other hand, the increasing complexity and shortened iteration cycle of software products bring in a decreasing average market life expectancy [15]. Thus, since 2000s, there is a great attention shift from hardware development and testing to improve software quality and reliability with the purpose of wining more market share.

Moreover, high reliability is desirable if software company plans to reduce the total cost of software product upon the economic point of view. It is undoubtedly that lower reliability software product not only results in the negative impact regarding customer satisfaction, but also brings in the additional cost occurred during the operation phase because of fixing a software fault in the operation phase costs more resource compared with in-house testing. Since the fixing cost for a software fault in the operation phase is much higher than in-

house testing phase, most of organizations try to minimize these expenditures occurred in the operation phase. That is why many high technology organizations need to release multiple versions for a software product instead of fixing software faults in the operation phase to improve product reliability and introduce new features to improve the user experience as well.

Therefore, it is necessary to develop a practical and applicable model which can capture the software failure growth trend, predict the number of failures, predict software reliability given a specific period of operation time, propose the optimal release time of new products, and schedule the delivery time for the next release based on the predetermined level of reliability. Software reliability models are applied to evaluate software reliability and capture the failure growth trend in the past few decades. There are several ways to measure software reliability. A practical and common one is model software reliability by utilizing the past failure behavior obtained from testing phase.

There are two main types of software reliability models: the deterministic and the probabilistic. The number of distinct operators and operands, the number of errors, and the number of machine instructions in the program are investigated in deterministic software reliability model. Performance measures of the deterministic type are obtained by analyzing the program texture and do not involve any random event.

Two well-known models are: Halstead’s software metric and McCabe’s cyclomatic complexity metric. Halstead’s software metric is used to estimate the number of errors in

the program [16]. McCabe's cyclomatic complexity metric is used to determine the upper bound for estimating the number of remaining software defects [17]. In sum, these models provide a quantitative way of software measurement, however, they mainly focus on analyzing program texture and have not considered any random event [18] in the models.

The probabilistic model considers the failure occurrence and the fault removal as probabilistic events. The probabilistic software reliability models can be classified into different groups as stated in references [18, 19].

(1) Error seeding
(2) Failure rate
(3) Curve fitting
(4) Reliability growth
(5) Markov structure
(6) Time series
(7) Nonhomogeneous Poisson process

In this dissertation, we firstly focus on investigating the significant environmental factors affecting software reliability during single-release and multi-release software development, correlation between environmental factors, significant environmental factors in each development phase based on the applications of statistical learning methodologies. Meanwhile, comparisons of the significant environmental factors, correlation between environmental factors, and principle components between the development of single-

release and multi-release software product are discussed. Secondly, we propose a two-phase software reliability model incorporating software dependency and imperfect fault removal process. Different types of software faults are defined. A two-phase debugging process is also proposed according to different types of software faults. Thirdly, we develop a multi-release software reliability model in consideration of the remaining faults from previous release and the newly introduced faults, generated from the new-introduced features for the development of current release. Fourthly, given the significant impact of environmental factors on software reliability, we incorporate single/multiple environmental factor(s) in software reliability models. We not only consider environmental factors in the models but also the randomness caused by these environmental factors under the Martingale framework. All software reliability models developed in this dissertation are based on nonhomogeneous Poisson process assumption. Thus, we will provide detailed discussion of nonhomogeneous Poisson process in the next section.

1.2.3 General Theory of Nonhomogeneous Poisson Process

Nonhomogeneous Poisson Process (NHPP) has been successfully applied to model software reliability in the past few decades. Let $N(t)$ denotes the cumulative number of software failures by time *t*. The counting process $\{N(t), t \geq 0\}$ is said to be a NHPP with intensity function $\lambda(t), t \geq 0$. The probability of exactly *n* failures occurring during the time interval $(0, t)$ for the NHPP is given by

$$P\{N(t) = n\} = \frac{[m(t)]^n}{n!} e^{-m(t)} \quad \text{for } n = 0, 1, 2, \ldots \tag{1.1}$$

$$m(t) = E[N(t)] = \int_0^t \lambda(s)ds \tag{1.2}$$

where $m(t)$ is the expected number of failures up to time *t*, which is also known as the mean value function.

Note that the forms of mean value function vary with different assumptions. In NHPP, the stationary assumption is relaxed compared with Poisson process. In other words, $N(t)$ is Poisson-distributed with a time-dependent failure intensity function $\lambda(t)$, while Poisson process holds the stationary assumption, $m(t) = \lambda t$.

A general NHPP model includes the following assumptions.

(1) The failure process has an independent increment.

(2) The failure rate of the process is given by

$$P\{N(t + \Delta t) - N(t) = 1\} = \lambda(t)\Delta t + o(\Delta t)$$

(3) During a small interval Δt, the probability of more than one failure is negligible, that is

$$P\{N(t + \Delta t) - N(t) \geq 2\} = o(\Delta t)$$

where $o(\Delta t)$ represents a quantity which tends to be zero for a small Δt. The instantaneous failure intensity function $\lambda(t)$ is defined as

$$\lambda(t) = \lim_{\Delta t \to 0} \frac{R(t) - R(\Delta t + t)}{\Delta t R(t)} = \frac{f(t)}{R(t)} \tag{1.3}$$

where

$$R(t) = P[N(t) = 0] = e^{-m(t)}.$$

Given $\lambda(t)$, the mean value function $m(t)$ is

$$m(t) = \int_0^t \lambda(s) ds. \tag{1.4}$$

One of the main objectives of NHPP software reliability model is to derive appropriate mean value function $m(t)$. The failure intensity function is equivalent to the derivative of mean value function, $\lambda(t) = m'(t)$. Different assumptions on the fault detection and fault removal process lead to different failure intensity function, $\lambda(t)$. Reliability and other related measurements can be obtained by solving the differential equation, $m'(t)$. The least square estimate or maximum likelihood estimate mostly will be utilized to estimate the unknown parameters.

Software reliability $R(t)$ is defined as the probability that a software failure does not occur in $(0, t)$, that is

$$R(t) = P[N(t) = 0] = e^{-m(t)} \quad (1.5)$$

In general, during time interval $(t, t + x)$, software reliability can be described as

$$R(x|t) = P[N(t + x) - N(t) = 0] = e^{-[m(t+x)-m(t)]} \quad (1.6)$$

1.3 Importance of Software Testing

The commonest and easiest way to improve software reliability is focusing on in-house testing. Myers [20] defined software testing as a process of executing a program with the intent of finding errors. There are two fundamental rules in software testing. Firstly, it is intended to detect as many faults as possible during in-house testing phase and remove the detected faults from software system. Secondly, software failure data will be collected to predict system reliability, estimate the remaining faults, and schedule the product delivery date.

Owing to the fact that software debugging, testing and verification are accounted for 50 – 70% of a software product's development cost. Indeed, software testing is always defined as a difficult, time-consuming, and expensive section in software development [21, 22]. Software debugging cost even goes higher if debugging is carried out in the operation phase. In practice, it is unlikely to release bug-free software product owning to its nature limitation. Post-deployment failures are inevitable in a complex software.

It is generally accepted that the longer time spent on software testing, the less faults that software will carry and the more reliable of software will be. However, this is not a practical approach. Exhaustive testing to execute all possible inputs unlikely happens since too many possible combinations result in little improvement on system reliability [23, 24]. Moreover, full execution tracing is usually impractical for the complex software program due to the limitation of cost and resource [21]. Furthermore, after software reaches a certain level of refinement, any further effort on removing faults will cause an exponentially increase in the total development cost but not much increase in reliability assessment [25, 26]. Thus, how to test software efficiently and meet the determined reliability is challenging task for both researchers and practitioners.

1.4 Transitions of Environmental Factors Affecting Software Reliability

In this section, we firstly revisit 32 environmental factors defined in reference [27] fifteen years ago and analyze their impact on software reliability during software development based on a current survey distributed to software development practitioners. Secondly, given the application of agile development and increasing popularity of multi-release software products in many organizations, we conduct a new study investigating the impact level of these 32 environmental factors on affecting software reliability in the development of multi-release software to provide a sound and concise guidance to software practitioners and researchers [30].

Software development process has gone through a great change during the past one and half decades. The rise of the Internet had led to a rapid growth in the demand for

international information display and email systems on the World Wide Web. Software programmers are required to handle various illustrations, maps, photographs, and other images, plus simple animations at a rate we have never seen before. The high technology has an ever-increasing impact on daily life, which drives the software release cycle becomes shorter than before, for instance, many companies have shortened their software release cycle from traditional 18 months to 3 months, in order to respond the fast-changing and competitive market [31, 32].

Moreover, as the high technology gets more involved in our everyday life, there are a wide variety of computational devices like mobile phones, tablet PCs, laptops, desktops, notebooks, and so on [33], which also brings more challenges to software developers such as application maintenance, device consistency, and dynamic version settings [34]. Customers also have more requirements on the specific design and functionality of the software product. A user-friendly interface, involved in the interaction amongst users, designers, hardware system and software systems, has been emphasized to a great extent nowadays.

Furthermore, for practitioners and researchers, the programming skills, programming language, domain knowledge, and even the programmer organization and team size are different compared with fifteen years ago. Finally, software development is distributed across multiple locations as the development of globalization [35]. Some studies [36 - 38] stated that the cross-site work takes much longer time and requires much more effort, even though the work size and complexity is similar.

Given a great number of changes stated above, thus, it is time to investigate the transition of significant environmental factors affecting software reliability, the time distribution in each phase during software development, the principle components, the significant environmental factors in each phase, and the correlations between some of the environmental factors, and compare the findings with previous studies [27, 28] for the single-release software product and multi-release software product [29, 30].

1.5 Importance of Multi-Release Software Product

As the software development moves further away from the rigid and monolithic model, the importance of software multiple releases is brought to the vanguard. It is unlikely to deliver all features that customers wanted in the single release because of the limited budget, unavailable resource, estimated risk, and constrained working schedules. Staying competitive in the market and keeping profitable for a software product are difficult with having only a single release especially when rival releases a new release carrying more attractive features and satisfying more customer requirements [39]. As a result of multiple releases planning, software organization has more competitive and overwhelming advantages to balance the competing stakeholder's demands and benefits according to the available resource [40, 41].

On the other hand, large software system continually desires to align with the changing customer requirements for the sake of market share. In order to obtain the feedback from the users earlier, figure out what customers really look for, and assign a lower software

development cost, a certain portion of increments on the requirements for multi-release product is essential for the growth of an organization [42 - 44]. Thus, software organization needs to modify the parts of the existing modules to extend the current functionality, usability, and understandability by adding new features and correcting the problems from previous release [45, 46].

Additionally, agile software development is getting more attention in recent years. Agile is an iterative and team-based approach, which emphasizes the rapid delivery of an application in complete functional components [203]. The wide adoption of agile methodology also promotes software multiple releases.

Hence, software multiple releases are critical to keep software company stay competitive in the market. Only a few researchers proposed multi-release software reliability models [47 - 50], however, most reliability models only optimized software cost model to obtain the optimal release time, instead of considering software faults from different releases. Thus, this section focuses on developing software reliability model in terms of multi-release software incorporating the remaining faults from previous release and the newly introduced faults resulting from the newly introduced features in the development of the next release.

1.6 Overview of the Dissertation

This reminder of this dissertation is stated as follows. In Chapter 1, a general introduction of research background, current research limitation, and the motivation of this dissertation

are discussed. In Chapter 2, literatures that are related to NHPP software reliability model are reviewed. In Chapter 3, the objectives of this dissertation are presented.

In Chapter 4, a comparison analysis of environmental factors affecting software reliability is presented for the development of single-release software product and multi-release software product. This chapter aims to compare what have been changed for the past fifteen years regarding the environmental factors affecting software reliability in the development of single-release software product, and compare the environmental factors affecting software reliability, principles components, and development phase between single-release and multi-release software development.

In Chapter 5, we firstly develop a software reliability model in consideration of dependent fault detection process and imperfect fault removal processes. Later, we define a two-phase software debugging process considering different types of software faults in each phase. Thus, a two-phase software reliability model is proposed in this chapter addressing two main topics, software dependency, and imperfect fault removal process.

Most software reliability models in literature focus on single-release software product. However, nowadays, it is critical to release multiple version software product, given the increasing adoption of agile methodology in software development and the customer-oriented market. Thus, in Chapter 6, a multi-release software reliability model is proposed to capture the remaining faults from previous release and the newly introduced software faults along with the newly added features.

In Chapter 7, due to the significant impacts of environmental factors affecting software reliability during software development, as discussed in Chapter 4, therefore, we firstly propose a single-environmental-factor software reliability model under the Martingale framework to reflect the impact of a significant environmental factor, *Percentage of Reused Modules*. Later, we propose a multiple-environmental-factors software reliability model under the Martingale framework to address more practical and applicable issues in the real world by incorporating multiple environment factors contributed to the reliability improvement in the development process.

In Chapter 8, we conclude this dissertation and discuss the future research.

CHAPTER 2

LITERATURE REVIEW

In this chapter, we review several research articles with respect to the topics investigated in this dissertation.

In Section 2.1, we review the literatures defined and discussed different types of factors and their impact on software quality/reliability in software development process, e.g., environmental factors, success factors, situational factors.

There are two main types of software reliability models, the deterministic models and the probabilistic models. In Section 2.2, we present the review on software reliability models in terms of the deterministic models and the probabilistic models. The probabilistic models can be classified into different groups, such as error seeding, failure rate, curve fitting, reliability growth, Markov structure, time series, and nonhomogeneous Poisson process (NHPP).

This dissertation mainly focuses on the development of software reliability models based on NHPP assumption. Thus, Section 2.3 reviews a great number of literatures regarding NHPP software reliability models in terms of different assumptions on testing effort, testing coverage, fault removal efficiency, time-delay fault removal, and environmental factor impact under perfect debugging/imperfect debugging process. Meanwhile, the

software reliability models developed in Chapters 5, 6 and 7 consider software fault classification and fault dependency, multi-release software product, and the impact of single/multiple environmental factor(s) on software reliability, respectively. Hence, Sections 2.3.1, 2.3.2, and 2.3.3 review the related literatures along with the research topics focused in Chapters 5, 6 and 7, respectively.

2.1 Environmental Factors in Software Development

Although no general definition has been given to define what are the environmental factors affecting software reliability during software development process, there have been many related works in literature defined different types of factors in software development from various perspectives.

Zhang and Pham [27] defined 32 environmental factors to characterize the impact of these environmental factors affecting software reliability during software development process for single-release software product. 32 environmental factors are defined from the four phases of software development, general information, and the interaction with hardware systems. Software development is divided into four phases in this study: Analysis phase, Design phase, Coding phase and Testing phase. The authors conducted a survey investigation and obtained empirical quantitative and qualitative data form mangers, software engineers, designers, programmers, and testers whom participate in software development practices. This paper identified the importance of factors in software development and analyzed the correlation between factors. Later, Zhang et al. [28]

provided an exploratory analysis to further analyze the detailed relationship of these environmental factors.

Sawyer and Guinan [113] presented the effects on software development performance depend on the production method of software development and the social process of how people work together in software development environment.

Roberts Jr. et al. [114] proposed five factors which are essential to implement a system development methodology, e.g., organizational system development methodology (SDM) transition, functional management involvement/support, SDM transition, use of models, and external support.

Chow and Cao [89] collected the survey data from 109 agile projects from a diverse group of organizations with different sizes, industries and geographic locations to provide empirical information for the statistical analysis. Based on the multiple regression analysis, the critical success factors are identified to be a correct delivery strategy, a proper practice of agile software engineering techniques, and a high-caliber team. Three other factors that could be critical to certain success dimensions are identified to be a good agile project management process, an agile-friendly team environment, and a strong customer involvement.

Misra et al. [88] conducted a large-scale survey-based study to identify the success factors from the perspective of agile software development practitioners who have successfully

adopted agile software development in their projects. This study identified nine out of the fourteen hypothesized factors have statistically significant relationship with “success”. The important success factors are: customer satisfaction, customer collaboration, customer commitment, decision time, corporate culture, control, personal characteristics, societal culture, and training and learning.

Clarke and O’Connor [90] conducted the research on the situational factors affecting the software development process. Rigorous data coding techniques from Grounded Theory have been applied in this study. They concluded that the resulting reference framework of situational factors consists of eight classifications and 44 factors that inform the software process. On the other hand, this framework also provides useful information for practitioners who are challenged with defining and maintaining software development process.

In this dissertation, we present a comparison analysis, which has been published in *Journal of Systems and Software* [29], to compare the changes of 32 environmental factors affecting software reliability after Zhang and Pham [27] firstly proposed this idea. The above research mainly discussed the environmental factors/success factors/situational factors in the development process of single-release software. As the application of the principles of agile and lean software development, software multiple releases are very common in the modern society. Software multiple releases not only make company easily balance the competing stakeholder’s demand and benefits but also increase reliability and customer satisfaction level during each release, thus, we conduct another survey study, which has

been published in *Journal of Systems and Software* [30], to investigate the impact of environmental factors on software reliability in the development of multi-release software and compare with the findings in reference [29].

2.2 Software Reliability Model

In the past few decades, software failure data, collected mostly in the testing phase, are applied to study the behavior of software system, including software reliability given a specific period of time, failure growth rate, the number of remaining faults in the system, and the optimal release time. Hence, a great number of research have been focused on the development of software reliability models in the past four decades with various assumptions regarding testing methodology, testing coverage, fault removal efficiency, fault dependency, delay debugging, optimal release time, and so on.

The classification of software reliability model was presented by different researchers [57 - 60]. One of the most widely utilized classification methodologies categorizes software reliability models into two types: the deterministic models and the probabilistic models [19].

The deterministic models are used to study: (1) the element of a program by counting the number of distinct operators, operands, errors, and instruction; (2) the control flow of a program by counting the branches; (3) the data flow of a program (data sharing and passing). There are two well-known models: Halstead's software metric [16] and McCabe's cyclomatic complexity metric [17]. These models provided an innovative and

pioneering quantitative approach to analyze and measure the performance of software system at that time; however, random event is not involved, hence, these models are not suitable to apply in the modern software.

The probabilistic models take into account the failure detection and failure removal as probabilistic events during software development. The classification of probabilistic software reliability model is given by references [19, 61]: (1) error seeding; (2) failure rate; (3) curve fitting; (4) reliability growth; (5) Markov structure; (6) time series; (7) NHPP.

In error seeding class, the number of errors in a program is estimated by applying the multi-stage sampling technique. Mills' error seeding model [62] proposed an error seeding method to estimate the number of errors in a program by introducing seeded errors into the program. Cai [63] modified Mills' model by diving software into two parts: Part 0 and Part 1. Tohma et al. [64] introduced a reliability model based on the hypergeometric distribution to estimate the number of errors in the program.

In failure rate class, these studies focused on how failure rates change at the failure time during the failure intervals. The number of faults in the program is a discrete function, thus, the failure rate of a program is a discrete function as well. The Jelinski-Moranda model [65] is one of the earliest software reliability models, which states the program failure rate at the i^{th} failure interval is given by

$$\lambda(t_i) = \emptyset[N - (i-1)] \quad \text{for } i = 1, 2, \dots, N \tag{2.1}$$

where ∅ is a proportional constant and represents the contribution of one fault makes to the overall program, and *N* is the number of initial faults in the program.

The Schick-Wolverton model [66] modified the Jelinski-Moranda model by assuming the failure rate at the i^{th} time interval increases with time t_i since last debugging. Later, Moranda [67] proposed a reliability model considering the program failure rate function as initially a constant *D* and decreases geometrically at failure times.

In curve fitting class, the models use statistical regression analysis to illustrate the relationship amongst software complexity, the number of faults, and failure rate in the software. Linear regression analysis, nonlinear regression analysis, or time series approach is applied between the dependent and independent variables. Estimation of errors, complexity, and failure rate are investigated in the modeling. Belady and Lehman [68] introduced a model by applying time series approach to estimate software complexity. Miller and Sofer [69] also proposed a model to estimate software failure rate by assuming the failure rate is a monotonically non-increasing function.

In reliability growth class, the improvement of program reliability is measured and predicted via the testing phase by reliability growth models. The failure rate is a function of time, or the number of testing cases in this group of models. Coutinho [70] pointed out that the failure rate is a function of the cumulative number of failures and testing time. Wall

and Ferguson [71] proposed a model that is similar to Weibull model to predict software failure rate during testing.

In the group of Markov structure models, the assumption is that the failure of the modules is independent of each other. Goel and Okumoto [72] proposed a linear Markov model with imperfect debugging. Meanwhile, the transition probability of the model was stated. Littlewoods [73] developed a reliability model incorporating the transitions between modules while operating. Two types of failures are considered: failure from each module, as a Poisson failure process, and failure from interface between modules. Yamada et al. [74] performed a software safety model to illustrate software's time-dependent behavior using Markov process.

In time series model group, Auto-Regressive Integrated Moving Average (ARIMA) method is applied to study software reliability. Singpurwalla and Soyer [75] introduced several ramifications into a random coefficient auto-regressive process of order 1 to describe software reliability. In addition, several research papers [8, 75 - 78] also used time series approach to address software reliability prediction in testing phase and operation phase.

Since this dissertation mainly focuses on the development of software reliability modeling based on the NHPP assumption. Therefore, the literature review on NHPP software reliability models will be carried out in the next section.

2.3 NHPP Software Reliability Model

NHPP has been successfully applied on modeling software reliability since Goel and Okumoto [79] firstly proposed their innovative model in 1979 which assumed a constant fault detection rate and a constant total software fault content. In addition, they made assumptions that all software faults are mutually independent from the failure detection point of view. The failure intensity at any time is proportional to the remaining number of faults in the software program. Based on the fundamental assumptions of Goel-Okumoto model, a great number of NHPP software reliability models have been proposed in the past four decades to address different scenarios in software fault detection and fault correction process.

Ohba [80] proposed the hyper-exponential growth model in consideration of different clusters of modules in a program. Each module contains different initial number of errors and different failure rate, which are all assumed as constants in the software reliability model. It is well-known that the sum of exponential distribution is a hyper-exponential distribution. Thus, the system software reliability model is more like the summation of each module's reliability model.

Yamada and Osaki [81] also proposed a software reliability model which is similar to the model proposed by Ohba [80]. Software can be divided into K modules. The probability of faults for each module will be taken into consideration. The fault detection rate is same within modules, however, various between modules. The total number of errors in the

software is assumed as a constant and there are no new errors will be introduced during fault detection process.

Time-dependent fault detection rate is applied in modeling software reliability growth trend. The concept of S-shaped model is proposed to describe the behavior of the detected failures. Ohba et al. [82] discussed a NHPP model with S-shaped mean value function. Ohba and Yamada [83] also proposed a NHPP model with S-shaped mean value function and considered the cumulative number of detected faults often seems to perform S-shaped. They stated that some of the faults are not detectable before some other faults are removed. In addition, the probability of the failure detection at any time is proportional to the remaining faults in the software.

Around the same time, Yamada et al. [84 - 86] proposed several software reliability models considering software fault detection process as a learning process. Specifically, when software testers get more familiar with testing environment, specifications, and requirements, software fault detection rate will go higher.

Nakagawa [87] developed the connective NHPP model with S-curve forms. A group of modules called, main route modules, are tested first, followed by other modules. Even the failure intensity in the main route modules and other modules are similar, the failure growth curve performs as S-curve since the detection starts at different time points. Afterwards, S-shaped reliability models are further developed in many literatures [8, 9, 91 - 93].

NHPP perfect debugging model often assumes when a failure occurs, the fault that caused it can be immediately removed and no new faults are introduced [72, 83, 84, 94]. The concept of imperfect debugging is based on the assumptions [18, 19]: (1) when detected errors are removed, it is possible to introduce new errors; (2) the probability of finding an error in a program is proportional to the number of remaining errors in the program. Many reliability models are proposed based on NHPP imperfect debugging concept [18, 85, 91 - 93, 98 - 105, 155, 156]. For instance, Yamada et al. [105] developed two software reliability models incorporating imperfect debugging concept. New faults are sometimes introduced when faults originally are latent in software system. The test-effort functions are expressed by exponential and Rayleigh curves in this study.

Since testing phase plays an essential role in software development, a great number of software reliability models focus on testing coverage, testing efficiency, testing resource allocation, and so on. Pham and Zhang [97] presented testing coverage is a measure that enables software developers to evaluate the quality of the tested software and determine how much additional effort is needed to improve the quality and reliability. They introduced a generalized model incorporating the measurement of testing coverage into software reliability assessment. This model indicates that the failure intensity depends on both the rate at which the remaining faults are covered and the number of remaining faults at current time t divided by the current fractional population of uncovered faults.

Zhang et al. [106] developed a software reliability model based on imperfect debugging considering new faults can be introduced while debugging and the detected faults may not

be removed completely. They defined the fault removal efficiency in the study, which presented a new idea for the later research.

Huang and Lyu [107] studied the impact of testing effort and testing efficiency on the modeling of software reliability and the cost for optimal release time. Inoue and Yamada [99] developed a software reliability model by formulating the relationship between the alternative testing-coverage evaluation function and the number of detected faults. Later, Huang [108] incorporated both a generalized logistic testing-effort function and the change-point parameter into software reliability model.

Li et al. [109] incorporated logistic testing coverage function to develop software reliability model. Lin and Huang [110] incorporated the Weibull-type testing effort function with the multiple change-points into software reliability modeling. Moreover, Chatterjee and Singh [9] proposed a software reliability model by considering a logistic-exponential testing coverage function and an imperfect debugging process.

Time-delay fault removal are also discussed in many literatures. Xie and Zhao [111] generalized Schneidewind's model by assuming a continuous time-dependent delay function which quantifies the expected delay in correcting the detected faults. Delay is treated as an increasing function of time *t*. The faults are easily to be corrected in the early state of testing and become difficult to detect as time goes by.

Hwang and Pham [112] developed a generalized NHPP software reliability model considering quasi-renewal time-delay fault removal. The time delay is defined as the interval between fault detection and fault removal. Time-delay is considered as a time-dependent function, described by a quasi-renewal process in this study. This model provides a more relaxed assumption in software testing and debugging, which is very close to the practical testing and debugging process.

The testing resource allocation during the testing phase, which is usually depicted by the testing effort function, which is affected not only by the fault detection rate but also the time to correct a detected fault. In the research paper published by Peng et al. [204], the authors firstly incorporated testing effort function and fault introduction into the fault detection process and then developed the model considering fault correction prosses as a fault detection process with a correction effort. Various specific paired fault detection process and fault correction process models are obtained based on different assumptions of fault introduction and correction effort.

2.3.1 Software Reliability Model with Different Fault Classification

Many literatures state that there exists more than one type of software fault in the program [117 - 126]. Different fault classes are categorized by practitioners and researchers to describe the characteristics of software faults that cause failures during testing and operation phase [117, 118, 124 – 126]. The limits and challenges in the dependability of computer systems in terms of the fault class, such as physical faults, design faults, and interaction faults, are discussed in references [124, 125] as well.

Ohba [80] discussed two types of software faults, mutually independent faults and mutually dependent faults. Tokuno and Yamada [104] proposed an imperfect debugging software reliability model with two types of software failures involved. The first type is caused by the fault latent in the system, which is described by a geometrically decreasing function; the second type fault is randomly regenerated in testing phase, which has a constant hazard rate.

Kapur and Younes [127] considered leading error and dependent error in the model development. Pham [128, 129] also studied multiple failure types with different detection rate, but it was too simple to address the modern software products.

Goseva-Popstojanova and Trivedi [130] addressed the fault correlation and its impact on the software reliability assessment. Dai et al. [131] incorporated multiple types of software failures in software reliability model. Huang and Lin [132] incorporated fault dependence and delay debugging in the software reliability growth model.

Grottke et al. [117] studied the proportion of the various fault types and their evolvement with time based on the fault discovered in the on-board software for 18 JPL/NASA space missions. However, they did not provide a quantitative way to estimate the number of faults. Thus, an explicit method to quantify the behavior of different fault type, consider software dependency and imperfect fault removal is needed.

Our research, which has been published in *Computer Languages, Systems & Structures* [133], proposes a new NHPP software reliability model with a pioneering idea by considering software fault dependency and imperfect fault removal. In order to clearly explain software fault dependency, some facts and examples regarding the detection of different type of faults are discussed in Chapter 5. Two types of software faults are defined, Type I (independent) fault and Type II (dependent) fault, based on the consideration of fault dependency. Two phases debugging processes, Phase I and Phase II, are proposed according to the debugged software fault type. A small portion of software faults that software testers are not able to remove is also considered in both phases in the proposed model.

2.3.2 Multi-Release Software Reliability Model

Most software products are not introduced into the market with full capacities at their initial release. New features will be added, and existing features will be enhanced after launched software for a while. Extensive studies have been done for the release of single version software system for the past few decades. Modeling and predicting software failure behavior are investigated by many researchers as well. However, only a few researchers studied multi-release software reliability and introduced prediction models to explain software fault detection process and fault removal process for multi-release software products.

Garmabaki et al. [115] incorporated different severities level used to describe the difficulty of correcting faults in the upgrade process to develop a multi up-gradation software

reliability model. Faults are classified into two categories, simple fault and hard fault. The fault removal for the development of the new release depends on the fault from previous release and fault generated in that release.

Hu et al. [47] considered the effect of multiple releases regarding the fault detection process in software development. They assumed that there is no gap between the release of previous version and the development of next version. Moreover, optimal release time for each version is discussed in this study.

Kapur et al. [48] introduced the combined effect of schedule pressure and resource limitations by the use of Cobb-Douglas production function in software reliability modeling. The Cobb-Douglas function illustrates the total production output can be obtained by the amount of labor input, capital input, and total factor productivity. An optimal release planning problem is formulated in this study for software with multiple releases with the solution obtained by applying genetic algorithm method.

Pachauri et al. [50] proposed a software reliability growth model by considering fault reduction factors (FRFs) and extended this idea to multi-release software systems. FRFs is defined as the ratio of the total number of reduced faults to the total number of failures. FRFs is not a constant, which can be affected by other factors, such as resources allocation, debugging time lag, and imperfect debugging.

Yang et al. [49] incorporated fault detection and fault correction process in multi-release software reliability modeling. They considered there is a time-delay in fault repair after detecting faults. The time-delay function is explained by an exponential function or a gamma function. They also assumed the faults in a new version including both the undetected faults from last version and the newly introduced faults during the development process of the new version.

However, most literatures aimed to develop multi-release software reliability model only through optimizing software cost model to determine the optimal software release time except Yang et al. [49].

Our research, which has been published in *Annals of Operations Research* [116], focuses on the development of a multi-release software reliability model considering the remaining software faults from previous release and the newly introduced faults (from newly added features). Additionally, dependent fault detection process is taken into account in this research. In particular, the detection of a new software fault for developing the next release depends on the detection of the remaining faults from previous release and the detection of the newly introduced faults.

2.3.3 Environmental-Factor-Based Software Reliability Model

As discussed in Chapter 1, given the current trends of software development process, which are the adoption of software product lines, software development globalization, and the establishment of software ecosystems, the complication and human-centered software

development process needs to be addressed more appropriately. Meanwhile, environmental factors play significant impacts on affecting software reliability during development process [27 – 30, 88 – 90, 113, 114], as discussed in Section 2.1. Thus, how to incorporate the single/multiple environmental factors which present significant impact on reliability into software reliability model is critical to address modern software development in practice.

Only a few literatures incorporated random effect of the environments, or other factors, e.g., fault reduction factor, into software reliability models.

Teng and Pham [103] presented a new methodology for predicting software reliability in the field environment. A generalized random field environment (RFE) software reliability model which can cover both the testing phase and operating phase is proposed in this study by assuming all the random effect in the field environments can be captured by a unit-free environmental factor. Two specific RFE reliability models are developed by the use of the generalized RFE software reliability model, called the γ-RFE model and the β-RFE model, to describe different random effects in the operation phase.

Hsu et al. [134] integrated fault reduction factor into software reliability models. Fault reduction factor is proposed by Musa [135], which is generally defined the ratio of net fault reduction to failures experience, which could be influenced by many environmental factors, e.g., imperfect debugging or delay debugging. The authors firstly studied the trend of the fault reduction factor and considered it as a time-variable function, and then incorporated

the fault reduction factor in software reliability growth modeling to improve the accuracy of failure prediction.

Recently, Pham [136] incorporated the uncertainty of the operation environment into a software Vtub-shaped fault detection rate model. In particular, fault detection rate in this study is represented by a Vtub-shape function and the uncertainty of the operation environments is represented by a random variable, modeled as gamma distribution.

The first part of our research in this chapter, which has been published in *Annals of Operations Research*, incorporates one of the top 10 significant environmental factors, *Percentage of Reused Modules* (PoRM), to be a random variable which has random effect on fault detection rate. We then introduce the Martingale framework, specifically, Brownian motion and white noise process in the stochastic fault detection process to reflect the impact resulting from the randomness of environmental factor and to propose a single-environmental-factor software reliability model. Moreover, given the significance of these impact, we further propose a generalized software reliability model with multiple environmental factors under the Martingale framework.

CHAPTER 3

OBJECTIVES OF THE DISSERTATION

Software reliability growth model has been studied for a long time since 1970s. A great number of software reliability models have been proposed in consideration of different methodologies, assumptions, and field applications. In order to obtain software metrics such as the remaining faults in software system, failure growth rate, failure intensity, and optimal release time, past testing/operation failure data are usually employed to develop software reliability growth model.

Given the transitions of modern software development, how to predict software failures in terms of the practical development/application scenarios is critical yet challenging for researchers. Therefore, in this dissertation, we have not only integrated software practitioners' opinions from a wide variety of industries, but also developed software reliability models by addressing different practical problems observed in software development practices.

The objectives of this dissertation are stated as follows.

(1) Reinvestigate the environmental factors affecting software reliability in single-release software development and compare the findings with references [27, 28] to present an advanced analysis for all 32 environmental factors. Specifically,

- Investigate the correlation of some of the environmental factors and identify the methodology to reduce the dimension of related environmental factors, in order to apply the significant factors into software reliability model in the later research.
- Reveal the significant factors in each development phase.
- Examine the significance level of each development phase in the whole software development process.
- Compare the significant factors, principle components, significance levels of development phases, significant factors of each development phase, and time allocation of each development phase revealed in this study and previous findings [27, 28] to investigate the root cause of those differences.

(2) Investigate the environmental factors affecting software reliability in multi-release software development and compare the findings in Objective (1) to provide a comprehensive analysis for software development practices.

(3) Develop one-phase and two-phase NHPP software reliability models considering software fault dependency and imperfect fault removal process. Specifically,

- Define two types of software faults, Type I (independent fault) and Type II (dependent fault).
- Two phases debugging process are introduced. Each type of software fault will be detected and removed in different phase according to their own characteristics.
- Compare the descriptive and predictive ability of the proposed models with the existing NHPP models.

(4) Propose a multi-release software reliability model incorporating the remaining faults from previous release and newly introduced faults (from newly added features) for the development of the new release. The detection of the new faults depends on the remaining faults from previous release and the newly introduced faults. We also compare the predictive ability of the proposed model with other multi-release software reliability models.

(5) Given the findings from Objectives (1) and (2), incorporate single/multiple environmental factor(s) and the randomness caused by these environmental factor(s) in software reliability models under the Martingale framework. We also compare the predictive power of the software reliability models with and without incorporating environmental factors(s).

CHAPTER 4

ENVIRONMENTAL FACTORS IN SOFTWARE DEVELOPMENT

4.1 Environmental Factors in Single-Release Software Development

4.1.1 Research Motivation

Computer systems are widely applied on various areas to provide fast, reliable, and effective service to the targeted market nowadays. Software development process has gone through dramatic changes especially in the past one and half decades. The rise of the Internet has led to the rapid growths in many industries, such as semiconductor, pharmaceutical, online retailing, banking, financial service, and computer hardware & software. The high technology has an ever-increasing impact on our daily life. For example, the wide applications of computational devices like mobile phones, tablets, and laptops have brought the increasing conveniences for everyday life.

At the same time, software development process has become more critical and complicated, which also brings major effects on the growth of a company, including market share and position, customer loyalty, and the new product development. One of the main challenges for the software development team is to provide on-time, reliable, and high-quality software within budget [28]. On the other hand, software development is a complex and human-centric activity that is subject to many pitfalls if not appropriately organized [90]. Hence, improving software reliability has turned out to be one of the essential concerns for software practitioners and researchers.

Many software reliability models have been proposed during the past few decades, most of which focus on analyzing the software fault detection process as a function of time. Yet, most software reliability models are developed without considering other attributes during this complicated and human-centered software development process, testing process, and field operation.

Only a few literatures considered the impact of random environments and predicted the software reliability incorporating the random environmental factors. Teng and Pham [103] presented a new methodology for predicting software reliability in the field environment. A generalized random field environment (RFE) software reliability model which can cover both the testing phase and operating phase is proposed in this study by assuming all the random effect in the field environments can be captured by a unit-free environmental factor. Later, Hsu et al. [134] integrated fault reduction factor into software reliability models.

Zhang and Pham [27] defined 32 environmental factors in the software development process fifteen years ago. These factors not only included each phase of software development practice, but considered human nature, teamwork, and interactions of hardware system. They also provided a survey study to investigate the ranking based on the influence of the factors on software reliability and correlations between some of the factors. Moreover, Zhang et al. [28] presented another exploratory analysis to reduce the dimension of the factor space by applying factor analysis, analyze the relation between the years of experience, positions, and the opinions on software reliability improvement for 32

environmental factors. But, is there any changes of the findings after fifteen years? If there is any change, how can we quantify those changes?

4.1.2 Objectives

This study aims to reinvestigate the environmental factors and compare the findings with the references [27, 28] to provide an up-to-date ranking for all 32 environmental factors. The investigation is carried out by conducting a survey of environmental factors regarding their impact on software reliability from the perspective of managers, software engineers, programmers, and testers. We are going to investigate the correlation of some of the factors, identify methods to reduce the dimension of some related environmental factors, and identify the significant environmental factors in each development phase to provide a practical reference for software practitioners.

Statistical method such as principle component analysis (PCA) can be employed to perform the dimension deduction and correlation analysis. In addition, analysis of development life cycle is discussed in this study. The significant environmental factors are revealed in each development phase. Examination of the significance level of each development phase is included in the study as well. Finally, by comparing the ranking for the environmental factors, principle components of the environmental factors, significance levels of development phases, significant factors and time allocation of each development phase with previous findings, this study enables us to understand the root causes of these differences. The future software reliability models may also incorporate the significant factors defined from this study to improve the power of prediction.

4.1.3 Data Collection

We use the same survey from Zhang and Pham [27] included 32 environmental factors, which are defined from software development process and the information of software developer background. This study aims to investigate whether the significance levels of environmental factors during software development have changed after one and half decades. Thus, it is desired to maintain the homogeneity of the organization type and choose similar organizations to conduct the survey. Twenty organizations from diverse industries are selected to participate in the survey investigation.

Thirty-five survey forms are collected from 20 companies in various industries including semiconductor, pharmaceutical, online retailing, banking, financial service, computer hardware & software, IT service & consulting, and oil field service & equipment. Participants were asked to rank the environmental factors in terms of its impact on software reliability. Software developers in the participating organizations mainly focus on safety-critical, commercial, and inside users-oriented applications. To obtain a wide-ranging investigation, the years of experience and experience type on software development for the investigated participants are different. For instance, the software development experience type includes database, operation system, communication control, and language processor. In addition, participants have different positions, such as manager, software engineer, programmer, tester, and system administrator.

4.1.4 Findings and Results

Environmental Factor Analysis

Relative Weighted Method

The ranking for each environmental factor is from 0 to 7 in terms of its impact level on affecting software reliability. For example, if participants think one environmental factor is an extremely important factor based on its significance impact on software reliability, they should rank 7; if participants think the one environmental factor will not have impact on software reliability at all, then the rank for this environmental factor should be 0. Hence, the ranking summations for all 32 environmental factors are ranging from 0 to 224 in this study.

Relative weighted method is applied firstly to investigate the ranking proportion of each environmental factor in one survey form and all survey forms. First, summarize the original ranking provided by participants for all 32 environmental factors in each survey form. 31 out of 35 complete survey forms are used to calculate the ranking for the 32 environmental factors. The summations for all 32 environmental factors of our original ranking range from 64 to 196. Let r_{ij} be the score of the i^{th} factor in the j^{th} survey. Normalize r_{ij} for each survey by using

$$w_{ij} = \frac{r_{ij}}{\sum_{i=1}^{N} r_{ij}} \tag{4.1}$$

where N is the number of factors which are filled up by the j^{th} participant in the j^{th} survey.

The range of $\sum_{i=1}^{N} r_{ij}$ is from 64 to 196 as described earlier. Thus, by averaging w_{ij}, we obtain the final weight for the i^{th} factor by

$$w_i = \frac{\sum_{j=1}^{l} w_{ij}}{l} \tag{4.2}$$

where l is the number of complete survey forms.

Table 4.1 presents the final normalized weight for each environmental factor and the ranking of all 32 environmental factors. Generally, the environmental factor that has a higher normalized weight plays a more significant impact on the software reliability than the factor that has a lower weight; thereby, software developer should pay more attention to the environmental factors with higher normalized weight.

Principle Component Analysis of Environmental Factors

From Table 4.1, we notice that the environmental factors cover many aspects in the software development, and the majority of these factors have similarly impact on software reliability. A question one might ask now is whether some of these factors are correlated. If some of these factors are correlated, is it possible to reduce the dimension of these correlated factors?

PCA is a statistical analysis that can be used to reduce the dimensionality of a data set consisting of a large number of interrelated variables, while retaining as much of the variation present in the data set as possible [137].

Table 4. 1 Environmental factors ranking based on relative weight method

Rank	Factor	Factor name	Normalized weight
1	f8	frequency of program specification change	0.039283972
2	f22	testing effort	0.038359377
3	f21	testing environment	0.038250613
4	f25	testing coverage	0.037435855
5	f1	program complexity	0.036961340
6	f15	programmer skill	0.036862675
7	f6	percentage of reused modules	0.036405410
8	f12	relationship of detailed design to requirement	0.035234326
9	f24	testing methodologies	0.035101487
10	f19	domain knowledge	0.035036992
11	f16	programmer organization	0.034724575
12	f18	program workload(stress)	0.034271335
13	f23	testing resource allocation	0.034194468
14	f13	work standards	0.034183421
15	f11	requirements analysis	0.034042553
16	f20	human nature	0.033593929
17	f14	development management	0.033003212
18	f3	difficulty of programming	0.032899502
19	f4	amount of programming effort	0.032852885
20	f5	level of programming technologies	0.032745404
21	f26	testing tools	0.032457656
22	f27	documentation	0.032236361
23	f9	volume of program design documents	0.032076971
24	f10	design methodology	0.031998207
25	f17	development team size	0.030222175
26	f7	programming language	0.029878290
27	f2	program categories	0.028608644
28	f28	processors	0.027675094
29	f31	telecommunication devices	0.027071145
30	f32	system software	0.026832350
31	f30	input/output devices	0.026788903
32	f29	storage devices	0.026773022

The idea is that smaller dimension set of principle components can be used to capture the characteristics of the larger data set and provide a concise yet critical principle components for software developers. The top 10 most important environmental factors are selected based on the ranking results present by relative weight method in Table 4.1. The top 10 important environmental factors are f8, f22, f21, f25, f1, f15, f6, f12, f24, f19; four of them come from the Testing phase, two of them in the Analysis and Design phase, two of them in the Coding phase, and two of them in the General phase.

The original data set is ten-dimensional. PCA method will find the covariance matrix first, and then calculate the eigenvector and eigenvalue from this 10 by 10 covariance matrix. Eigenvectors enable us to capture the characteristics of the data set. Eigenvalue is ranked from the highest to the lowest in terms of the order of significance level of principle components. The eigenvalue of components, proportion, and cumulative proportion are illustrated in Table 4.2.

About 40% of the data variation can be explained by the first principle component, 16.7% of the data variation can be explained by the second principle component, and 11.8% of the data variation can be interpreted by the third principle component. Thus, about 69% of the data variation can be explained by the first three principle components. On the other hand, by subtracting the second eigenvalue 1.679 from the first eigenvalue 4.008, we obtain a difference is 2.329; by subtracting the third eigenvalue 1.175 from the second eigenvalue 1.679, we have a difference is 0.504; however, if we subtract the fourth eigenvalue 0.843

from the third eigenvalue, the difference is 0.332, which is smaller compared with the first two differences. Hence, the first three principle components are retained.

Table 4. 2 Eigenvalue of correlation matrix

Component	Eigenvalue	Proportion	Cumulative proportion
PC1	4.008	0.401	0.401
PC2	1.679	0.167	0.568
PC3	1.175	0.118	0.686
PC4	0.843	0.084	0.770
PC5	0.737	0.074	0.844
PC6	0.608	0.061	0.905
PC7	0.442	0.044	0.949
PC8	0.241	0.024	0.973
PC9	0.165	0.016	0.989
PC10	0.106	0.011	1.000

As seen from Table 4.2, three principle components are retained, PC1, PC2, and PC3. The loading coefficient, measures the contribution to the data variance between the principle components and related environmental factors, is also determined. If the loading coefficient is very small, or only strongly-correlated with less-significant principle components, such as PC4, PC5, PC6, PC7, PC8, PC9, PC10, these environmental factors have little or no contribution to the variation of the data set. Table 4.3 describes the environmental factors which are strongly correlated with the first three principle components. For example, testing coverage has the highest correlation with the first principle component compared with other environmental factors. Frequency of specification change and programmer skills have high correlation with the second principle component and the third principle component, respectively.

Table 4. 3 Principle components and strongly correlated factors

Component	Factor	Description	Loading coefficient
PC1	f25	testing coverage	0.400
	f21	testing environment	0.385
	f22	testing effort	0.379
	f24	testing methodologies	0.368
	f12	relationship of detailed design and requirement	0.359
	f6	percentage of reused code	0.269
PC2	f8	frequency of Specification change	0.559
	f19	domain knowledge	0.482
PC3	f15	programmer skills	0.664
	f1	program complexity	0.408

Hypothesis Testing

This study aims to investigate whether those environmental factors have the same significance impact on software reliability. Analysis of Variance (ANOVA) is a statistical method used to compare the mean of two or more samples set [138]. ANOVA is applied in this study to compare the significance level for these environmental factors. In order to compare all pairwise difference between factors and control the error rate within a level that you specify [139], Tukey method, a single-step multiple comparison procedure, is used to group environmental factors in terms of their mean values.

Table 4.4 depicts the final grouping using Tukey's method. Table 4.4 lists the environmental factors based on the significance impact on software reliability from the greatest to the least. Factors of the same numeric values are considered as of the same significant level. For example, based on the final grouping, the most impactful factor is

testing coverage; frequency of specification change is the second significant factor followed by testing environment and testing effort as the third and fourth critical factors. Factors listed from testing methodologies to processors belong to one significance level based on the Tukey method.

Correlation Analysis

The purpose of performing correlation analysis is to observe the relationship between variables and investigate the strength and direction of this relationship. Having the knowledge of the correlation of the environmental factors will provide a better understanding for software developers on resource allocation and testing efficiency during software development, in particular, Testing phase. The Pearson product-moment correlation coefficient, also called Pearson's *r*, is a measure of the linear correlation between two variables. The range for Pearson's *r* is from -1 to 1. If *r* is 0, meaning there is no relationship between variables; if *r* is 1, indicating a total positive correlation; if *r* is -1, referring a total negative correlation. For the output from Minitab, the absolute value of *r* is larger than 0.5 is chosen as the correlation factors for each environmental factor. Table 4.5 presents the correlation of the environmental factors.

Table 4. 4 Final grouping based on Tukey method

Factor	Description	N	Mean	Final grouping
f25	testing coverage	29	5.586	1
f8	frequency of program specification change	31	5.516	2
f21	testing environment	30	5.500	3
f22	testing effort	30	5.433	4
f24	testing methodologies	29	5.276	5
f15	programmer skill	31	5.226	5
f23	testing resource allocation	28	5.214	5
f12	relationship of detailed design to requirement	28	5.214	5
f1	program complexity	29	5.172	5
f16	programmer organization	30	5.100	5
f6	percentage of reused modules	30	5.067	5
f19	domain knowledge	30	5.033	5
f11	requirements analysis	30	5.000	5
f13	work standards	30	4.967	5
f18	program workload(stress)	29	4.966	5
f27	documentation	29	4.931	5
f4	amount of programming effort	29	4.897	5
f10	design methodology	29	4.862	5
f9	volume of program design documents	29	4.862	5
f3	difficulty of programming	28	4.857	5
f14	development management	28	4.786	5
f20	human nature	31	4.774	5
f5	level of programming technologies	30	4.733	5
f26	testing tools	29	4.621	5
f17	development team size	29	4.483	5
f7	programming language	30	4.233	5
f28	processors	28	4.179	5
f2	program categories	29	4.103	6
f32	system software	28	4.071	6
f30	input/output devices	29	4.043	7
f29	storage devices	29	3.966	8
f31	telecommunication devices	28	3.964	9

Table 4. 5 Correlation analysis for single-release software survey data

Factor	Description	Correlated factors		Pearson's *r*
f25	testing coverage	f18	program workload(stress)	0.773
		f24	testing methodologies	0.729
		f26	testing tools	0.661
		f27	documentation	0.660
		f12	relationship of detailed design to requirement	0.648
		f17	development team size	0.568
		f9	volume of program design documents	0.566
		f3	difficulty of programming	0.537
		f4	amount of programming effort	0.528
		f14	development management	0.528
		f22	testing effort	0.506
f8	frequency of program specification change	f31	telecommunication devices	-0.519
f21	testing environment	f22	testing effort	0.749
		f19	domain knowledge	0.719
		f16	programmer organization	0.668
		f13	work standards	0.576
		f4	amount of programming effort	0.526
f22	testing effort	f21	testing environment	0.749
		f13	work standards	0.651
		f18	program workload(stress)	0.610
		f16	programmer organization	0.564
		f25	testing coverage	0.506
f24	testing methodologies	f4	amount of programming effort	0.574
		f9	volume of program design documents	0.549
		f10	design methodology	0.506
		f12	relationship of detailed design to requirement	0.555
		f17	development team size	0.612
		f18	program workload(stress)	0.666
		f25	testing coverage	0.729
		f26	testing tools	0.540
		f27	documentation	0.533
		f31	telecommunication devices	0.527
		f32	system software	0.526
f15	programmer skill	f20	human nature	0.431
f23	testing resource allocation	f12	relationship of detailed design to requirement	0.766
		f14	development management	0.712
		f4	amount of programming effort	0.625

		f10	design methodology	0.618
		f3	difficulty of programming	0.579
		f27	documentation	0.556
		f1	program complexity	0.547
		f32	system software	0.536
		f9	volume of program design documents	0.533
		f26	testing tools	0.504
f12	relationship of detailed design to requirement	f23	testing resource allocation	0.766
		f32	system software	0.734
		f28	processors	0.659
		f25	testing coverage	0.648
		f31	telecommunication devices	0.641
		f14	development management	0.626
		f3	difficulty of programming	0.610
		f9	volume of program design documents	0.593
		f27	documentation	0.578
		f30	input/output devices	0.562
		f10	design methodology	0.555
		f24	testing methodologies	0.555
		f1	program complexity	0.552
		f4	amount of programming effort	0.536
f1	program complexity	f12	relationship of detailed design to requirement	0.552
		f23	testing resource allocation	0.547
		f29	storage devices	0.510
f16	programmer organization	f21	testing environment	0.668
		f22	testing effort	0.564
		f19	domain knowledge	0.536
f6	percentage of reused modules	f5	level of programming technologies	0.578
f19	domain knowledge	f21	testing environment	0.719
		f10	design methodology	0.544
		f16	programmer organization	0.536
		f4	amount of programming effort	0.524
f11	requirements analysis	f2	program categories	0.637
f13	work standards	f22	testing effort	0.651
		f21	testing environment	0.576
		f18	program workload(stress)	0.526
f18	program workload(stress)	f25	testing coverage	0.773
		f4	amount of programming effort	0.673
		f24	testing methodologies	0.666
		f27	documentation	0.628
		f22	testing effort	0.610

		f26	testing tools	0.572
		f9	volume of program design documents	0.557
		f3	difficulty of programming	0.550
		f13	work standards	0.526
f27	documentation	f9	volume of program design documents	0.731
		f18	program workload(stress)	0.628
		f10	design methodology	0.624
		f12	relationship of detailed design to requirement	0.578
		f14	development management	0.559
		f23	testing resource allocation	0.556
		f4	amount of programming effort	0.541
		f30	input/output devices	0.527
f4	amount of programming effort	f3	difficulty of programming	0.753
		f10	design methodology	0.697
		f9	volume of program design documents	0.677
		f18	program workload(stress)	0.673
		f23	testing resource allocation	0.625
		f14	development management	0.575
		f24	testing methodologies	0.574
		f27	documentation	0.541
		f12	relationship of detailed design to requirement	0.536
		f25	testing coverage	0.528
		f21	testing environment	0.526
		f19	domain knowledge	0.524
f10	design methodology	f9	volume of program design documents	0.840
		f4	amount of programming effort	0.697
		f27	documentation	0.624
		f23	testing resource allocation	0.618
		f14	development management	0.591
		f3	difficulty of programming	0.570
		f12	relationship of detailed design to requirement	0.555
		f32	system software	0.547
		f19	domain knowledge	0.544
		f24	testing methodologies	0.506
f9	volume of program design documents	f10	design methodology	0.840
		f27	documentation	0.731
		f4	amount of programming effort	0.677
		f12	relationship of detailed design to requirement	0.593
		f25	testing coverage	0.566
		f32	system software	0.559
		f18	program workload(stress)	0.557

		f24	testing methodologies	0.549
		f26	testing tools	0.546
		f14	development management	0.533
		f23	testing resource allocation	0.533
		f30	input/output devices	0.529
		f28	processors	0.525
f3	difficulty of programming	f4	amount of programming effort	0.753
		f14	development management	0.613
		f12	relationship of detailed design to requirement	0.610
		f23	testing resource allocation	0.579
		f10	design methodology	0.570
		f18	program workload(stress)	0.550
		f25	testing coverage	0.537
f14	development management	f23	testing resource allocation	0.712
		f12	relationship of detailed design to requirement	0.626
		f3	difficulty of programming	0.613
		f10	design methodology	0.591
		f4	amount of programming effort	0.575
		f27	documentation	0.559
		f9	volume of program design documents	0.533
		f25	testing coverage	0.528
f20	human nature	f32	system software	0.434
		f15	programmer skill	0.431
f5	level of programming technologies	f6	percentage of reused modules	0.578
f26	testing tools	f27	documentation	0.610
		f28	processors	0.610
		f30	input/output devices	0.609
		f18	program workload(stress)	0.572
		f25	testing coverage	0.561
		f9	volume of program design documents	0.546
		f24	testing methodologies	0.540
		f23	testing resource allocation	0.504
f17	development team size	f24	testing methodologies	0.612
		f25	testing coverage	0.568
f7	programming language	f30	input/output devices	0.525
f28	processors	f32	system software	0.871
		f30	input/output devices	0.806
		f31	telecommunication devices	0.795
		f12	relationship of detailed design to requirement	0.659
		f9	volume of program design documents	0.525
f2	program categories	f11	requirements analysis	0.637

f32	system software	f28	processors	0.871
		f31	telecommunication devices	0.826
		f30	input/output devices	0.799
		f9	volume of program design documents	0.559
		f10	design methodology	0.547
		f23	testing resource allocation	0.536
		f24	testing methodologies	0.526
f30	input/output devices	f28	processors	0.806
		f32	system software	0.799
		f31	telecommunication devices	0.748
		f26	testing tools	0.609
		f12	relationship of detailed design to requirement	0.562
		f9	volume of program design documents	0.529
		f27	documentation	0.527
		f7	programming language	0.525
f29	storage devices	f1	program complexity	0.510
f31	telecommunication devices	f32	system software	0.826
		f32	system software	0.799
		f28	processors	0.795
		f30	input/output devices	0.748
		f12	relationship of detailed design to requirement	0.641
		f24	testing methodologies	0.527
		f8	frequency of program specification change	-0.519

Development Life Cycle Phase Analysis

Do First Four Environmental Factor Groups have Same Impact?

Since Zhang and Pham [27] categorized these 32 environmental factors into five groups, General, Analysis and Design, Coding, Testing, Hardware Systems, we intend to investigate whether the first four groups have the same impact on software reliability or not in this study and compare with the results from Zhang et al. [28].

Table 4.6 describes the final grouping for each phase in software development process. The ranging of the mean score from participants for each development phase is from 4.722 to 5.225. Tukey grouping is applied here to group different development phases based on the mean value of participants' score. Finally, three final groups are present in Table 4.6. Testing phase, including testing effort, testing methodologies, testing coverage, etc., are in the Group 1. Analysis and Design phase and Coding phase are in the Group 2. General phase, including program complexity, program categories, amount of programming effort, etc., are in the Group 3.

The final grouping table exhibits the Testing phase has higher mean value compared with other development phases, which also implies that the Testing phase has higher significant impact level on affecting software reliability from the survey feedback. On the other hand, 40% of the environmental factors on the top 10 ranking environmental factors in Table 4.1 is from the Testing phase; in particular, there are four environmental factors from the Testing phase stay on the top 10 ranking. These comparisons provide the same conclusion

that Testing phase is the most critical phase on affecting software reliability during the software development process in this study.

Table 4. 6 Final grouping for development phase

Development phase	N	Mean	Tukey grouping		Final grouping
Testing	204	5.225	A		1
Analysis and Design	205	5.034	A	B	2
Coding	180	4.933	A	B	2
General	205	4.722		B	3

Significant Factors in Each Development Phase

As mentioned in the last section, General phase, Analysis and Design phase, Coding phase, and Testing phase are included in software development. Software projects are often considered as long-term projects [140]. Communication, coordination, and knowledge sharing with other team members are very common amongst software developers, as the software project grows larger [141]. However, an individual still has his/her own job responsibility. Thus, it is very helpful to provide the significant environmental factors in each development phase to software developers based on their impact on software reliability.

Backward elimination method is applied in this study to eliminate the non-significant environmental factors in each development phase. The variables of the backward elimination for each development phase are the environmental factors in this phase. The responses of the backward elimination are the improvement of the accuracy on software

reliability, which are collected from the participants as well and the value may vary from 0 to 100. Table 4.7 presents the significant factors, parameter estimate, and *p*-value for each phase.

Backward elimination starts with all the predictors, all environmental factors in each development phase in the model, eliminates the predictor based on the *p*-value as compared with Alpha-to-Remove value and stops when the *p*-value for all left predictors are less than or equal to Alpha-to-remove value [139]. Parameters are also calculated in the backward elimination for the predicators in the final linear model. Hence, the parameter estimate can be positive or negative, which depends on this environmental factor is positively or negatively correlated with the improvement of software reliability. As seen from the results, f24, testing methodologies is negatively correlated with the improvement of software reliability. In practice, as the testing methodology becomes more complicated, it becomes easier for the software testers to make mistakes during testing. As a result, the improvement of software reliability will decrease, which explains the coefficient of testing methodologies in the backward elimination method is negative.

In the General phase, f6, percentage of reused code, is one of the significant factors, which is also on the top 10 ranking environmental factors by relative weighted method from Table 4.1. f8, frequency of program specification change, and f12, relationship of detailed design to requirement, are significant factors in the Analysis and Design phase, which are ranked 1 and 8 on top 10 ranking environmental factors, respectively. It is noticeable that significant environmental factors identified by using the backward elimination method for

each development phase are also have high rankings based on the relative weighted method described in Table 4.1.

Table 4. 7 Significant environmental factor in each development phase

Development phase	**Significant factor**	**Description**	**Parameter estimate**	***p*-value**
General	f4	amount of programming effort	8.2	0.0001
	f6	percentage of reused code	4.7	0.0130
Analysis and Design	f8	frequency of program specification change	5.9	0.0060
	f12	relationship of detailed design to requirement	5.7	0.0140
Coding	f15	programmer skill	6.5	0.0320
	f18	program workload (stress)	8.9	0.0001
Testing	f23	testing resource allocation	8.3	0.0010
	f24	testing methodologies	-7.8	0.0170
	f25	testing coverage	10.8	0.0020

4.1.5 Comparisons

The above sections have discussed the environmental factor analysis and development life cycle phase analysis based on the survey we conducted. What is left for this study is to investigate whether the significance rankings of 32 environmental factors on affecting software reliability have changed after fifteen years; if they have changed, what are the reasons that caused these changes.

In this section, results comparisons between the analysis in this study and those in the references [27, 28] will be discussed in the following section. These comparisons will be helpful for software developers to identify up-to-date significant environmental factors in software development and improve working efficiency on software reliability

improvement. Moreover, incorporating these up-to-date significant environmental factors into software reliability modeling will enhance prediction for the software reliability analysis [103].

Ranking of Environmental Factors

Since the participants and the time of the two investigations are not identical, and thus the ranking results are different. What is interesting is that most of the top 10 ranking environmental factors from fifteen years ago still stay on the top 10 ranking nowadays except f11 falls to rank 15 based on the relative weight method. Additionally, the order of importance for each environmental factor has changed compared with the previous analysis [27]. The comparison of the top 10 ranking between the new analysis and the findings from Zhang and Pham [27] are illustrated in Table 4.8.

As described in Table 4.8, f11, requirement analysis, are replaced by f19, domain knowledge. F11, requirement analysis, usually, analyzes the requirements from current or potential customers. Software developer will generate the updated specifications for the new product in consideration of the customer specifications. f19, domain knowledge, refers to the developer's knowledge of the input space and output target. Since the wide application of computing operation systems and a great amount of sales data analysis, software development team has better understanding in terms of customer requirement compared with 15 years ago. Domain knowledge is critical nowadays because of insufficient domain knowledge not only causes problems for coding and testing procedures, but also delays the software delivery date and affect software reliability in the field

operation. Domain knowledge has more influence for the entire development process than requirement analysis in terms of software reliability improvement. Therefore, domain knowledge is listed in the top 10 most important factors in this study.

F8, frequency of program specification change, takes the place of f1, program complexity, is the most important environmental factors in this study. Program specifications are generated by software developers based on the customer requirements. First of all, if the frequency of changing program specification is very high, it will certainly lower the software developer's working efficiency. Moreover, frequently changed specifications lead to more code changes and testing, in most cases, will delay software release and affect software robustness and reliability.

There are other reasons that cause frequent specification changes. 1. Different software development crew may understand customer requirements differently. The employee turnover rates nowadays can be as high as 40%, or even higher [144]. The newly hired developers may need to change specifications based on their understanding of customer requirement, which is different with the old employees. At the same time, there is a time lag between the new employees receiving training and they are able to perform as well as the experienced employee. This gap will reduce productivity and service quality [144]. 2. Time-to-market (TTM) has driven the needs to speed up the development process starting from the specification generation to the final product. Sometimes, marketing department promises some features to the customers, however the development team is not able to

deliver. The discrepancy between marketing team and developers would inevitably cause specification changes.

Nowadays, many companies intend to shorten the software release cycle. Beck and Andres [143] claims shorter release cycle can bring more benefits for both the companies and users. However, there is some literatures reported the shorter release cycle makes it impossible to test the adequate configurations for software product [142]. Hence, if the frequency of changing specifications is high in such a short inter-release time, it will lead to a severe software reliability issue for the released product.

Therefore, the impact of the frequency of program specification change on affecting software reliability is the highest amongst all the other environmental factors in this study.

Table 4. 8 Comparison of new ranking and previous ranking

	New ranking		**Ranking from Zhang and Pham [27]**	
Rank	**Factor**	**Description**	**Factor**	**Description**
1	f8	frequency of program specification change	f1	program complexity
2	f22	testing effort	f15	programmer skill
3	f21	testing environment	f25	testing coverage
4	f25	testing coverage	f22	testing effort
5	f1	program complexity	f21	testing environment
6	f15	programmer skill	f8	frequency of program specification change
7	f6	percentage of reused modules	f24	testing methodologies
8	f12	relationship of detailed design to requirement	f11	requirements analysis
9	f24	testing methodologies	f6	percentage of reused modules
10	f19	domain knowledge	f12	relationship of detailed design to requirement

Principle Components of Environmental Factors

PCA is applied to determine the principle components which are able to capture the characteristics of the survey data as provided in Section 4.1.4. Zhang et al. [28] applied factor analysis to investigate common factors which can represent most of the variation of the data. This study selects the top 10 important environmental factors for principle component analysis, f11, requirement analysis, and f5, level of programming technologies, are not included in this study, while Zhang et al. [28] selected the top 11 environmental factors for factor analysis.

Table 4. 9 Comparison of principle components

Principle components			**Common factors from Zhang et al. [28]**		
Principle Component	**Factor**	**Description**	**Common factor**	**Factor**	**Description**
PC1	f25	testing coverage	C1	f21	testing environment
	f21	testing environment		f22	testing effort
	f22	testing effort		f5	level of programming technologies
	f24	testing methodologies		f12	relationship of detailed design and requirements
	f12	relationship of detailed design and requirements	C2	f24	testing methodologies
	f6	percentage of reused code		f25	testing coverage
PC2	f8	frequency of specification change		f6	percentage of reused code
	f19	domain knowledge			
PC3	f15	programmer skills	C3	f11	requirements analysis
	f1	program complexity		f8	frequency of specification change
			C4	f15	programmer skills
				f1	program complexity

Generally speaking, three principle components are retained in this study, while four common factors are revealed in Zhang et al. [28], as described in Table 4.9. The first

principle component is the combination of the first common factor and the second common factor, but f5, level of programming technologies is not included. PCA provides less number of principle components compared with the number of common factors. The principle components in this study also deliver a clear and comprehensive interpretation in terms of investigating strong correlated factors.

Significance Level of Each Development Phase based on Tukey Grouping and SNK Grouping

Tukey method is used in this study to group the four development phases in terms of their mean scores. SNK multiple comparison test was applied in Zhang et al. [28]. The hypothesis for these two methods is that all development phases have the same significant level.

As seen in the left side of Table 4.10, there are three final groups based on Tukey method, yet the previous study [28] only has one final group. Testing phase and General phase are not staying in one group. It is reasonable to be separated into different groups, due to the fact that the environmental factors from the Testing phase have higher occupancies on the top 10 ranking, compared with the factors from other phases from the above analysis. These changes can also be interpreted as the shift of attention of modern software development. Analysis and Design play more important roles in software development than fifteen years ago.

Table 4. 10 Final grouping comparison

New grouping			Grouping from Zhang et al. [28]		
Development phase	**Mean**	**Final grouping**	**Development phase**	**SNK grouping**	**Mean**
Testing	5.225	1	Testing	A	5.430
Analysis and Design	5.034	2	Coding	A	5.350
Coding	4.933	2	General	A	5.240
General	4.722	3	Analysis and Design	A	5.030

Significant Environmental Factors in Each Development Phase

Linear regression backward elimination method is employed in both survey studies. Due to the participants are not the same, each survey feedback may provide different answer for each environmental factor. The response for the linear regression backward elimination method are the improvement of software reliability, which may vary from 0 to 100.

The significant environmental factors comparison is present in Table 4.11. In the General phase, f6, percent of reused modules, is the most significant factor in both analyses, which is also on the top 10 ranking environmental factors. In the Analysis and Design phase, f8, frequency of program specification change, has high significant level in both studies. f12, relationship of detailed design to requirement, is another significant factor. Detailed design will be compared with customer requirements at the end of design phase, inspections will be performed, and misunderstanding part will be removed. Customer satisfaction has gained more attention in such a competitive and fast-changing technology environment and this is ultimately one of the key attributes to the company growth. Hence, f12, relationship

of detailed design to requirement, is the significant environmental factor in the Analysis and Design phase.

The time spent on coding in software development largely depends on programmers' skills since software programming is a complex exercise nowadays. Moreover, how to handle stress is another important issue addressed in this competitive society. The software development is still considered as a complicated yet resource-limited activity; at the same time, customers demand more reliable and safer software products. Thus, testing resource allocation, testing methodologies, and testing coverage are undoubtedly the most significant factors in the Testing phase.

Table 4. 11 Comparison of significant factors in each development phase

New significant factors				**Results from Zhang et al. [28]**			
Phase	**Factor**	**Description**	***p*-value**	**Phase**	**Factor**	**Description**	***p*-value**
General	f4	amount of programming effort	0.0001	General	f1	program complexity	0.0001
	f6	percentage of reused code	0.0130		f6	percentage of reused code	0.0907
Analysis and Design	f8	frequency of program specification change	0.0060	Analysis and Design	f8	frequency of program specification change	0.0635
	f12	relationship of detailed design to requirement	0.0140		f10	design methodology	0.0063
Coding	f15	programmer skill	0.0320		f13	work standards	0.0068
	f18	program workload (stress)	0.0001	Coding	f17	development team size	0.0192
Testing	f23	testing resource allocation	0.0010		f19	domain knowledge	0.0341
	f24	testing methodologies	0.0170	Testing	f21	testing environment	0.0001
	f25	testing coverage	0.0020				

Again, the surveys are conducted from the perspective of the improvement and the impact on software reliability. The ranking of these environmental factors could have a wide variation depends on the opinion of participants. Compared with the studies conducted fifteen years ago [27, 28], most of the significant factors still stay on the top 10 lists, even though the importance orders are slightly different. This confirms that both analyses provide valuable information about what software developers shall focus on during different development stages.

Time Allocation of Each Development Phase in Software Development

The percentages of the time allocation of each development phase in this study are: 22% on the Analysis phase, 20% on the Design phase, 34% on the Coding phase, 24% on the Testing phase. Zhang and Pham's [27] analysis concluded that the percentages of time spent on Analysis, Design, Coding and Testing phase are 25%, 18%, 36%, and 21%, respectively. The time allocation for each development only has slight difference, but also reflect the current emphasis of software development on the Analysis phase with the increasing customer involvement, less coding, and more testing.

4.1.6 Conclusions of Comparison Analysis between Current Study and Previous Findings

We revisit the 32 environmental factors defined in Zhang and Pham [27], reinvestigate their impact on software reliability to provide an up-to-date environmental factors analysis

and development life cycle phase analysis, and compare the results with the previous findings [27, 28] to analyze the difference of the impact of environmental factors on software reliability and each development phase in software development process. Conclusions can be drawn as follows.

1. Most environmental factors listed on the top 10 lists in the previous study also stay on the top 10 lists in this study. However, the order of importance for environmental factors has changed. In this study, the frequency of program specification change is the most significant environmental factor amongst all 32 environmental factors in terms of the impact on affecting software reliability. Additionally, the frequency of program specification change can also be considered in the future software reliability modeling development.

2. Three principle components, PC1, PC2, and PC3, as presented in Table 4.9, are able to represent 69% of the data variation of the collected survey of environmental factors. The three principle components, determined by PCA, are slightly different with the previous findings.

3. The impacts of four development phases on software reliability are different in this study compared with what was discussed in Zhang et al. [28]. The earlier study indicated that four development phase shared equal importance in terms of improving software reliability. In our study, though, Testing phase still has the highest impact on software reliability, Analysis and Design phase has moved up as

the second significant phase. This reflects an emphasis on the front-end analysis and design which will help to straighten the development of software while making a more stringent TTM requirement.

4. In response to the improvement on software reliability, the significant factors in each development phase have different orderings as compared with the previous study based on the backward elimination method.

5. The time allocation for each development phase, Analysis, Design, Coding, and Testing are slightly different as compared with the previous study.

4.2 Environmental Factors in Multi-Release Software Development

4.2.1 Research Motivation

As software systems are more deeply embedded in our everyday life, the dependence of our modern society on complex, intelligent and reliable large-scale software systems is rapidly growing than ever [145]. Meanwhile, the possibility of carrying more faults in the large-scale software systems is higher than decades ago. Software failures are increasingly common in the filed environment given the increasing complexity of software products [146].

To continually align with the fast-changing customer's requirements and provide reliable products to the market, most companies will release multiple versions of the software

product since it is unlikely to deliver all the features in a single release and satisfy all the constraints within the limited resources [1, 46, 48, 147].

The principles of multi-release software are: adding new features in the next release and fixing the remaining faults from previous release due to the fact that bug-free software product is not likely to be delivered in any release [42]. Software multiple releases not only make company easily balance the competing stakeholder's demand and benefits, but also increase reliability and customer satisfaction level during each release [40, 41].

The resources and constraints for the development of multi-release software are different. Software development team needs to select the corresponding features included in the next release with respect to customers' feedback and market requirements. Since software will be released in increments for multi-release software, thus, coupling this concept with other principles such as continuous unit testing and pair programming will better arrange the cost distribution [148].

During the past decades, software version planning and release has been studied by many researchers. Szoke [149] developed a staged-delivery global optimized model for agile release planning. Li et al. [150] proposed a multi-objective optimization technique to optimize three main objectives with respect to cost, revenue, and uncertainty for robust next release problem. Etgar et al. [151] explored several optimization approaches to determine the content and release date for each release to provide optimal net present value.

However, all the past research related to release planning or multi-release software reliability modeling did not investigate the impact of environmental factors on software reliability in the development of multi-release software. It is very pragmatic yet interesting to investigate what are the impact of environmental factors affecting software reliability in the development of multi-release software.

Therefore, it is plausible to conduct a new study to investigate the impact of these environmental factors on software reliability for multi-release software development and compare the differences between single-release software and multi-release software.

4.2.2 Objectives

As discussed in the previous section, the emphasis of the development process for multi-release software is different with single-release software. For example, how to select the desirable features in which release, and how to determine the removal percentage of the detected software faults in each release. This study aims to revisit the environmental factors in terms of their impact on software reliability for multi-release software development. This study is carried out by conducting a survey of environmental factors affecting software reliability for the next release's development.

Firstly, we need to investigate the significant environmental factors affecting reliability in the development of multi-release software. Secondly, the correlation between environmental factors. Is it possible to reduce the dimension of those variables to provide concise and sound information for software researchers and practitioners?

Moreover, the significant environmental factors in each development phase and the significance level of each development phase are investigated to provide a helpful time/resource allocation matrix for software development team. Thirdly, we also compare the significant environmental factors during the whole development process, principle components, significant environmental factors in each development phase, and significance level of each development phase between the development of single-release software and multi-release software.

At the end of this study, other statistical methods in terms of variable selection is also applied in this dissertation to provide an insightful matrix for readers according to their selection priority. Software practitioners can choose the results coming from different methodologies based on their requirements for the reference of multi-release software development.

4.2.3 Data Collection

To align with the latest survey data analysis focused on the significant environmental factors on affecting software reliability during the development of single-release software [29] and maintain the similarity for the comparison of environmental factors, we still use the same 32 environmental factors firstly defined in Zhang and Pham [27].

Forty-five survey responses are collected from various industries including computer software, internet, banking, semiconductor, online retailing, financial service, IT service &

consulting and research institution. Participants ranked the environmental factors based on the impact of each environmental factor on software reliability during the development of multi-release software. The software development experiences, software applications, years of experience, and job title are also provided from the participants. All the participants are currently working in IT Department in different industries or working on software development in high technology companies in favor of the validity and reliability of the survey response. Hence, we are expecting the survey response data is sound, valid, and reliable.

4.2.4 Findings and Results

Environmental Factor Analysis

Relative Weighted Method

We are using the same method as described in Section 4.1.4.1.1, the summation for all survey responses ranges from 87 to 191. Equations (4.1) and (4.2) are used for the normalization of each score in the original survey and final weight calculation, respectively. Note that L is the number of complete form, which is 45 in this multi-release study.

As a result, we are able to calculate the normalized weight for each environmental factor and find their ranking, as presented in Table 4.12. Program complexity is the most important factor, which has higher impact on affecting software reliability for the development of next release than other environmental factors. In general, the higher

ranking of an environmental factor in Table 4.12, the higher impact of this environmental factor on software reliability in the development of multi-release software.

Table 4. 12 Environmental factors ranking by relative weighted method

Rank	Factor	Description	Normalized weight
1	f1	program complexity	0.0390879087
2	f11	requirement analysis	0.0384558837
3	f8	frequency of program specification change	0.0368853086
4	f22	testing effort	0.0361218996
5	f12	relationship of detailed design to requirement	0.0355715603
6	f4	amount of programming effort	0.0353399281
7	f25	testing coverage	0.0342799173
8	f18	program workload (stress)	0.0342002348
9	f6	percentage of reused modules	0.0341796701
10	f5	level of programming technologies	0.0340814368
11	f15	programmer skills	0.0337374978
12	f21	testing environment	0.0337126257
13	f23	testing resource allocation	0.0330611582
14	f10	design methodology	0.0329230683
15	f24	testing methodologies	0.0328771780
16	f19	domain knowledge	0.0327225665
17	f27	documentation	0.0322377808
18	f3	difficulty of programming	0.0320263075
19	f26	testing tools	0.0312195731
20	f14	development management	0.0311128846
21	f16	programmer organization	0.0306560899
22	f2	program categories	0.0303179007
23	f20	human nature	0.0300672895
24	f17	development team size	0.0293613617
25	f9	volume of program design documents	0.0291258278
26	f13	work standards	0.0286965882
27	f28	processors	0.0248516495
28	f32	system software	0.0247222791
29	f29	storage devices	0.0246144443
30	f7	programming language	0.0226114305
31	f30	input/output devices	0.0211775686
32	f31	telecommunication devices	0.0199631816

Principle Component Analysis of Environmental Factors

In order to keep the similarity of this study with the previous study, described in Section 4.1, and compare what are differences between these two studies regarding single/multi release software, we are using the same methodology to analyze environmental factors.

PCA is applied to reduce the dimension of the environmental factors for multi-release software survey study. The top 10 most important environmental factors based on the relative weighted method are selected to perform PCA.

The top 10 environmental factors are f1, f11, f8, f22, f12, f4, f25, f18, f6, and f5. We notice that the ranking is quite different compared with the findings from Section 4.1.4.1.2. The group of significant environmental factors affecting software reliability is also different between the development of multi-release software and single-release software. The covariance matrix of 10 × 10 will be calculated to obtain the eigenvalues and eigenvectors. Eigenvalues, from the highest to the lowest, are listed in Table 4.13 in terms of their impact level on the principle components.

As seen from Table 4.13, four principle components will be retained, which are PC1, PC2, PC3, and PC4. The first four principle components are able to address over 75% of the data variation. 38.2% of the data variation is explained by the first principle component; about 13% of the data variation is explained by the second principle component. Likewise, the third and the fourth principle component are able to represent 12.7% and 11.1% of the data

variation, respectively. All other principle components only represent less than 10% of the variation, respectively. For the principle components PC5, PC6, PC7, PC8, PC9, and PC10, the compensation of adding one more principle component is considerately high according to the contributions that they bring in.

Table 4. 13 Eigenvalue and proportion of the principle components

Component	Eigenvalue	Proportion	Cumulative proportion
PC1	3.9836	0.3821	0.3821
PC2	1.2567	0.1303	0.5124
PC3	1.1596	0.1273	0.6397
PC4	1.0539	0.1113	0.751
PC5	0.8239	0.0832	0.8342
PC6	0.6142	0.0654	0.8996
PC7	0.4195	0.0429	0.9425
PC8	0.2871	0.0241	0.9666
PC9	0.2418	0.0192	0.9858
PC10	0.1596	0.0141	1.0000

After retaining the principles components of environmental factors in the development of multi-release software product. The next question we want to discuss is the contribution of each environmental factor to each principle component. Specifically, each principle component is the linear combination of the environmental factors.

Table 4.14 presents the environmental factors which are strongly-correlated with the principle components. The loading coefficient is utilized to measure the contribution of each environmental factor to the principle components. Relationship of detailed design to requirement is highly correlated with the first principle component; amount of

programming effort is highly correlated with the second principle component; program complexity and frequency of specification change are highly correlated with the third and fourth principle component, respectively.

Table 4. 14 Principle component associated with strong-correlated environmental factors

Component	Factor	Description	Loading coefficient
PC1	f12	relationship of detailed design to requirement	-0.399
	f25	testing coverage	-0.383
	f22	testing effort	-0.383
	f11	requirement analysis	-0.323
	f5	level of programming technologies	-0.294
PC2	f18	program workload (stress)	0.520
	f4	amount of programming effort	-0.308
PC3	f6	percentage of reused modules	-0.526
	f1	program complexity	0.509
PC4	f8	frequency of program specification change	0.629

Hypothesis Testing

There is a common question that software practitioners often bring up: do these 32 environmental factors have the same impact on software reliability in the development of multi-release software? In this section, one-way ANOVA is applied to compare the significance level of those environmental factors. From the R output, F-statistics for the environmental factors analysis is 8.342 with a *p*-value less than $2e^{-16}$; while the value of $F_{31,1408}$ is 1.459. Statistically, the null hypothesis is rejected when *F* value is larger than *F* critical value of 1.459. Therefore, the significance level of each environmental factor on

software reliability assessment is different during the development of multi-release software.

Correlation Analysis

Table 4.15 presents the Pearson's *r* associated with each pair of environmental factors in the development of multi-release software. Due to the limitation of the page, we only choose the absolute value of Pearson's *r* is larger than 0.45 to present.

Table 4. 15 Correlation analysis for multi-release software survey data

Factor	Description	Correlated factors		Pearson's *r*
f1	program complexity	f5	level of programming technologies	0.452
f11	requirement analysis	f22	testing effort	0.574
		f29	storage devices	-0.565
		f5	level of programming technologies	0.561
		f12	relationship of detailed design to requirement	0.553
		f28	processors	-0.539
		f23	testing resource allocation	0.485
		f25	testing coverage	0.477
f22	testing effort	f25	testing coverage	0.764
		f23	testing resource allocation	0.684
		f26	testing tools	0.679
		f21	testing environment	0.671
		f27	documentation	0.632
		f5	level of programming technologies	0.627
		f17	development team size	0.584
		f14	development management	0.573
f12	relationship of detailed design to requirement	f10	design methodology	0.554
		f11	requirement analysis	0.553
		f6	percentage of reused modules	0.496
f4	amount of programming effort	f3	difficulty of programming	0.646

		f15	programmer skills	0.524
		f7	programming language	0.476
f25	testing coverage	f24	testing methodologies	0.801
		f22	testing effort	0.764
		f21	testing environment	0.712
		f26	testing tools	0.704
		f23	testing resource allocation	0.690
		f17	development team size	0.624
		f14	development management	0.570
		f13	work standards	0.550
		f27	documentation	0.539
		f5	level of programming technologies	0.534
		f11	requirement analysis	0.477
		f2	program categories	0.463
		f32	system software	0.451
f18	program workload (stress)	f20	human nature	0.497
		f19	domain knowledge	0.491
		f17	development team size	0.467
f6	percentage of reused modules	f12	relationship of detailed design to requirement	0.496
		f5	level of programming technologies	0.484
f5	level of programming technologies	f22	testing effort	0.627
		f11	requirement analysis	0.561
		f25	testing coverage	0.534
		f26	testing tools	0.510
		f6	percentage of reused modules	0.484
		f23	testing resource allocation	0.474
		f1	program complexity	0.452
f15	programmer skills	f7	programming language	0.647
		f19	domain knowledge	0.603
		f16	programmer organization	0.591
		f4	amount of programming effort	0.524
f21	testing environment	f25	testing coverage	0.711
		f22	testing effort	0.671
		f23	testing resource allocation	0.643
		f17	development team size	0.627
		f26	testing tools	0.570
		f20	human nature	0.538
		f24	testing methodologies	0.475
f23	testing resource allocation	f26	testing tools	0.842

		f25	testing coverage	0.690
		f22	testing effort	0.685
		f24	testing methodologies	0.668
		f21	testing environment	0.643
		f17	development team size	0.558
		f14	development management	0.555
		f9	volume of program design documents	0.492
		f11	requirement analysis	0.485
		f13	work standards	0.479
		f5	level of programming technologies	0.474
f10	design methodology	f14	development management	0.604
		f9	volume of program design documents	0.563
		f12	relationship of detailed design to requirement	0.554
f24	testing methodologies	f25	testing coverage	0.800
		f13	work standards	0.715
		f14	development management	0.694
		f23	testing resource allocation	0.668
		f22	testing effort	0.625
		f26	testing tools	0.621
		f9	volume of program design documents	0.613
		f17	development team size	0.599
		f27	documentation	0.564
		f32	system software	0.508
		f31	telecommunication devices	0.470
f19	domain knowledge	f29	storage devices	0.741
		f7	programming language	0.653
		f16	programmer organization	0.631
		f15	programmer skills	0.603
		f20	human nature	0.564
		f17	development team size	0.513
		f18	program workload (stress)	0.491
f27	documentation	f14	development management	0.642
		f22	testing effort	0.632
		f24	testing methodologies	0.564
		f13	work standards	0.551
		f25	testing coverage	0.539
		f31	telecommunication devices	0.512
		f17	development team size	0.493
f3	difficulty of programming	f4	amount of programming effort	0.646
		f7	programming language	0.538
f26	testing tools	f23	testing resource allocation	0.843
		f25	testing coverage	0.703
		f22	testing effort	0.679

		f24	testing methodologies	0.621
		f21	testing environment	0.570
		f5	level of programming technologies	0.510
		f32	system software	0.492
		f17	development team size	0.459
		f2	program categories	0.451
f14	development management	f24	testing methodologies	0.694
		f27	documentation	0.642
		f10	design methodology	0.604
		f22	testing effort	0.573
		f23	testing resource allocation	0.554
		f13	work standards	0.498
f16	programmer organization	f19	domain knowledge	0.631
		f31	telecommunication devices	0.611
		f29	storage devices	0.596
		f15	programmer skills	0.592
		f17	development team size	0.522
		f28	processors	0.504
f2	program categories	f13	work standards	0.715
		f9	volume of program design documents	0.472
		f25	testing coverage	0.463
		f26	testing tools	0.451
f20	human nature	f19	domain knowledge	0.564
		f7	programming language	0.563
		f21	testing environment	0.538
		f18	program workload (stress)	0.497
f17	development team size	f32	system software	0.663
		f21	testing environment	0.627
		f24	testing methodologies	0.599
		f22	testing effort	0.584
		f31	telecommunication devices	0.572
		f23	testing resource allocation	0.560
		f30	input/output devices	0.560
		f15	programmer skills	0.550
		f16	programmer organization	0.522
		f19	domain knowledge	0.513
		f27	documentation	0.493
		f14	development management	0.483
		f7	programming language	0.477
		f18	program workload (stress)	0.467
		f26	testing tools	0.459

f9	volume of program design documents	f2	program categories	0.472
		f10	design methodology	0.563
		f13	work standards	0.584
		f14	development management	0.500
		f23	testing resource allocation	0.492
		f24	testing methodologies	0.613
f13	work standards	f2	program categories	0.715
		f24	testing methodologies	0.715
		f9	volume of program design documents	0.584
		f27	documentation	0.551
		f25	testing coverage	0.550
		f14	development management	0.498
		f23	testing resource allocation	0.479
f28	processors	f29	storage devices	0.887
		f30	input/output devices	0.668
		f31	telecommunication devices	0.561
		f16	programmer organization	0.506
		f19	domain knowledge	0.469
		f7	programming language	0.459
		f11	requirement analysis	-0.539
f32	system software	f17	development team size	0.663
		f30	input/output devices	0.654
		f31	telecommunication devices	0.650
		f25	testing coverage	0.571
		f24	testing methodologies	0.508
		f26	testing tools	0.492
		f14	development management	0.474
		f22	testing effort	0.467
f29	storage devices	f28	processors	0.886
		f30	input/output devices	0.683
		f31	telecommunication devices	0.619
		f16	programmer organization	0.596
		f7	programming language	0.503
		f19	domain knowledge	0.471
		f11	requirement analysis	-0.565
f7	programming language	f4	amount of programming effort	0.476
		f15	programmer skills	0.647
		f17	development team size	0.477
		f19	domain knowledge	0.653
		f20	human nature	0.563
		f28	processors	0.459
		f29	storage devices	0.503
		f30	input/output devices	0.467

f30	input/output devices	f31	telecommunication devices	0.829
		f29	storage devices	0.683
		f28	processors	0.668
		f32	system software	0.654
		f17	development team size	0.560
		f16	programmer organization	0.497
		f7	programming language	0.468
f31	telecommunication devices	f30	input/output devices	0.839
		f32	system software	0.650
		f29	storage devices	0.619
		f16	programmer organization	0.612
		f17	development team size	0.572
		f28	processors	0.561
		f14	development management	0.488

Development Phase Analysis

Significance Level of Each Development Phase

We are interested in whether these five development phases have the same impact on software reliability during the development of multi-release software. Thereafter the comparison between the development of single-release software and multi-release software in terms of the significance level of each development phase will be drawn later. One-way ANOVA method is utilized to compare the significance level. The hypothesis is presented as follows.

$$H_0: \mu_{General} = \mu_{Analysis\&design} = \mu_{Coding} = \mu_{Testing} = \mu_{Hardware\ system}$$

$$H_a: not\ all\ equal$$

From the R output, the *F*-statistic for the development phase is 35.98 with *p*-value less than $2e^{-16}$. The *F* critical value, $F_{4,1435}$, is 2.378 from *F* table, which is much less than 35.98. Therefore, during the development of multi-release software, the impact of each development phase on software reliability is different.

Figure 4.1 illustrates the boxplot of each development phase. General phase, Analysis and Design phase, Coding phase, and Testing phase have the similar mean value. From the output of Tukey multiple comparison of mean, we consider the General, Analysis and Design, Coding and Testing as a group. The grouping distribution is different with the previous findings from Section 4.1, for which, we will discuss later.

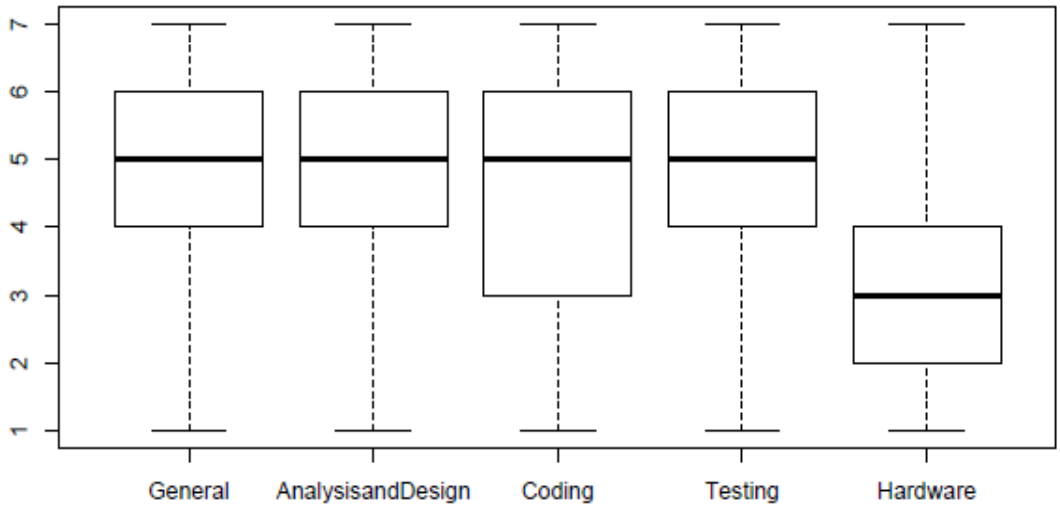

Figure 4. 1 Boxplot for each development phase

Significant Environmental Factors in Each Development Phase

We also use the backward elimination to identify the significant environmental factors in each phase during multi-release software development. For the variable selection process, we use *Reliability* obtained from survey as response variable and the environmental factors in each phase as explanatory variables. For instance, in the General phase, f1, f2, f3, f4, f5, f6, and f7 are explanatory variable, the *Reliability* value is obtained from survey.

Table 4.16 reveals the significant environmental factors in each development phase for multi-release software. Most significant factors in the General, Analysis and Design phase also stay on top 10 ranking environmental factors calculated by relative weighted method. It is very interesting to see human nature is one of the significant factors in the Coding phase. Human nature refers to the characteristics of software developer, such as the ability to avoid making mistakes. As the software release cycle becomes shorter, the software development cycle is also getting shorter accordingly. In such a scenario, the consequence of human nature on affecting software reliability is getting more attention in the development of multi-release software.

Table 4. 16 Significant factors in each development phases for multi-release software

Development phase	**Significant factor**	**Description**	**Parameter estimate**	***p*-value**
General	f4	amount of programming effort	-0.074	0.017500
	f5	level of programming technologies	0.090	0.006600
	f7	programming language	0.049	0.023000
Analysis and Design	f11	requirement analysis	0.080	0.002600
Coding	f20	human nature	0.092	0.000006
Testing	f23	testing resource allocation	0.099	0.002500

4.3 Comparisons between Single-Release and Multi-Release Software

4.3.1 Ranking of Environmental Factors

There are 60% environmental factors on the top 10 ranking for single-release software still stay on the top 10 ranking for multi-release software, however, the order of importance has changed. The detailed comparison of the top 10 ranking between single-release software and multi-release software is presented in Table 4.17.

Table 4. 17 Comparison of ranking between multi-release and single-release

	Multi-release software		Single-release software	
Rank	**Factor**	**Description**	**Factor**	**Description**
1	f1	program complexity	f8	frequency of program specification change
2	f11	requirement analysis	f22	testing effort
3	f8	frequency of program specification change	f21	testing environment
4	f22	testing effort	f25	testing coverage
5	f12	relationship of detailed design to requirement	f1	program complexity
6	f4	amount of programming effort	f15	programmer skills
7	f25	testing coverage	f6	percentage of reused modules
8	f18	program workload (stress)	f12	relationship of detailed design to requirement
9	f6	percentage of reused modules	f24	testing methodologies
10	f5	level of programming technologies	f19	domain knowledge

Most software companies frequently release new updated versions to stay competitive in the market. Depending on the purpose of the new release, some version may be released on a frequent basis, for instance, fix urgent faults in the previous software release; some version may be released on a long-term basis since it will bring in major updates [152].

Additionally, large software companies like Ericsson increasingly apply the principles of agile and lean software development in an iterative manner to quickly respond to customers' feedback [153]. It is understood that the requirements analysis and design and have the customers involved take longer time for the multi-release software development. The testing is relatively on a smaller scope and takes shorter time compared with single-release software. Likewise, this study also supports this point of view. For example, as seen in Table 4.17, there are four environmental factors are coming from the Testing phase for single-release software, while only two environmental factors are coming from the Testing phase for multi-release software.

Environmental factors, f11, f4, f18, and f5, are revealed in the top 10 ranking for the first time. f11, f4, f18, f5 refers to requirement analysis, amount of programming effort, program workload (stress), and level of programming technologies, respectively. Environmental factor, f11, requirement analysis, usually comes from customers, is the fundamental factor for the following design and coding work. The specifications that software developers generate mostly based on the requirements from customers. It is also known that continuous delivery of customer requirement is critical for software development company in this very market-driven environment [153], thus, it is reasonable that requirement analysis has significant impact on affecting software reliability in the development of multi-release software. f4 and f5 are coming from the General phase and both stay in the top 10 ranking in this study. f4, amount of programming effort, which may directly improve the efficiency and reliability of the product. f5 refers to the level of programming technologies. The programming technologies are classified into four

categories: design techniques, documentation techniques, programming techniques and development techniques. f5 represents a comprehensive consideration for software development team. Due to the short release of the software and the quick response to customer requirements, programming technologies have more significant impact on affecting the quality and reliability of multi-release software.

4.3.2 Principle Components of Environmental Factors

Four principle components are obtained in multi-release software survey study, represented over 75% variation for the collected data; three principle components were generated in the previous findings for single-release software, explained 69% variation of the collected data. The detailed comparisons are explained in Table 4.18. In sum, software developers need to choose the corresponding results according to the product requirements.

4.3.3 Significance Level of Each Development Phase

Tukey method is utilized on both studies to compare the four development phases. The comparison of significance level of each development phase is illustrated in Table 4.19. The null hypothesis for both studies is all development phases have the same significance level in terms of their impact on software reliability. Three groups are categorized for single-release software development. Testing phase is the most important phase on affecting software reliability; Analysis, Design, and Coding phases have the same significance level; General phase has the least significance level. However, in the development of multi-release software, all of four development phases have the equal

impact on affecting software reliability. As we discussed earlier, the application of agile and lean software development method results in short iteration and short release to quickly respond customers' feedback. Thus, the significance level of the Testing phase in the development of multi-release software is not as significant as it in the development of single-release software.

Table 4. 18 Comparisons of principle components between single-release and multi-release software

Principle components for multi-release software			**Principle components for single-release software**		
Principle components	**Factor**	**Description**	**Principle components**	**Factor**	**Description**
PC1	f12	relationship of detailed design to requirement	PC1	f25	testing coverage
	f25	testing coverage		f21	testing environment
	f22	testing effort		f22	testing effort
	f11	requirement analysis		f24	testing methodologies
	f5	level of programming technologies		f12	relationship of detailed design to requirement
PC2	f18	program workload (stress)		f6	percentage of reused modules
	f4	amount of programming effort	PC2	f8	frequency of program specification change
PC3	f6	percentage of reused modules		f19	domain knowledge
	f1	program complexity	PC3	f15	programmer skills
PC4	f8	frequency of program specification change		f1	program complexity

4.3.4 Significant Environmental Factors in Each Development Phase

Backward elimination method is applied in both studies to extract the significant environment factors of each development phase for single/multi release software. The comparisons are illustrated in Table 4.20.

Most of the significant factors in each development phase are also on the top 10 ranking environmental factors in these two studies. The significant factors in each development phase are different between single-release software and multi-release software. For multi-release software, the significant factors in the General phase are f4, amount of programming effort, f5, level of programming technologies, and f7, programming language; while the significant factors in the General phase for single-release software are f4, amount of programming effort and f6, percentage of reused modules. Only f4, amount of programming effort, is the significant factor in the General phase in both studies. Analysis and Design phase have the similar choice of significant factors in these two studies since f11, requirement analysis, and f12, relationship of detailed design to requirement, both focus on the alignment of customer requirements.

In the Coding phase, the interesting finding is that multi-release software development more focuses on human nature. Human nature refers to the developers' characteristics, including the ability to avoid making working mistakes, careless work omission. Given the short-release cycle of multi-release software, on-time project delivery is very critical thus any working mistake could affect the coding efficiency and delay the product release.

In the Testing phase, testing resource allocation is the significant factor in both studies. Technical-related environmental factors like testing methodology and coverage are more emphasized in the development of single-release software. In sum, the selection of the

significant factors in these two studies reflects the difference of the development methodology.

Table 4. 19 Comparison of final grouping

Grouping (multi-release software)			**Grouping (single-release software)**		
Development phase	**Mean**	**Final grouping**	**Development phase**	**Mean**	**Final grouping**
General	4.540	1	General	5.225	1
Analysis and Design	4.641	1	Analysis and Design	5.034	2
Coding	4.470	1	Coding	4.933	2
Testing	4.707	1	Testing	4.722	3

Table 4. 20 Comparison of significant factors in each development phase

Significant factors for multi-release software				**Significant factors for single-release software**			
Phase	**Sig. factor**	**Description**	***p*-value**	**Phase**	**Sig. factor**	**Description**	***p*-value**
General	f4	amount of programming effort	0.01750	General	f4	amount of programming effort	0.0001
	f5	level of programming technologies	0.00660		f6	percentage of reused module	0.0130
	f7	programming language	0.02300	Analysis and Design	f8	frequency of program specification change	0.0060
Analysis and Design	f11	requirement analysis	0.00260		f12	relationship of detailed design to requirement	0.0140
Coding	f20	human nature	0.00001	Coding	f15	programmer skills	0.0320
					f18	program workload (stress)	0.0001
Testing	f23	testing resource allocation	0.00250	Testing	f23	testing resource allocation	0.0010
					f24	testing methodologies	0.0170
					f25	testing coverage	0.0020

4.4 Other Statistical Learning Method to Select Environmental Factors

We apply lasso regression in this section and compare with the results from other variable selection methods for multi-release survey data. Why do we choose lasso regression to fit the model instead of least squares? It is the trade-off between the variance and bias for the sake of the model prediction accuracy and interpretability [154]. The least squares estimates tend to have low bias if the relationship between the response and the predictors is linear. If $n \gg p$, that is, if the number of observations, *n*, is much larger that the number of variables, *p*, the least squares tends to have low variance and performs well on the test observations. However, if *n* is not larger than *p*, then variability can be occurred in the least squares estimates. Overfitting and poor prediction may also appear in the future observation, which are not used in the model training [154]. We have 45 survey responses while 32 environmental factors are presented for multi-release survey data. The number of observations is not much larger than the number of variables. To reduce the variance at the cost of increasing bias, we often shrink the estimated coefficients. Therefore, lasso regression, a shrinkage regression method, in which some coefficients will be estimated towards zero, hence, is applied to perform variable selection.

As the tuning parameter λ is sufficiently large, some coefficients can be exactly equal to zero in lasso regression. By applying lasso regression, the non-zero-coefficient variables are f11, f20, f23, represent requirement analysis, human nature, and testing resource allocation, respectively. They are considered as the significant predictors on affecting reliability in the development of multi-release software from lasso regression.

Different variable selection methods may provide different sets of significant factors, but similarity could still exist in these sets. For instance, relative weighted method applied in all 32 environmental factors and the backward elimination method applied in each development phase both conclude f11, requirement analysis, is an important environmental factor. Moreover, f20, human nature, and f23, testing resource allocation, are selected as significant factor in each development phase addressed in Section 4.2. There are also some drawbacks on selecting variables using lasso regression. We understand that other variables which are highly-correlated with the selected variables may be eliminated in the selection process based on the mathematical theory, however, some of them may possess more practical meaning than the selected variables. From this point of view, Table 4.15 provides detailed explanation on their correlations.

4.5 Conclusions of Environmental Factor Studies in Development of Single-Release and Multi-Release Software

This chapter aims to investigate and compare the impact of environmental factors on affecting software reliability in the development of single-release and multi-release software. Comparisons are concluded as follows.

1. 60% similarity of environmental factors listed on the top 10 ranking in these two studies. However, the order of importance has changed. The importance level of the Testing phase in the development of multi-release software is not as significant as in single-release software because of the increasing application of agile software development method. Correspondingly, factors related to the Testing phase are not

important as before, while factors related to customer requirement/feedback analysis are getting more attention in the development of multi-release software.

2. Given the emphasis of multi-release software development is shifting more to the Analysis and Design phase, all four development phases in multi-release software are categorized as one group, while three groups are retained for single-release software.

3. Significant factors in each development phase are not the same for single and multi-release software. The weight of testing-related and programmer-skill-related factors decreases, as the weight of customer-related factors increases.

4. Other statistical learning method, e.g., lasso regression, is applied to analyze thee multi-release survey data. Requirement analysis, human nature, and testing resource allocation are considered as the significant predictors on predicting reliability for the development of multi-release software. Different variable selection method has its own limitation, researchers and practitioners will choose the corresponding results depends on their applications.

In this chapter, survey responses from different industries/projects/regions are considered as one group to investigate the significance level of environmental factors on affecting software reliability in the development of single-release and multi-release software. Several research directions can be addressed in the future research. For example, the

comparison of the findings amongst industries, projects, or even regions can be further discussed since the variance exists in those groups are considerably worthy to be investigated. Incorporating single/multiple significant environmental factor(s) in software reliability modeling is also plausible.

CHAPTER 5

SOFTWARE RELIABILITY MODELS CONSIDERING FAULT DEPENDENCY AND IMPERFECT FAULT REMOVAL

5.1 Research Motivation

Software technologies have been greatly adopted in many critical applications, such as air traffic control system, national security defend system, network/grid system, and consumer appliance [157]. Given such an increasing expectation on the performance of software-related product, the reliability and quality of software product have been studied by many practitioners and researchers since 1970s. Thus, a great number of attempts and approaches are proposed to measure software reliability and reliability improvement during the testing/operation phase in the past four decades, as reviewed in Chapter 2. NHPP is considered as one of the most effective mathematical tools to model software fault growth process since software faults come on a discrete-time scale.

The common assumptions for the existing software reliability growth models are summarized as follows. (1) Software faults are mutually independent. (2) The program only has one type of software fault; hence, the difficulty level of detecting software faults is the same. In other words, software detection rate is always the same without considering different fault type. (3) The detected software faults are perfectly removed during the testing phase. It is unlikely to remove all the faults during the testing phase for the modern software product in consideration of the limited resource, estimated risk, constrained schedules, and multi-release consideration [39]. The above assumptions are proposed

mainly because of the simplicity consideration of the mathematical modeling for fault detection process. However, it may not be realistic.

As discussed in Chapter 2, a typical software program usually suffers from more than one type of software fault [117 - 123]. Different fault classes are categorized by practitioners and researchers to describe the characteristics of software faults that cause failures during testing and operation phase [117, 124]. Some fault classes are discussed as follows.

First, solid (hard) faults, often corresponding to Bohrbugs, and soft (elusive) faults, referring to Mandelbugs. Even Bohrbug and Mandelbug are discussed by many practitioners. we still do not have consistent definitions given in the literature for Bohrbug and Mandelbug [120 - 121]. The definitions given in this dissertation are referred from Grottke et al. [117]. Bohrbug is defined as an easily isolated fault that manifests consistently under a well-defined set of condition due to its activation is lack of complexity, while Mandelbug is associated with complex activation and/or error propagation behavior. The “Complexity” in Mandelbug may be triggered by the interaction of software application and its system environment (hardware, operation system, and other applications), the influence of the operation sequences, the influence of inputs, and the time lag between fault activation and failure occurrence [122, 123].

Secondly, related faults and independent faults. Laprie et al. [118] stated that software faults are either related or independent. Related faults manifest themselves as similar errors and lead to the common-mode failures, while independents faults usually cause distinct

errors and separate failures. In other words, the failure mode of related faults is similar, but the failure mode of independent faults is distinct.

Thirdly, Lapire [126] also identified and discussed the limits and challenges in the dependability of computer systems in terms of the fault class, such as physical faults, design faults, and interaction faults.

Given the practical consideration of different types of software faults discussed above, and the lack of software reliability models incorporating both software fault dependency and imperfect fault removal, therefore, we include both software fault dependency and imperfect fault removal in a NHPP software reliability model in this chapter.

Two main concerns: ***Software Fault Dependency*** and ***Software Imperfect Fault Removal***, are included in this chapter. A brief discussion regarding these two topics are presented below.

Software Fault Dependency

As discussed above, different types of software faults are defined in the literature based on different criteria. For example, solid faults and soft faults are defined with respect to the existence of "complexity". "Complexity" is caused by a time lag between the fault activation and failure occurrence, or the influence of interaction between software and application environment [117]. Independent faults and related faults are also defined according to the independence of fault manifestation. Those literatures are commonly

known in software engineering area. But when it comes to develop software reliability growth model, software fault dependency is often neglected for the sake of model simplicity. Only a few literatures address fault dependence/different types of software faults in the modeling, as discussed in Chapter 2.

In order to explain the fault dependency, we adopt an example of correct and faulty program from Huang and Lin [132], as revealed in Figure 5.1. The left side of Figure 5.1 is the correct program, while the right side of Figure 5.1 is the faulty program.

There are two misusing operators in the faulty program, located in line 30 and line 105, respectively. The fault located in line 30 in the faulty program is caused by a misusing operator. After executing line 30, it will lead to the wrong input for line 42. It is expected that line 105 will not print out the right output since another fault exists in line 105. Besides removing the fault in line 105, the fault existing in line 30 still needs to be corrected. Thus, in the segment discussed below, fault located in line 30 is Type I (independent) fault, while fault located in line 105 is Type II (dependent) fault based on the definition given in this study.

The detailed discussion for Type I and Type II fault will be described in Section 5.3. Note that in the proposed software reliability model for one-phase debugging process, we assume Type I (independent) faults have been removed in the preliminary testing, thus, only consider Type II (dependent) faults in the program for the sake of simplification. The

proposed software reliability model for two-phase debugging process consider both types of faults in the program.

Correct program ***Faulty program***

Line number	**Code**	**Line number**	**Code**	**Remarks**
:	:	:	:	
30	key = key % 5	30	key = key / 5	#misusing operator
:		:		
42	while (key < 5)	42	while (key < 5)	
43	{	43	{	
:	:	:	:	
70	count = count * 2	70	count = count * 2	
:	:	:	:	
80	}	80	}	
:		:		
105	print (count % 5)	105	print (count / 5)	#misusing operator
106	print ("end")	106	print ("end")	

Figure 5. 1 Example of Type I (independent) fault and Type II (dependent) fault

Software Imperfect Removal

In practice, it is unlikely to remove all the detected faults in software testing phase due to the limited resource, estimated risk, constrained schedules, and multi-release consideration. These literatures [106, 158 – 161] incorporated fault removal efficiency in the software reliability models. We also consider imperfect fault removal in this chapter. Specifically, detected faults cannot be perfectly removed in software testing phase.

5.2 Proposed Software Reliability Model for One-Phase Debugging Process

As discussed in Figure 5.1, we define two types of software faults, Type I (independent) fault and Type II (dependent) fault. For the sake of model simplification, we first consider one-phase debugging process to model the failure growth. We only consider Type II faults in one-phase debugging process and assume all Type I faults have been removed in the preliminary testing phase. Thus, as seen in Figure 5.2, we assume at $t = 0$, the detection of Type II faults starts.

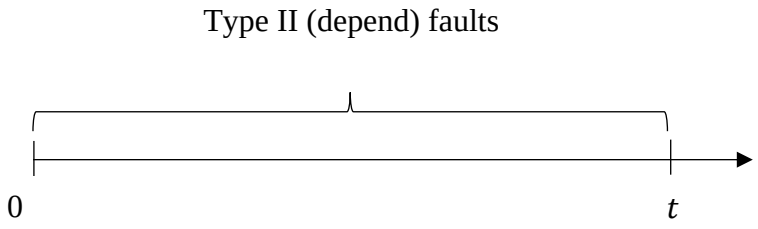

Figure 5. 2 One-phase debugging process

The assumptions for the proposed one-phase software reliability model are described as follows.

(1) Software detection follows a NHPP process.

(2) This is a fault-dependent detection process.

(3) Fault detection is a process of a learning curve phenomenon.

(4) Fault is not removed perfectly upon detection.

(5) The debugging process may introduce new errors into software. This is an imperfect debugging process, but the maximum faults contained in the software is L.

(6) The software failure intensity $\lambda(t)$ is explained as the percentage of the removed errors in the software product.

(7) The non-removed software error rate is assumed to be a constant.

The notations for this section are given as follows.

L	Maximum number of software faults in the program
b	Asymptotic unit of software fault detection rate
β	Shape parameter of the learning curve
c	Non-removable error rate per unit of time
m_0	Expected number of software failures at time *t = 0*
$m(t)$	Expected number of software failures by time t
$b(t)$	Fault detection rate function
$c(t)$	Non-removable fault rate function due to the limitation of testing resource, the skill and experience of the programmer, and multi-release consideration for software organization

A NHPP software reliability model with fault-dependent detection, imperfect fault removal, and the maximum number of faults is formulated as follows

$$\frac{dm(t)}{dt} = b(t)m(t)\left[1 - \frac{m(t)}{L}\right] - c(t)m(t) \tag{5.1}$$

where $m(t)$ represents the expected number of software failures detected by time t. L denotes the maximum number of software faults in the program. $b(t)$ is the fault detection rate per individual fault per unit of time. $c(t)$ represents the non-removed error rate per unit of time.

$(1-\frac{m(t)}{L})$ indicates the proportion of software faults are going to detect in every debugging effort. $b(t)m(t)\left[1-\frac{m(t)}{L}\right]$ is the percentage of detected dependent errors by time t. $c(t)m(t)$ represents the non-removed errors by time t. Hence, $b(t)m(t)\left[1-\frac{m(t)}{L}\right]-cm(t)$ represents the proportion of the removed errors in the software by time t. $\lambda(t)=\frac{dm(t)}{dt}$ is the failure intensity function for the whole software system by time t.

The marginal condition for the above equation is given as

$$m(t_0)=m_0, \quad m_0>0 \tag{5.2}$$

Given software testers perform a preliminary testing to remove the Type I software faults before officially starting this one-phase debugging process. Thus, in this chapter, we assume $m_0>0$ by taking into consideration of those errors. The general solution for (5.1) is easily obtained as

$$m(t)=\frac{e^{\int_{t_0}^{t}(b(\tau)-c(\tau))d\tau}}{\frac{1}{L}\int_{t_0}^{t}e^{\int_{t_0}^{\tau}(b(s)-c(s))ds}b(\tau)d\tau+\frac{1}{m_0}} \tag{5.3}$$

We assume fault detection is a process of a learning curve phenomenon, which is addressed in equation (5.4). Non-removed rate $c(t)$ is a constant, given in equation (5.5).

$$b(t) = \frac{b}{1 + \beta e^{-bt}}, \qquad b > 0, \qquad \beta > 0 \tag{5.4}$$

$$c(t) = c, \qquad c > 0 \tag{5.5}$$

Substituting equations (5.4) – (5.5) into equation (5.3), we obtain

$$m(t) = \frac{\beta + e^{bt}}{\frac{b}{L(b-c)}[e^{bt} - e^{ct}] + \frac{1+\beta}{m_0}e^{ct}} \tag{5.6}$$

5.3 Proposed Software Reliability Model for Two-Phase Debugging Process

In the last section, we discuss one-phase debugging process with dependent fault detection process and imperfect fault removal along with the maximum number of software faults exists in the program. We also assume Type I software faults have been eliminated in the pre-analysis testing phase.

In this section, we propose a two-phase software reliability model in consideration of software fault dependency and imperfect fault removal process to model software failure growth.

Firstly, two types of software faults are defined, Type I and Type II. Type I software fault is defined as an independent and easy-detected fault, which is detected and corrected in Phase I. Type II software fault is defined as a dependent and difficult-detected fault, which is detected and corrected in Phase II. In particular, the detection of Type II faults depends on the faults that have already detected. Hence, two-phase debugging process is proposed accordingly.

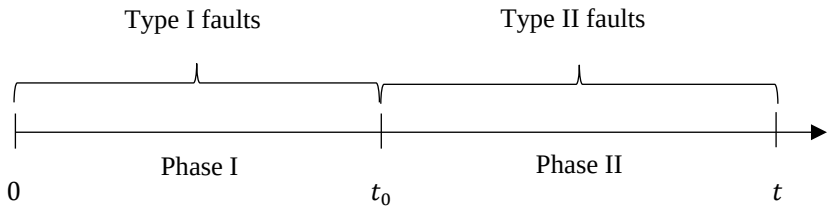

Figure 5. 3 Phase I and Phase II associated with corresponding fault type

Figure 5.3 describes two-phase debugging process and their corresponding fault type. Secondly, there exists a small portion of software faults in both phases that software programmer is not able to remove owing to the fact that programmer's domain knowledge and limited testing resources.

Thirdly, new software faults will be introduced to detect the existing software faults. In this study, we assume that no new Type I fault will be introduced in Phase II and left-over Type I faults from Phase I are still not able to be detected in Phase II. In the future research,

we will consider the new-introduced Type I faults and the left-over Type I faults from Phase I could both be detected in Phase II.

The assumptions for the proposed two-phase software reliability model are given as follows.

(1) Software detection follows a NHPP process.
(2) Software debugging is imperfect. New software faults will be introduced into the program to detect the existing faults.
(3) Type I fault is detected and removed in Phase I; Type II fault is detected and removed in Phase II. In this section, we do not consider Type I fault detection in Phase II and the left-over Type I faults from Phase I are still not able to be detected in Phase II.
(4) In both phases, there exists a certain portion of software faults that software development team are not able to remove.
(5) The fault detection rate and non-removable fault rate are different in Phase I and Phase II due to different software fault type.
(6) Debugging time is negligible.

The notations for this section are given as follows.

a The number of initial fault in the program

α Fault introduction rate per detected fault

b_1	Asymptotic unit of software fault detection rate in Phase I
β	Shape parameter of the learning curve in Phase I
c_1	Non-removable software fault in Phase I
b_2	Asymptotic unit software fault detection rate in Phase II
γ	Shape parameter of the learning curve in Phase II
c_2	Non-removable software fault rate in Phase II
$m_1(t)$	Expected number of software failures in Phase I by time t, $t \in [0, t_0]$
$m_2(t)$	Expected number of software failures in Phase II by time t, $t \in (t_0, \infty)$
$m_1(t_0)$	Expected number of software failures in Phase I by time t_0
$m_2(t_0)$	Expected number of software failures in Phase II by time t_0
$a_1(t)$	Total fault content function in Phase I
$a_2(t)$	Total fault content function in Phase II
$b_1(t)$	Fault detection rate function in Phase I
$b_2(t)$	Fault detection rate function in Phase II
$c_1(t)$	Non-removable fault rate function in Phase I due to the limitation of testing resource, the skill and experience of the programmer, and multi-release consideration for software organization
$c_2(t)$	Non-removable fault rate function in Phase II due to the limitation of testing resource, the skill and experience of the programmer, and multi-release consideration for software organization
$y'(t)$	Failure increasing rate during time interval $(t,\ t + \Delta t)$
$y(t)$	Observed cumulative number of failures by time t
$y(t + \Delta t)$	Observed cumulative number of failures by time $t + \Delta t$

The proposed two-phase software reliability model considering software fault dependency and imperfect fault removal is formulated as follows

$$\frac{dm_1(t)}{dt} = b_1(t)[a_1(t) - m_1(t)] - c_1(t)m_1(t), \qquad t \leq t_0 \qquad (5.7)$$

$$\frac{dm_2(t)}{dt} = \frac{b_2(t)}{a_2(t)}m_2(t)[a_2(t) - m_2(t)] - c_2(t)m_2(t), \qquad t > t_0 \qquad (5.8)$$

where $m_1(t)$ is the expected number of software failures in Phase I by time t, $t \in [0, t_0]$. $m_2(t)$ is the expected number of software failures in Phase II by time $t, t \in (t_0, \infty)$. $a_1(t)$ and $a_2(t)$ represent the total fault content function in Phase I and Phase II, respectively. $b_1(t)$ and $b_2(t)$ denote the fault detection rate function in Phase I and Phase II, respectively. $c_1(t)$ and $c_2(t)$ describe non-removable fault rate function in Phase I and Phase II, respectively, due to the limitation of testing resource, the skill and experience of the programmer, and multi-release consideration of software organization.

The connection between Phase I and II is given as follows

$$m_1(t_0) = m_2(t_0) \qquad (5.9)$$

where $m_1(t_0)$ represents the expected number of software failures in Phase I by time t_0 and $m_2(t_0)$ represents the expected number of software failures in Phase II by time t_0.

Note that we consider time t_0 belongs to Phase I for the latter parameter estimation. The fault content in Phase II is obtained by

$$a_2(t) = a_1(t_0) - m_1(t_0) \tag{5.10}$$

where $a_1(t_0)$ is the total fault content function in Phase I at time t_0.

5.3.1 Phase I Software Reliability Model

As illustrated in Figure 5.3, software testers only detect and remove Type I fault, which is the independent and easy-detected software fault in Phase I. New software faults will be introduced into system while executing debugging, therefore, this is an imperfect debugging process and the total software faults content in Phase I is

$$a_1(t) = a(1 + \alpha t), \quad a > 0, \qquad \alpha > 0 \tag{5.11}$$

where a is the number of initial fault in the program and α is the fault introduction rate per detected fault.

The fault detection rate in Phase I is given by

$$b_1(t) = \frac{b_1}{1 + \beta e^{-b_1 t}}, \quad b_1 > 0, \qquad \beta > 0 \tag{5.12}$$

where b_1 is the asymptotic unit of software fault detection rate in Phase I and β is the parameter applied to determine the shape of the learning curve in Phase I.

The non-removable fault rate in Phase I is given by

$$c_1(t) = c_1, \qquad c_1 > 0 \tag{5.13}$$

where c_1 denotes the non-removable software fault in Phase I. The initial condition for Phase I is given as

$$m_1(t = 0) = 0 \tag{5.14}$$

Substituting equations (5.11) - (5.14) into equation (5.7), we obtain the mean value function $m_1(t)$ given as follows

$$m_1(t) = \frac{ab_1[(b_1 + c_1)(1 + \alpha t)e^{(b_1+c_1)t} - \alpha e^{(b_1+c_1)t} + \alpha - b_1 - c_1]}{(b_1 + c_1)^2(\beta + e^{b_1 t})e^{c_1 t}}, t \in [0, t_0] \tag{5.15}$$

5.3.2 Phase II Software Reliability Model

Software testers are going to detect and remove Type II fault in Phase II. Note that we do not consider the detection of Type I fault in Phase II in this study and the left-over Type I faults from Phase I are still not able to be detected in Phase II.

The fault content function $a_2(t)$ is obtained from equations (5.9) and (5.10)

$$a_2(t) = a_1(t_0) - m_1(t_0) = a(1 + \alpha t_0) - m_1(t_0) \tag{5.16}$$

The fault detection rate function $b_2(t)$ in Phase II is described as

$$b_2(t) = \frac{b_2}{1 + \gamma e^{-b_2 t}}, \quad b_2 > 0, \qquad \gamma > 0 \tag{5.17}$$

where b_2 is the asymptotic unit software fault detection rate in Phase II and γ determines the shape of the learning curve in Phase II.

The non-removable fault rate in Phase II is given by

$$c_2(t) = c_2, \quad c_2 > 0 \tag{5.18}$$

where c_2 represents the non-removable software fault rate in Phase II.

Substituting equations (5.9) - (5.10), (5.15) - (5.18) into equation (5.8), the mean value function $m_2(t)$ is obtained as follows

$$m_2(t) = \frac{\gamma + e^{b_2 t}}{\frac{b_2 e^{c_2 t}}{[a(1+\alpha t_0) - m_1(t_0)](b_2 - c_2)}[e^{(b_2 - c_2)t} - e^{(b_2 - c_2)t_0}] + \frac{\gamma + e^{b_2 t_0}}{m_1(t_0)}}, t \in (t_0, \infty) \quad (5.19)$$

where $m_1(t_0) = \frac{ab_1[(b_1 + c_1)(1 + \alpha t_0)e^{(b_1 + c_1)t_0} - \alpha e^{(b_1 + c_1)t_0} + \alpha - b_1 - c_1]}{(b_1 + c_1)^2(\beta + e^{b_1 t_0})e^{c_1 t_0}}$.

5.4 Parameter Estimation and Comparison Criteria

Parameter Estimation

In practice, parameter estimation will be achieved by applying least square estimate (LSE) and maximum likelihood estimation. For example, minimizing the equation (5.20) or maximizing the equation (5.21).

$$S = \sum_{i=1}^{n} [m(t_i) - y_i]^2 \quad (5.20)$$

$$LLF = \sum_{i=1}^{n} (y_i - y_{i-1}) \log[m(t_i) - m(t_{i-1})] - m(t_n) - \sum_{i=1}^{n} \log(y_i - y_{i-1})! \quad (5.21)$$

where y_i is the observed number of failures at time t_i. $m(t_i)$ is the predicted data. We apply LSE to minimize the equation (5.20) to estimate the parameters. t_0 has already determined in last section. Thus, we have eight unknown parameters from equations (5.15) and (5.19) that need to be estimated. The Genetic Algorithm (GA) is employed to solve the optimization function. MATLAB and R software are used to solve the optimization function and estimate parameters.

Comparison Criteria

(1) Mean Squared Error (MSE)

$$MSE = \frac{\sum_{i=1}^{n}[m(t_i) - y_i]^2}{n - N} \tag{5.22}$$

where *n* is the total number of observations. y_i is the observed failure data at t_i. $m(t_i)$ is the predicted failure data at t_i. *N* represents the number of unknown parameters in each model. The MSE measures the distance of a model estimate from the observed data.

(2) Predictive-Ratio Risk (PRR) and Predictive Power (PP) [18]

$$PRR = \sum_{i=1}^{n}\left[\frac{m(t_i) - y_i}{m(t_i)}\right]^2 \tag{5.23}$$

$$PP = \sum_{i=1}^{n} \left[\frac{m(t_i) - y_i}{y_i} \right]^2 \tag{5.24}$$

The PRR and PP are calculated to compare the power of different models. The PRR measures the distance of the model estimates from the actual data against the model estimates; while the PP measures the distance of the model estimates from the actual data against the actual data.

(3) Variation

The Variation is defined as [162]

$$Variation = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} [y_i - m(t_i) - Bias]^2} \tag{5.25}$$

where

$$Bias = \frac{1}{n} \sum_{i=1}^{n} [m(t_i) - y_i].$$

(4) The Akaike information criterion (AIC) not only measures the ability of a model to maximize its likelihood function, but also assigns the penalty for increasing the number of estimated parameters.

$$AIC = -2 * log(LF) + 2 * N \tag{5.26}$$

where *LF* is the maximum value of the likelihood function. *N* is the number of estimated parameters.

For all the criteria discussed above, the smaller of the criteria, the better fit of the model.

5.5 Numerical Examples for One-Phase Software Reliability Model

Numerical Example 1

Telecommunication system data, reported by Zhang et al. [163], are applied to validate the proposed one-phase software reliability model. System test data consists of two phases of test data. In each phase, the system records the cumulative number of faults by each week. 356 system test hours were observed in each week for Phase I data, as seen in Table 5.1. 416 system test hours were observed in each week for Phase II data, as seen in Table 5.2. Parameter estimate was carried out by the GA method.

To provide a better comparison of our proposed one-phase software reliability model with the other existing models, described in Table 5.3, we have analyzed Phase I as well as Phase II system test data in this section.

Table 5. 1 Phase I system test data

Week index	Exposure time (System test hours)	Failures	Cumulative failures	Week index	Exposure time (System test hours)	Failures	Cumulative failures
1	356	1	1	12	4272	2	15
2	712	0	1	13	4628	4	19
3	1068	1	2	14	4984	0	19
4	1424	1	3	15	5340	3	22
5	1780	2	5	16	5696	0	22
6	2136	0	5	17	6052	1	23
7	2492	0	5	18	6408	1	24
8	2848	3	8	19	6764	0	24
9	3204	1	9	20	7120	0	24
10	3560	2	11	21	7476	2	26
11	3916	2	13	-	-	-	-

Table 5. 2 Phase II system test data

Week index	Exposure time (System test hours)	Failures	Cumulative failures	Week index	Exposure time (System test hours)	Failures	Cumulative failures
1	416	3	3	12	4992	2	25
2	832	1	4	13	5408	5	30
3	1248	0	4	14	5824	2	32
4	1664	3	7	15	6240	4	36
5	2080	2	9	16	6656	1	37
6	2496	0	9	17	7072	2	39
7	2912	1	10	18	7488	0	39
8	3328	3	13	19	7904	0	39
9	3744	4	17	20	8320	3	42
10	4160	2	19	21	8736	1	43
11	4576	4	23	-	-	-	-

Table 5. 3 Mean value function for all compared models

Model	**Mean Value Function**
Goel-Okumoto (G-O)	$m(t) = a(1 - e^{bt})$
Delayed S-shaped	$m(t) = a[1 - (1 + bt)e^{-bt}]$
Inflection S-shaped	$m(t) = \frac{a(1 - e^{-bt})}{1 + \beta e^{-bt}}$
Yamada imperfect debugging	$m(t) = a(1 - e^{-bt})\left(1 - \frac{\alpha}{b}\right) + \alpha a t$
PNZ Model	$m(t) = \frac{a[(1 - e^{-bt})\left(1 - \frac{\alpha}{b}\right) + \alpha t]}{1 + \beta e^{-bt}}$
Pham-Zhang model	$m(t) = \frac{1}{1 + \beta e^{-bt}}\left[(c + a)(1 - e^{-bt}) - \frac{ab}{b - \alpha}(e^{-\alpha t} - e^{-bt})\right]$
Dependent-parameter model	$m(t) = \alpha(1 + \gamma t)(\gamma t + e^{-\gamma t} - 1)$
Dependent-parameter model with $m_0 \neq 0$, $t_0 \neq 0$,	$m(t) = m_0\left(\frac{\gamma t + 1}{r t_0 + 1}\right)e^{-\gamma(t - t_0)} + \alpha(\gamma t + 1)\left[\gamma t - 1 + (1 - \gamma t_0)e^{-\gamma(t - t_0)}\right]$
Loglog fault-detection rate model	$m(t) = N(1 - e^{-(a^{t^b} - 1)})$
Proposed model	$m(t) = \frac{\beta + e^{bt}}{\frac{b}{L(b - c)}[e^{bt} - e^{ct}] + \frac{1 + \beta}{m_0}e^{ct}}$

In the proposed one-phase software model, when $t = 0$, the initial number of faults in the software satisfies $0 < m_0 \leq y_1$, where y_1 is the number of observed failures at time $t = 1$. At the same time, m_0 must be an integer. The interpretation of this constraint is that the software tester often completes preliminary testing to eliminate trivial errors, Type I faults in this study, before officially starting testing. The cause of these trivial errors could be human mistakes or other simple settings. Since we consider the maximum number of faults that the software to contain in the modeling, these eliminated trivial errors will be counted into the total number of software faults.

Tables 5.4 and 5.5 summarize the results of the estimated parameters and corresponding criteria value (MSE, PRR, PP, AIC) for the proposed one-phase software reliability model and other existing models. Both system test data present as an S-shaped curve. Thus, the existing models such as Goel-Okumoto model is not able to perfectly capture the characteristic of the two system test datasets.

For Phase I system test data, the estimated parameters are $\hat{m}_0 = 1, \hat{L} = 49.7429,\ \hat{\beta} = 0.2925, \hat{b} = 0.6151, \hat{c} = 0.292$. As seen in Table 5.4, MSE and PRR values for the proposed model are 0.630 and 0.408, which are the smallest among all ten models listed here. Inflection S-shaped model has the smallest PP value. However, the PP value for the proposed model is 0.526, which is only slightly higher than 0.512.

Moreover, the PRR value for the inflection S-shaped model is much higher than that of the proposed model. The AIC value for the proposed model is 65.777, which is just slightly higher than the smallest AIC value, 63.938. Thus, we conclude that the proposed model is the best fit for Phase I system test data compared with the other nine models. Figure 5.4 presents the comparison of actual cumulative failures and cumulative failures predicted the proposed model.

Table 5. 4 Parameter estimates and model comparison (Phase I system test data)

Model	**MSE**	**PRR**	**PP**	**AIC**	**Parameter Estimates**
Goel-Okumoto (G-O)	5.944	1.818	8.165	66.211	$\hat{a} = 62.040$ $\hat{b} = 0.024$
Delayed S-shaped	1.609	14.546	0.981	64.230	$\hat{a} = 44.221$ $\hat{b} = 0.101$
Inflection S-shaped	0.709	1.714	0.512	63.938	$\hat{a} = 27.247$ $\hat{b} = 0.269$ $\hat{\beta} = 17.255$
Yamada imperfect debugging	2.602	0.840	0.757	66.710	$\hat{a} = 1.864$ $\hat{b} = 0.250$ $\hat{\alpha} = 0.842$
PNZ Model	2.479	2.954	0.690	68.611	$\hat{a} = 1.556$ $\hat{b} = 0.324$ $\hat{\alpha} = 0.969$ $\hat{\beta} = 0.999$
Pham-Zhang model	3.429	1.982	1.187	70.617	$\hat{a} = 13.394$ $\hat{b} = 0.267$ $\hat{\alpha} = 0.511$ $\hat{\beta} = 9.013$ $\hat{c} = 12.034$
Dependent-parameter model	15.741	287.191	3.768	77.541	$\hat{\alpha} = 0.087$ $\hat{\gamma} = 0.952$
Dependent-parameter model with $m_0 \neq 0$, $t_0 \neq 0$	13.477	2.136	1.189	77.621	$\hat{\alpha} = 6206.000$ $\hat{\gamma} = 0.005$ $t_0 = 1.000$ $m_0 = 1.000$
Loglog fault-detection rate model	71.241	11.736	15.475	93.592	$\widehat{N} = 15.403$ $\hat{a} = 1.181$ $\hat{b} = 0.567$
Proposed model	0.630	0.408	0.526	65.777	$\widehat{m}_0 = 1.000$ $\hat{L} = 49.743$ $\hat{\beta} = 0.293$ $\hat{b} = 0.615$ $\hat{c} = 0.292$

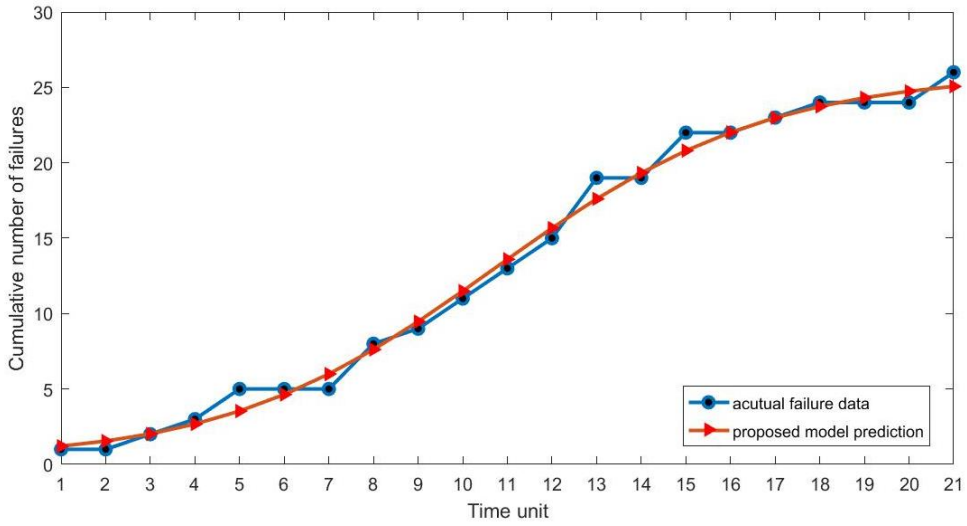

Figure 5. 4 Comparison of actual predicted failures (Phase I system test data)

For Phase II system test data, the estimated parameters are $\widehat{m}_0 = 3, \hat{L} = 59.997,\ \hat{\beta} = 0.843, \hat{b} = 0.409, \hat{c} = 0.108$. The proposed model presents the smallest MSE, PRR, PP and AIC value in Table 5.5. Thus, we conclude that the proposed model is the best fitting for Phase II test data among all other models. Figure 5.5 plots the comparison of the actual cumulative failures and cumulative failures predicted the proposed model.

Moreover, the proposed model provides the maximum number of faults contained in software, for instance, $L = 60$ for Phase II test data. Assume that the company releases software at week 21, 43 faults will be detected upon this time based on the actual observations; however, the fault may not be perfectly removed upon detection as discussed in Section 5.1. The remaining faults revealed in the operation field, mostly, are Mandelbugs [164]. Given the maximum number of faults in the software, it is very helpful for the

software developer to better predict the remaining errors and decide the release time for the next version.

Table 5. 5 Parameter estimates and model comparison (Phase II system test data)

Model	**MSE**	**PRR**	**PP**	**AIC**	**Parameter Estimates**
Goel-Okumoto (G-O)	6.607	0.687	1.099	74.752	$\hat{a} = 98295.000$ $\hat{b} = 5.2E-8$
Delayed S-shaped	3.273	44.267	1.429	77.502	$\hat{a} = 62.300$ $\hat{b} = 2.85E-4$
Inflection S-shaped	1.871	5.938	0.895	73.359	$\hat{a} = 46.600$ $\hat{b} = 5.78E-4$ $\hat{\beta} = 12.200$
Yamada imperfect debugging	4.982	4.296	0.809	78.054	$\hat{a} = 1.500$ $\hat{b} = 0.001$ $\hat{\alpha} = 0.004$
PNZ Model	1.994	6.834	0.957	75.501	$\hat{a} = 45.990$ $\hat{b} = 6.0E-4$ $\hat{\alpha} = 0$ $\hat{\beta} = 13.240$
Pham-Zhang model	2.119	6.762	0.952	77.502	$\hat{a} = 0.060$ $\hat{b} = 6.0E-4$ $\hat{\alpha} = 1.0E-4$ $\hat{\beta} = 13.200$ $\hat{c} = 45.900$
Dependent-parameter model	43.689	601.336	4.530	101.386	$\hat{\alpha} = 3.0E-6$ $\hat{\gamma} = 0.490$
Dependent-parameter model with $m_0 \neq 0$, $t_0 \neq 0$	35.398	2.250	1.167	87.667	$\hat{\alpha} = 890996.000$ $\hat{\gamma} = 1.2E-6$ $t_0 = 832.000$ $m_0 = 4.000$
Loglog fault-detection rate model	219.687	13.655	4.383	114.807	$\hat{N} = 231.920$ $\hat{a} = 1.019$ $\hat{b} = 0.489$
Proposed model	1.058	0.163	0.144	68.316	$\hat{m}_0 = 3.000$ $\hat{L} = 59.997$ $\hat{\beta} = 0.843$ $\hat{b} = 0.409$ $\hat{c} = 0.108$

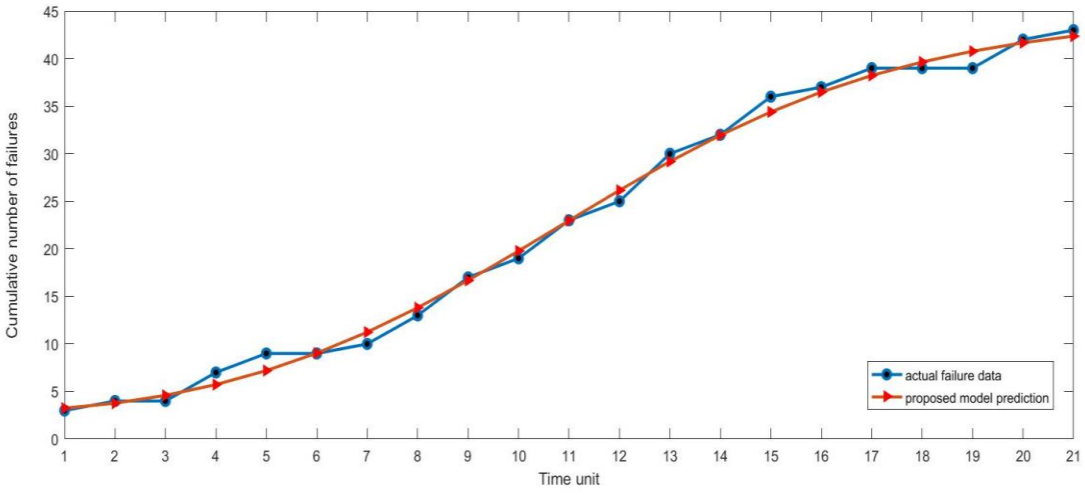

Figure 5. 5 Comparison of actual predicted failures (Phase II system test data)

Numerical Example 2

Wood [165] provides software failure data including four major releases of software products at Tandem Computers. Eight NHPP models were studied in Wood [165] and it was found that the G-O models provided the best performance in terms of goodness of fit. By fitting our model into the same subset of data, from week 1 to week 9, we predict the cumulative number of faults from week 10 to week 20 and compare the results with the G-O model and Zhang–Teng–Pham model [106].

Table 5.6 describes the predicted number of software failures from each model. The AIC value for the proposed model is not the smallest AIC value present in Table 5.6; however, we still conclude that the proposed model is the best fit for this dataset, since the other three

Table 5. 6 Comparison of G-O, Zhang-Teng-Pham model and the proposed model

Testing Time (weeks)	**CPU hours**	**Defects found**	**Predicted total defects by G-O**	**Predicted total defects by Zhang-Teng-Pham model**	**Predicted total defects by proposed model**
1	519	16	-	-	-
2	968	24	-	-	-
3	1,430	27	-	-	-
4	1,893	33	-	-	-
5	2,490	41	-	-	-
6	3,058	49	-	-	-
7	3,625	54	-	-	-
8	4,422	58	-	-	-
9	5,218	69	-	-	-
10	5,823	75	98	74.7	75.5
11	6,539	81	107	80.1	80.8
12	7,083	86	116	85.2	85.1
13	7,487	90	123	90.1	88.5
14	7,846	93	129	94.6	91.2
15	8,205	96	129	98.9	93.2
16	8,564	98	134	102.9	94.7
17	8,923	99	139	106.8	95.8
18	9,282	100	138	110.4	96.6
19	9,641	100	135	111.9	97.2
20	10,000	100	133	112.2	97.6
Predicted MSE	-	-	1359.222	82.660	10.120
Predicted AIC	-	-	149.600	186.468	169.667
Predicted PRR	-	-	0.756	0.041	0.007
Predicted PP	-	-	1.395	0.050	0.006

criteria (MSE, PRR and PP) indicate that the proposed model is significantly better than other models.

The GA method is applied here to estimate the parameter. Parameter estimates for the proposed model are given as $\widehat{m}_0 = 3,\ \hat{L} = 181,\ \hat{\beta} = 0.5001, \hat{b} = 0.602,\ \hat{c} = 0.274.$

We propose a one-phase software reliability model that incorporates dependent fault detection and imperfect fault removal, along with the maximum number of faults contained in the software. To our knowledge, not many research have included dependent fault detection in software reliability models. We also estimate the maximum number of faults in the software to provide software measurement metrics, such as remaining errors, failure rate, and software reliability.

5.6 Numerical Examples for Two-Phase Software Reliability Model

Empirical Data Analysis for Software Failure Data Set

We employ three software failure datasets to illustrate the effectiveness of the proposed two-phase model. The first dataset is the failure data from a real-time control system. This monitoring software has about 200 modules with an average of 1000 lines of high-level language in each module [112]. Table 5.7 shows the first failure dataset (DS1) which were detected during the 111 days testing period. The second dataset (DS2) is obtained from testing data of Release 3 [166] in a wireless network switching center, as seen in Table 5.8.

The third fault (DS3) tracking data [49] are collected and organized on Firefox from Bugzilla (https: //bugzilla.mozilla.org/), as seen in Table 5.9.

The proposed model has two phases. Usually, software development team knows t_0 value in real software testing. However, for the given datasets, we do not know t_0. Hence, the first step is to determine t_0 in the model validation.

Table 5. 7 Dataset 1 (DS 1)

Day	**Cumulative failures**	**Day**	**Cumulative failures**	**Day**	**Cumulative failures**	**Day**	**Cumulative failures**
1	5	29	254	57	448	85	473
2	10	30	259	58	451	86	473
3	15	31	263	59	453	87	475
4	20	32	264	60	460	88	475
5	26	33	268	61	463	89	475
6	34	34	271	62	463	90	475
7	36	35	277	63	464	91	475
8	43	36	293	64	464	92	475
9	47	37	309	65	465	93	475
10	49	38	324	66	465	94	475
11	80	39	331	67	465	95	475
12	84	40	346	68	466	96	476
13	108	41	367	69	467	97	476
14	157	42	375	70	467	98	476
15	171	43	381	71	467	99	476
16	183	44	401	72	468	100	477
17	191	45	411	73	469	101	477
18	200	46	414	74	469	102	477
19	204	47	417	75	469	103	478
20	211	48	425	76	469	104	478
21	217	49	430	77	470	105	478
22	226	50	431	78	472	106	479
23	230	51	433	79	472	107	479
24	234	52	435	80	473	108	479
25	236	53	437	81	473	109	480
26	240	54	444	82	473	110	480
27	243	55	446	83	473	111	481
28	252	56	446	84	473	-	-

Table 5. 8 Dataset 2 (DS 2)

Week	Cumulative failures	Week	Cumulative failures	Week	Cumulative failures	Week	Cumulative failures
1	5	10	46	19	105	28	156
2	6	11	53	20	110	29	156
3	13	12	63	21	117	30	164
4	13	13	70	22	123	31	166
5	22	14	71	23	128	32	169
6	24	15	74	24	130	33	170
7	29	16	78	25	136	34	176
8	34	17	90	26	141	35	180
9	40	18	98	27	148	36	181

Table 5. 9 Dataset 3 (DS 3)

Week	Cumulative failures	Week	Cumulative failures	Week	Cumulative failures	Week	Cumulative failures
1	9	22	44	43	60	64	94
2	12	23	45	44	60	65	99
3	16	24	45	45	60	66	102
4	25	25	46	46	61	67	104
5	27	26	47	47	62	68	105
6	29	27	47	48	62	69	105
7	29	28	49	49	62	70	106
8	32	29	50	50	62	71	10
9	34	30	50	51	62	72	107
10	35	31	50	52	64	73	108
11	36	32	50	53	65	74	108
12	36	33	51	54	66	75	109
13	39	34	52	55	73	76	112
14	39	35	53	56	76	77	113
15	40	36	54	57	81	78	113
16	40	37	55	58	83	79	115
17	40	38	55	59	87	80	115
18	41	39	55	60	88	81	116
19	42	40	55	61	92	-	-
20	43	41	56	62	94	-	-
21	43	42	59	63	94	-	-

Empirical data analysis is utilized to determine the value of t_0 based on the practical interpretation of Phase I and Phase II. The software failure increasing rate is desirable in this analysis. We formulate the failure increasing rate as follows

$$y'(t) = \lim_{\Delta t \to 0} \frac{y(t + \Delta t) - y(t)}{\Delta t} \tag{5.27}$$

where $y'(t)$ illustrates the failure increasing rate during time interval $(t,\ t + \Delta t)$. $y(t)$ is the observed cumulative number of failures by time t. $y(t + \Delta t)$ denotes the observed cumulative number of failures by time $t + \Delta t$. Given different value of Δt, we are interested in investigating the pattern of $y'(t)$.

Dataset 1 (failure data from real-time control system [112])

Figure 5.6 illustrates the failure increasing rate $y'(t)$ in terms of different Δt for Dataset 1. We notice that $y'(t)$ has two peaks, $t = 14$ and $t = 41$. Software testers often need some time to get familiar with the algorithm and failure mode. Software failure increasing rate $y'(t)$ usually stays stable at first, then presents a significantly increasing pattern until it gets peak due to tester's growing debugging experience. Afterwards, it declines to a stabilized rate. The similar pattern is repeated for debugging another type of software fault. Thus, we conclude that the maximum failure increasing rate for Type I fault is manifested at time $t = 14$ and the maximum failure increasing rate for Type II fault is manifested at time $t = 41$. Therefore, $t_0 = 35$.

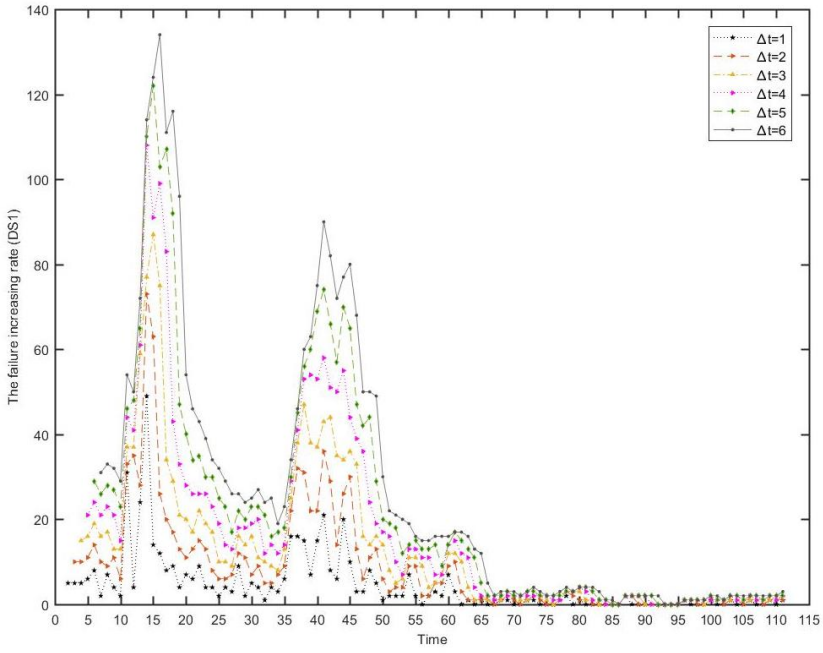

Figure 5. 6 Software failure increasing rate with respect to different value of Δt (DS1)

Dataset 2 (failure data from wireless network switching center [166])

Figure 5.7 illustrates the failure increasing rate $y'(t)$ in terms of different Δt for Dataset 2. The failure increasing rate performs several peaks during the testing period. Considering that each type of software fault should have its maximum failure rate, thus, we conclude at time $t = 13$, Type I fault have its maximum failure rate and at time $t = 18$, Type II fault have its maximum failure rate. Therefore, t_0 is determined as 15 in this failure dataset.

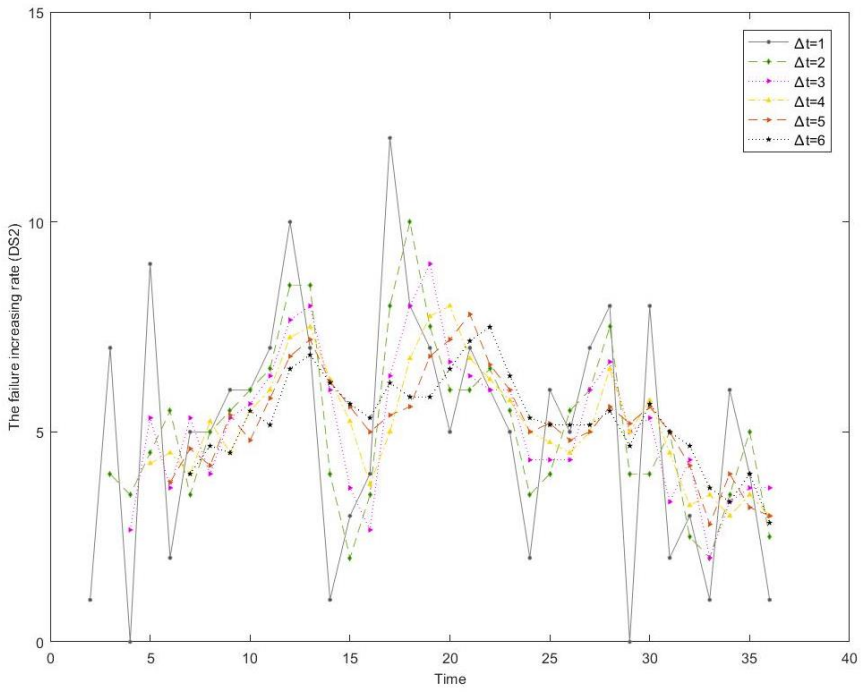

Figure 5. 7 Software failure increasing rate with respect to different value of Δt (DS2)

Dataset 3 (failure data from online bug tracking system [49])

Figure 5.8 illustrates the failure increasing rate $y'(t)$ in terms of different Δt for Dataset 3 by applying equation (5.27). Given the nature characteristics of failure growing, we consider $t_0 = 50$ is employed to distinguish Phase I and Phase II for Dataset 3.

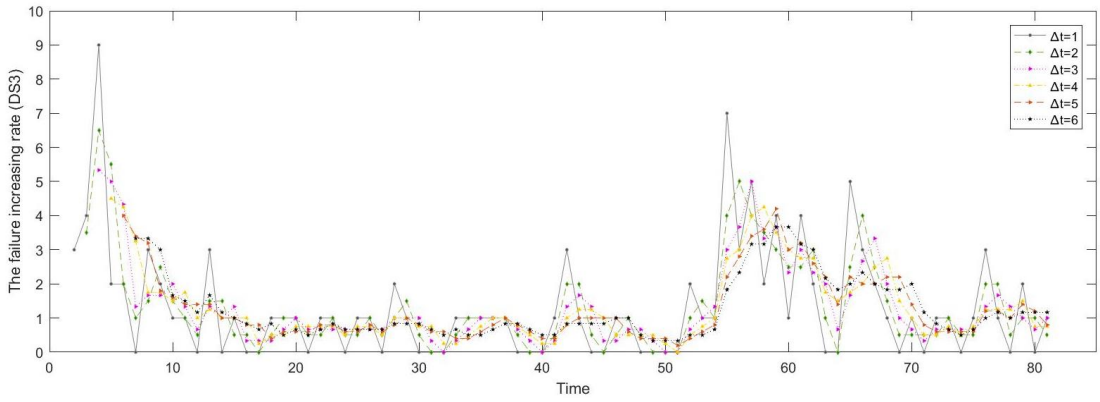

Figure 5. 8 Software failure increasing rate with respect to different value of Δt (DS3)

Numerical Example 1

We conclude that $t_0 = 35$ for the real-time control systems software failure data from Section 5.6.1. Thus, Phase I is defined when $t \in [0, 35]$; Phase II is defined when $t \in (35, 111]$. Table 5.10 describes the comparison of software reliability models. Parameter estimates and criteria comparisons for all models are displayed in Table 5.10 by the use of GA method.

The proposed model has the smallest MSE, PP, and Variation value. PRR assigns a larger penalty to the model which has underestimated the cumulative number of failures. Even the proposed model does not perform the best PRR value, however, given such a significant improvement on MSE, PP and Variation value, we still conclude that the proposed model presents better prediction than other software reliability models. The mean value function of the proposed model and the observed data is plotted in Figure 5.9.

As predicted by the proposed model, software tester successfully removed 273 Type I (independent) software failures at the end of Phase I. At the end of Phase II, software testers remove 471 software failures including 273 Type I (independent) software faults and 198 Type II (dependent) software faults. However, 504 software faults still exist in the program. Thus, multiple version software release planning is utilized by most software organization those days to deal with the faults after initial release.

Table 5. 10 Parameter estimates and model comparison (DS1)

Model	**Parameter Estimates**	**MSE**	**PRR**	**PP**	**Variation**
Yamada imperfect debugging model	$\hat{a} = 591.800$ $\hat{b} = 0.024$ $\hat{\alpha} = 0.002$	6000.690	6.460	31.440	132.910
PNZ model	$\hat{a} = 470.760$ $\hat{b} = 0.075$ $\hat{\alpha} = 0.0002$ $\hat{\beta} = 4.693$	320.400	1.570	2.070	17.820
G-O model	$\hat{a} = 497.290$ $\hat{b} = 0.031$	1006.080	5.110	33.000	31.620
Delayed S-shaped model	$\hat{a} = 488.400$ $\hat{b} = 0.066$	344.200	16.328	2.174	18.180
Inflection S-shaped model	$\hat{a} = 482.020$ $\hat{b} = 0.070$ $\hat{\beta} = 4.146$	301.220	1.670	2.490	17.230
Pham-Zhang IFD model	$\hat{a} = 482.000$ $\hat{b} = 0.081$ $\hat{d} = 0.007$	450.147	36.376	4.328	22.400
Proposed model	$\hat{a} = 600.000$ $\hat{b}_1 = 0.348$ $\hat{c}_1 = 0.883$ $\hat{\alpha} = 0.018$ $\hat{\beta} = 64.650$ $\hat{b}_2 = 0.092$ $\hat{c}_2 = 0.017$ $\hat{\gamma} = 0.00013$	51.450	5.310	1.630	6.980

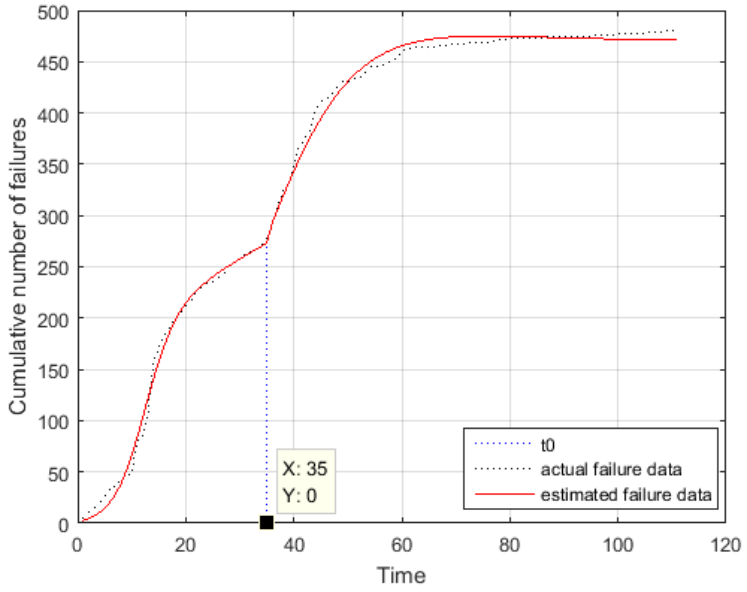

Figure 5. 9 Comparison of failure data prediction and actual data (DS1)

Numerical Example 2

As discussed in Section 5.6.1, $t_0 = 15$ for the failure data from wireless network switching center. Thus, Phase I is defined when $t \in [0, 15]$; Phase II is defined when $t \in (15, 36]$. The proposed model presents the smallest MSE, PRR, PP, and Variation value, as explained in Table 5.11. The mean value function of the proposed model and the observed data is plotted in Figure 5.10. As predicted by the proposed model, software tester successfully removed 73 Type I software failures at the end of Phase I. At the end of Phase II, software testers remove 175 software failures including 73 Type I software failures and 102 Type II software failures. However, 1345 software faults still exist in the program.

Table 5. 11 Parameter estimates and model comparison (DS2)

Model	**Parameter Estimates**	**MSE**	**PRR**	**PP**	**Variation**
Yamada imperfect debugging model	$\hat{a} = 250.040$ $\hat{b} = 0.021$ $\hat{\alpha} = 0.022$	57.320	0.550	1.050	9.370
PNZ model	$\hat{a} = 300.670$ $\hat{b} = 0.024$ $\hat{\alpha} = 0.013$ $\hat{\beta} = 0.423$	35.550	0.550	1.140	5.940
G-O model	$\hat{a} = 463.060$ $\hat{b} = 0.014$	70.560	1.360	3.820	12.150
Delayed S-shaped model	$\hat{a} = 280.340$ $\hat{b} = 0.061$	36.860	64.270	2.210	6.750
Inflection S-shaped model	$\hat{a} = 191.000$ $\hat{b} = 0.157$ $\hat{\beta} = 15.207$	45.160	4.730	1.400	7.280
Pham-Zhang IFD model	$\hat{a} = 192.910$ $\hat{b} = 0.116$ $\hat{d} = 0.013$	41.560	261.570	4.160	7.580
Proposed model	$\hat{a} = 240.000$ $\hat{b}_1 = 0.0001$ $\hat{c}_1 = 0.0001$ $\hat{\alpha} = 0.376$ $\hat{\beta} = 8.217$ $\hat{b}_2 = 0.074$ $\hat{c}_2 = 0.158$ $\hat{\gamma} = 0.0001$	31.360	0.420	0.820	5.090

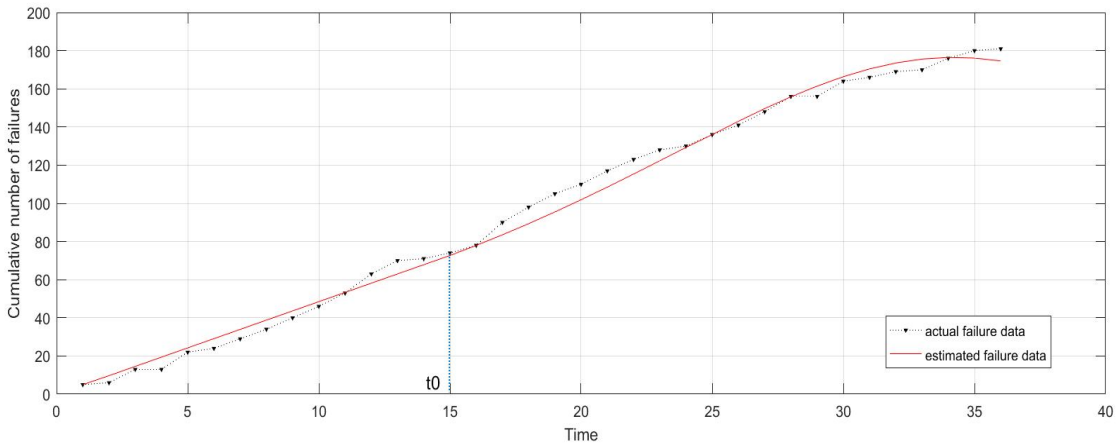

Figure 5. 10 Comparison of failure data prediction and actual data (DS2)

Numerical Example 3

$t_0 = 50$ is determined in Section 5.6.1 for the failure data of the online bug tracking system. Phase I is defined when $t \in [0, 50]$; Phase II is defined when $t \in (50, 81]$. As seen from Table 5.12, the proposed model has the smallest MSE, PRR, PP, and Variation value. The mean value function of the proposed model and the observed data is plotted in Figure 5.11. Software tester successfully removed 56 Type I software failures at the end of Phase I and 54 Type II software failures at the end of Phase II. However, there are still 617 software faults left in the program after testing phase.

Table 5. 12 Parameter estimates and model comparison (DS3)

Model	**Parameter Estimates**	**MSE**	**PRR**	**PP**	**Variation**
Yamada imperfect debugging model	$\hat{a} = 129.560$ $\hat{b} = 0.014$ $\hat{\alpha} = 0.005$	91.500	44.660	5.880	10.210
PNZ model	$\hat{a} = 120.060$ $\hat{b} = 0.021$ $\hat{\alpha} = 0.005$ $\hat{\beta} = 0.423$	100.290	50.340	6.270	10.050
G-O model	$\hat{a} = 170.080$ $\hat{b} = 0.012$	97.890	34.720	5.380	10.630
Delayed S-shaped model	$\hat{a} = 280.340$ $\hat{b} = 0.063$	185.320	6571.610	11.920	15.750
Inflection S-shaped model	$\hat{a} = 195.010$ $\hat{b} = 0.015$ $\hat{\beta} = 0.804$	101.310	64.340	6.760	12.180
Pham-Zhang IFD model	$\hat{a} = 120.00$ $\hat{b} = 0.045$ $\hat{d} = 1 \times 10^{-6}$	187.630	6948.020	12.050	15.730
Proposed model	$\hat{a} = 41.780$ $\hat{b}_1 = 0.013$ $\hat{c}_1 = 0.015$ $\hat{\alpha} = 0.355$ $\hat{\beta} = 0.909$ $\hat{b}_2 = 0.050$ $\hat{c}_2 = 0.101$ $\hat{\gamma} = 1.5 \times 10^{-5}$	21.790	3.330	1.360	4.650

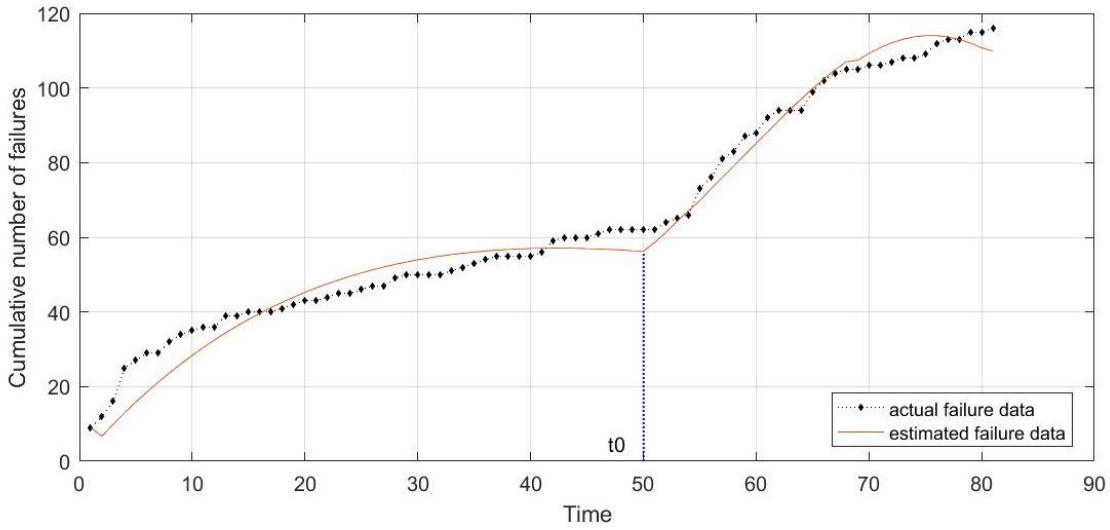

Figure 5. 11 Comparison of failure data prediction and actual data (DS3)

Reliability Prediction

As an illustration for quantifying reliability assessment, we will present reliability prediction for Dataset 1, discussed in Section 5.6.2, for the proposed two-phase software reliability model. Other datasets just follow the same calculation by applying equation (5.28). Since the parameters have been estimated in Table 4, software reliability within $(t, t+x)$ is determined by

$$R(x|t) = e^{-[m(t+x)-m(t)]} \tag{5.28}$$

Let $t = 111$, and vary *x* from 0 to 8, then we provide the reliability prediction for the field operation. The reliability prediction curve for time *x* given *t* is illustrated in Figure 5.12.

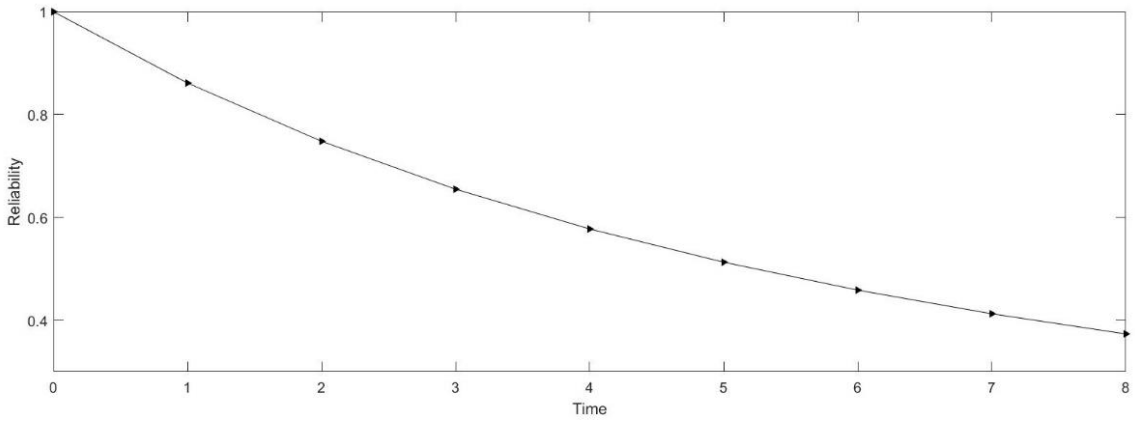

Figure 5. 12 Reliability prediction

5.7 Conclusions

This chapter provides a practical and pioneering idea for the researchers since different types of software faults are defined mainly depend on their detection dependency. We first propose a one-phase software reliability model considering the program only has Type II fault in the official testing phase while Type I faults have been removed in the preliminary testing phase. Imperfect fault removal process is also considered in this model.

Later, we clearly define Type I (independent and easy-detected) software fault and Type II (dependent and difficult-detected) software fault according to the classification discussed in literature. Correspondingly, two phases (Phase I & II) debugging processes are defined based on the debugged software fault type. Moreover, a small portion of software fault retained at the end of Phase I & II with the realistic consideration. Thus, a two-phase software reliability model is proposed.

Limitations also exist in this study. First, we assume only Type II (dependent) software faults will be detected in Phase II. We could be able to detect both Type I (independent) fault and Type II (dependent) fault in Phase II in the real application, of which we have not considered in this chapter and will be addressed in the future research. Secondly, empirical data analysis is employed to find t_0 since we do not know this value. In reality, we are expecting software testers have an estimate of this value based on their experience.

CHAPTER 6

MULTI-RELEASE SOFTWARE RELIABILITY MODELING INCORPORATING DEPENDENT SOFTWARE FAULT DETECTION PROCESS

6.1 Research Motivation

In the previous chapter, we have discussed software reliability models subject to software fault dependency along with imperfect fault removal process in one/two phase software debugging process for a single-release software product. In addition, we notice that it still exists a portion of software faults have not been detected or have already been detected but not removed at the end of testing phase due to the limited testing time and resources, the nature of software and market requirement.

Thereby, it is the time for software company to decide either develop a new product or just release new version of the current product to solve issues brought by the remaining faults in the program, which could cause failures under certain configurations in the operation phase.

As software development moves further away from the rigid and monolithic model, the importance of software multiple release is brought to the vanguard [147]. Most of the software organizations release the initial version with sufficient functionalities to meet the customer requirements and occupy a certain portion of market share at first. However, it is unlikely to deliver all features that customers wanted in the single release given the

consideration of limited budget, unavailable resource, estimated risk, and constrained schedules.

Staying competitive in the market and keeping profitable for a software product unlikely happen in this increasing-innovational society if only has a single release especially when rival has a new release carrying more attractive features and satisfying more customer requirements [147]. From this point of view, multiple-release planning not only makes software organization easily balance the competing stakeholder's demands and benefits according to the available resource but lower the risk of not satisfying customer requirements [40, 41].

On the other hand, large software system continually needs to align with the changing customer requirements for the sake of market share. In order to get the feedback earlier and figure out what customer really wants, and assigning a lower software development cost, with a certain portion of increments on requirement for multiple release product is essential for the growth of an organization [42 - 44, 167]. Thus, it is plausible for software company to modify the parts of the existing modules to extend the current functionality, usability, and understandability by adding new features and correcting the issues from previous release [45, 46].

Since multi-release is critical for modern software product, release planning is becoming a popular research topic in the past few years. Release planning is a very complex problem. It has to take into account the consequence of feedback and update from customers, the

demands of potential customers, market feedback, defects from the previous release and other technical and non-technical constraints [40, 45]. Many researchers have studied software release planning problems [39 - 43, 168, 169].

It is generally considered reliability as a key factor in software quality measurement owing to the fact that it qualifies software failures and misbehaviors. Nonhomogeneous Poisson process (NHPP) is considered as one of the most effective models to study software reliability, as discussed in Chapter 2. Nevertheless, most of them only can be applied on a single release. How to model software reliability based on a multiple release perspective just starts gaining researcher's attention not very long.

In this chapter, we take into account two types of software faults for developing the next release: (1) *faults from previous releases*: remaining faults from previous release since it is unlikely to detect and remove all faults within limited resources; (2) *newly introduced faults*: new features are added in the next release, which also brings new software faults into the next release. We also assume that the detection of software fault for the next release's development depends on the detection of the remaining faults from previous release and the newly introduced faults. To the extent of our knowledge, we have not seen any research focus on the remaining faults from previous release, newly introduced faults, and dependent fault detection process in multi-release software reliability modeling.

6.2 Multi-Release Software Reliability Model Framework

Multi-Release Software Reliability Model

The notations of this section are given as follows.

$a(t)$	Total fault content function from previous release
$b(t)$	Total fault content function for the newly added features
$d(t)$	Fault detection rate function for the next release
$m(t)$	Expected number of software failures by time t
$N(t)$	Total number of software failures in the time interval $[0, t]$
$\lambda(t)$	Failure intensity function $\lambda(t) = d[m(t)]/dt$
m_0	Expected number of software failures at $t = 0$
C_0	Coefficient association with the general function
a	Total fault content from previous release
b	Total fault content from newly added features
d	Fault detection rate for the development of the next release

It is unlikely to get bug-free software product within limited resources and tightened schedules. Software detection process still follows a NHPP process for developing the next release. The cumulative number of detected faults $N(t)$ follows Poisson Process, presented as follows

$$\Pr\{N(t) = n\} = \frac{\big(m(t)\big)^n \exp\big(-m(t)\big)}{n!}, \qquad \text{for } n = 0,1,2, \ldots$$

where $m(t)$ is the mean value function of the counting process $N(t)$.

Two types of software faults will be addressed in this chapter. Remaining faults from previous release (Part I) and the newly introduced faults (Part II) will be both incorporated with the aim of developing the next release. Fault detection is a dependent process. We assume the detection of a software fault in the development of the next release depends on the fault detected from Part I and Part II.

Thus, the multi-release software reliability modeling is formulated as follows

$$\frac{dm(t)}{dt} = d(t)[a(t) - m(t)][b(t) - m(t)]m(t) \tag{6.1}$$

where $m(t)$ represents the expected number of software failures by time t. $d(t)$ denotes the fault detection rate function. $a(t)$and $b(t)$ represent the total remaining faults from previous release, and the total fault content of the newly added features, respectively. In this model, we assume

$$a(t) = a, \qquad b(t) = b, \qquad d(t) = d \tag{6.2}$$

where a is the total fault content from previous release, b is the total fault content from the newly added features and d is the fault detection rate for the development of the next release.

Substituting equation (6.2) into equation (6.1), we obtain a general solution for the mean value function $m(t)$, given as follows

$$e^{dt+C_0} = m(t)^{\frac{1}{ab}}[m(t)-a]^{\frac{1}{a(a-b)}}[m(t)-b]^{-\frac{1}{b(a-b)}} \tag{6.3}$$

where C_0 is a constant. In this study, we consider the initial solution of the function $m(t)$ is given as follows

$$m(t=0) = m_0 \tag{6.4}$$

where m_0 is unknown and $m_0 \geq 0$. At time $t = 0$, the expected number of initial software failures is m_0. Since multiple software releases are considered in this study, the expected number of software failures at the beginning of next release should be less than or equal to the expected number of failures at the end of previous release. We are also supported by references [40, 45, 46, 48 - 50] by stating that it is unlikely to remove all the software faults for each release due to the limitation of all available resource, including software programmer's domain knowledge and other environmental factors.

Substituting equation (6.4) into equation (6.3), we obtain

$$e^{C_0} = m_0{}^{\frac{1}{ab}}(m_0-a)^{\frac{1}{a(a-b)}}(m_0-b)^{-\frac{1}{b(a-b)}} \tag{6.5}$$

Thus, the solution for the function $m(t)$, as seen in equation (6.1), is obtained by solving the following equation

$$e^{dt} = \left[\frac{m(t)}{m_0}\right]^{\frac{1}{ab}} \left[\frac{m(t)-a}{m_0-a}\right]^{\frac{1}{a(a-b)}} \left[\frac{m(t)-b}{m_0-b}\right]^{-\frac{1}{b(a-b)}} \tag{6.6}$$

Multi-Release Software Reliability Function Discussion

Let

$$g(t) = e^{dt+C_0} \tag{6.7}$$

and

$$f(x) = x^{\frac{1}{ab}} (x-a)^{\frac{1}{a(a-b)}} (x-b)^{-\frac{1}{b(a-b)}} \tag{6.8}$$

We now present the following results.

Lemma 1: The solution $m(t)$ of the equation (6.3) will be obtained by solving the following function

$$g(t) = f[m(t)] \tag{6.9}$$

and

- If $g(0) > \max\{f[m(t)]\}$, then there exists no solution, as illustrated in Figure 6.1.
- Otherwise, there exists at least one solution for the function *m*(*t*), as illustrated in Figure 6.2.

where the function $g(t)$ and $f(x)$ are given in equations (6.7) and (6.8), respectively.

We need to prove: (1) Function $g(t)$, as stated in equation (6.7), is convex. (2) Function $f(x)$, as stated in equation (6.8), is concave.

Proof of Lemma 1:

(1) Since *d* is non-negative and $g^{"}(t) = d^2 e^{dt+C_0} > 0$, thus function $g(t)$ is convex.

(2) Since $f(x) = x^{\frac{1}{ab}} (x-a)^{\frac{1}{a(a-b)}} (x-b)^{-\frac{1}{b(a-b)}}$,

Then

$$f'(x) = \frac{1}{ab} x^{\frac{1}{ab}-1} (x-a)^{\frac{1}{a(a-b)}} (x-b)^{-\frac{1}{b(a-b)}} + \frac{1}{a(a-b)} x^{\frac{1}{ab}} (x-a)^{\frac{1}{a(a-b)}-1} (x-b)^{-\frac{1}{b(a-b)}} - \frac{1}{b(a-b)} x^{\frac{1}{ab}} (x-a)^{\frac{1}{a(a-b)}} (x-b)^{-\frac{1}{b(a-b)}-1}$$

Then

$$f'(x) = x^{\frac{1}{ab}}(x-a)^{\frac{1}{a(a-b)}}(x-b)^{-\frac{1}{b(a-b)}}\left[\frac{1}{abx} + \frac{1}{a(a-b)(x-a)} - \frac{1}{b(a-b)(x-b)}\right]$$

$$= x^{\frac{1}{ab}}(x-a)^{\frac{1}{a(a-b)}}(x-b)^{-\frac{1}{b(a-b)}}\frac{(ab+a+b)-(a+b)x}{abx(x-a)(x-b)}$$

$$= f(x)\frac{(ab+a+b)-(a+b)x}{abx(x-a)(x-b)}$$

$$= [f(x)]^{-1}\frac{(ab+a+b)-(a+b)x}{ab}$$

Thus

$$f''(x) = -[f(x)]^{-2}f'(x)\frac{(ab+a+b)-(a+b)x}{ab} - [f(x)]^{-1}\frac{a+b}{ab}$$

$$= -\frac{f'(x)\left\{[f(x)]^{-1}\frac{(ab+a+b)-(a+b)x}{ab}\right\}}{f(x)} - [f(x)]^{-1}\frac{a+b}{ab}$$

$$= -\frac{[f'(x)]^2}{f(x)} - \frac{\frac{a+b}{ab}}{f(x)} < 0$$

Note that $f(x) = g(t) > 0$ in equation (6.3), $a > 0$ and $b > 0$, hence, function $f(x)$ in equation (6.8) is concave.

Since the function $g(t)$ is convex, and $f(x)$ is concave, the results in Lemma 1 follow accordingly.

Q.E.D

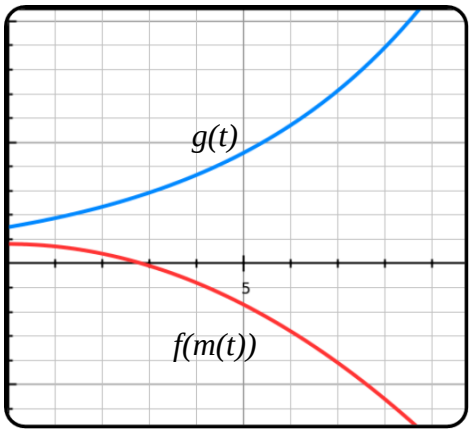

Figure 6. 1 Illustration of solution – Part I

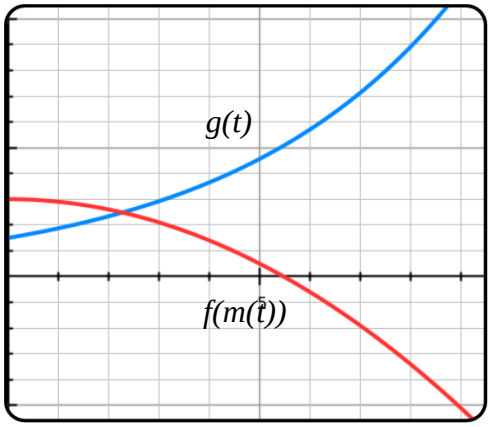

Figure 6. 2 Illustration of solution – Part II

6.3 Parameter Estimation and Comparison Criteria

Parameter Estimation

Most software reliability models use the least square estimate (LSE) or maximum likelihood estimation to estimate the parameters carried in the model. For example, minimizing the equation (6.10) or maximizing the equation (6.11).

$$f(t) = \sum_{i=1}^{n} (m(t_i) - y_i)^2 \tag{6.10}$$

$$LLF = \sum_{i=1}^{n} (y_i - y_{i-1}) \log[m(t_i) - m(t_{i-1})] - m(t_n) - \sum_{i=1}^{n} \log(y_i - y_{i-1})! \tag{6.11}$$

We apply LSE to minimize the equation (6.12) to estimate the parameters. Since $g(t) = f(m(t))$, indeed, $log[g(t)] = log[f\big(m(t)\big)]$. The optimization function is given by

$$\min S(a, b, d) = \sum_{a,b,d} \{\log[f(y_i)] - \log[g(t_i)]\}^2 \tag{6.12}$$

where y_i is the observed number of failures at time t_i. $g(t_i) = e^{dt_i + C_0}$.

The *Lemma 1* presented in last section is to demonstrate that there could exist the solutions and explain the behaviors of the proposed model. The Genetic Algorithm (GA) is employed to solve the optimization function as given in equation (6.12). The schematic diagram of the algorithm [170] is described in Figure 6.3. We use Matlab Optimization Toolbox to solve the optimization function and estimate parameters.

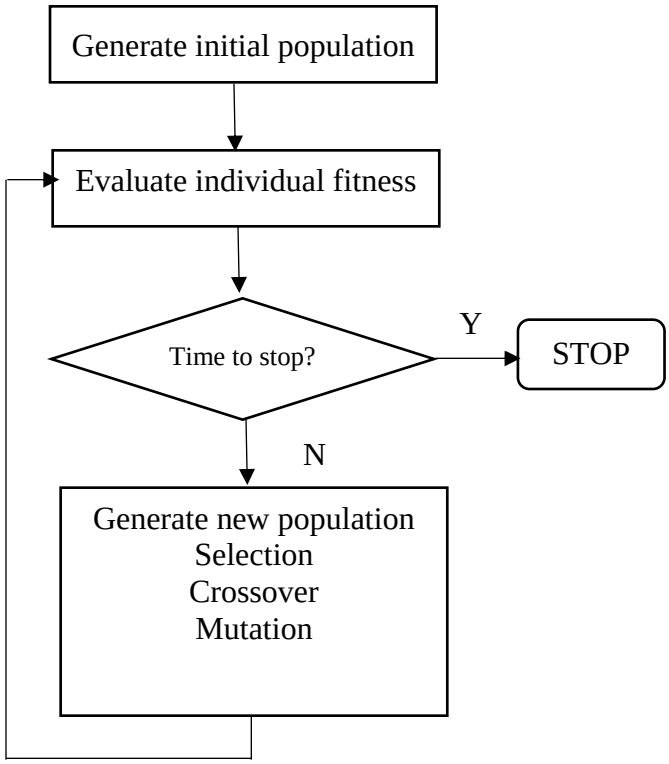

Figure 6. 3 Schematic diagram of Generic Algorithm

Comparison Criteria

The detailed discussion of comparison criteria is described in Chapter 5.

6.4 Numerical Examples

Two numerical applications are given to validate the multi-release software reliability model. We employ two datasets both collected from Open Source Software (OSS) project. OSS is a new way to build a global-based large software system, which differs in many perspectives with the traditional software engineering [171]. The evolution process of OSS

is much faster than the traditional close source software. Widespread OSS projects bring in a great change in terms of software development paradigms and software architectures [171 - 173].

Numerical Example 1

The Juddi OSS project data, adopted as numerical example 1, is shown in Table 6.1. Failure dataset from week 1 to week 31 are considered as Release 1; failure dataset from week 32 to week 49 are considered as Release 2; failure dataset from week 50 to week 61 are considered as Release 3. In this chapter, we use Release 2 to validate our proposed model.

First, Table 6.2 summarizes all the models we will compare for those two numerical examples. We notice that the proposed model has the best performance in terms of all criteria present in Table 6.3. Figure 6.4 illustrates the comparison between model prediction and observed failure data. The *x*-axis represents week index. All models presented in Table 6.3 only consider single-release software product except the proposed model. In other words, they didn't consider the remaining faults from the previous release since most of models assume all software faults will be removed before software company release the product.

Table 6. 1 Failure data of numerical example 1

Week	Cumulative Failures	Week	Cumulative Failures	Week	Cumulative Failures	Week	Cumulative Failures
1	10	17	131	33	210	49	221
2	12	18	136	34	210	50	234
3	20	19	137	35	211	51	249
4	31	20	137	36	213	52	267
5	33	21	139	37	213	53	273
6	41	22	144	38	214	54	279
7	47	23	155	39	217	55	290
8	54	24	160	40	217	56	297
9	64	25	160	41	217	57	314
10	74	26	169	42	218	58	336
11	77	27	170	43	218	59	345
12	99	28	180	44	218	60	387
13	110	29	193	45	218	61	393
14	118	30	194	46	221	-	-
15	120	31	195	47	221	-	-
16	127	32	210	48	221	-	-

Table 6. 2 Software reliability models

Model	Mean value function
G-O model	$m(t) = a(1 - e^{-bt})$
Inflection S-shaped model	$m(t) = \frac{a(1 - e^{-bt})}{1 + \beta e^{-bt}}$
Delayed S-shaped model	$m(t) = a(1 - (1 + bt)e^{-bt})$
Yamada imperfect debugging model	$m(t) = a[1 - e^{-bt}]\left[1 - \frac{\alpha}{b}\right] + \alpha at$
PNZ model	$m(t) = \frac{a[(1 - e^{-bt})\left(1 - \frac{\alpha}{b}\right) + \alpha t]}{1 + \beta e^{-bt}}$
Pham-Zhang IFD	$m(t) = a - ae^{-bt}[1 + (b + d)t + bdt^2]$
Dependent-parameter model	$m(t) = \alpha(1 + \gamma t)(\gamma t + e^{-\gamma t} - 1)$
Proposed Model	$e^{dt} = \left[\frac{m(t)}{m_0}\right]^{\frac{1}{ab}} \left[\frac{m(t) - a}{m_0 - a}\right]^{\frac{1}{a(a-b)}} \left[\frac{m(t) - b}{m_0 - b}\right]^{-\frac{1}{b(a-b)}}$

Table 6. 3 Parameter estimates and model comparison of numerical example 1

Model	**Parameter Estimates**	**MSE**	**PRR**	**PP**	**Variation**
G-O model	$\hat{a} = 227.000$ $\hat{b} = 2.420$	131.840	0.041	0.046	4.799
Inflection S-shaped model	$\hat{a} = 230.000$ $\hat{b} = 9.070$ $\hat{\beta} = 830.670$	226.793	0.064	0.074	4.935
Delayed S-shaped model	$\hat{a} = 225.010$ $\hat{b} = 4.330$	93.698	0.030	0.033	4.304
Yamada imperfect debugging model	$\hat{a} = 231.000$ $\hat{b} = 2.180$ $\hat{\alpha} = 3.64 \times 10^{-5}$	254.213	0.072	0.083	5.747
PNZ model	$\hat{a} = 229.000$ $\hat{b} = 6.380$ $\hat{\alpha} = 1 \times 10^{-6}$ $\hat{\beta} = 52.690$	211.036	0.056	0.064	4.787
Pham-Zhang IFD model	$\hat{a} = 234.000$ $\hat{b} = 9.200$ $\hat{d} = 98.950$	368.638	0.101	0.120	5.601
Dependent-parameter model	$\hat{\alpha} = 364.850$ $\hat{\gamma} = 0.063$	23163.480	85316.330	8.153	108.378
Proposed Model	$\hat{a} = 254.840$ $\hat{b} = 221.000$ $\hat{m}_0 = 100.000$ $\hat{d} = 2.36 \times 10^{-4}$	34.467	0.011	0.011	3.322

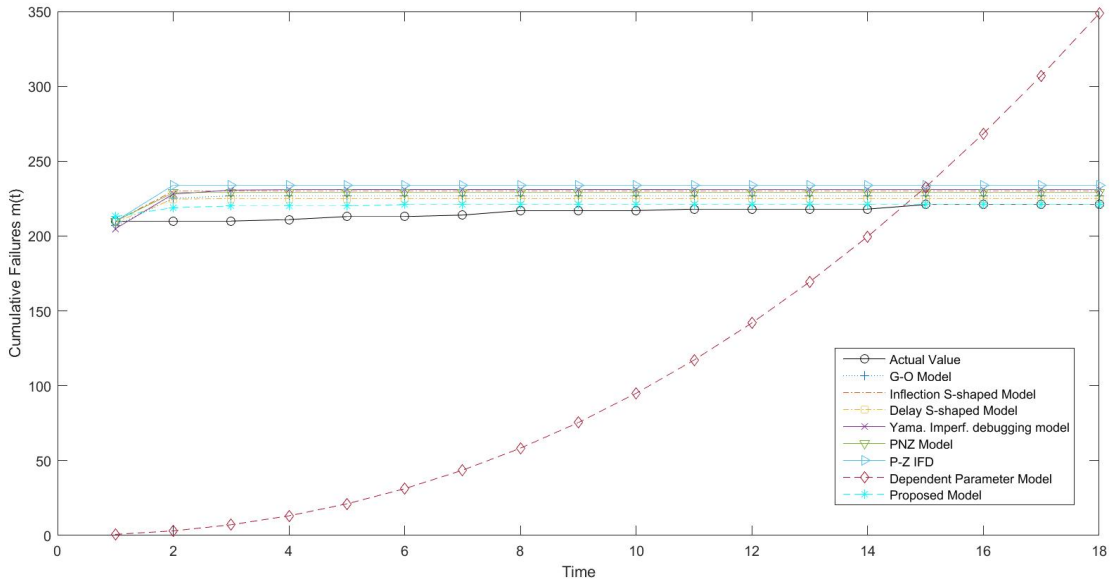

Figure 6. 4 Comparison of proposed model and other models of numerical example 1

Numerical Example 2

Apache 2.0 [174] is available on the website since 2002. The first two releases are employed to verify the proposed model, as shown in Table 6.4. The failure data from day 1 to day 18 is taken into account as Release 1; failure data from day 19 to day 164 is considered as Release 2.

The proposed model provides the smallest MSE, PP, and Variation, as seen in Table 6.5. The PRR value even though is not the smaller one, however, 0.155 is just slightly higher than 0.117. It is thus considering the proposed model presents the best performance to model this dataset. Figure 6.5 also plots the comparison between the predicted values and the observed values to provide an intuitive sense of model fitting. The *x*-axis represents week index in the figure.

Table 6. 4 Failure data of numerical example 2

Day	Cumulative Failures	Day	Cumulative Failures	Day	Cumulative Failures	Day	Cumulative Failures
1	1	15	20	28	36	49	51
2	3	16	22	29	37	50	52
3	5	17	25	30	39	51	53
4	8	18	26	31	40	57	54
5	11	19	27	32	41	66	55
7	13	22	30	35	44	70	56
8	14	23	31	38	45	81	57
9	15	24	32	39	46	164	58
10	16	25	34	42	47	-	-
11	17	26	35	43	48	-	-

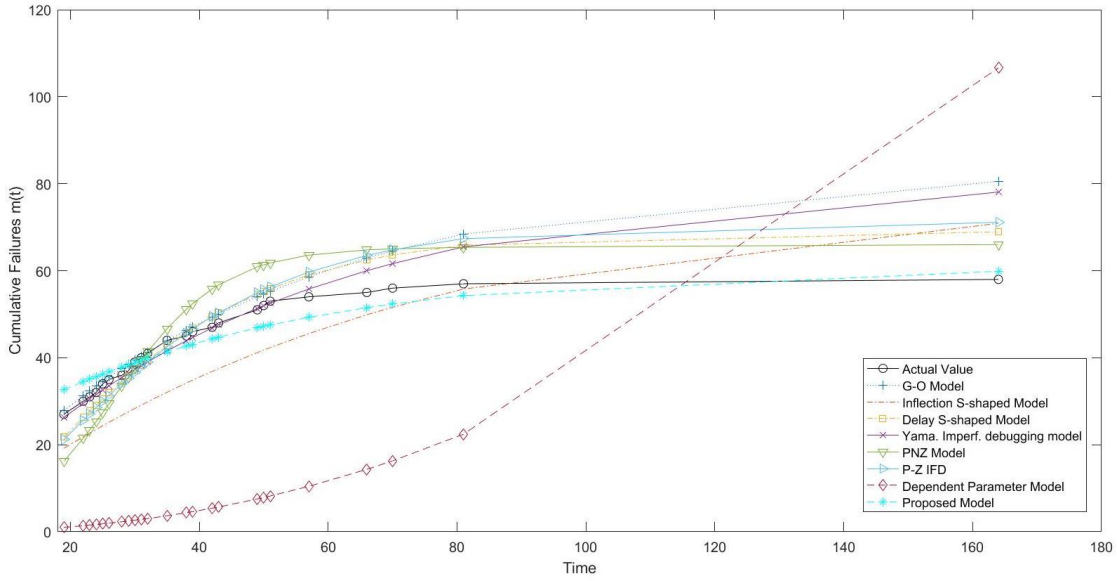

Figure 6. 5 Comparison of proposed model and other models of numerical example 2

Table 6. 5 Parameter estimates and model comparison of numerical example 2

Model	**Parameter Estimates**	**MSE**	**PRR**	**PP**	**Variation**
G-O model	$\hat{a} = 82.990$ $\hat{b} = 0.022$	38.317	0.169	0.268	8.449
Inflection S-shaped model	$\hat{a} = 75.017$ $\hat{b} = 0.019$ $\hat{\beta} = 0.274$	104.096	2.299	1.304	17.380
Delayed S-shaped model	$\hat{a} = 69.000$ $\hat{b} = 0.060$	21.784	0.230	0.223	4.930
Yamada imperfect debugging model	$\hat{a} = 79.690$ $\hat{b} = 0.021$ $\hat{\alpha} = 1 \times 10^{-4}$	26.670	0.117	0.178	5.241
PNZ model	$\hat{a} = 65.000$ $\hat{b} = 0.121$ $\hat{\alpha} = 1 \times 10^{-4}$ $\hat{\beta} = 26.178$	64.302	1.129	0.754	8.484
Pham-Zhang IFD model	$\hat{a} = 71.200$ $\hat{b} = 0.058$ $\hat{d} = 9.15 \times 10^{-5}$	30.828	0.309	0.300	5.656
Dependent-parameter model	$\hat{\alpha} = 101.110$ $\hat{\gamma} = 0.007$	1587.346	3843.289	18.719	71.169
Proposed Model	$\hat{a} = 90.038$ $\hat{b} = 61.001$ $\hat{m}_0 = 20.000$ $\hat{d} = 1.165 \times 10^{-5}$	13.197	0.155	0.171	3.630

In summary, the proposed model has considered a dependent fault detection process. Specifically, the newly detected faults depend on the detection of the remaining faults from previous release and the newly introduced faults. In order to detect a new fault, we need to detect the corresponded faults from the remaining faults from previous release and the newly introduced faults first. Therefore, there is only a small portion of software faults detected for developing the next release.

6.5 Conclusions

It is unlikely to deliver all the features in a single release for modern software products. The proposed software reliability model provides a new paradigm to integrate the dependent fault detection process and different types of software faults in multiple software releases. Due to the resource limitation, there also exists a portion of undetected software faults for the current release. Thus, how to incorporate the remaining faults from previous release into the development for the next release is an important issue for software practitioners.

As an effort to reflect the development of multi-release software, the remaining faults from previous release, the newly introduced faults, and the dependent fault detection process are discussed in this chapter. In order to accurately illustrate the performance of the proposed model, we employ two datasets both collected from OSS project to validate the usage of the model. The behavior of software reliability function is studied as well. We are currently investigating the new features adding in the next release and the remaining faults from previous release as fixed numbers in this study, which can be extended as a random number or as a time-dependent function corresponding to its optimal profit and release time for the organization. The impact of environmental factors [29, 30] during the software development process can be considered into the future research as well.

CHAPTER 7

MARTINGALE-BASED SOFTWARE RELIABILITY MODEL INCORPORATING SINGLE/MULTIPLE ENVIRONMENTAL FACTOR(S)

7.1 Research Motivation

The increasing power and versatility of software systems has led to their widespread applications in our modern society [176]. Since software system has become one of the essential elements in different aspects of our daily life and an important factor in numerous critical industrial application and software system is expected to be even more ubiquitous in the coming years, there are a great demand for high-quality software products [177, 178]. However, delivering high quality software products for real-world applications is not easy [179].

Despite of being widely studied and of interested to the global market, software quality is still a complex and costly task for the researchers and practitioners. Meanwhile, we are facing an increasing-complexity software development environment due to the three trends discussed in Chapter 1. The first trend is the wide-spread adoption of software product lines. The adoption of software product lines will reduce the release time to the market, decrease the development cost, optimize resource assignment, and achieve the commonality in user experience between different products. However, the adoption of software product lines also brings a new level of dependency into the product and organization, which causes the added complexity for software development.

The second trend is the globalization of software development within many organizations. Many software companies have either placed several software development sites globally or have partnered with remoted companies, located mostly in India and China. However, it could increase the complexity of the product and organization as well.

The third trend is the adoption of software ecosystems. In recent years, software development has transitioned from a predominantly solo activity of developing standalone programs within a single organization, to a highly distributed and collaborative environment that depends on or contributed to large and complex software ecosystems which could be placed world-wide [180]. Software developers are able to contribute to multiple projects, accordingly, the project boundaries blur, not just program architecture and paradigm, and even how they are authored. Software developers not only focus on how to write the code, but also the contribution they make, the connection that they build within development-related communities by establishing a participatory culture [181 - 183]. Since the development of new functionality can be occurred outside of the platform, App-store styled approaches are introduced by many companies to provide this feature to the market [184 – 186]. However, software ecosystems build dependencies between components and their associated organizations which did not exist earlier.

Hence, in recent years, most software developments have shifted their attention from building the system toward composing system from the existing open source, commercial,

and internal developed components. Under such transitions, the goal of software development will be more concentrated on the creation and functionality.

Given large-scale software development is an increasing-complexity, effort-consuming, and expensive activity, how we can assure software quality in one of the challenge problem in industry. One of the fundamental quality characteristics is the reliability. Many nonhomogeneous Poisson process (NHPP) software reliability growth models have been developed regarding various testing/operation scenarios for the sake of remaining faults estimation, failure rate prediction, and software reliability prediction given a specific time of interest since 1970s. However, most of existing NHPP software reliability models or software fault prediction frameworks based on neural networks, vector machines, or other machine learning techniques did not incorporate the impact of environmental factors in software reliability/fault prediction.

As discussed above, given the current trends of software development process, adoption of software product lines, software development globalization, and the establishment of software ecosystems, the complication and human-centered software development process needs to be addressed more appropriately in software reliability model in order to accurately predict failures. Thus, how to incorporate the environmental factors which present such significant impacts on reliability into software reliability models is critical to address modern software development in practice.

Thus, in the early stage of this dissertation, presented in Chapter 4, we studied the impact the environmental factors affecting software reliability in single-release and multi-release software development. In this chapter, we will incorporate single and multiple environmental factor(s) in software reliability models to improve the model prediction power and applicability.

First, we consider one of the top 10 critical environmental factors as concluded in Chapter 4 [29, 30], *Percentage of Reused Modules* (PoRM), to be a random variable which has random effect on software fault detection rate. The data collected from several industries is applied to obtain the distribution of PoRM. We then introduce the Martingale framework, specifically, Brownian motion and white noise process in the stochastic fault detection process to reflect the randomness resulting from influence of environmental factor. Thus, a single-environmental-factor software reliability model incorporating these considerations is proposed in Section 7.2.

Secondly, we consider multiple environmental factors which have significant impacts on software reliability during software development process into software reliability models. Martingale framework, in particular, Brownian motion, and white noise process is introduced to reflect the stochastic fault detection process and the randomness caused by these environmental factors. To the best of our knowledge, we have not seen any research incorporates multiple environmental factors and their random impact on fault detection process in software reliability models. The proposed generalized multiple-environmental-factors software reliability model incorporating these considerations will be presented in

Section 7.3. We further present two specific software reliability models incorporating two important factors from previous study, PoRM, and *Frequency of Program Specification Change* (FoPSC) in Section 7.3.

Numerical examples for the single-environmental-factor and multiple-environmental-factors software reliability model are illustrated in Sections 7.4 and 7.5, respectively. Moreover, we are interested in the predictive accuracy comparison of the proposed models with and without considering environmental factor(s), therefore, failure prediction comparison is discussed in Section 7.6. We conclude this chapter in Section 7.7.

7.2 Single-Environmental-Factor Software Reliability Model

In this section, we consider environmental factor, PoRM, to be a random variable, performs random effect on software fault detection rate. The distribution of PoRM will be determined in Section 7.2.1, given the data collected in a wide variety of industries. The introduction of Martingale process and the framework of other software reliability models considering random environments are discussed in Section 7.2.2. Finally, we propose a single-environmental-factor software reliability model in Section 7.2.3.

The notations that are used in this section is explained as follows.

$m(t,\eta)$	Expected number of software failure detected by time t considering environmental factor and its impact
$N(t)$	Total fault content function in the software by time t

$h(t,\eta)$	Time-dependent fault detection rate per unit of time considering random effect
$h(t)$	Time-dependent fault detection rate per unit of time
$G(t,\eta)$	Impact of PoRM on failure detection rate per unit of time
$M(t,w)$	Martingale with respect to the filtration $(\mathrm{F}_t, t \geq 0)$
$\dot{M}(t,w)$	Derivative of $M(t,w)$ with respect to time t
$f(\eta)$	Probability density function of PoRM
$v(t)$	Measures the impact of time on PoRM
$F^*(s)$	Laplace transform of $f(\eta)$
$B(t)$	Brownian motion
$\dot{B}(t)$	Standard white noise
δ	Dirac Delta measure
θ, γ	Parameters of gamma distribution
η	Random variable, which represents PoRM
λ_0	Coefficient along with $G(t,\eta)$
b, c, a, k	Coefficient in function $h(t)$, $v(t)$, and $N(t)$
y_i	Observed number of software failures at time t_i

Environmental Factor (PoRM) Distribution

Although some existing software reliability models have considered the environmental factors in the modeling, most of them assume the distribution of environmental factor based on their knowledge and model consideration. For example, Teng and Pham [103] described

the random environmental factor is 1 in in-house testing phase and a random variable in operation phase. They assumed that this random variable follows gamma distribution or beta distribution in their proposed reliability models. But, there is no real data to support this assumption.

In this study, the definition of PoRM, adopted from references [27 - 30], is presented as follows

$$PoRM = \frac{S_0}{S_N + S_0} \tag{7.1}$$

where S_0 represents kiloline of code for the existing modules and S_N represents kiloline of code for the new modules.

In this section, we employ the real-world data collected from several industries to illustrate the distribution of PoRM. Participants were asked to provide PoRM in their industries based on the relevant working experience. All the participants are currently working in IT Department in various industry including computer software, banking, semiconductor, online retailing, IT service & research institution or working on software development in high-tech company in favor of the validity and reliability of the survey response. The collected PoRM are illustrated in Figure 7.1.

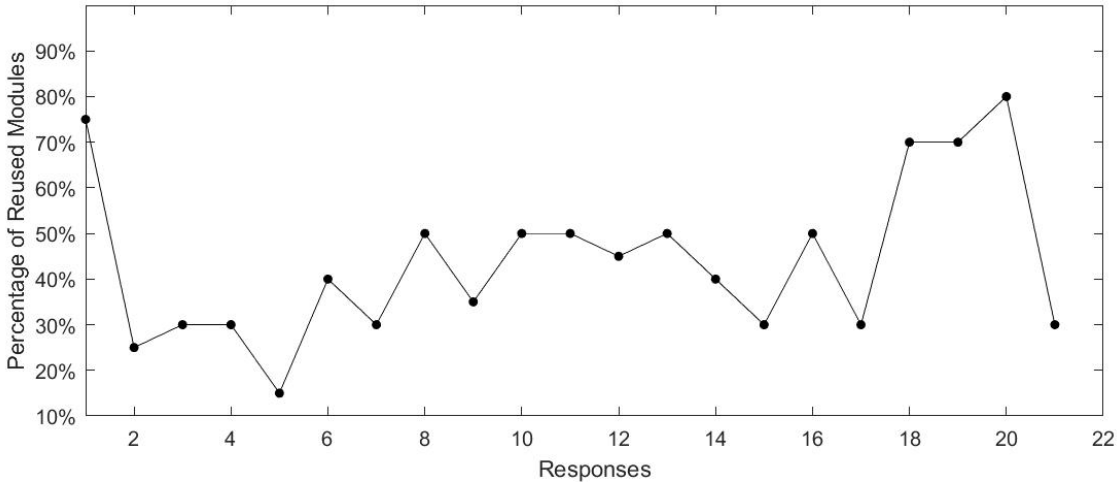

Figure 7. 1 Data collection of PoRM

Eight different distributions commonly applied to model environmental factors are adopted in the model comparison to estimate the distribution of PoRM. The maximum likelihood estimation (MLE) is applied to compare the effectiveness of each model. The log-likelihood values for all models are revealed in Table 7.1.

Table 7. 1 Log-likelihood value comparison

Normal Dist.	**Weibull Dist.**	**Beta Dist.**	**Exponential Dist.**	**Gamma Dist.**	**Inverse Gaussian Dist.**	**Log-logistic Dist.**	**Lognormal Dist.**
7.041	7.639	7.830	-3.782	8.174	8.002	7.739	8.015

Gamma distribution will be employed to model the distribution of PoRM. The parameters of gamma distribution are also obtained from model comparison: $\theta = 14.726, \gamma = 6.487$.

$$f(\eta) = \frac{\theta^{\gamma}\eta^{\gamma-1}e^{-\theta\eta}}{\Gamma(\gamma)} \tag{7.2}$$

where η is a random variable and represents PoRM. θ and γ are the parameters along with gamma distribution, respectively.

Related Works

We first provide the definition of software fault and software failure in order to clearly address the software reliability models discussed below. Software fault is caused by an incorrect step, process, or data definition in a computer program [187]. Failure is defined as a system or component is not able to perform its required functions within specified performance requirements. Software failure is the manifestation of software fault [187].

Many NHPP software reliability models have been proposed for the past four decades to address different assumptions. The failure processes are described by NHPP property with the mean value function at time t, $m(t)$, and the failure intensity of the software, $\lambda(t)$, which is also the derivative of the mean value function. Most existing NHPP software reliability models are developed based on the model given as follows

$$\frac{d}{dt}m(t) = h(t)[N(t) - m(t)] \tag{7.3}$$

where $m(t)$ is the expected number of software failures detected by time t, $N(t)$ is the total number of fault content by time t, and $h(t)$ is the time-dependent fault detection rate

per unit of time. The underlying assumption of equation (7.3) is the failure intensity is proportional to the residual fault content in the software. Many NHPP software reliability models are developed based on the different formulation of $h(t)$ and $N(t)$ in equation (7.3). Some examples are given in Table 7.2.

Table 7. 2 Some existing NHPP software reliability models

NHPP model	**Functions**	**Mean value function**
Geol-Okumoto	$h(t) = b$ $N(t) = a$	$m(t) = a(1 - e^{-bt})$
Delayed S-shaped	$h(t) = \frac{bt^2}{1+bt}$ $N(t) = a$	$m(t) = a(1 - (1+bt)e^{-bt})$
Inflection S-shaped	$h(t) = \frac{b}{1+\beta e^{-bt}}$ $N(t) = a$	$m(t) = \frac{a(1-e^{-bt})}{1+\beta e^{-bt}}$
Yamada imperfect debugging	$h(t) = b$ $N(t) = a(1+\alpha t)$	$m(t) = a[1 - e^{-bt}]\left[1 - \frac{\alpha}{b}\right] + \alpha at$
PNZ	$h(t) = \frac{b}{1+\beta e^{-bt}}$ $N(t) = a(1+\alpha t)$	$m(t) = \frac{a[(1-e^{-bt})\left(1-\frac{\alpha}{b}\right)+\alpha t]}{1+\beta e^{-bt}}$
Pham-Zhang IFD	$h(t) = \frac{b^2 t}{1+bt} + \frac{d}{1+dt}$ $N(t) = a$	$m(t) = a - ae^{-bt}(1 + (b+d)t + bdt^2)$

The software reliability models presented in Table 7.2, along with other deterministic models, have not taken into account the effect of randomness from the development/operation environment. To capture the effect of random environments, a stochastic process is incorporated in the fault detection process by imposing $h(t)$ to be $h(t, \eta)$, where η is the random effect. The equation (7.3) will be described as

$$\frac{d}{dt}m(t,\eta) = h(t,\eta)[N(t) - m(t,\eta)] \tag{7.4}$$

For example, in Pham [136], the author considered

$$h(t,\eta) = h(t)\,\eta, \quad N(t) = N \tag{7.5}$$

where η is a random variable and follows gamma distribution with parameter α and β. By substituting equation (7.5) to equation (7.4), the explicit solution of the mean value function is expressed as [136]

$$m(t) = N\left[1 - \left(\frac{\beta}{\beta + \int_0^t h(s)ds}\right)^{\alpha}\right] \tag{7.6}$$

The above formulation, $h(t,\eta) = h(t)\,\eta$, proposed in Pham [136], is referred as dynamic multiplicative noise model in recent publication from Pham and Pham [188]. They [188] also proposed a dynamic additive noise model shown as follows

$$h(t,w) = h(t) + \dot{M}(t,w) \tag{7.7}$$

where $\dot{M}(t,w)$ denotes the derivative of M with respect to time t. $M(t)$ is defined as a martingale with respect to the filtration $(\mathcal{F}_t, t \geq 0)$ in this function. One worth-noting martingale property [189, 190] applied in equation (7.7) is

$$E\left(\int_v^t h(s,w)ds\right) = \int_v^t h(s)ds \tag{7.8}$$

Brownian motion [189] is also a martingale. Let $\{B(t): t \geq 0\}$ be Brownian motion, Mikosch [189] mentioned that $\{B(t): t \geq 0\}$ and $\{B^2(t) - t: t \geq 0\}$ are martingale with respect to the nature filtration $(\mathbb{F}_t, t \geq 0)$. For the model consideration in this chapter, we also choose $M(t)$ to be Brownian motion since Brownian motion is deeply researched by many literatures.

A stochastic process $\{B(t): t \geq 0\}$ is called Brownian motion [190] with start in $x \in \mathbb{R}$ if

(1) $B(0) = x$.

(2) The process has independent increments. For example, the increments $B(t_n) - B(t_{n-1}), B(t_{n-1}) - B(t_{n-2}), \ldots, B(t_2) - B(t_1)$ are independent for all time $0 \leq t_1 \leq t_2 \leq \cdots \leq t_n$.

(3) The increments $B(t+s) - B(t)$ are normally distributed, for all $t \geq 0, s \geq 0$.

If $B(0) = 0,$ i.e. the motion starts from the origins, this is called standard Brownian motion.

Proposed Single-Environmental-Factor Software Reliability Model

The proposed NHPP software reliability model in this study incorporates both the dynamic multiplicative and additive noise characteristics discussed in Section 7.2.2. Meanwhile, one of the top 10 environmental factors from the recent comparison survey studies in Chapter 4, modeled as a gamma distribution with parameter θ and r estimated from real-collected

data from various industries, is incorporated in the model development. In sum, environmental factor, PoRM, described as gamma distribution from real data, and the randomness caused by PoRM, illustrated by the martingale, specifically, Brownian motion, are taken into account for the proposed software reliability model.

The assumptions for the proposed model are given as follows.

(1) The fault removal process follows a NHPP.
(2) The failure of software system subjects to the manifestation of the remaining faults in the software program.
(3) All faults in the software are independent.
(4) The failure intensity is proportional to the remaining faults in the software.
(5) The environmental factor, PoRM, is described as gamma distribution based on the real data collected from various industry. The impact of PoRM is explained by an additive portion along with the traditional fault detected process.
(6) The randomness caused by the environmental factor is explained as a martingale, particularly, Brownian motion in this study.

The proposed NHPP software reliability model is formulated as

$$\frac{d}{dt}m(t,\eta) = \left[h(t) + \lambda_0 G(t,\eta) + \dot{B}(t)\right][N(t) - m(t,\eta)] \tag{7.9}$$

$$\mathrm{m}(0) = 0 \tag{7.10}$$

where $m(t,\eta)$ is the expected number of failures by time t. $h(t)$ is the fault detection rate per unit of time without considering the effect of environmental factor. η is a random variable and represents the environmental factor, PoRM. $G(t,\eta)$ is a time-dependent function and represents the impact of PoRM on failure detection rate per unit of time. λ_0 is the coefficient along with $G(t,\eta)$. $\dot{B}(t)$ denotes a standard Gaussian white noise, specifically

$$\frac{d}{dt}B(t) = \dot{B}(t) \tag{7.11}$$

where $B(t)$ is a Brownian motion.

The covariance of $\dot{B}(t)$ is

$$E\left(\dot{B}(t)\dot{B}(u)\right) = \delta(u-t), 0 < t < u \tag{7.12}$$

where δ is the Dirac Delta measure.

From [188], similarly, we obtain the general solution for equation (7.9) as follows

$$m(t,\eta) = N(t) - N(0)e^{-\int_0^t \left(h(s)+\lambda_0 G(s,\eta)+\dot{B}(s)\right)ds} - \int_0^t e^{-\int_u^t \left(h(s)+\lambda_0 G(s,\eta)+\dot{B}(s)\right)ds} N'(u)du \tag{7.13}$$

The detailed derivation is presented in Appendix I, located at the last page of this chapter. Since $\dot{B}(t)$ denotes a standard Gaussian white noise, we also have

$$\int_0^t \big(h(s) + \lambda_0 G(s,\eta) + \dot{B}(s)\big)ds = \int_0^t \big(h(s) + \lambda_0 G(s,\eta)\big)ds + B(t) - B(0)$$
$$= \int_0^t \big(h(s) + \lambda_0 G(s,\eta)\big)ds + B(t) \quad (7.14)$$

The mean value function is expressed as

$$\overline{m}(t) = N(t) - N(0)e^{-\int_0^t h(s)ds} e^{\frac{t}{2}} e^{-\int_0^t \lambda_0 G(s,\eta)ds}$$
$$- \int_0^t e^{-\int_u^t h(s)ds} e^{\frac{t-u}{2}} e^{-\int_u^t \lambda_0 G(s,\eta)ds} N'(u)du \quad (7.15)$$

Let

$$G(t,\eta) = \eta v(t) \quad (7.16)$$

where $v(t)$ is a time-dependent function and measures the impact of time on PoRM.

The probability density function of the environmental factor, η, described as a gamma distribution with the parameter θ and r, is given as follows

$$f(\eta) = \frac{\theta^{\gamma}\eta^{\gamma-1}e^{-\theta\eta}}{\Gamma(\gamma)} \tag{7.17}$$

We apply the expectation on equation (7.15) with respect to η. The mean value function is obtained

$$\begin{aligned}\overline{m}_{\eta}(t) &= N(t) - N(0)e^{-\int_0^t h(s)ds}e^{\frac{t}{2}}\left[\int_0^{\infty} e^{-\eta\int_0^t \lambda_0 v(s)ds} f(\eta)\, d\eta\right] \\ &\quad - \int_0^{\infty}\int_0^{t} e^{-\int_u^t h(s)ds}e^{\frac{t-u}{2}}e^{-\int_u^t \lambda_0\eta v(s)ds}N'(u)f(\eta)du\, d\eta \\ &= N(t) - N(0)e^{-\int_0^t h(s)ds}e^{\frac{t}{2}}\left[\int_0^{\infty} e^{-\eta\int_0^t \lambda_0 v(s)ds} f(\eta)\, d\eta\right] \\ &\quad - \int_0^{t} N'(u)e^{-\int_u^t\left(h(s)-\frac{1}{2}\right)ds}\left[\int_0^{\infty} e^{-\eta\int_u^t \lambda_0 v(s)ds} f(\eta)d\eta\right]du\end{aligned} \tag{7.18}$$

Compute the equations with Laplace Transform

$$\int_0^{\infty} xe^{-sx}f(x)dx = -\frac{dF^*(s)}{ds} \tag{7.19}$$

The Laplace Transform of gamma probability density function is

$$F^*(s) = \left(\frac{\theta}{\theta+s}\right)^{\gamma} \tag{7.20}$$

Therefore

$$\overline{m}_{\eta}(t) = N(t) - N(0)e^{-\int_0^t h(s)ds}e^{\frac{t}{2}}F^*\left(\int_0^t \lambda_0 v(s)ds\right)$$

$$-\int_0^t N'(u)e^{-\int_u^t\left(h(s)-\frac{1}{2}\right)ds}F^*\left(\int_0^t \lambda_0 v(s)ds\right)du$$

$$= N(t) - N(0)e^{-\int_0^t h(s)ds}e^{\frac{t}{2}}\left[\frac{\theta}{\theta+\int_0^t \lambda_0 v(s)ds}\right]^{\gamma}$$

$$-\int_0^t N'(u)e^{-\int_u^t\left(h(s)-\frac{1}{2}\right)ds}\left[\frac{\theta}{\theta+\int_0^t \lambda_0 v(s)ds}\right]^{\gamma}du$$

$$= N(t) - N(0)e^{-\int_0^t h(s)ds}e^{\frac{t}{2}}\left[\frac{\theta}{\theta+\int_0^t \lambda_0 v(s)ds}\right]^{\gamma}$$

$$-\left[\frac{\theta}{\theta+\int_0^t \lambda_0 v(s)ds}\right]^{\gamma}\int_0^t N'(u)e^{-\int_u^t\left(h(s)-\frac{1}{2}\right)ds}du \qquad (7.21)$$

The equation (7.21) provides a general solution for the proposed single-environmental-factor software reliability model in considerations of PoRM and the randomness caused by this environmental factor. Different formulation for $h(t)$, $v(t)$ and $N(t)$ with respect to different assumptions can be plugged in equation (7.21) to obtain the final solution. In this study, we apply the formulation as follows.

Let

$$h(t) = \frac{b}{1+ce^{-bt}} \qquad (7.22)$$

$$v(t) = e^{-at} \qquad (7.23)$$

$$N(t) = \frac{1}{k} e^{kt} \tag{7.24}$$

As described in Melo et al. [191], the application process of the reused module is described as follows: (1) Reused module selection. (2) The adaption of the reused module to the new objective. (3) The integration to the new-developed software product. The impact of PoRM on the software fault detection rate will decrease as software development moves to the later phase. Thus, we use an exponentially decreasing function e^{-at} to represent $v(t)$ in this study. Meanwhile, due to $\frac{1}{k} e^{kt}$ is a monotonically increasing function and the inspiration from reference [188], we employ $\frac{1}{k} e^{kt}$ to represent $N(t)$, the nature growth of the software failures.

Substituting equations (7.22) - (7.24) to equation (7.21), the mean value function is expressed as

$$m(t,\eta) = \frac{1}{k} e^{kt} - \frac{1}{k} \frac{c+1}{c+e^{bt}} e^{\frac{t}{2}} \left[\frac{\theta}{\theta + \frac{\lambda_0}{a}(1 - e^{-at})} \right]^{\gamma}$$

$$- \left[\frac{\theta}{\theta + \frac{\lambda_0}{a}(1 - e^{-at})} \right]^{\gamma} \frac{e^{\frac{t}{2}}}{c+e^{bt}} \int_0^t N'(u) \left[c e^{-\frac{u}{2}} + e^{\left(b - \frac{1}{2}\right)u} \right] du$$

$$= \frac{1}{k}e^{kt} - \frac{e^{\frac{t}{2}}}{c+e^{bt}}\left[\frac{\theta}{\theta+\frac{\lambda_0}{a}(1-e^{-at})}\right]^{\gamma}\left[\frac{c+1}{k} - \frac{c}{k-\frac{1}{2}}e^{\left(k-\frac{1}{2}\right)t} - \frac{1}{b+k-\frac{1}{2}}e^{\left(b+k-\frac{1}{2}\right)t} + \frac{c}{k-\frac{1}{2}} + \frac{1}{b+k-\frac{1}{2}}\right] \quad (7.25)$$

Given the definition of $m(t)$ is the expected number of software failures by time *t*, we do know $m(t)$ is an increasing function and we are also able to prove $when\ t \to \infty, m(t) \to \infty$ for equation (7.25) based on the numerical example.

We believe the behavior of equations (7.21) and (7.25) mainly depends on the formulation of $N(t)$. For example,

(1) If $N(t)$ is a constant, then $m(t)$ converges to this constant.

(2) If $N(t)$ is a time-dependent function and can converge to a finite number, then $m(t)$ converges to this finite number.

(3) If $N(t)$ is a time-dependent function and cannot converge, then $m(t)$ cannot converge.

But in this dissertation, we are not going to cover the mathematical proof for the above description. The investigation of the function behavior will be further studied in the future research.

7.3 Multiple-Environmental-Factors Software Reliability Model

We first propose a generalized software reliability model considering multiple environmental factors under the Martingale framework in Section 7.3.1, and then we present two specific software reliability models incorporating two important factors from previous study, PoRM, and FoPSC.

The notations that are used in this section is explained as follows.

$m\left(t, \underbrace{\eta_1, \eta_2, \dots, \eta_n}\right)$	Expected number of software failure detected by time t considering multiple environmental factors and their impact
$G_i\left(t, \underbrace{\eta_1, \eta_2, \dots, \eta_n}\right)$	Impact of environmental factor *1, 2, 3, …, n* on failure detection rate per unit of time
$\mathrm{HG}([\beta_2], [\beta_1 + \beta_2], s)$	Generic hypergeometric function
$N(t)$	Total fault content function in the software by time t
$h(t)$	Time-dependent fault detection rate per unit of time
$f(\eta_i)$	Probability density function of environmental factor 1, 2, 3, …, n
$f(\eta_1)$	Probability density function of PoRM
$f(\eta_2)$	Probability density function of FoPSC
$B(t)$	Brownian motion
$\dot{B}(t)$	Standard white noise
$v_i(t)$	Impact of time on environmental factor η_i
$v_1(t)$	Impact of time on environmental factor η_1

$v_2(t)$	Impact of time on environmental factor η_2
$F^*_{\eta_i}(s)$	Laplace transform of $f(\eta_i)$
$F^*_{\eta_1}(s)$	Laplace transform of $f(\eta_1)$
$F^*_{\eta_2}(s)$	Laplace transform of $f(\eta_2)$
η_i	Random variable and represents environmental factor 1, 2, 3, …, n
η_1	Random variable and represents PoRM
η_2	Random variable and represents FoPSC
λ_i	Coefficient along with $G_1(t,\eta_1)$, $G_2(t,\eta_2)$, $G_3(t,\eta_n)$, … , $G_n(t,\eta_n)$
δ	Dirac Delta measure
θ_1, γ_1	Parameters of gamma distribution, which denotes PoRM
θ_2, γ_2	Parameters of gamma distribution, which denotes FoPSC
β_1, β_2	Parameters of beta distribution, which denotes FoPSC
b, c, a, k	Coefficients in function $h(t)$, $v(t)$, and $N(t)$
y_i	Observed number of software failures at time t_i

7.3.1 A Generalized Multiple-Environmental-Factors Software Reliability Model

We present the following assumptions to develop the generalized multiple-environmental-factors software reliability model.

(1) Software fault removal process follows the NHPP.

(2) Software failure intensity is proportional to the remaining faults in the software program.

(3) Software failures subject to the manifestation of the remaining faults in the software program.

(4) All software faults in the program are independent.

(5) Multiple environmental factors are considered in the proposed model. All environmental factors are independent in this study. We do not consider correlation between environmental factors in this study.

(6) The randomness imposed on the software fault detection rate, caused by the introduction of environmental factors, is modeled by Martingale process, specifically, Brownian motion.

Hence, we propose a generalized multiple-environmental-factors software reliability model, given as follows

$$\frac{d}{dt} m\left(t, \underbrace{\eta_1, \eta_2, \dots, \eta_n}\right) = \left[h(t) + \sum_{i=1}^{n} \lambda_i G_i\left(t, \underbrace{\eta_1, \eta_2, \dots, \eta_n}\right) + \dot{B}(t)\right]\left[N(t) - m\left(t, \underbrace{\eta_1, \eta_2, \dots, \eta_n}\right)\right] \quad (7.26)$$

$$m(0) = 0 \quad (7.27)$$

where $\underbrace{\eta_1, \eta_2, \dots, \eta_n}$ represents *n*-dimensional vector. $m\left(t, \underbrace{\eta_1, \eta_2, \dots, \eta_n}\right)$ represents the expected number of software failures by time *t* considering multiple environmental factors. $h(t)$ is the software fault detection rate per unit of time without the impact of environmental factors. η_i is random variable represented environmental factor 1, 2, 3, …,

n. $G_i\left(t, \underbrace{\eta_1, \eta_2, \dots, \eta_n}\right)$ is the time-dependent function, represents the impact of environmental factor 1, 2, 3, …, n on software fault detection rate per unit of time. λ_i is the coefficient associated with $G_1(t, \eta_1)$, $G_2(t, \eta_2)$, $G_3(t, \eta_n), \dots, G_n(t, \eta_n)$. $N(t)$ is the fault content function by time *t*. $\dot{B}(t)$ is a standard white noise, is given as follows

$$\frac{d}{dt}B(t) = \dot{B}(t) \tag{7.28}$$

where $B(t)$ denotes Brownian motion.

Given $\dot{B}(t)$ is the white noise process, in other words, $\dot{B}(t)$ is a Gaussian process with the covariance structure given as follows

$$E\left(\dot{B}(t)\dot{B}(u)\right) = \delta(u - t), 0 < t < u \tag{7.29}$$

where δ is the Dirac Delta measure. Equation (7.29) will have the impact on $h(t) + \sum_{i=1}^{n} \lambda_i G_i\left(t, \underbrace{\eta_1, \eta_2, \dots, \eta_n}\right) + \dot{B}(t)$, specifically, we obtain

$$\int_0^t [h(s) + \sum_{i=1}^{n} \lambda_i G_i\left(s, \underbrace{\eta_1, \eta_2, \dots, \eta_n}\right) + \dot{B}(s)]ds$$

$$= \int_0^t \left[h(s) + \sum_{i=1}^{n} \lambda_i G_i\left(s, \underbrace{\eta_1, \eta_2, \dots, \eta_n}\right)\right] ds + B(t) - B(0)$$

$$= \int_0^t \left[h(s) + \sum_{i=1}^{n} \lambda_i \, G_i \left(s, \underbrace{\eta_1, \eta_2, \dots, \eta_n} \right) \right] ds + B(t) \tag{7.30}$$

Given the general solution discussed in last section, the general solution for equation (7.26) is provided as follows

$$m\left(t, \underbrace{\eta_1, \eta_2, \dots, \eta_n}\right) = N(t) - N(0) e^{-\int_0^t \left(h(s) + \sum_{i=1}^{n} \lambda_i G_i \left(s, \underbrace{\eta_1, \eta_2, \dots, \eta_n} \right) + \dot{B}(s) \right) ds} - \int_0^t e^{-\int_u^t \left(h(s) + \sum_{i=1}^{n} \lambda_i G_i \left(s, \underbrace{\eta_1, \eta_2, \dots, \eta_n} \right) + \dot{B}(s) \right) ds} N'(u) du \tag{7.31}$$

Substituting equation (7.30) to equation (7.31), the mean value function is obtained as

$$\overline{m}\left(t, \underbrace{\eta_1, \eta_2, \dots, \eta_n}\right) = N(t) - N(0) e^{-\int_0^t h(s) ds} e^{\frac{t}{2}} e^{-\int_0^t \sum_{i=1}^{n} \lambda_i G_i \left(s, \underbrace{\eta_1, \eta_2, \dots, \eta_n} \right) ds} - \int_0^t e^{-\int_u^t h(s) ds} e^{\frac{t-u}{2}} e^{-\int_u^t \sum_{i=1}^{n} \lambda_i G_i \left(s, \underbrace{\eta_1, \eta_2, \dots, \eta_n} \right) ds} N'(u) du \tag{7.32}$$

Let

$$G_i \left(t, \underbrace{\eta_1, \eta_2, \dots, \eta_n} \right) = \eta_i v_i(t) \tag{7.33}$$

where $v_i(t)$ is a time-dependent function and measures the impact of time on environmental factor $\eta_1, \eta_2, \eta_3, \dots, \eta_n$.

As discussed in the model assumption, we consider all environmental factors are independent, and environmental factor 1, 2, 3, …, and n are represented by η_i, where $i = 1, 2, \dots, n$. η_i is considered as random variable in this study, which could follow different distribution function. In order to solve equation (7.32), we apply the expectation on both sides of equation (7.32) with respect to $\eta_1, \eta_2, \eta_3, \dots, \eta_n$. Thus, the mean value function is expressed as

$$\begin{aligned}\overline{m}_{\underbrace{\eta_1,\eta_2,\dots,\eta_n}}(t) &= N(t) - N(0)e^{-\int_0^t h(s)ds}e^{\frac{t}{2}}\left[\prod_{i=1}^{n}\int_0^{\infty} e^{-\int_u^t \lambda_i\eta_i v_i(s)ds} f(\eta_i)d\eta_i\right] \\ &\quad - \int_0^t e^{-\int_u^t h(s)ds}e^{\frac{t-u}{2}}\left[\prod_{i=1}^{n}\int_0^{\infty} e^{-\int_u^t \lambda_i\eta_i v_i(s)ds} f(\eta_i)d\eta_i\right]N'(u)du \\ &= N(t) - N(0)e^{-\int_0^t h(s)ds}e^{\frac{t}{2}}\left[\prod_{i=1}^{n}\int_0^{\infty} e^{-\int_u^t \lambda_i\eta_i v_i(s)ds} f(\eta_i)d\eta_i\right] \\ &\quad - \int_0^t N'(u)e^{-\int_u^t\left(h(s)-\frac{1}{2}\right)ds}\left[\prod_{i=1}^{n}\int_0^{\infty} e^{-\int_u^t \lambda_i\eta_i v_i(s)ds} f(\eta_i)d\eta_i\right]du \qquad (7.34)\end{aligned}$$

Equation (7.34) is the generalized mean value function in consideration of multiple environmental factors, and the randomness caused by these environmental factors, modeled as Brownian motion. Because each environmental factor is modeled as a random variable, which follows certain distribution, there is another advantage regarding the formulation of equation (7.34) stated as follows. If we know the distribution of each factor, we use the Laplace Transform of the probability density function and will have high possibility to obtain a close form solution of the mean value function.

7.3.2 Specific Multiple-Environmental-Factors Software Reliability Models

Two Specific Environmental Factors

PoRM

The same definition and distribution given in Section 7.2.1 will be adopted for the development of specific multiple-environmental-factors software reliability models.

FoPSC

In 1980, Lehman [3] summarized the Laws of Program Evolution. The first law, continuing change, expressed the universally observed fact that large programs are never completed. They just continue to evolve. Changes of specifications occur since the initial development until product delivery, which posing a considerate amount of risk to software cost but brining new opportunity to add value and improve reliability [192]. Changes of specifications, in the early 1990s, studied by Harker et al. [193] mostly due to the source described as follows. (1) Fluctuations within organization or the market. (2) Increased understanding of requirements. (3) Consequence of system-usage. (4) Customer migratory issues. (5) Adaption issues. Later, many literatures proposed other explanations for the changes of specifications mainly from the perspectives of product strategy, hardware/software environment/interaction, testability and functionality enhancement [194 - 196].

Meanwhile, FoPSC is one of the significant environmental factors in the recent survey study investigating the impact of environmental factors affecting software reliability during development process, stated in Chapter 4. We define FoPSC as the total times of all the specifications have been changed in all the historical versions in software development. But in this study, we will use the percentage of all the changes in a project to estimate the parameters. We employ the dataset provided in references [197, 201] to estimate the distribution of FoPSC, as shown in Figure 7.2.

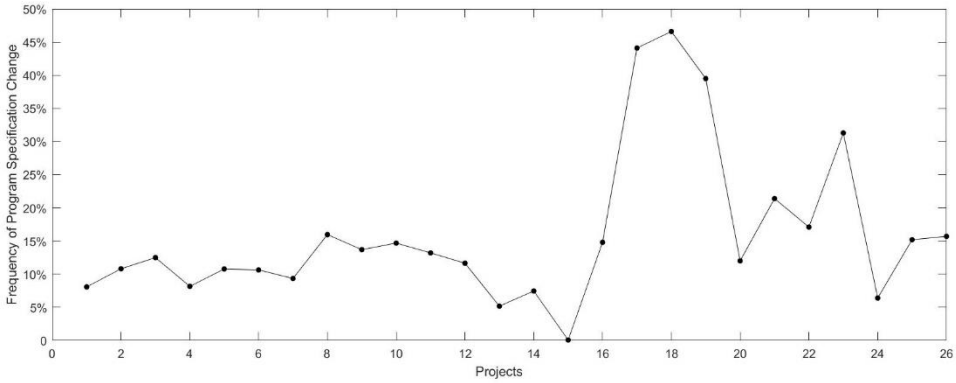

Figure 7. 2 Data collection of FoPSC

Gamma distribution or beta distribution is good choice for FoPSC, given the model characteristics. Thus, we compare the Log-likelihood value for gamma distribution and beta distribution, and it demonstrates beta distribution is better fit for FoPSC. The parameter estimate of beta distribution is also obtained, which is written as FoPSC $\sim Beta\ (1.411, 7.409)$. In Section 7.3.2.2, we provide the specific models with FoPSC follows gamma distribution and beta distribution, but later in the numerical

examples illustration, we only use beta distribution to estimate parameters and compare model predictive power.

Model Framework

In this section, we use two environmental factors, PoRM and FoPSC, as discussed in the previous section, to substitute in the generalized mean value function considering multiple environmental factors.

The mean value function with two environmental factors is obtained from equation (7.34) as follows

$$\bar{m}_{\underbrace{\eta_1,\eta_2,\ldots,\eta_n}}(t) = N(t) - N(0)e^{-\int_0^t h(s)ds}e^{\frac{t}{2}}\left(\prod_{i=1}^{2}\int_0^{\infty} e^{-\int_0^t \lambda_i\eta_i v_i(s)ds} f(\eta_i)d\eta_i\right) - \int_0^t N'(u)e^{-\int_u^t\left(h(s)-\frac{1}{2}\right)ds}\left(\prod_{i=1}^{2}\int_0^{\infty} e^{-\int_0^t \lambda_i\eta_i v_i(s)ds} f(\eta_i)d\eta_i\right)du \tag{7.35}$$

where η_1 denotes PoRM and η_2 denotes FoPSC. λ_1 and λ_2 is the coefficient associated with the function $G_1(t,\eta_1)$ and $G_2(t,\eta_2)$, respectively. $v_1(t)$ and $v_2(t)$ are time-dependent functions and measure the impact of time on environmental factor $\eta_{1,}\,\eta_2$, respectively.

Note that PoRM follows gamma distribution as discussed in the previous section, which is $\eta_1 \sim \text{Gamma}(\gamma_1, \theta_1)$ The probability density function for PoRM is given as follows

$$f(\eta_1) = \frac{{\theta_1}^{\gamma_1} {\eta_1}^{\gamma_1 - 1} e^{-\theta_1 \eta_1}}{\Gamma(\gamma_1)} \tag{7.36}$$

where γ_1 and θ_1 are the parameters of the gamma distribution, which represents PoRM.

Next, we present the specific software reliability models when FoPSC follows gamma distribution and beta distribution. Software researchers and practitioner will choose the one fits best for the forthcoming scenarios.

First, let FoPSC follows gamma distribution, which is $\eta_2 \sim \text{Gamma}(\gamma_2, \theta_2)$, then

$$f(\eta_2) = \frac{{\theta_2}^{\gamma_2} {\eta_2}^{\gamma_2 - 1} e^{-\theta_2 \eta_2}}{\Gamma(\gamma_2)} \tag{7.37}$$

where γ_2 and θ_2 are the parameters of the gamma distribution, which represents FoPSC.

The mean value function with gamma-distributed PoRM and gamma-distributed FoPSC is written as

$$\bar{m}_{\underbrace{\eta_1,\eta_2,\ldots,\eta_n}}(t)$$

$$= N(t) - N(0)e^{-\int_0^t h(s)ds}e^{\frac{t}{2}}\left[\frac{\theta_1}{\theta_1 + \int_0^t \lambda_1 v_1(s)ds}\right]^{\gamma_1}\left[\frac{\theta_2}{\theta_2 + \int_0^t \lambda_2 v_2(s)ds}\right]^{\gamma_2}$$

$$-\left[\frac{\theta_1}{\theta_1 + \int_0^t \lambda_1 v_1(s)ds}\right]^{\gamma_1}\left[\frac{\theta_2}{\theta_2 + \int_0^t \lambda_2 v_2(s)ds}\right]^{\gamma_2}\int_0^t N'(u)e^{-\int_u^t\left(h(s)-\frac{1}{2}\right)ds}\,du \qquad (7.38)$$

Let $h(t)$ and $N(t)$ be equations (7.22) and (7.24), respectively, $v_1(t) = e^{-a_1 t}$, and $v_2(t) = e^{-a_2 t}$, the mean value function is written as

$$\bar{m}_{\underbrace{\eta_1,\eta_2,\ldots,\eta_n}}(t) = \frac{1}{k}e^{kt}$$

$$-\frac{e^{\frac{t}{2}}}{c + e^{bt}}\left[\frac{\theta_1}{\theta_1 + \frac{\lambda_1}{a_1}(1 - e^{-a_1 t})}\right]^{\gamma_1}\left[\frac{\theta_2}{\theta_2 + \frac{\lambda_2}{a_2}(1 - e^{-a_2 t})}\right]^{\gamma_2}\left[\frac{c+1}{k}\right.$$

$$\left.-\frac{c}{k-\frac{1}{2}}e^{\left(k-\frac{1}{2}\right)t} - \frac{1}{b+k-\frac{1}{2}}e^{\left(b+k-\frac{1}{2}\right)t} + \frac{c}{k-\frac{1}{2}} + \frac{1}{b+k-\frac{1}{2}}\right] \qquad (7.39)$$

Secondly, let FoPSC follows beta distribution, which is $\eta_2 \sim \text{Beta}(\beta_1, \beta_2)$, then

$$f(\eta_2) = \frac{\Gamma(\beta_1 + \beta_2){\eta_2}^{\beta_1 - 1}(1 - \eta_2)^{\beta_2 - 1}}{\Gamma(\beta_1)\Gamma(\beta_2)} \qquad (7.40)$$

where β_1 and β_2 are the parameters of beta distribution.

The Laplace transform of η_2 is given as follows [103]

$$F_{\eta_2}^*(s) = e^{-s} \times HG([\beta_2], [\beta_1 + \beta_2], s) \tag{7.41}$$

where $\mathrm{HG}([\beta_2], [\beta_1 + \beta_2], s)$ is the generic hypergeometric function such that

$$HG([a_1, a_2, \dots, a_m], [b_1, b_2, \dots, b_n], s) = \sum_{k=0}^{\infty} \left[\frac{s^k \prod_{i=1}^{m} \frac{\Gamma(a_i + k)}{\Gamma(a_i)}}{\prod_{i=1}^{n} \frac{\Gamma(b_i + k)}{\Gamma(b_i)} k!} \right].$$

Therefore

$$F_{\eta_2}^*(s) = e^{-s} \left[\sum_{j=0}^{\infty} \frac{\Gamma(\beta_1 + \beta_2)\Gamma(\beta_2 + j)}{\Gamma(\beta_2)\Gamma(\beta_1 + \beta_2 + j)} \right] = \sum_{j=0}^{\infty} \frac{\Gamma(\beta_1 + \beta_2)\Gamma(\beta_2 + j)}{\Gamma(\beta_2)\Gamma(\beta_1 + \beta_2 + j)} \times \frac{s^j e^{-s}}{j!}$$

$$= \sum_{j=0}^{\infty} \frac{\Gamma(\beta_1 + \beta_2)\Gamma(\beta_2 + j) \times Poisson(j, s)}{\Gamma(\beta_2)\Gamma(\beta_1 + \beta_2 + j)} \times \mathrm{Poisson}(j, s) \tag{7.42}$$

where $Poisson(j, s) = \frac{s^j e^{-s}}{j!}$.

The mean value function with gamma-distributed PoRM and beta-distributed FoPSC is written as

$$\bar{m}_{\underbrace{\eta_1,\eta_2,\ldots,\eta_n}}(t) = N(t) -$$

$$N(0)e^{-\int_0^t h(s)ds}e^{\frac{t}{2}}\left[\frac{\theta_1}{\theta_1+\int_0^t \lambda_1 v_1(s)ds}\right]^{\gamma_1}\left[\sum_{j=0}^{\infty}\frac{\Gamma(\beta_1+\beta_2)\Gamma(\beta_2+j)\times Poisson\left(j,\int_0^t \lambda_2 v_2(s)ds\right)}{\Gamma(\beta_2)\Gamma(\beta_1+\beta_2+j)}\right] -$$

$$\left[\frac{\theta_1}{\theta_1+\int_0^t \lambda_1 v_1(s)ds}\right]^{\gamma_1}\left[\sum_{j=0}^{\infty}\frac{\Gamma(\beta_1+\beta_2)\Gamma(\beta_2+j)\times Poisson\left(j,\int_0^t \lambda_2 v_2(s)ds\right)}{\Gamma(\beta_2)\Gamma(\beta_1+\beta_2+j)}\right]\int_0^t N'(u)e^{-\int_u^t\left(h(s)-\frac{1}{2}\right)ds}\,du \tag{7.43}$$

where $Poisson\left(j,\int_0^t \lambda_2 v_2(s)ds\right) = \frac{\left(\int_0^t \lambda_2 v_2(s)ds\right)^j e^{-\int_0^t \lambda_2 v_2(s)ds}}{j!}$.

Let $h(t)$ and $N(t)$ be equations (7.22) and (7.24), respectively, $v_1(t) = e^{-a_1 t}$, and $v_2(t) = e^{-a_2 t}$, the mean value function is written as

$$\bar{m}_{\underbrace{\eta_1,\eta_2,\ldots,\eta_n}}(t)$$

$$= \frac{1}{k}e^{kt}$$

$$-\frac{e^{\frac{t}{2}}}{c+e^{bt}}\left[\frac{\theta_1}{\theta_1+\frac{\lambda_1}{a_1}(1-e^{-a_1 t})}\right]^{\gamma_1}\left[\sum_{j=0}^{\infty}\frac{\Gamma(\beta_1+\beta_2)\Gamma(\beta_2+j)\times Poisson\left(j,\frac{\lambda_2}{a_2}(1-e^{-a_2 t})\right)}{\Gamma(\beta_2)\Gamma(\beta_1+\beta_2+j)}\right]\left[\frac{c+1}{k}\right.$$

$$\left.-\frac{c}{k-\frac{1}{2}}e^{\left(k-\frac{1}{2}\right)t}-\frac{1}{b+k-\frac{1}{2}}e^{\left(b+k-\frac{1}{2}\right)t}+\frac{c}{k-\frac{1}{2}}+\frac{1}{b+k-\frac{1}{2}}\right] \tag{7.44}$$

where $Poisson\left(j,\frac{\lambda_2}{a_2}(1-e^{-a_2 t})\right) = \frac{\left[\frac{\lambda_2}{a_2}(1-e^{-a_2 t})\right]^j e^{-\frac{\lambda_2}{a_2}(1-e^{-a_2 t})}}{j!}$.

7.4 Numerical Examples for Single-Environmental-Factor Software Reliability Model

In the following experiments, we choose two applications, DS1 and DS2, to validate the proposed single-environmental-factor software reliability model and compare the performance with other existing software reliability models. The comparison criteria are discussed in Chapter 5. DS1 and DS2 are both collected from Open Source Software (OSS) project. OSS has attracted significant attention in the past decade. Some report shows that a few major OSS products have surpassed their commercial counterparts in terms of the market share and quality evaluation [198]. Not only individuals are attracted by the features of OSS, but many software companies and government-supported organizations [199]. A research study conducted by *CIO Magazine* [200] found that IT community is growing better by using open source development model and OSS will dominate as the Web server application platform and server operating system. The majority of the companies are using open source today for web development. To align with the new transitions in software development, OSS project data are employed to illustrate the performance of the proposed model.

Numerical Example 1

Dataset 1 (DS1) is extracted from Apache OSS project. It was collected from Sep 2010 to April 2013. The collected data are described in Table 7.3. The column named *Failures* in Table 7.3 represents the number of software failures detected between time unit *t-1* and *t*. The column named *Cumulative failures* in Table 7.3 represents the cumulative software failure by time *t*. We compare the models discussed in Table 7.4. The parameter estimates

and model comparisons are presented in Table 7.5. In this dataset, we use the first 24 time units to estimate the parameters.

As presented in Table 7.5, the proposed single-environmental-factor model has the smallest MSE, PRR, and Variation. It is worth noting that MSE is the most critical criteria in terms of model selection. The PP value for the proposed model is 0.311, which is just slightly larger than the smallest PP value 0.228, however, all other three criteria for G-O model is significantly larger than the proposed model. Thus, we conclude that the proposed model has the best fit. Figure 7.3 and 7.4 also illustrate the comparison between the actual failure data and the predicted failure data from all the models discussed in Table 7.5. The proposed model yields very close fittings and predictions of software failures.

Table 7. 3 DS1 failure data

Time unit	**Failures**	**Cumulative failures**	**Time unit**	**Failures**	**Cumulative failures**	**Time unit**	**Failures**	**Cumulative failures**
1	6	6	12	4	66	23	6	102
2	6	12	13	0	66	24	22	124
3	6	18	14	4	70	25	3	127
4	8	26	15	5	75	26	1	128
5	13	39	16	5	80	27	1	129
6	6	45	17	2	82	28	0	129
7	8	53	18	10	92	29	0	129
8	2	55	19	1	93	30	0	129
9	3	58	20	1	94	31	4	133
10	3	61	21	2	96	32	3	136
11	1	62	22	0	96	-	-	-

Table 7. 4 Model comparisons

Model	**Mean value function**
G-O model	$m(t) = a(1 - e^{-bt})$
Inflection S-shaped model	$m(t) = \frac{a(1 - e^{-bt})}{1 + \beta e^{-bt}}$
Delayed S-shaped model	$m(t) = a(1 - (1 + bt)e^{-bt})$
Yamada imperfect debugging model	$m(t) = a[1 - e^{-bt}]\left[1 - \frac{\alpha}{b}\right] + \alpha at$
PNZ model	$m(t) = \frac{a[(1 - e^{-bt)}\left(1 - \frac{\alpha}{b}\right) + \alpha t]}{1 + \beta e^{-bt}}$
Pham-Zhang IFD	$m(t) = a - ae^{-bt}(1 + (b + d)t + bdt^2)$
Proposed single-environmental-factor model	$m(t) = \frac{1}{k}e^{kt} - \frac{e^{\frac{t}{2}}}{c + e^{bt}}\left[\frac{\theta}{\theta + \frac{\lambda_0}{a}(1 - e^{-at})}\right]^{\gamma}\left(\frac{c+1}{k} - \frac{c}{k - \frac{1}{2}}e^{\left(k - \frac{1}{2}\right)t} - \frac{1}{b + k - \frac{1}{2}}e^{\left(b + k - \frac{1}{2}\right)t} + \frac{c}{k - \frac{1}{2}} + \frac{1}{b + k - \frac{1}{2}}\right)$

Table 7. 5 DS1 parameter estimates and model comparison

Model	**MSE**	**PRR**	**PP**	**Variation**	**Parameter Estimates**
G-O model	36.561	0.315	0.228	6.076	$\hat{a} = 201.250$ $\hat{b} = 0.033$
Inflection S-shaped model	68.326	0.789	0.483	7.992	$\hat{a} = 150.030$ $\hat{b} = 0.096$ $\hat{\beta} = 1.830$
Delayed S-shaped model	110.047	20.902	2.123	10.544	$\hat{a} = 131.400$ $\hat{b} = 0.144$
Yamada imperfect debugging model	60.193	0.401	0.300	74.941	$\hat{a} = 185.180$ $\hat{b} = 0.033$ $\hat{\alpha} = 0.010$

PNZ model	54.515	0.523	0.333	6.795	$\hat{a} = 161.010$ $\hat{b} = 0.069$ $\hat{\alpha} = 0.001$ $\hat{\beta} = 0.930$
Pham-Zhang IFD	143.253	40.461	2.665	11.076	$\hat{a} = 143.045$ $\hat{b} = 0.129$ $\hat{d} = 0.001$
Proposed single-environmental-factor model	35.173	0.254	0.311	5.390	$\hat{k} = 0.014$ $\hat{b} = 0.589$ $\hat{c} = 0.039$ $\hat{a} = 75.000$ $\widehat{\lambda_0} = 2.001$

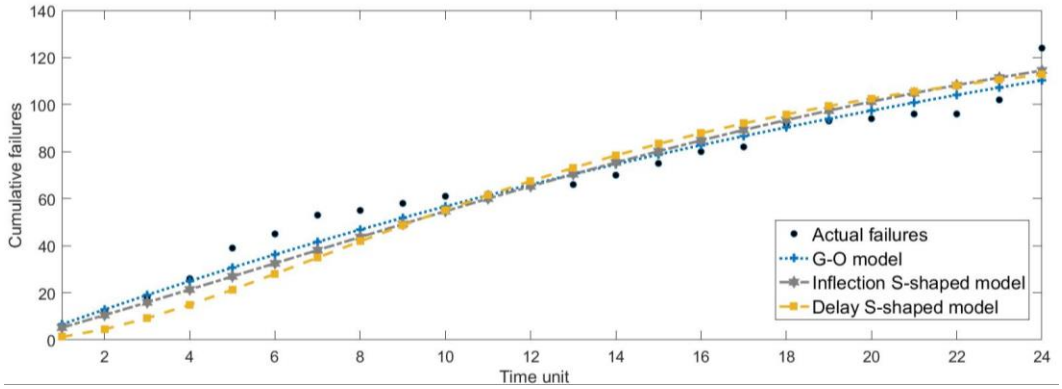

Figure 7. 3 DS1 comparison of actual failure data and predicted failure data – Part I

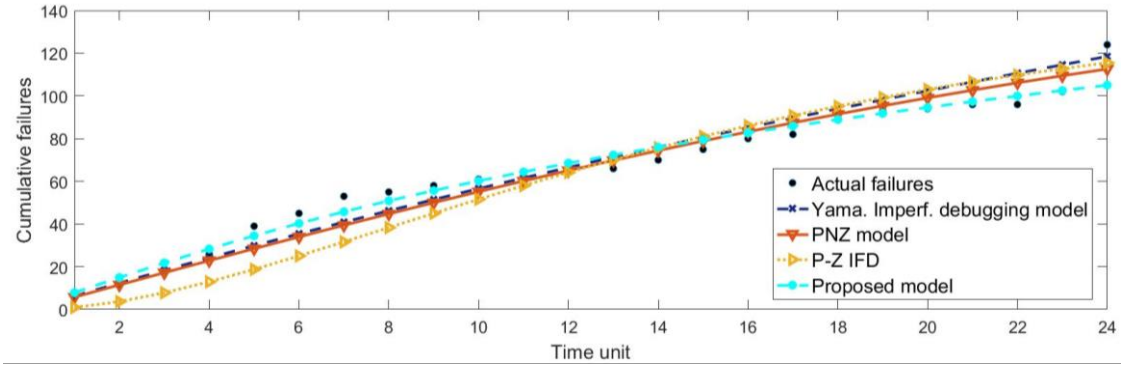

Figure 7. 4 DS1 comparison of actual failure data and predicted failure data – Part II

Other software reliability measures are also calculated to provide a comprehensive understanding of the prediction power for the proposed model. In this study, we are interested in the estimated software failures for each time unit, which is obtained by $m(t) - m(t-1)$, and the estimated time to detect all the 136 actual software failures that already have been appeared in the program, as seen in Table 7.3. Figure 7.5 illustrates the estimated software failures for 16 time units, ranging from time unit 25 to 40, which gives an estimate for software tester in terms of the number of software failures occurred in the operation field during each time unit and further helps software multiple release. Moreover, the estimated time unit to detect all the 136 actual software failures is 38.40 based on the proposed model. Software tester use this information to estimate how much time will be spent on one project and assign the corresponding testing resource. In sum, those measures are helpful for software testing team to optimally assign the available testing resource to each ongoing project, decide when to stop testing, and plan software multiple release.

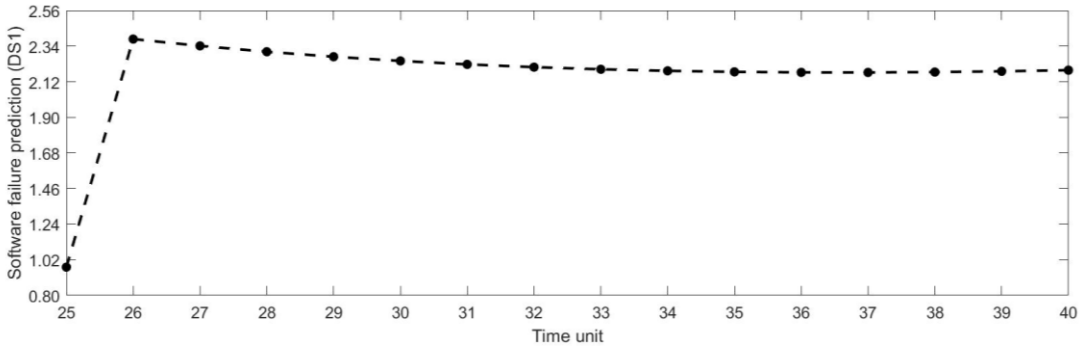

Figure 7. 5 DS1 software failure prediction from time unit 25 to 40

Numerical Example 2

DS2 is also one of Apache OSS project data. It was collected from Feb 2009 to Feb 2014. 33 sets of failure data are presented in Table 7.6. The column named *Failures* in Table 7.6 represents the number of software failures detected between time unit *t-1* and *t*. The column named *Cumulative failures* in Table 7.6 represents the cumulative software failure by time *t*. The parameter estimates and model comparisons are described in Table 7.7.

Table 7. 6 DS2 failure data

Time unit	Failures	Cumulative failures	Time unit	Failures	Cumulative failures	Time unit	Failures	Cumulative failures
1	7	7	12	22	88	23	11	140
2	2	9	13	11	99	24	4	144
3	8	17	14	8	107	25	0	144
4	9	26	15	2	109	26	5	149
5	2	28	16	7	116	27	0	149
6	5	33	17	3	119	28	9	158
7	5	38	18	4	123	29	13	171
8	7	45	19	1	124	30	1	172
9	10	55	20	0	124	31	1	173
10	9	64	21	0	124	32	12	185
11	2	66	22	5	129	33	0	185

As seen from Table 7.7, the proposed model has the smallest MSE and Variation. Although PNZ model has the smallest PRR and PP, the MSE for PNZ model is much larger than the proposed model. Given MSE is the most critical for model selection, therefore, the best fitting model is still the proposed model. Figure 7.6 and 7.7 illustrate the comparison between the actual failure data and the predicted failure data from all the models discussed

in Table 7.7. The proposed model subjects to a very close fittings and predictions on the cumulative software failures of DS2.

Table 7. 7 DS2 parameter estimates and model comparison

Model	**MSE**	**PRR**	**PP**	**Variation**	**Parameter Estimates**
G-O model	136.477	1.285	3.462	11.892	$\hat{a} = 181.250$ $\hat{b} = 0.056$
Inflection S-shaped model	176.235	5.292	1.513	12.794	$\hat{a} = 179.230$ $\hat{b} = 0.193$ $\hat{\beta} = 13.159$
Delayed S-shaped model	67.922	27.778	1.536	8.221	$\hat{a} = 200.090$ $\hat{b} = 0.112$
Yamada imperfect debugging model	69.510	0.625	1.180	103.480	$\hat{a} = 230.250$ $\hat{b} = 0.034$ $\hat{\alpha} = 0.008$
PNZ model	90.902	0.372	0.418	8.809	$\hat{a} = 300.130$ $\hat{b} = 0.048$ $\hat{\alpha} = 0.001$ $\hat{\beta} = 1.321$
Pham-Zhang IFD	74.124	528.248	2.576	8.240	$\hat{a} = 189.960$ $\hat{b} = 0.134$ $\hat{d} = 0.010$
Proposed single-environmental-factor model	63.989	4.895	1.224	7.576	$\hat{k} = 0.009$ $\hat{b} = 0.626$ $\hat{c} = 1.078$ $\hat{a} = 51.725$ $\widehat{\lambda_0} = 25.346$

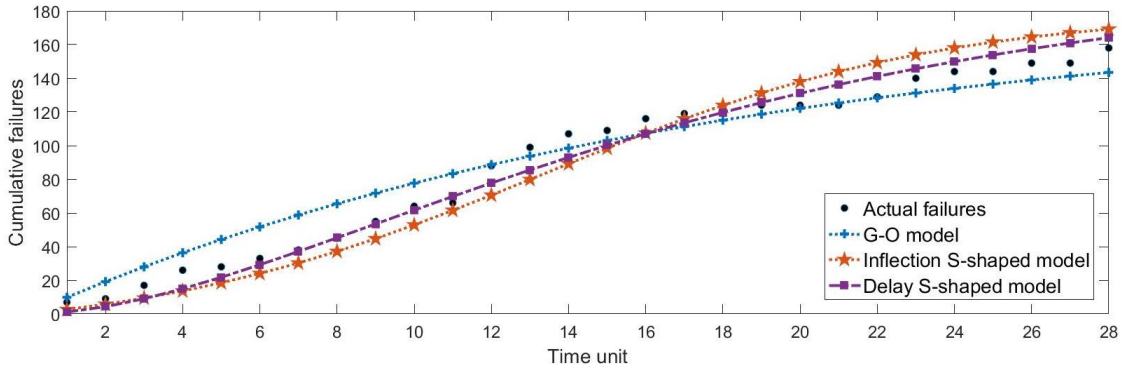

Figure 7. 6 DS2 comparison of actual failure data and predicted failure data – Part I

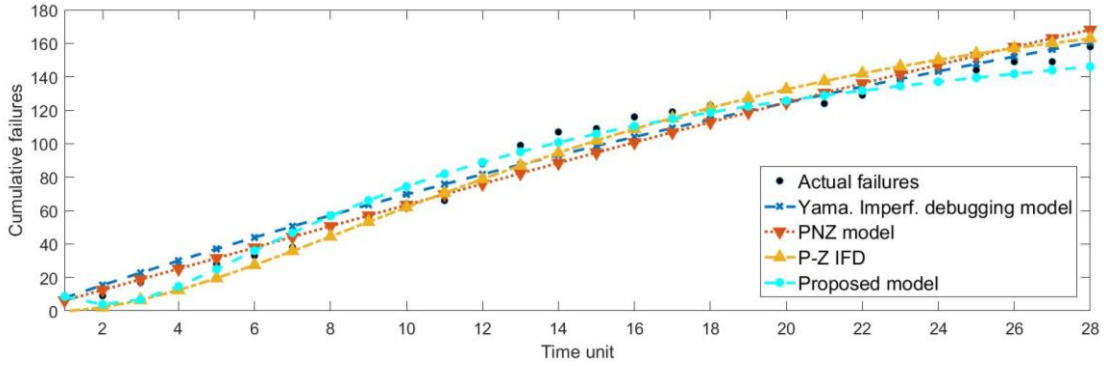

Figure 7. 7 DS2 comparison of actual failure data and predicted failure data – Part II

We also provide other software reliability measures for DS2. Figure 7.8 presents the estimated failures for each time unit, ranging from time unit 29 to 44. Moreover, the estimated time unit to detect all 185 software failures that have been already appeared in the program is 50.92.

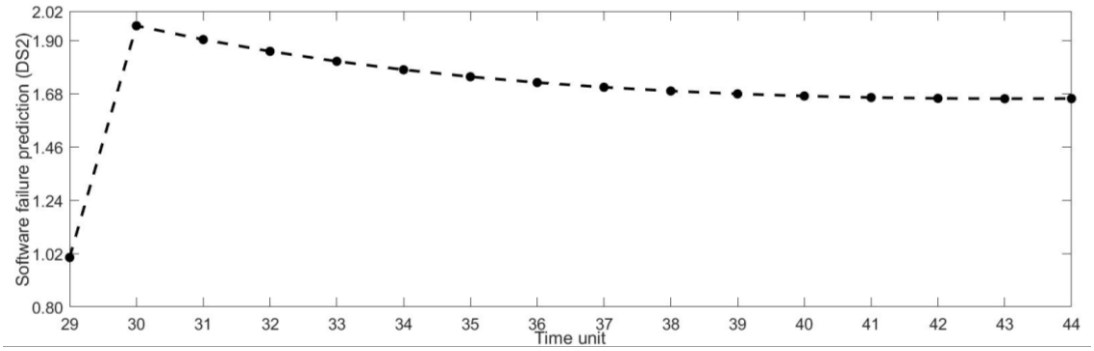

Figure 7. 8 DS2 software failure prediction for time unit 29 to 44

Reliability Prediction

Once the parameters are obtained, the software reliability within $(t, t+x)$ is determined as

$$R(x|t) = e^{-[m(t+x)-m(t)]} \tag{7.45}$$

Figure 7.9, and 7.10 presents the reliability prediction for DS1 and DS2 by varying x from time unit 0 to 1.2, respectively. All other models did not take into consideration of environmental factor (PoRM) and a dynamic fault detection process. As a result, they cannot present reliability prediction well since OSS projects are influenced by the randomness caused by environmental factors significantly than traditional projects. Thus, we did not incorporate reliability prediction from other models.

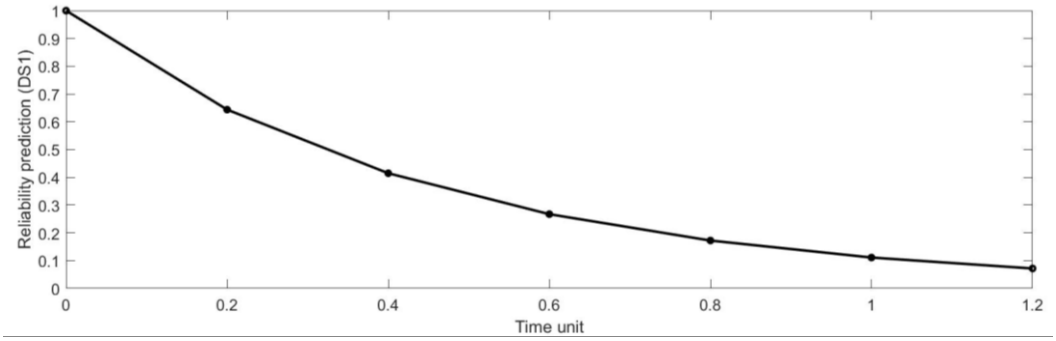

Figure 7. 9 DS 1 reliablity predicton

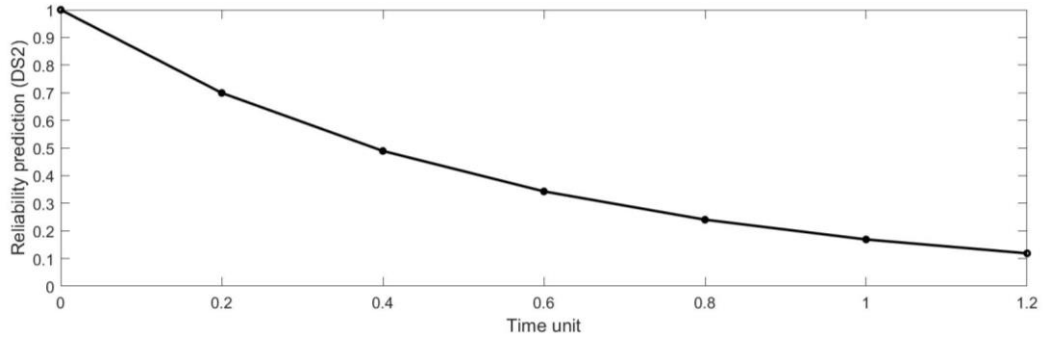

Figure 7. 10 DS2 reliablity predicton

7.5 Numerical Examples for Multiple-Environmental-Factors Software Reliability Model

We use the same data set as in Section 7.4.2 to compare the prediction power of the proposed specific multiple-environmental-factors software reliability model, specifically, with gamma-distributed PoRM and beta-distributed FoPSC. The model comparison and parameter estimate are presented in Table 7.8. The proposed multiple-environmental-

factors model, with gamma-distributed PoRM and beta-distributed FoPSC, performs better accuracy of prediction failures, compared with the single-environmental-factor model.

Table 7. 8 Parameter estimates and model comparison

Model	**MSE**	**PRR**	**PP**	**Variation**	**Parameter Estimates**
G-O model	136.477	1.285	3.462	11.892	$\hat{a} = 181.250$ $\hat{b} = 0.056$
Inflection S-shaped model	176.235	5.292	1.513	12.794	$\hat{a} = 179.230$ $\hat{b} = 0.193$ $\hat{\beta} = 13.159$
Delayed S-shaped model	67.922	27.778	1.536	8.221	$\hat{a} = 200.090$ $\hat{b} = 0.112$
Yamada imperfect debugging model	69.510	0.625	1.180	103.480	$\hat{a} = 230.250$ $\hat{b} = 0.034$ $\hat{\alpha} = 0.008$
PNZ model	90.902	0.372	0.418	8.809	$\hat{a} = 300.130$ $\hat{b} = 0.048$ $\hat{\alpha} = 0.001$ $\hat{\beta} = 1.321$
Pham-Zhang IFD	74.124	528.248	2.576	8.240	$\hat{a} = 189.960$ $\hat{b} = 0.134$ $\hat{d} = 0.010$
Proposed single-environmental-factor model	63.989	4.895	1.224	7.576	$\hat{k} = 0.009$ $\hat{b} = 0.626$ $\hat{c} = 1.078$ $\hat{a} = 51.725$ $\widehat{\lambda_0} = 25.346$
Proposed multiple-environmental-factor model (gamma-distributed PoRM and beta-distributed FoPSC)	44.780	0.990	0.327	5.421	$\hat{k} = 0.008$ $\hat{b} = 0.597$ $\hat{c} = 0.900$ $\widehat{a_1} = 100.000$ $\widehat{\lambda_1} = 40.148$ $\widehat{a_2} = 113.644$ $\widehat{\lambda_2} = 29.289$

7.6 Discussion of Impact of Environmental Factor

To emphasize the significance of incorporating environmental factor(s) in software reliability models in this chapter, the comparison between the software reliability model with and without environmental factor(s) will be discussed in this section. Without considering environmental factors(s), equation (7.26) will be formulated as

$$\frac{d}{dt}m(t,\eta) = \left[h(t) + \dot{B}(t)\right][N(t) - m(t,\eta)] \tag{7.46}$$

This formulation has detailed explanation in Pham and Pham [188]. Substituting equations (7.22) - (7.24) into equation (7.45), the new mean value function without considering environmental factor(s) is obtained as follows

$$m(t) = \frac{1}{k}e^{kt} - \frac{e^{\frac{t}{2}}}{c + e^{bt}}\left(\frac{c+1}{k} - \frac{c}{k-\frac{1}{2}} - \frac{1}{b+k-\frac{1}{2}}\right) - \frac{ce^{kt}(c+e^{bt})^{-1}}{k-\frac{1}{2}} - \frac{e^{(k+b)t}(c+e^{bt})^{-1}}{b+k-\frac{1}{2}} \tag{7.47}$$

As an illustration for the comparison, DS1 will be used to compare the mean value function with environmental factor (equation (7.25)) and the mean value function without environmental factor (equation (7.47)). The parameter estimates for the mean value function without environmental factor are $\hat{k} = 0.109, \hat{c} = 0.045, \hat{b} = 0.501$. Using the criteria described previously, we have $MSE = 103.238$, which is significantly higher than

the model considering environmental factor, 35.173, as shown in Table 7.5. We also present Figure 7.11, which compares the actual failure data, the failure prediction by the model with environmental factor, and the failure prediction by the model without environmental factor. We notice that the failure prediction will be more accurate with considering environmental factor for OSS project data. Therefore, incorporating environmental factor in software reliability model will significantly improve the predictive accuracy.

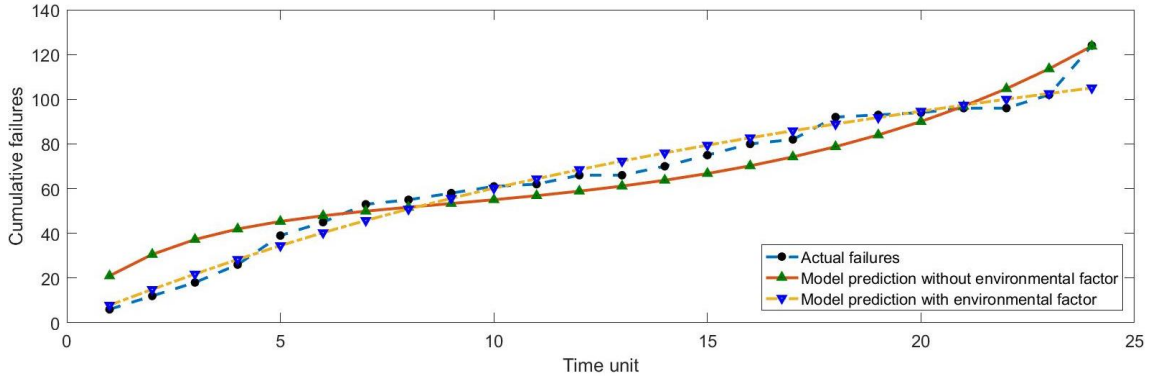

Figure 7. 11 Comparison of failure prediction

7.7 Conclusions

To the best of our knowledge, environmental factor(s), and martingale framework, specifically, Brownian motion and white noise process have not been simultaneously employed in NHPP software reliability model. A considerable amount of software development has shifted their attention from building a new system toward composing system from the existing open source platform. On the other hand, there is an increasing trend on the adoption of software ecosystem. The development of new functionality can be

occurred outside of the platform. For example, App-store styled approaches are getting popular in software community. Therefore, it is of great importance to incorporate environmental factor(s) in the software reliability model because it brings a significant impact on open source project compared with traditional software development.

There are several research directions can be pursued in the next step. For instance, (1) the total number of fault content is considered as a time-dependent function in this study. The randomness caused by environmental factors may also affect this time-dependent function; (2) environmental factors are correlated; (3) the function (equations (21) and (34)) behavior can be further investigated.

Appendix I

Given a generalized mean value function

$$\frac{d}{dt}m(t,\eta) = h(t,\eta)[N(t) - m(t,\eta)]$$

$$m(0) = 0$$

The general solution for the above function is

$$m(t,\eta) = e^{-\int_0^t h(s,\eta)dt}\int_0^t e^{\int_0^u h(s,\eta)ds}h(u)N(u)du = \int_0^t h(u)N(u)e^{-\int_u^t h(s,\eta)ds}du$$

Since

$$\frac{d}{du}\left(e^{-\int_u^t h(s,\eta)ds}\right) = e^{-\int_u^t h(s,\eta)ds} * h(u,\eta)$$

Thus, we have

$$m(t,\eta) = \int_0^t N(u)d[e^{-\int_u^t h(s,\eta)ds}]$$

$$= \left[N(t)e^{-\int_u^t h(s,\eta)ds}\right]_{u=t} - \left[N(t)e^{-\int_u^t h(s,\eta)ds}\right]_{u=0} - \int_0^t e^{-\int_u^t h(s,\eta)ds}d[N(u)]$$

$$= N(t) - N(0)e^{-\int_0^t h(s,\eta)ds} - \int_0^t e^{-\int_u^t h(s,\eta)ds}N'(u)du$$

CHAPTER 8

CONCLUSIONS AND FUTURE RESEARCH

8.1 Conclusions

At the early stage of this dissertation, in Chapter 4, firstly, a comparison analysis for environmental factors affecting software reliability during single-release software development is carried out. The data collection is conducted by survey from twenty organizations, a diverse group of industries is selected to participate in the survey investigation. Participants were asked to rank the environmental factors in light of their impact on software reliability. The significant environmental factors in software development process/each development phase, principle components, and significant level of development phases are revealed and compared with the previous findings [27, 28]. The correlation between environmental factors, how to reduce the dimension of these correlated environmental factors, and development phase analysis are discussed as well.

Later, another investigation of environmental factors affecting reliability in multi-release software development is studied. We further compare the significant factors and other findings between the development of singe-release and multi-release software.

In Chapter 5, software faults are classified into two groups, Type I (independent) faults and Type II (dependent) faults. Two phases software debugging process are introduced according to different types of faults. We firstly propose a one-phase NHPP software reliability model. We assume there is only Type II faults in this model given the Type I

faults have been removed in the preliminary testing phase. Later, a two-phase NHPP software reliability model is developed in consideration of fault dependency and imperfect fault removal. The descriptive and predictive ability of the proposed models is examined in the numerical examples.

In Chapter 6, we aim to develop a NHPP software reliability model for multi-release software product. The remaining faults from previous release and the newly introduced faults (from newly added features) are both incorporated in the model development. In addition, the detection of the new faults in the development of the next release depends on the remaining faults from previous release and the newly introduced faults for developing the next release.

In Chapter 7, the software reliability models incorporating single/multiple environmental factor(s) under the Martingale framework are proposed. We have not only considered the impact of significant factor(s) on software reliability, revealed in Chapter 4, but also the randomness caused by these factors under the Martingale framework in the model development.

Chapter 4 has been published in *Journal of Systems and Software*, as cited in references [29, 30]. Chapter 5 has been published in *Computer Languages, Systems & Structures* and *Vietnam Journal of Computer Science*, as cited in references [133, 174]. Chapter 6 has been published in *Annals of Operations Research*, as cited in reference [116]. Chapter 7 has been published in *Annals of Operations Research*, as cited in reference [202].

8.2 Future Research

Given the study discussed in Chapter 4, there could exists correlations between environmental factors in software development, and we have not considered this correlation in the single and multiple-environmental-factor(s) models in Chapter 7. The first research problem is presented as follows.

Problem 1: Develop software reliability models considering multiple environmental factors, the randomness caused by these factors, and the correlation between the factors.

All the software reliability models developed in this dissertation still focus on the methodologies applied in the Testing phase and the defects found in the Testing phase. As the agile development and other new development methodologies are applied in industry, how to quantify software quality and reliability is interesting to investigate. The second research problem is described as follows.

Problem 2: Develop policies/models to quantify software quality and reliability in the earlier phases of software development, not waiting until the Testing phase.

Software-embedded systems have been greatly adopted in a wide array such as consumer, automotive, medical, commercial and military applications. High quality and reliability of software-embedded systems have been highlighted as leverages to achieve competitive advantages of producing secure and reliable goods. Hence, advanced reliability models are

necessitated to improve the prediction power of the whole system considering critical factors. In general, software-embedded systems consist of hardware and software systems; accordingly, the system failures are classified into three categories: hardware, software, and hardware-software-interaction failures. Thus, the third research problem is proposed as follows.

Problem 3: Develop system reliability model considering three types of failures, hardware, software, and hardware-software-interaction.

REFERENCES

[1] Febrero, F., Calero, C., & Moraga, M. Á. (2016). Software reliability modeling based on ISO/IEC SQuaRE. *Information and Software Technology*, *70*, 18-29.

[2] Hartz, M. A., Walker, E. L., & Mahar, D. (1997). Introduction to Software Reliability: A State of The Art Review. The Center.

[3] Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, *68*(9), 1060-1076.

[4] Catelani, M., Ciani, L., Scarano, V. L., & Bacioccola, A. (2011). Software automated testing: A solution to maximize the test plan coverage and to increase software reliability and quality in use. *Computer Standards & Interfaces*, *33*(2), 152-158.

[5] Han, S., Dang, Y., Ge, S., Zhang, D., & Xie, T. (2012). Performance debugging in the large via mining millions of stack traces. *Proceedings of the 34th International Conference on Software Engineering*, IEEE, 145-155.

[6] Dang, Y., Ge, S., Huang, R., & Zhang, D. (2011). Code clone detection experience at Microsoft. *Proceedings of the 5th International Workshop on Software Clones*, ACM, 63-64.

[7] Chang, I. H., Pham, H., Lee, S. W., & Song, K. Y. (2014). A testing-coverage software reliability model with the uncertainty of operating environments. *International Journal of Systems Science: Operations & Logistics*, *1*(4), 220-227.

[8] Chatterjee, S., Misra, R. B., & Alam, S. S. (1997). Prediction of software reliability using an auto regressive process. *International Journal of Systems Science*, *28*(2), 211-216.

[9] Chatterjee, S., & Singh, J. B. (2014). A NHPP based software reliability model and optimal release policy with logistic–exponential test coverage under imperfect debugging. *International Journal of System Assurance Engineering and Management*, *5*(3), 399-406.

[10] Patterson, D. A., & Hennessy, J. L. (2013). Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann Publishers.

[11] Moravec, H. (1998). When will computer hardware match the human brain. *Journal of Evolution and Technology*, *1*(1), 10.

[12] Lyu, M. R. (1996). Handbook of Software Reliability Engineering. IEEE Computer Society.

[13] Lyu, M. R. (2007). Software reliability engineering: A roadmap. *2007 Future of Software Engineering*, IEEE Computer Society, 153-170.

[14] https://www.computerworlduk.com/galleries/infrastructure/top-software-failures-recent-history-3599618/#22.

[15] Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, RTI Project, 7007(011).

[16] Halstead, M. H. (1977). Elements of Software Science. Elsevier.

[17] McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, *4*, 308-320.

[18] Pham, H. (2007). System Software Reliability. Springer Science & Business Media.

[19] Pham, H. (2000). Software Reliability. Springer Science & Business Media.

[20] Myers, G. J., Sandler, C., & Badgett, T. (2011). The Art of Software Testing. John Wiley & Sons.

[21] Ohmann, P., & Liblit, B. (2017). Lightweight control-flow instrumentation and postmortem analysis in support of debugging. *Automated Software Engineering*, *24*(4), 865-904.

[22] Hailpern, B., & Santhanam, P. (2002). Software debugging, testing, and verification. *IBM Systems Journal*, *41*(1), 4-12.

[23] Weyuker, E. J. (2004). How to judge testing progress. *Information and Software Technology*, *46*(5), 323-328.

[24] Kaner, C., Falk, J., & Nguyen, H. Q. (2000). Testing Computer Software. Second Edition. Dreamtech Press.

[25] Pham, H., & Zhang, X. (1999). Software release policies with gain in reliability justifying the costs. *Annals of Software Engineering*, *8*(1-4), 147-166.

[26] Pham, H., & Zhang, X. (1999). A software cost model with warranty and risk costs. *IEEE Transactions on Computers*, *48*(1), 71-75.

[27] Zhang, X., & Pham, H. (2000). An analysis of factors affecting software reliability. *Journal of Systems and Software*, *50*(1), 43-56.

[28] Zhang, X., Shin, M. Y., & Pham, H. (2001). Exploratory analysis of environmental factors for enhancing the software reliability assessment. *Journal of Systems and Software*, *57*(1), 73-78.

[29] Zhu, M., Zhang, X., & Pham, H. (2015). A comparison analysis of environmental factors affecting software reliability. *Journal of Systems and Software*, *109*, 150-160.

[30] Zhu, M., & Pham, H. (2017). Environmental factors analysis and comparison affecting software reliability in development of multi-release software. *Journal of Systems and Software*, *132*, 72-84.

[31] HP Applications Handbook. (2012). Shorten release cycles by bringing developers to application lifecycle management. http://www.hp.com/hpinfo/newsroom/press_kits/2011/optimization2011/Lifecycle.pdf.

[32] Khomh, F., Dhaliwal, T., Zou, Y., & Adams, B. (2012). Do faster releases improve software quality? An empirical case study of Mozilla Firefox. *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, IEEE, 179-188.

[33] Gallud, J. A., Peñalver, A., López-Espín, J. J., Lazcorreta, E., Botella, F., Fardoun, H. M., & Sebastián, G. (2012). A proposal to validate the user's goal in distributed user interfaces. *International Journal of Human-Computer Interaction*, *28*(11), 700-708.

[34] Eisenstein, J., Vanderdonckt, J., & Puerta, A. (2001). Applying model-based techniques to the development of UIs for mobile computers. *Proceedings of the 6th International Conference on Intelligent User Interfaces*, ACM, 69-76.

[35] Ramasubbu, N., & Balan, R. K. (2007). Globally distributed software development project performance: An empirical analysis. *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM, 125-134.

[36] Herbsleb, J. D., Mockus, A., Finholt, T. A., & Grinter, R. E. (2000). Distance, dependencies, and delay in a global collaboration. *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, ACM, 319-328.

[37] Herbsleb, J. D., Mockus, A., Finholt, T. A., & Grinter, R. E. (2001). An empirical study of global software development: Distance and speed. *Proceedings of the 23rd International Conference on Software Engineering*, IEEE Computer Society, 81-90.

[38] Herbsleb, J. D., & Mockus, A. (2003). An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*, *29*(6), 481-494.

[39] Saliu, O., & Ruhe, G. (2005). Software release planning for evolving systems. *Innovations in Systems and Software Engineering*, *1*(2), 189-204.

[40] Ruhe, G., & Momoh, J. (2005). Strategic release planning and evaluation of operational feasibility. *Proceedings of the 38th Annual Hawaii International Conference on System Science*, IEEE, 313b-313b.

[41] Svahnberg, M., Gorschek, T., Feldt, R., Torkar, R., Saleem, S. B., & Shafique, M. U. (2010). A systematic review on strategic release planning models. *Information and Software Technology*, *52*(3), 237-248.

[42] Maurice, S., Ruhe, G., & Saliu, O. (2006). Decision support for value-based software release planning. *Value-based Software Engineering*, Springer, 247-261.

[43] Greer, D., & Ruhe, G. (2004). Software release planning: an evolutionary and iterative approach. *Information and Software Technology*, *46*(4), 243-253.

[44] Missbauer, H. (2002). Aggregate order release planning for time-varying demand. *International Journal of Production Research*, *40*(3), 699-718.

[45] Al-Emran, A., & Pfahl, D. (2007). Operational planning, re-planning and risk analysis for software releases. *International Conference on Product Focused Software Process Improvement*, Springer, 315-329.

[46] Gorschek, T., & Davis, A. M. (2008). Requirements engineering: In search of the dependent variables. *Information and Software Technology*, *50*(1-2), 67-75.

[47] Hu, Q. P., Peng, R., Xie, M., Ng, S. H., & Levitin, G. (2011). Software reliability modelling and optimization for multi-release software development processes. *Proceedings of 2011 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, IEEE, 1534-1538.

[48] Kapur, P. K., Pham, H., Aggarwal, A. G., & Kaur, G. (2012). Two dimensional multi-release software reliability modeling and optimal release planning. *IEEE Transactions on Reliability*, *61*(3), 758-768.

[49] Yang, J., Liu, Y., Xie, M., & Zhao, M. (2016). Modeling and analysis of reliability of multi-release open source software incorporating both fault detection and correction processes. *Journal of Systems and Software*, *115*, 102-110.

[50] Pachauri, B., Dhar, J., & Kumar, A. (2015). Incorporating inflection S-shaped fault reduction factor to enhance software reliability growth. *Applied Mathematical Modelling*, *39*(5-6), 1463-1469.

[51] Bosch, J., & Bosch-Sijtsema, P. (2010). From integration to composition: On the impact of software product lines, global development and ecosystems. *Journal of Systems and Software*, *83*(1), 67-76.

[52] Hallsteinsen, S., Hinchey, M., Park, S., & Schmid, K. (2008). Dynamic software product lines. *Computer*, *41*(4).

[53] Clements, P., & Northrop, L. (2002). Software Product Lines. Addison-Wesley.

[54] Cascio, W. F., & Shurygailo, S. (2003). E-leadership and virtual teams. *Organizational Dynamics, 31*(4), 362-376.

[55] Messerschmitt, D. G., & Szyperski, C. (2005). Software Ecosystem: Understanding an Indispensable Technology and Industry. MIT Press Books, 1.

[56] Jansen, S., Brinkkemper, S., & Finkelstein, A. (2007). Providing transparency in the business of software: A modeling technique for software supply networks. *Establishing the Foundation of Collaborative Networks*, Springer, 677-686.

[57] Bastani, F. B., & Ramamoorthy, C. V. (1986). Input-domain-based models for estimating the correctness of process control programs. *Reliability Theory*, 321-378.

[58] Goel, A. L. (1985). Software reliability models: Assumptions, limitations, and applicability. *IEEE Transactions on Software Engineering*, *12*, 1411-1423.

[59] Musa, J. D., Iannino, A., & Okumoto, K. (1990). Software reliability. *Advances in Computers, 30*, 85-170.

[60] Mellor, P. (1987). Software reliability modelling: The state of the art. *Information and Software Technology*, *29*(2), 81-98.

[61] Xie, M. (1991). Software Reliability Modelling. World Scientific.

[62] Mills, H. (1972). On the Statistical Validation of Compute Programs. *IBM Federal Systems Division Report*, 72-6015.

[63] Cai, K. Y. (1998). On estimating the number of defects remaining in software. *Journal of Systems and Software*, *40*(2), 93-114.

[64] Tohma, Y., Yamano, H., Ohba, M., & Jacoby, R. (1991). The estimation of parameters of the hypergeometric distribution and its application to the software reliability growth model. *IEEE Transactions on Software Engineering*, *17*(5), 483-489.

[65] Jelinski, Z., & Moranda, P. (1972). Software reliability research. *Statistical Computer Performance Evaluation*, 465-484.

[66] Schick, G. J., & Wolverton, R. W. (1978). An analysis of competing software reliability models. *IEEE Transactions on Software Engineering*, *2*, 104-120.

[67] Moranda, P. B. (1981). An error detection model for application during software development. *IEEE Transactions on Reliability*, *30*(4), 309-312.

[68] Belady, L. A., & Lehman, M. M. (1976). A model of large program development. *IBM Systems journal*, *15*(3), 225-252.

[69] Miller, D. R., & Sofer, A. (1985). Completely monotone regression estimates of software failure rates. *Proceedings of the 8th International Conference on Software Engineering*, IEEE Computer Society, 343-348.

[70] Coutinho, J. D. S. (1973). Software reliability growth. *Proceedings of the IEEE Symposium on Computer Software Reliability*, 58-64.

[71] Wall, J. K., & Ferguson, P. A. (1977). Pragmatic software reliability prediction. *Annual Reliability and Maintainability Symposium*, 485-488.

[72] Goel, A. L., & Okumoto, K. (1979). A Markovian model for reliability and other performance measures of software systems. *National Computer Conference*, IEEE, 769-774.

[73] Littlewood, B. (1979). Software reliability model for modular program structure. *IEEE Transactions on Reliability*, *28*(3), 241-246.

[74] Yamada, S., Tokuno, K., & Kasano, Y. (1998). Quantitative assessment models for software safety/reliability. *Electronics and Communications in Japan (Part II: Electronics)*, *81*(5), 33-43.

[75] Singpurwalla, N. D., & Soyer, R. (1985). Assessing (software) reliability growth using a random coefficient autoregressive process and its ramifications. *IEEE Transactions on Software Engineering*, *12*, 1456-1464.

[76] Ho, S. L., & Xie, M. (1998). The use of ARIMA models for reliability forecasting and analysis. *Computers & Industrial Engineering*, *35*(1-2), 213-216.

[78] Xie, M., & Ho, S. L. (1999). Analysis of repairable system failure data using time series models. *Journal of Quality in Maintenance Engineering*, *5*(1), 50-61.

[79] Goel, A. L., & Okumoto, K. (1979). Time-dependent error-detection rate model for software reliability and other performance measures. *IEEE Transactions on Reliability*, *28*(3), 206-211.

[80] Ohba, M. (1984). Software reliability analysis models. *IBM Journal of Research and Development*, *28*(4), 428-443.

[81] Yamada, S., & Osaki, S. (1985). Software reliability growth modeling: Models and applications. *IEEE Transactions on Software Engineering*, *12*, 1431-1437.

[82] Ohba, M., Yamada, S., Takeda, K., & Osaki, S. (1982). S-shaped software reliability growth curve: How good is it?. *Proceedings of COMPSAC'82*, 38-44.

[83] Ohba, M., & Yamada, S. (1984). S-shaped software reliability growth models. *Proceedings of the 4th International Colloquium on Reliability and Maintainability*, 430-436.

[84] Yamada, S., Ohba, M., & Osaki, S. (1983). S-shaped reliability growth modeling for software error detection. *IEEE Transactions on Reliability*, *32*(5), 475-484.

[85] Yamada, S., Ohba, M., & Osaki, S. (1984). S-shaped software reliability growth models and their applications. *IEEE Transactions on Reliability*, *33*(4), 289-292.

[86] Yamada, S., Ohtera, H., & Narihisa, H. (1986). Software reliability growth models with testing-effort. *IEEE Transactions on Reliability*, *35*(1), 19-23.

[87] Nakagawa, Y. (1994). A connective exponential software reliability growth model based on analysis of software reliability growth curves. *IEICE Trans*, *77*, *433-442*.

[88] Misra, S. C., Kumar, V., & Kumar, U. (2009). Identifying some important success factors in adopting agile software development practices. *Journal of Systems and Software*, *82*(11), 1869-1890.

[89] Chow, T., & Cao, D. B. (2008). A survey study of critical success factors in agile software projects. *Journal of Systems and Software*, *81*(6), 961-971.

[90] Clarke, P., & O'Connor, R. V. (2012). The situational factors that affect the software development process: Towards a comprehensive reference framework. *Information and Software Technology*, *54*(5), 433-447.

[91] Pham, H. (1993). Software reliability assessment: Imperfect debugging and multiple failure types in software development. EGandG-RAAM-10737, Idaho National Engineering Laboratory.

[92] Pham, H., & Zhang, X. (1997). An NHPP software reliability model and its comparison. *International Journal of Reliability, Quality and Safety Engineering*, *4*(03), 269-282.

[93] Pham, H., Nordmann, L., & Zhang, Z. (1999). A general imperfect-software-debugging model with S-shaped fault-detection rate. *IEEE Transactions on Reliability*, *48*(2), 169-175.

[94] Hossain, S. A., & Dahiya, R. C. (1993). Estimating the parameters of a non-homogeneous Poisson-process model for software reliability. *IEEE Transactions on Reliability*, *42*(4), 604-612.

[95] Pham, H. (2014). Loglog fault-detection rate and testing coverage software reliability models subject to random environments. *Vietnam Journal of Computer Science*, *1*(1), 39-45.

[96] Pham, H., & Wang, H. (2001). A quasi-renewal process for software reliability and testing costs. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, *31*(6), 623-631.

[97] Pham, H., & Zhang, X. (2003). NHPP software reliability and cost models with testing coverage. *European Journal of Operational Research*, *145*(2), 443-454.

[98] Pham, L., & Pham, H. (2000). Software reliability models with time-dependent hazard function based on Bayesian approach. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, *30*(1), 25-35.

[99] Inoue, S., & Yamada, S. (2004). Testing-coverage dependent software reliability growth modeling. *International Journal of Reliability, Quality and Safety Engineering*, *11*(04), 303-312.

[100] Jones, C. (1996). Software defect-removal efficiency. *Computer*, *29*(4), 94-95.

[101] Kapur, P. K., GUPTA, A., & Jha, P. C. (2007). Reliability analysis of project and product type software in operational phase incorporating the effect of fault removal efficiency. *International Journal of Reliability, Quality and Safety Engineering*, *14*(03), 219-240.

[102] Kapur, P. K., Pham, H., Anand, S., & Yadav, K. (2011). A unified approach for developing software reliability growth models in the presence of imperfect debugging and error generation. *IEEE Transactions on Reliability*, *60*(1), 331-340.

[103] Teng, X., & Pham, H. (2006). A new methodology for predicting software reliability in the random field environments. *IEEE Transactions on Reliability*, *55*(3), 458-468.

[104] Tokuno, K., & Yamada, S. (2000). An imperfect debugging model with two types of hazard rates for software reliability measurement and assessment. *Mathematical and Computer Modelling*, *31*(10-12), 343-352.

[105] Yamada, S., Hishitani, J., & Osaki, S. (1991). Test-effort dependent software reliability measurement. *International Journal of Systems Science*, *22*(1), 73-83.

[106] Zhang, X., Teng, X., & Pham, H. (2003). Considering fault removal efficiency in software reliability assessment. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, *33*(1), 114-120.

[107] Huang, C. Y., & Lyu, M. R. (2005). Optimal release time for software systems considering cost, testing-effort, and test efficiency. *IEEE Transactions on Reliability*, *54*(4), 583-591.

[108] Huang, C. Y. (2005). Performance analysis of software reliability growth models with testing-effort and change-point. *Journal of Systems and Software*, *76*(2), 181-194.

[109] Li, H., Li, Q., & Lu, M. (2008). Software reliability modeling with logistic test coverage function. *Proceedings of the 19th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 319-320.

[110] Lin, C. T., & Huang, C. Y. (2008). Enhancing and measuring the predictive capabilities of testing-effort dependent software reliability models. *Journal of Systems and Software, 81*(6), 1025-1038.

[111] Xie, M., & Zhao, M. (1992). The Schneidewind software reliability model revisited. *Proceedings of the 3rd International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 184-192.

[112] Hwang, S., & Pham, H. (2009). Quasi-renewal time-delay fault-removal consideration in software reliability modeling. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, *39*(1), 200-209.

[113] Sawyer, S., & Guinan, P. J. (1998). Software development: Processes and performance. *IBM Systems Journal*, *37*(4), 552-569.

[114] Roberts, T. L., Gibson, M. L., Fields, K. T., & Rainer, R. K. (1998). Factors that impact implementing a system development methodology. *IEEE Transactions on Software Engineering*, *24*(8), 640-649.

[115] Garmabaki, A. H., Aggarwal, A. G., & Kapur, P. K. (2011). Multi up-gradation software reliability growth model with faults of different severity. *Proceedings of 2011 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, IEEE, 1539-1543.

[116] Zhu, M., & Pham, H. (2017). A multi-release software reliability modeling for open source software incorporating dependent fault detection process. *Annals of Operations Research*. https://doi.org/10.1007/s10479-017-2556-6.

[117] Grottke, M., Nikora, A. P., & Trivedi, K. S. (2010). An empirical investigation of fault types in space mission system software. *Proceedings of 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, 447-456.

[118] Laprie, J. C., Arlat, J., Beounes, C., & Kanoun, K. (1990). Definition and analysis of hardware-and software-fault-tolerant architectures. *Computer*, *23*(7), 39-51.

[119] Avizienis, A. (1985). The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, *12*, 1491-1501.

[120] Grottke, M., & Trivedi, K. S. (2005). A classification of software faults. *Journal of Reliability Engineering Association of Japan*, *27*(7), 425-438.

[121] Grottke, M., & Trivedi, K. S. (2007). Fighting bugs: Remove, retry, replicate, and rejuvenate. *Computer*, *40*(2).

[122] Shetti, N. M. (2003). Heisenbugs and Bohrbugs: Why are they different. *Techn. Ber.* Rutgers, The State University of New Jersey.

[123] Alonso, J., Grottke, M., Nikora, A. P., & Trivedi, K. S. (2013). An empirical investigation of fault repairs and mitigations in space mission system software. *Proceedings of 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, 1-8.

[124] Yazdanbakhsh, O., Dick, S., Reay, I., & Mace, E. (2016). On deterministic chaos in software reliability growth models. *Applied Soft Computing*, *49*, 1256-1269.

[125] Deswarte, Y., Kanoun, K., & Laprie, J. C. (1998). Diversity against accidental and deliberate faults. *Proceedings of the Computer Security, Dependability and Assurance: From Needs to Solution*, IEEE, 171-181.

[126] Laprie, J. C. (1995). Dependable computing: Concepts, limits, challenges. *Special Issue of the 25th International Symposium on Fault-Tolerant Computing*, 42-54.

[127] Kapur, P. K., & Younes, S. (1995). Software reliability growth model with error dependency. *Microelectronics Reliability*, *35*(2), 273-278.

[128] Pham, H. (1996). A software cost model with imperfect debugging, random life cycle and penalty cost. *International Journal of Systems Science*, *27*(5), 455-463.

[129] Pham, H., & Deng, C. (2003). Predictive-ratio risk criterion for selecting software reliability models. *Proceedings of the 9th International Conference on Reliability and Quality in Design*, 17-21.

[130] Goseva-Popstojanova, K., & Trivedi, K. (1999). Failure correlation in software reliability models. *Proceeding of the 10th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 232-241.

[131] Dai, Y. S., Xie, M., & Poh, K. L. (2005). Modeling and analysis of correlated software failures of multiple types. *IEEE Transactions on Reliability*, *54*(1), 100-106.

[132] Huang, C. Y., & Lin, C. T. (2006). Software reliability analysis by considering fault dependency and debugging time lag. *IEEE Transactions on Reliability*, *55*(3), 436-450.

[133] Zhu, M., & Pham, H. (2017). A two-phase software reliability modeling involving with software fault dependency and imperfect fault removal. *Computer Languages, Systems & Structures*, *53*, 27-42.

[134] Hsu, C. J., Huang, C. Y., & Chang, J. R. (2011). Enhancing software reliability modeling and prediction through the introduction of time-variable fault reduction factor. *Applied Mathematical Modelling*, *35*(1), 506-521.

[135] Musa, J. D. (1975). A theory of software reliability and its application. *IEEE Transactions on Software Engineering*, *3*, 312-327.

[136] Pham, H. (2014). A new software reliability model with Vtub-shaped fault-detection rate and the uncertainty of operating environments. *Optimization*, *63*(10), 1481-1490.

[137] Jolliffe, I. T. (2002). Principal Component Analysis. Springer.

[138] Montgomery, D. C. (2017). Design and Analysis of Experiments. John Wiley & Sons.

[139] Montgomery, D. C., Peck, E. A., & Vining, G. G. (2012). Introduction to Linear Regression Analysis. John Wiley & Sons.

[140] Lo, D., Li, J., & Khoo, S. C. (2011). Mining iterative generators and representative rules for software specification discovery. *IEEE Transactions on Knowledge and Data Engineering*, *23*(2), 282-296.

[141] Panagiotou, D., & Mentzas, G. (2011). Leveraging software reuse with knowledge management in software development. *International Journal of Software Engineering and Knowledge Engineering*, *21*(05), 693-723.

[142] Porter, A., Yilmaz, C., Memon, A. M., Krishna, A. S., Schmidt, D. C., & Gokhale, A. (2006). Techniques and processes for improving the quality and performance of open-source software. *Software Process: Improvement and Practice*, *11*(2), 163-176.

[143] Beck, K., & Gamma, E. (2000). Extreme Programming Explained: Embrace Change. Addison-Wesley Professional.

[144] Van der Aa, Z., Bloemer, J., & Henseler, J. (2012). Reducing employee turnover through customer contact center job quality. *The International Journal of Human Resource Management*, *23*(18), 3925-3941.

[145] Mohanty, R., Ravi, V., & Patra, M. R. (2013). Hybrid intelligent systems for predicting software reliability. *Applied Soft Computing*, *13*(1), 189-200.

[146] Sahoo, S. K., Criswell, J., & Adve, V. (2010). An empirical study of reported bugs in server software with implications for automated bug diagnosis. *2010 ACM/IEEE 32nd International Conference on Software Engineering*, IEEE, *1*, 485-494.

[147] Saliu, O., & Ruhe, G. (2005). Software release planning for evolving systems. *Innovations in Systems and Software Engineering*, *1*(2), 189-204.

[148] Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., & Kern, J. (2001). Manifesto for agile software development. www.agilemanifesto.org.

[149] Szőke, Á. (2011). Conceptual scheduling model and optimized release scheduling for agile environments. *Information and Software Technology*, *53*(6), 574-591.

[150] Li, L., Harman, M., Letier, E., & Zhang, Y. (2014). Robust next release problem: Handling uncertainty during optimization. *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, ACM, 1247-1254.

[151] Etgar, R., Gelbard, R., & Cohen, Y. (2017). Optimizing version release dates of research and development long-term processes. *European Journal of Operational Research*, *259*(2), 642-653.

[152] Fritsch, B. L., Bibr, V., Blagojevic, V., Goring, B. R., Shenfield, M., & Vitanov, K. B. (2010). *U.S. Patent No. 7,747,995*. Washington, DC: U.S. Patent and Trademark Office.

[153] Staron, M., Meding, W., & Palm, K. (2012). Release readiness indicator for mature agile and lean software development projects. *International Conference on Agile Software Development*, Springer, 93-107.

[154] James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). An Introduction to Statistical Learning. Springer.

[155] Fang, C. C., & Yeh, C. W. (2016). Effective confidence interval estimation of fault-detection process of software reliability growth models. *International Journal of Systems Science*, *47*(12), 2878-2892.

[156] Xie, M., & Yang, B. (2003). A study of the effect of imperfect debugging on software development cost. *IEEE Transactions on Software Engineering*, *29*(5), 471-473.

[157] Rana, R., Staron, M., Berger, C., Hansson, J., Nilsson, M., Törner, F., Meding, W., & Höglund, C. (2014). Selecting software reliability growth models and improving their predictive accuracy using historical projects data. *Journal of Systems and Software*, *98*, 59-78.

[158] Yan, H. (2013). NHPP software reliability growth model incorporating fault detection and debugging. *2013 IEEE 4th International Conference on Software Engineering and Service Science (ICSESS)*, IEEE, 225-228.

[159] Li, Q., & Pham, H. (2017). A testing-coverage software reliability model considering fault removal efficiency and error generation. *PloS one*, *12*(7), e0181524.

[160] Huang, C. Y. (2005). Cost-reliability-optimal release policy for software reliability models incorporating improvements in testing efficiency. *Journal of Systems and Software*, *77*(2), 139-155.

[161] Huang, C. Y., Luo, S. Y., & Lyu, M. R. (1999). Optimal software release policy based on cost and reliability with testing efficiency. *Proceedings of the 23rd Annual International Computer Software and Applications Conference*, IEEE, 468-473.

[162] Huang, C. Y., & Kuo, S. Y. (2002). Analysis of incorporating logistic testing-effort function into software reliability modeling. *IEEE Transaction on Reliability, 51*(3), 261-270.

[163] Zhang, X., Jeske, D. R., & Pham, H. (2002). Calibrating software reliability models when the test environment does not match the user environment. *Applied Stochastic Models in Business and Industry, 18*(1), 87-99.

[164] Vaidyanathan, K., & Trivedi, K. S. (2001). Extended classification of software faults based on aging. *Fast Abstracts, Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE.

[165] Wood, A. (1996). Predicting software reliability. *Computer, 29*(11), 69-77.

[166] Jeske, D. R., & Zhang, X. (2005). Some successful approaches to software reliability modeling in industry. *Journal of Systems and Software, 74*(1), 85-99.

[167] Mehlawat, M. K. (2013). A multi-choice goal programming approach for COTS products selection of modular software systems. *International Journal of Reliability, Quality and Safety Engineering, 20*(06), 1350026.

[168] Carlshamre, P., Sandahl, K., Lindvall, M., Regnell, B., & Natt och Dag, J. N. (2001). An industrial survey of requirements interdependencies in software product release planning. *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, IEEE, 84-91.

[169] Ho, J., & Ruhe, G. (2013). Releasing sooner or later: An optimization approach and its case study evaluation. *Proceedings of the 1st International Workshop on Release Engineering*, IEEE, 21-24.

[170] Kachitvichyanukul, V. (2012). Comparison of three evolutionary algorithms: GA, PSO, and DE. *Industrial Engineering and Management Systems, 11*(3), 215-223.

[171] Raymond, E. S. (2001). The cathedral and the bazaar: Musings on Linux and open source by an accidental revolutionary (2nd ed.). Sebastapol, CA: O'Reilly.

[172] Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., & Zhai, C. (2006). Have things changed now? An empirical study of bug characteristics in modern open source software. *Proceedings of the 1st workshop on Architectural and System Support for Improving Software Dependability*, ACM, 25-33.

[173] Li, X., Li, Y. F., Xie, M., & Ng, S. H. (2011). Reliability analysis and optimal version-updating for open source software. *Information and Software Technology*, *53*(9), 929-936.

[174] Zhu, M., & Pham, H. (2016). A software reliability model with time-dependent fault detection and fault removal. *Vietnam Journal of Computer Science*, *3*(2), 71-79.

[175] https://www.apache.org/.

[176] Fiondella, L., Rajasekaran, S., & Gokhale, S. S. (2013). Efficient software reliability analysis with correlated component failures. *IEEE Transactions on Reliability*, *62*(1), 244-255.

[177] Condori-Fernandez, N., & Lago, P. (2018). Characterizing the contribution of quality requirements to software sustainability. *Journal of Systems and Software*, *137*, 289-305.

[178] El-Sebakhy, E. A. (2009). Software reliability identification using functional networks: A comparative study. *Expert Systems with Applications*, *36*(2), 4013-4020.

[179] Ponnurangam, D., & Uma, G. V. (2005). Fuzzy complexity assessment model for resource negotiation and allocation in agent-based software testing framework. *Expert Systems with Applications*, *29*(1), 105-119.

[180] Storey, M. A., Zagalsky, A., Figueira Filho, F., Singer, L., & German, D. M. (2017). How social and communication channels shape and challenge a participatory culture in software development. *IEEE Transactions on Software Engineering*, *43*(2), 185-204.

[181] Singer, L., Figueira Filho, F., Cleary, B., Treude, C., Storey, M. A., & Schneider, K. (2013). Mutual assessment in the social programmer ecosystem: An empirical investigation of developer profile aggregators. *Proceedings of 2013 conference on Computer Supported Cooperative Work*, ACM, 103-116.

[182] Wenger, E. C., & Snyder, W. M. (2000). Communities of practice: The organizational frontier. *Harvard Business Review*, *78*(1), 139-146.

[183] Jenkins, H., Purushotma, R., Weigel, M., Clinton, K., & Robison, A. J. (2009). Confronting the Challenges of Participatory Culture: Media Education for the 21st Century. MIT Press.

[184] Harman, M., Jia, Y., & Zhang, Y. (2012). App store mining and analysis: MSR for app stores. *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR)*, IEEE, 108-111.

[185] Ghazawneh, A., & Henfridsson, O. (2013). Balancing platform control and external contribution in third-party development: The boundary resources model. *Information Systems Journal*, *23*(2), 173-192.

[186] Basole, R. C., & Karla, J. (2011). On the evolution of mobile platform ecosystem structure and strategy. *Business & Information Systems Engineering*, *3*(5), 313.

[187] Radatz, J., Geraci, A., & Katki, F. (1990). IEEE standard glossary of software engineering terminology. *IEEE Standard*, 610121990(121990), 3.

[188] Pham, T., & Pham, H. (2017). A generalized software reliability model with stochastic fault-detection rate. *Annals of Operations Research*. https://doi.org/10.1007/s10479-017-2486-3.

[189] Mikosch, T. (1998). Elementary Stochastic Calculus, with Finance in View. World Scientific.

[190] Mörters, P., & Peres, Y. (2010). Brownian Motion. Cambridge University Press.

[191] Melo, W. L., Briand, L., & Basili, V. R. (1998). Measuring the impact of reuse on quality and productivity in object-oriented systems. Technical Report, University of Maryland, Department of Computer Science, College Park, MD, USA.

[192] McGee, S., & Greer, D. (2010). Sources of software requirements change from the perspectives of development and maintenance. *International Journal on Advances in Software*, *3*(1-2), 186-200.

[193] Harker, S. D., Eason, K. D., & Dobson, J. E. (1993). The change and evolution of requirements as a challenge to the practice of software engineering. *Proceedings of IEEE International Symposium on Requirements Engineering*, IEEE, 266-272.

[194] Nurmuliani, N., Zowghi, D., & Powell, S. (2004). Analysis of requirements volatility during software development life cycle. *Proceedings of 2004 Australian Software Engineering Conference*, IEEE, 28-37.

[195] Carlshamre, P. (2002). Release planning in market-driven software product development: Provoking an understanding. *Requirements Engineering*, *7*(3), 139-151.

[196] Nurmuliani, N., Zowghi, D., & Williams, S. P. (2004). Using card sorting technique to classify requirements change. *Proceedings of the 12th IEEE International Requirements Engineering Conference*, IEEE, 240-248.

[197] Shi, L., Wang, Q., & Li, M. (2013). Learning from evolution history to predict future requirement changes. *Proceedings of the 21st IEEE International Requirements Engineering Conference*, IEEE, 135-144.

[198] Wheeler, D. A. (2007). Why open source software/free software (OSS/FS, FLOSS, or FOSS)? Look at the numbers. https://www.dwheeler.com/oss_fs_why.html.

[199] Zhou, Y., & Davis, J. (2005). Open source software reliability model: An empirical approach. *ACM SIGSOFT Software Engineering Notes*, ACM, *30*(4),1-6.

[200] Cosgrove, L. (2003). Confidence in open source growing. CIO Research Report.

[201] Loconsole, A., & Borstler, J. (2005). An industrial case study on requirements volatility measures. *Proceedings of the 12th Asia-Pacific Software Engineering Conference* (*APSEC'05)*, IEEE.

[202] Zhu, M., & Pham, H. (2018). A Software reliability model incorporating martingale process with gamma-distributed environmental factors. *Annals of Operations Research*. https://doi.org/10.1007/s10479-018-2951-7.

[203] https://www.seguetech.com/waterfall-vs-agile-methodology/.

[204] Peng, R., Li, Y. F., Zhang, W. J., & Hu, Q. P. (2014). Testing effort dependent software reliability model for imperfect debugging process considering both detection and correction. *Reliability Engineering & System Safety*, *126*, 37-43.

[205] Zhang, X. (1999). Software reliability and cost models with environmental factors. Doctor of Philosophy Dissertation, Department of Industrial and Systems Engineering, Rutgers, The State University of New Jersey, New Brunswick, New Jersey.

[206] Teng, X. (2001). A non-homogeneous Poisson process software reliability growth model for N-version programming systems. Doctor of Philosophy Dissertation, Department of Industrial and Systems Engineering, Rutgers, The State University of New Jersey, New Brunswick, New Jersey.