AL al-Bayt **University**

AL al-Bayt **University**
Prince Hussein bin Abdullah Faculty of Information Technology
Computer Science Department

THE PNZ SOFTWARE RELIABILITY MODEL REVISITED

By
KHALID HASSAN MOHAMED EDRIS
Enrollment No.: 0620901016

SUPERVISOR
DR. OMAR SHATNAWI

December 2009

THE PNZ SOFTWARE RELIABILITY MODEL REVISITED

إعادة النظر في نموذج (PNZ) لعول البرمجيات

By

KHALID HASSAN MOHAMED EDRIS

Enrollment No.: 0620901016

SUPERVISOR

DR. OMAR SHATNAWI

The members of Committee discussion — signature

DR. OMAR SHATNAWI .....

PROF.DR. ADNAN AL-SMADI .....

DR. KHALED BATIHA .....

DR. RAED SHATNAWI .....

Submitted this thesis to complement the requirements for obtaining a master's degree in computer science at the Faculty of Prince Hussein bin Abdullah Information Technology at the Al-albayt University

Discussed and recommended / modify / reject on Dec. 30, 2009

Dedication

To my dear mother

To my dear father

To my Faithful wife

To my son

To my brothers and sisters

To all my family

To my friends

To all my teachers

I dedicate them this research.

Acknowledgment

I express my thanks and appreciation to Dr. Omar Shatnawi who prefer to supervise this thesis, and thank him for the effort made for me, and what I found his support, respect, appreciation, cooperation, and to overcome the difficulties that stood in my way during the preparation of this thesis.

I thank the members of the Committee discussion on the approval to discuss this thesis and to evaluate it.

Contents

Tables list

Figures list

Appendixes list

Abstract

In the present scenario, computer systems are indispensable for society and their need and importance are increasing rapidly. To meet this increasing demand, the complexity of the software products to construct such computer systems, has enhanced to a considerable extent. During the development of such complex software systems, many software failures may occur. To reduce these faults, thorough testing of the software is required so that a highly reliable software system can be developed. Over the past three decades, there have been several attempts at modeling the processes associated with software failures based on various underlying assumptions related to how software is tested. These models are collectively known as software reliability growth model (SRGMs). It is important to note that due to the complexity of software design, it is not expected that any single model can incorporate all factors which are thought to influence software reliability.

In this thesis, we show how beginning with very simple assumptions, non-homogenous Poisson process (NHPP) type of continuous time SRGM, are gradually made more realistic with the incorporation of imperfect debugging, involvement of a learning-process in debugging and introduction of new faults. The applicability of the resultant generalized model is demonstrated through several actual software reliability data sets obtained from different software development projects. The proposed generalized model is also checked against different components of the model, including existing one (e.g., PNZ, Inflection S-shaped, Fault Generation, Imperfect, Delayed S-shaped, Exponential) thus highlighting its applicability. The software reliability data sets were deliberately chosen from different testing environments where the growth curves ranging from purely exponential to highly S-shaped. The results are fairly encouraging in terms of goodness of fit, predictive validity and software reliability evaluation measures.

The major contribution of this thesis is its introduction the concept of two types of imperfect debugging during software fault removal phenomenon with logistic fault removal rate. Most of the SRGMs discussed in the literature seldom differentiate between the failure observation and fault removal processes. In real software development environment, the number of failures observed need not be same as the number of fault removed. If the number of failures observed is more than the number of faults removed then we have the case of imperfect debugging. Due to the complexity of

the software system and the incomplete understanding of the software requirements, specifications and structure, the testing team may not be able to remove the fault perfectly on the detection of the failure and the original fault may remain or get replaced by another fault. While the first phenomenon is known as *imperfect debugging*, the second is called *fault generation*. In case of imperfect debugging the fault-content of the software is not changed, but just because of incomplete understanding of the software, the detected fault is not removed completely. But in case of fault generation the fault-content increases as the testing progresses and removal results in introduction of new faults while removing old ones. To model learning, fault removal rate has been taken as logistic function. Actual software reliability data cited from real software development projects have been used to demonstrate the applicability of the proposed model.

The results of the proposed model are encouraging in terms of goodness of fit criteria, predictive validity criterion, and software reliability evaluation measures for software reliability data due to its applicability and flexibility.

Chapter 1

Introduction and Overview

Today, computer systems are indispensable in our daily lives, and their importance and need have increased immensely. Successful operation of any computer system depends largely on its software components. Thus, its very important to ensure the quality of the underlying software in the sense that it performs its functions that its designed and built for. Software development process is often called software development lifecycle (SDLC), because it describes the life of a software product from its inception to its implementation. Every software development process includes system requirements, as it is input and a delivered product as its output. Many lifecycle models have been proposed, based on the tasks involved in developing and maintaining software, but they all consists of the following stages: requirement and specification, design and coding, testing, and operation and maintenance. Faults can be introduced during any of these stages and hence it is not possible to produce fault-free software due to human imperfection. A fault occurs when a human makes a mistake, called an error, in performing activities related to the software. A fault can reside in any development or maintenance system. Faults manifest themselves in terms of failures, when the software is executed. A failure is a departure from the system's required behavior. It can be discovered before or after system delivery, during testing, or during operation and maintenance. Testing phase in the software development process aims at detecting and removing faults, and hence making the software more reliable. It is this phase, which is amenable to mathematical modeling.

The most used way to verify and validate the software is by testing. Software testing involves running the software and checking for unexpected behavior in software output. The successful test can be considered to be one, which reveals the presence of the latent faults. The process of locating the faults and designing the procedures to remove them is called the debugging process. The chronology of failure occurrence and fault removals can be utilized to provide an estimate of the software reliability and the level of fault content. The software reliability model is the tool, which can be used to evaluate the software quantitatively, develop test status, schedule status and monitor the change in reliability performance.

1.1 Software Reliability Engineering

Software Reliability Engineering (SRE) is a practice that helps you develop software that is more reliable, and helps you develop it faster and cheaper. It is a standard, proven, widespread best practice that is widely applicable to systems that include software. Software reliability engineering is low in cost and its implementation has virtually no schedule impact (Musa, 2004).

Software reliability engineering works by quantitatively characterizing and applying two things about the product: the expected relative use of its functions and its required major quality characteristics. The major quality characteristics are reliability, availability, delivery date and life-cycle cost. In applying software reliability engineering, you can vary the relative emphasis you place on these factors. When you have characterized use, software reliability engineering guides you in substantially increasing development efficiency by focusing resources on functions in proportion to use and criticality. It also maximizes test efficiency by making test highly representative of use in the field. Increased efficiency increases the effective resource pool available to add customer value. Software reliability engineering is centered around a very important software attribute: reliability. Software reliability is one of the attributes of software quality, a multidimensional property including other customer satisfaction factors like functionality, usability, performance, serviceability, capability, installability, maintainability and documentation (Musa, 2004).

Software Reliability should be defined as the probability of failure-free software operation for a specified period of time in a specified environment (ANSI/IEEE, 1991). Software Reliability is generally accepted as the key factor in software quality since it quantifies software failures—the most unwanted event which makes software useless or even harmful to the whole system and malfunctioning software may kill people. As a result, it is regarded the most important factor contributing to customer satisfaction. In fact, ISO 9000-3 specifies field failures as the basic requirement for quality metrics (Lyu, 1996).

Achieving highly reliable software in the customer's perspective is a demanding job to all software engineers and reliability engineers.

Four technical methods are applicable to achieve reliable software systems (Lyu, 1996):

- **Fault Avoidance.** The interactive refinement of the user's system requirement, the engineering of the software specification process, the use of good software design methods, the enforcement of structured programming discipline and the encouragement of writing clear code are the general approaches to avoid faults in the software. These guidelines have been, and will continue to be, the fundamental techniques in preventing software faults from being created.

Recently, formal methods have been attempted in the research community in attacking the software quality problem. In formal-methods approaches, requirement specifications are developed and maintained using mathematically trackable languages and tools.

Current studies in this area have been focused on language issues and environmental supports, which include at least the following goals:

1. Executable specifications for systematic and precise evaluation,
2. Proof mechanisms for software verification and validation, and
3. Development procedures which follow incremental refinement for step-by-step verification.

Every work item, be it a specification or a test case, is subject to mathematically verification for its correctness and appropriateness.

Another fault avoidance technique, particularly popular in the software development community, is software reuse. The crucial measures of success in this area are the capability to prototype and evaluate reusable synthesis techniques. This is why Object-Oriented Paradigms and Techniques are receiving much attention nowadays—largely due to their inherent properties in enforcing software reuse.

- **Fault Removal.** When formal methods are in full swing, formal design proofs might be available to achieve mathematical proof-of-correctness for programs. Also fault-monitoring assertions could be employed through executable specifications, and test cases could be automatically generated to achieve efficient software verification. However, before this happens, practitioners will have to rely mostly on software testing techniques to remove existing faults. Microsoft, for example, allocates as many software testers as software developers, and employs a "buddy" system which binds the developer of every software component with its tester for their daily work. The key question to reliability engineers, then, is how to derive testing quality measures (e.g., test coverage factors) and establish their relationships to reliability.

Another practical fault removal scheme which has been widely implemented in industry is formal inspection. A formal inspection is a rigorous process focused on finding faults, correcting faults and verifying the corrections. Formal inspection is carried out by a small group of peers with a vested interest in the work product during pretest phases of the lifecycle. Many companies have claimed its success (Lyu, 1996).

- **Fault Tolerance.** Fault tolerance is the survival attribute of computing systems or software in their ability to deliver continuous service to their users in the presence of faults. Software fault tolerance is concerned with all the techniques necessary to enable a system to tolerate software faults remaining in the system after its development. These software faults may or may not manifest themselves during system operations, but when they do, software fault tolerance techniques should provide the necessary mechanisms to the software system to prevent system failure from occurring.

In a single-version software environment, the techniques for partially tolerating software design faults include monitoring techniques, atomicity of actions, decision verification and exception handling. In order to fully recover from activated design faults, multiple versions of software developed via design diversity are introduced, in which functionally equivalent yet independently developed software versions are applied in the system to provide ultimate tolerance to software design faults. The main approaches include the recovery blocks technique, the N-version programming technique and the N self-checking programming technique. These approaches have found a wide range of applications in the aerospace industry, the nuclear power industry, the health care industry, the telecommunications industry and the ground transportation industry.

- **Fault\Failure Forecasting.** Fault\failure forecasting involves formulation of the fault\failure relationship, an understanding of the operational environment, the establishment of reliability models, the collection of failure data, the application of reliability models by tools, the selection of appropriate models, the analysis and interpretation of results, and the guidance for management decisions. This has been the main focus of Software Reliability Modelling.

Due to the intrinsic complexity of modern software systems, software reliability engineers have to apply a combination of the above methods for the delivery of reliable software systems. These four areas are also the main theme of the state of the art for software engineering covering a wide range of disciplines.

Who has used Software Reliability Engineering?

ATandT's International Definity project shows the benefits that result from applying SRE and related technologies. In comparison with a previous release that did not use these technologies, reliability, customer satisfaction and sales all increased by a factor of 10. The system test interval and test costs decreased by a factor of two; project development time by 30% and maintenance costs by a factor of 10. Other organizations such as Alcatel, Bellcore, CNES (France), ENEA (Italy), Ericsson Telecom, France Telecom, Hewlett Packard, Hitachi, IBM, Lockheed-Martin, Lucent Technologies, Microsoft, Mitre, Motorola, NASA's Jet Propulsion Laboratory, NASA's Space Shuttle, Nortel, Raytheon, Saab Military Aircraft, Tandem Computers, the US Air Force and the US Marine Corps have also used SRE profitably (Lyu, 1996).

1.2 Software versus Hardware Reliability

Software reliability is similar to hardware reliability in that both are stochastic processes and can be described by probability distributions. However, software reliability is different from hardware reliability in the sense that software does not wear out, burn out or deteriorate, i.e., its reliability does not decrease with time. Moreover, software generally enjoys reliability growth during testing and operation since software faults can be detected and removed when software failures occur. On the other hand, software may experience reliability decrease due to abrupt changes of its operational usage or incorrect modifications to the software. Software is also continuously modified throughout its lifecycle. The malleability of software makes it inevitable for us to consider variable failure rates.

Unlike hardware faults which are mostly physical faults, software faults are design faults which are harder to visualize, classify, detect and correct. As a result, software reliability is a much more difficult measure to obtain and analyze than hardware reliability. Usually hardware reliability theory relies on the analysis of stationary processes, because only physical faults are considered. However, with the increase of systems complexity and the introduction of design faults in software, reliability theory based on stationary process becomes unsuitable to address non-stationary phenomena such as reliability growth or reliability decrease experienced in software. This makes software reliability a challenging problem which requires an employment of several methods to attack (Lyu, 1996).

Because of this difference in the effect of faults, software reliability must be defined differently from hardware reliability. When hardware is repaired, it is returned to its previous level of reliability; the hardware's reliability is maintained. But when the software is repaired, its reliability may actually increase or decrease. Thus, the goal of hardware reliability engineering is stability; the goal of software reliability engineering is reliability growth (Pfleeger, 2006).

1.3 Software Engineering

Developing software system is generally a quite complex and time consuming process. Moreover, the nature of and complexity of software requirements have drastically changed in the last few decades and users all over the world have become much more demanding in terms of cost, schedule quality. These three parameters, all being desirable, have an apparent contradiction at times which can only be resolved by optimum design of software using well established software engineering methodologies. Software Engineering Methodologies constitute the framework that guides software developers in optimally developing the software systems. These frameworks define the different phases of software development (such as planning, requirements analysis, design testing and maintenance). The choice of which methodology to use in a specific development process is closely related to the size, complexity, reliability and maintainability of the software, and to the environment it is supposed to function in.

Software cost now forms the major component of a computer system's cost. Software is currently expensive to develop and is often unreliable. The goal of the software engineering is to face this "software problem". Software is not just a set of computer programs but comprises programs and associated data and documentation.

The main problems for software development currently are: high cost, low quality, and frequent changes causing rework. Software engineering is the discipline that aims to provide methods and procedures for developing software systems (Lyu, 1996).

1.3.1 Software Life-Cycle

A software lifecycle provides a systematic approach to developing, using, operating, and maintaining a software system. The standard IEEE Computer Dictionary has defined the software lifecycle as: "That period of time in which the software is

conceived, developed and used." There are many different definitions of software lifecycle (Pfleeger, 2006), (Pressman, 2001).

A software lifecycle consists of the following five successive phases: analysis (requirements and functional specifications), design, coding, testing and operating. The **analysis phase** is the first step in the software development process. It is also the most important phase in the whole process and the foundation of building a successful software product. The purpose of the analysis phase is to define the requirements and provides specifications for the subsequent phases and activities. The **design phase** is concerned with building the system to perform as required. There are two stages of design: system architecture design and detailed design. The system architecture design includes system structure and the system architecture document. System structure design is the process of partitioning a software system into smaller parts. The system architecture document describes system components, subsystems and interfaces. Detailed design is about designing the program and the algorithmic details. **Coding** involves translating the design into the code of a programming language, beginning when the design document is baselined. Coding comprises of the following activities: identifying reusable modules, code editing, code inspection and final test planning. **Testing** is the verification and validation activity for the software product. The goals of the testing phase are: to affirm the quality of the product by finding and eliminating faults in the program, to demonstrate the presence of all specified functionality in the product, and to estimate the operational reliability of the software. During the **testing phase**, program components are combined into the overall software code and testing is preformed according to a developed test (Software Verification and Validation) plan. The final phase in the software lifecycle is **operation**. The operating phase usually contains activates such as installation, training, support and maintenance (Lyu, 1996).

1.3.2 Software Verification and Validation

Verification and Validation (VV) are two ways to check whether the design satisfies the user's requirements. According to (ANSI/IEEE, 1991):

Software Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. **Software Validation** is the process of evaluating a system or

component during or at the end of the development process to determine whether it satisfies specified requirements (Lyu, 1996).

1.4 Software Reliability Measurement

Software reliability measurement includes two types of activities, reliability estimation and reliability prediction:

- **Reliability Estimation.** This activity determines current software reliability by applying statistical inference techniques to failure data obtained during system test or operation. This is a measure regarding the achieved reliability from the past until the current point. Its main purpose is to assess the current reliability and determine whether a reliability model is a good fit in retrospect.

- **Reliability Prediction.** This activity determines future software reliability based upon available software metrics and measures. Depending on the software development stage, prediction involves different techniques:

When failure data are available (e.g., software is in system test or operation stage), the estimation techniques can be used to parameterize and verify software reliability models, which can perform future reliability prediction. This definition is also referred to as **Reliability Prediction**.

When failure data are not available (e.g., software is in the design or coding stage), the metrics obtained from the software development process and the characteristics of the resulting product can be used to determine reliability of the software upon testing or delivery. This is referred to as **Early Prediction**.

Most current Software Reliability Models (SRMs) fall in the estimation category to do reliability prediction. Nevertheless, a few early prediction models were proposed and described in the literature. An SRM specifies the general form of the dependence of the failure process on the principal factors that affect it: fault introduction, fault removal and the operational environment.

1.4.1 Definition of Software Reliability

The three major components in the definition of software reliability: time, failure and operational environment:

- **Time.** Reliability quantities are defined with respect to time, although it is possible to define them with respect to other bases like program runs. We are concerned

with three types of time: the execution time for a software system is the CPU time that is actually spent by the computer in executing the software, the calendar time is the time people normally experience in terms of days, weeks, months, etc., and the clock time is the elapsed time from start to end of computer execution in running the software. In measuring clock time, the periods during which the computer is shut down are not counted.

It is generally accepted that execution time is more adequate than calendar time for software reliability measurement and modelling. However, reliability quantities must ultimately be related back to calendar time for easy human interpretation, particularly when managers, engineers, and customers want to compare them across different systems. As a result, translations between calendar time and execution time are required. The technique for such translations is described in (Musa et al., 1987). If execution time is not readily available, approximations such as clock time, weighted clock time, staff working time, or units that are natural to the application, such as transactions or test cases executed, may be used.

- **Failure.** A Software failure is an incorrect result with respect to the specification or an unexpected software behavior perceived by the user at the boundary of the software system, while a software fault is the identified or hypothesized cause of the software failure.

When a time basis is determined, failures can be expressed in several ways: the cumulative failure function, the failure intensity function, the failure rate function and the mean time to failure function. The cumulative failure function (also called the mean value function) denotes the average cumulative failures associated with each point of time. The failure intensity function represents the rate of change of the cumulative failure function.

The failure rate function (or called the hazard rate, or the rate of occurrence of failures) is defined as the instantaneous failure rate at a time *t*, given that the system has not failed up to *t*. The mean time to failure (MTTF) function represents the expected time that the next failure will be observed. (MTTF is also known as MTBF, mean time between failures.) Note that the above four measures are closely-related and could be translated with one another.

- **Operational Profile.** The operational profile of a system is defined as the set of operations that the software can execute along with the probability with which they will occur. An operation is a group of runs which typically involve similar processing.

1.4.2 Fault/Failure Data Collection

Two types of failure data, namely, failure-count data and time-between-failures data, can be collected for the purpose of software reliability measurement. Failure-count (or failures per time period) data tracks the number of failures observed per unit of time. Time-between-failures (or inter-failure times) data tracks the intervals between consecutive failures.

These data are usually used by practitioners when analyzing and predicting reliability applications. Some software reliability models can handle both types of data. The time-between-failures approach involves recording the individual times at which failure occurred. The failure-count approach characterized by counting the number of failures occurring during a fixed period (e.g., test session, hour, week, day). Using this method, the collected data are a count of the number of failures in the interval. The time-between-failures approach always provides higher accuracy in the parameters estimates with current tools but involves more data collection efforts than the interval approach. The practitioners must trade off the cost of data collection with the accuracy reliability level required by the model predictions.

Many reliability modelling programs have the capability to estimate model parameters from either failure-count or time-between-failures data, as statistical modeling techniques can be applied to both (Lyu, 1996), (Musa et al., 1987), (Musa, 2004), (Pham, 2000), (Pfleeger, 2006). However, if a program accommodates only one type of data, it may be required to transform the other type. If the expected input is failure-count data, it may be obtained by transforming time-between-failures data to cumulative failure times and then simply counting the number of failures whose cumulative times occur within a specified time period. If the expected input is time-between-failures data, converting the failure-count data can be achieved by either randomly or uniformly allocating the failures for the specified time intervals, and then by calculating the time periods between adjacent failures.

1.5 Software Reliability Modelling

There are two main types of software reliability models: the deterministic and the probabilistic, more details can be found in (Xie, 1991), (Kapur et al., 1999), (Pham, 2000). The deterministic model is used to study the number of distinct operands in a program as well as the number of errors and the number of machine instructions in the

program. Performance measures of the deterministic type are obtained by analyzing the program texture and do not involve any random event. The probabilistic model represents the failure occurrences and the fault removals as probabilistic events.

The Probabilistic Software Reliability Models can be classified into different groups (Pham, 2000):

- **Error Seeding Models.** The error seeding group of models estimates the number of errors in a program by using the multistage sampling technique. Errors are divided into indigenous errors and induced/seeded errors. The unknown number of indigenous errors is estimated from the number of induced errors and the ratio of the two types of errors obtained from the debugging data.

- **Failure Rate Models.** The failure rate group of models is used to study the program failure rate per fault at the failure intervals. This group of models studies how failure rates change at the failure time during the failure intervals. As the number of remaining faults changes, the failure rate of the program changes accordingly. Since the number of faults in the program is a discrete function, the failure rate of the program is also a discrete function with discontinues at the failure times.

- **Curve Fitting Models.** The curve fitting group models uses statistical regression analysis to study the relationship between software complexity and the number of faults in a program, the number of changes, or failure rate. This group of models finds a functional relationship between dependent and independent variables by using the methods of linear regression, nonlinear regression, or time series analysis. The independent variables, for example, are the number of modules changed in the maintenance phase, time between failures, programmer's skill, program size, etc.

- **Reliability Growth Models.** The reliability growth group of models measures and predicts the improvement of reliability programs through testing process. The growth model represents the reliability or failure rate of a system as a function of time or the number of test cases.

- **Markov Structure Models.** A Markov process has the property that the future behavior of the process depends only on the current state and is independent of its past history. The Markov structure group of modules is a general way of representing the failure process of software. This group of modules can also be used to study the reliability and interrelationship of the modules. It is assumed that failures of the modules are independent of each other. This assumption seems reasonable at the

module level since they can be designed, coded and tested independently, but may not be true at the system level.

- **Non-Homogeneous Poisson Process (NHPP) Models.** The NHPP group of models provides an analytical framework for describing the software failure-occurrence or fault-removal phenomenon during testing. The main issue in the NHPP model is to estimate the mean value function of the cumulative number of failures experienced or faults removed up to a certain point in time. This group of models will be reviewed in detail in the next chapter.

1.6 Basic Definitions and Acronyms Used

We give below the definitions of the terms and acronyms used in this Thesis. These definitions are cited from (Musa, 1999), (Kapur, 1999), (Pfleeger, 1998) and from research papers.

Bug. A mistake in interpreting a requirement, a syntax error in a piece of code, or the (as-yet-unknown) cause of a system crash.

Calendar Time. Chorological time including time in which a computer may not be running.

Clock Time. Elapsed time from start to end of a program execution including wait time on a running computer.

Debugging Process. The process of analyzing the cause of the software failure, locating the faulty part of the software and implementing the necessary steps to remove the software fault.

Deterministic. Possessing the property of having one value at a given time.

Environment. The set of all possible input states/space with their associated probabilities of occurrence.

Estimation. Determination of software reliability model parameters and quantities from fault detection data.

Machine Execution / CPU Time. Time spent by the processor in executing a program.

Failure. A departure from the system's required behavior. It can be discovered before or after system delivery, during testing, or during operation and maintenance.

Failure Intensity. Failures per natural or time unit, an alternative way of expressing reliability.

Fault. A fault occurs when a human makes a mistake, called an **Error**, in performing some software activity. For example, a designer may misunderstand a requirement and create a design that does not match the actual intent of the requirement analyst and the user.

Fault Density. Faults per line of deliverable executable source code (LOC).

Homogenous. Possessing characteristics that do not vary with time.

Imperfect Debugging. When the debugging process does not lead to the removal of the cause of the failure.

Least Square Estimation. A method of parameter estimation in which the parameters are selected to minimize the sum of squares of deviation of the estimated failure/fault data from the observed ones.

Maximum Likelihood Estimation. A form of parameter estimation in which the parameters are selected that maximize the possibility that the data that have been observed could have occurred.

Mean Value Function. A function that expresses the average value of the number of events experienced by a random process at each point in time.

Prediction. Determination of software reliability model parameters and quantities from characteristics of the software product and development process.

Reliability. Probability that a system or a capability of a system will continue to function without failure for a specified period in a specified environment. The period may be specified in natural or time units.

Test Occasions/Cases. A test case can be a single computer test run executed in an hour, day, week or even month. Therefore, it includes the computer test run and length of time spent to visually inspect the software source code. Whereas, a computer test run is a set of software input variables arranged in a certain manner to test the functional performance of a particular part of the software system.

Acronyms

SRGM	Software Reliability Growth Model	**SSE**	Sum of Squared Errors
NHPP	Non-Homogeneous Poisson Process	**AIC**	Akaike Information Criterion
MLE	Maximum Likelihood Estimate	**RPE**	Relative Prediction Error
PGF	Probability Generating Function	**DS**	Data Set
RMSPE	Root Mean Square Prediction Error	$\mathbf{R^2}$	R Squared

1.7 Structure of the Thesis

The following is a brief of the remaining Chapters:

Chapter 2 is divided into two Sections. Section 1 investigates the concepts and the description of NHPP based SRGMs. Section 2 reviews some of the well-documented and established NHPP based SRGMs.

Chapter 3 is divided into two Sections. Section 1 presents the proposed model that incorporates fault generation and imperfect debugging with learning-process. Section 2 presents a discrete version of the proposed model.

Chapter 4 is divided into three Sections. Section 1 presents the validation methods in terms of goodness-of-fit and predictive validity metrics, Section 2 derives some important software reliability evaluation measures, and Section 3 provides the parameters estimation technique.

Chapter 5 is divided into four Sections. All of these Sections show the applicability of the NHPP based SRGMs and the proposed model by applying them on software fault-detection-count data set cited from four real software development projects in terms of goodness of fit, predictive validity and software reliability evaluation measures. The results are very encouraging due to their applicability and flexibility as they can capture different reliability growth curves ranging from purely exponential to highly S-shaped.

We conclude the thesis in **Chapter 6**.

Chapter 2

Literature Review

The importance of modelling and analysis of software failure-occurrence or fault-removal phenomena has been well recognized and many studies have addressed this problem. An important objective of most of these investigations has been to develop analytical models for the fault removal phenomena in order to compute quantities of interest such as the number of faults removed, the number of remaining faults and the software reliability function. Such quantities are useful for planning purposes, in both the development and the operational phases of the software systems. In particular, software reliability models (SRMs) that describe software failure-occurrence or fault-removal phenomena in the system testing phase are called software reliability growth models (SRGMs). The models are useful in measuring reliability for the quality control and testing process control of software development. Many models have been proposed by many researchers. A few models have actually been applied to several software management tools which aid the software quality or reliability measurement and testing–progress control in the testing phase. Among others, Nonhomogeneous Poisson process (NHPP) models have been discussed in many applications because the models can be easily applied in actual software development. It forms one of the main classes of the existing SRGMs, due to its mathematical tractability and wide applicability. NHPP models are useful in describing failure processes, providing trends such as reliability growth and fault-content. SRGMs consider the debugging process as a counting process characterized by the value function of a NHPP. Software reliability can be estimated once the mean value function is determined. Model parameters are usually determined using either Maximum Likelihood Estimation or Least-square estimation methods.

The SRGMs are classified into two groups. The first group contains models, which use the machine execution (i.e., CPU) time or calendar time as a unit of fault detection/removal period. Such models are called continuous time models. The second group contains models, which use the number of test occasions/cases as a unit of fault detection period. Such models are called discrete time models, since the unit of software fault detection period is countable. A test case can be a single computer test run executed in an hour, day, week or even month. Therefore, it includes the computer test

run and length of time spent to visually inspect the software source code. A large number of models have been developed in the first group while there are fewer in the second group. Discrete time models in software reliability are important and a little effort has been made in this direction.

2.1 NHPP Models

Stochastic processes are used for the description of a system's operation over time. There are two main types of stochastic processes: continuous and discrete. Among discrete processes, counting processes in reliability engineering are widely used to describe the appearance of events in time (e.g., failures, number of perfect repairs, etc). The simplest counting process is a Poisson process. The Poisson process plays a special role to many applications in reliability engineering (Pham, 1999).

As a general class of well-developed stochastic process model in reliability engineering, NHPP models have been successfully used in studying hardware reliability problems. They are especially useful to describe failure processes which possess certain trends such as reliability growth and deterioration. Therefore, an application of NHPP models to software reliability analysis is then easily implemented.

The model provides the expected number of faults/failures at a given time. (Goel and Okumoto, 1979) proposed a exponential model based on the concept the expected number of faults removed per unit time is proportional to the current fault content and (Yamada and Osaki, 1985) proposed a discrete version of the model. (Yamada et al., 1983) proposed a delayed S-shaped model based on the concept of failure observation and the corresponding fault removal phenomenon and (Kapur et al., 1999) proposed a discrete version of the model. (Ohba, 1984) proposed the inflection S-shaped model. (Yamada et al., 1985), (Huang, et al., 2007), (Kapur et al., 2009) further proposed testing-effort dependent models which assumes the testing-effort to follow either exponential, Weibull, logistic, or Rayleigh distribution. (Kapur and Garg, 1990) modified exponential model by introducing the concept of imperfect debugging.

In the real life software development projects, the non-uniform testing is more popular and hence the S-shaped growth curve has been observed in many software development projects. The cause of S-shapedness has been attributed to different reasons. (Yamada et al., 1983) attributed it to time delay between the fault removal and the initial failure observation which is result of the unskilledness of the testing team at the early stages of

the test. Also, (Ohba, 1984) attributed it to the mutual dependency between software faults. (Yamada et al., 1985) ascribed it to the non-uniform distribution of the testing-effort. (Bittanti et al., 1988) accrued it to the increased fault detection rate later in the testing phase. (Kapur et al., 1992) ascribed it to the learning process of the test team.

Later, few SRGMs were developed taking into account causes of the S-shapedness (Kapur et al., 1999), (Pham, 2000), (Shatnawi and Kapur, 2008), (Shatnawi, 2009(a)), (Shatnawi, 2009(b)). Also some more exponential models were developed to cater for different situations during testing (Yamada et al., 1992), (Kapur et al., 1999), (Pham, 2000), (Zhang and Pham, 2000), (Kapur et al., 2008), (Shatnawi, 2009(b)). As a result we have a large number of SRGMs each being based on a particular set of assumptions that suits a specific testing environment.

All the mentioned SRGMs have been proposed for the testing phase and it is generally assumed that operational profile is similar to the testing phase, which may not be the case in practice. Very few attempts have been made to model the failure phenomenon of commercial software during its operational use. One of the reasons for this can be attributed to the inability of software engineers to measure the growth in usage of software while it is in the market. It is unlike the testing phase where testing-effort follows a definite pattern (Kenny, 1988), (Shatnawi, 2004).

The most important criterion in a model selection is the validity of the model assumptions and the relevance of these assumptions to the real testing environment. Besides, the performance of the model in terms of its ability to regenerate the past failure data and to predict the future of the failure observation process are two other important criteria. The model selection problem is a tedious task in the presence of a large number of SRGMs. To reduce the difficulty of model selection, flexible modelling approach has been used in this thesis. The ability of the model to fit different growth curves with enough variability reflects its flexibility and thus the flexible model is the one which can represent the fault removal process over a wide range of software testing environment.

2.1.1 A General Description of Continuous Time Model (Lyu, 1996), (Musa et al., 1987), (Musa, 2004), (Pham, 2000), (Pfleeger, 2006)

Let [$N(t)$; $t \geq 0$] denotes a discrete counting process representing the cumulative number of failures experienced (fault removed) up to time t, i.e., $N(t)$, is said to be an NHPP with intensity function $\lambda(t)$, if it satisfies the following conditions:

1. There are no failures experienced at time t=0, i.e., $N(t=0)=0$ with probability 1.
2. The process has independent increments, i.e., the number of failures experienced in (t, $t+\Delta t$], i.e., $N(t+\Delta t)-N(t)$, is independent of the history. Note this assumption implies the Markov property that the $N(t+\Delta t)$ of the process depends only on the present state $N(t)$ and is independent of it is past state $N(x)$, for $x<t$.
3. The probability that a failure will occur during (t,$t+\Delta t$] is $\lambda(t)\Delta t+o(\Delta t)$, i.e., $\Pr[N(t+\Delta t)-N(t)=1]=\lambda(t)+o(\Delta t)$. Note that the function $o(\Delta t)$ is defined as

$$\lim_{\Delta t \to 0} \frac{o(\Delta t)}{\Delta t} = 0 \tag{2.1}$$

   In practice, it implies that the second or higher order effects of Δt are negligible.
4. The probability that more than one failure will occur during (t,$t+\Delta t$] is $o(\Delta t)$, i.e., $\Pr[N(t+\Delta t)-N(t)>1]=o(\Delta t)$.

Based on these NHPP assumptions, it can be shown that the probability that $N(t)$ is a given integer k is expressed by

$$\Pr[N(t)=k] = \frac{[m(t)]^k}{k!}\exp\{-m(t)\}, \quad k \geq 0 \tag{2.2}$$

The function $m(t)$ is called the mean value function and describes the expected cumulative number of failures in (0,t]. Hence, $m(t)$ is a very useful descriptive measure of the failure behavior.

The function $\lambda(t)$ which called the instantaneous failure intensity is defined as

$$\lambda(t) = \lim_{\Delta t \to 0} \frac{p[N(t+\Delta t) - N(t) > 0]}{\Delta t} \tag{2.3}$$

Given $\lambda(t)$, the mean value function $m(t)=E[N(t)]$ satisfies

$$m(t) = \int_0^t \lambda(s)ds \tag{2.4}$$

Inversely, knowing $m(t)$, the failure intensity function $\lambda(t)$ can be obtained as

$$\lambda(t) = \frac{dm(t)}{dt} \tag{2.5}$$

Generally, by using different nondecreasing function $m(t)$, we get different NHPP models.

Define the number of remaining software failure at time t by $\overline{N}(t)$ and we have that

$$\overline{N}(t) = N(\infty) - N(t) \tag{2.6}$$

where $N(\infty)$ is the number of faults which can be detected by infinite time of testing.

It follows from the standard theory of NHPP that the distribution of $\overline{N}(t)$ is Poisson with parameter $[m(\infty)-m(t)]$, that is

$$\Pr\left[\overline{N}(t) = k\right] = \frac{\left[m(\infty) - m(t)\right]^k}{k!} \exp\left\{m(\infty) - m(t)\right\}, \quad k \geq 0 \tag{2.7}$$

The reliability function at time t_o is exponential given by

$$R(t \mid t_o) = \exp\left\{-\left(m(t + t_o) - m(t)\right)\right\} \tag{2.8}$$

The above conditional reliability function $R(t|t_o)$ is called a software reliability function based upon a NHPP for a continuous time model.

2.1.2 A General Description of Discrete Time Model (Kapur et al, 2006), (yamada and Osaki, 1995)

During the software testing phase a software system is executed with a sample of test cases to remove software faults, which cause software failures.

A discrete counting process $[N(n);n\geq 0]$ is said to be an NHPP with mean value function $m(n)$, if it satisfies the following conditions:

1. There are no failures experienced at $n=0$, i.e., $N(n=0)=0$.
2. The counting process has independent increments, that is, for any collection of the numbers of test cases $n_1,n_2,\ldots,n_k$ where $(0<n_1<n_2<\ldots<n_k)$, the k random variables $N(n_1),N(n_2)\text{-}N(n_1),\ldots,N(n_k)\text{-}N(n_{k-1})$ are statistically independent.

For any of numbers of test cases n_i and n_j where $(0\leq n_i\leq n_j)$, we have

$$\Pr\left[N(n_i) - N(n_j) = x\right] = \frac{\left[m(n_i) - m(n_j)\right]^x}{x!} \exp\left\{-\left[m(n_i) - m(n_j)\right]\right\}, \quad x \geq 0 \tag{2.9}$$

The mean value function $m(n)$ which bounded above and is non-decreasing in n represents the expected cumulative number of faults detected by n test cases. Then the NHPP model with $m(n)$ is

$$\Pr\left[N(n) = x\right] = \frac{\left[m(n)\right]^x}{x!} \exp\left\{-m(n)\right\}, \quad x \geq 0 \tag{2.10}$$

As a useful software reliability growth index, the fault detection rate per fault (per test case) after the n^{th} test case is given by

$$q(n) = \frac{[m(n+1) - m(n)]}{[m(\infty) - m(n)]}, \quad n \geq 0 \tag{2.11}$$

where $m(\infty)$ represents the expected number of faults to be eventually detected.

Let $\overline{N}(n)$ denotes the number of faults remaining in the system after the n^{th} test case is given as

$$\overline{N}(n) = N(\infty) - N(n) \tag{2.12}$$

The expected value of $\overline{N}(n)$ is given by:

$$h(n) = m(\infty) - m(n) \tag{2.13}$$

which is equivalent to the variance of $\overline{N}(n)$. Suppose that n_d faults have been detected by n test cases. The conditional distribution of $\overline{N}(n)$, given that $N(n)=n_d$, is given by

$$\Pr\{\overline{N}(n) = y \mid N(n) = n_d\} = \frac{\{E(n)\}^y}{y!} \exp[-\{E(n)\}] \tag{2.14}$$

which means a Poisson distribution with mean $E(n)$, independent of n_d.

Now, the probability of no faults detected between the n^{th} and the $(n+h)^{th}$ test cases, given that n_d faults have been detected by n test cases, is given by:

$$R(h \mid n) = \exp[-\{m(n+h) - m(n)\}], \quad n, h \geq 0 \tag{2.15}$$

The above conditional reliability function $R(h|n)$is called a software reliability function based upon a NHPP for a discrete time model and is independent of n_d.

2.1.3 Comments on Using NHPP

Among the existing models, NHPP models have been widely applied by practitioners. The application of NHPP to reliability analysis can be found in elementary literature on reliability. The calculation of the expected number of failures/faults up to a certain point in time is very simple due to the existence of mean value function. The estimates of the parameters are easily obtained by using either the method of MLE or of least squares.

Other important advantages of NHPP models which should be stressed here are that NHPP's are closed under superposition and time transformation. We can easily incorporate two or more existing NHPP models by summing up the corresponding mean

value functions. The failure intensity of the superposed process is also just the sum of the failure intensity of the underlying processes.

It should be noted here that NHPP models are capable of coping with the case of non-homogenous testing and hence it is useful for a calendar time data as well as for the execution time data (Xie, 1990).

2.2 Some NHPP based SRGMs

A very large number of continuous time models has been developed in the literature to monitor the fault removal process and measure and predict the reliability of the software systems. During testing phase it has been observed that the relationship between the testing time and the corresponding number of faults removed is either Exponential or S-shaped or the mix of two. The following are some of the well-established models: models due to (Goel and Okumoto, 1979), (Yamada et al., 1983), (Ohba, 1984), (Kapur and Garg, 1990), (Yamada et al., 1992), and (Pham et al., 1999).

Several SRGMs have been developed in the literature to monitor the fault removal process and measure and predict the reliability of the software systems (Xie, 1990), (Kapur et al., 1999), (Musa et al., 1987), (Pham, 1999). During testing phase it has been observed that the relationship between the testing time and the corresponding number of faults removed is either Exponential or S-shaped or the mix of two. The following are some of the SRGMs of interest, which exhibit such behavior

1. Model due to (Goel and Okumoto, 1979) (purely exponential in nature)
2. Model due to (Yamada et al., 1983) (purely S-shaped in nature)
3. Models due to (Ohba, 1983) (flexible in nature)
4. Model due to (Kapur and Garg, 1990) (purely exponential in nature)
5. Model due to (Yamada et al., 1992) (purely exponential in nature)
6. Model due to (Pham et al., 1999) (flexible in nature)

Intensity functions for 3 and 6 have two points of inflection, which can be implicitly attributed to faults of different severity. Severity is determined by the time of its detection/removal. However it should be noted that the simple faults most of which are detected in the earlier stage of testing continue to reside in software till the end of testing. Therefore when we categorize faults, except simple faults, all other faults are relatively difficult, relatively hard and complex faults. Though many criteria can be defined for categorizing faults, the ability of test cases to force the fault detection has

been chosen for the purpose in this section. Under the assumption that testing is done uniformly, simple hard and complex faults manifest themselves at any time during testing but are generally concentrated at distinct time intervals.

Some of the general assumptions (apart from some special ones for specific models discussed) assumed in the models discussed in this section are as follows:

1. The fault detection / removal phenomenon is modelled by NHPP.
2. Software is subject to failure during execution caused by faults remaining in the software.
3. Failure rate of the software is equally affected by faults remaining in the software.
4. The number of faults detected at any time is proportional to the remaining number of faults in the software.

The following are models notations

a, b	Initial fault-content and rate of fault removal per remaining respectively, $a > 0$, $0 < b < 1$.
$m(t)$	Expected number of faults removed in $(0,t]$, i.e., the mean value function of NHPP.
r	Ratio of independent faults to the total number of faults in the software, $0 \leq r \leq 1$.
k_i ,k_f	Initial and final values of Fault Exposure Coefficient (FEC) respectively, $0 < k_i < 1$, $0 \leq k_f < 1$.
u, v	Rate at which failures are occurring and rate of additional fault removals respectively, $0 < u < 1$, $0 \leq v < 1$.
p	Probability of fault removal on a failure, $0 < p \leq 1$.
α	Fault introduction rate per removed faults per unit time, $\alpha \geq 0$.
β	Constant parameter in the logistic learning-process function, $\beta \geq 0$.

2.2.1 Exponential Model, (Goel and Okumoto, 1979)

Following differential equation results from assumption 4, we may write

$$\frac{d}{dt}m(t) = b\left(a - m(t)\right) \tag{2.16}$$

Solving the differential Equation (2.16) with the initial condition $m(t=0)=0$ gives

$$m(t) = a\left(1 - \exp(-bt)\right) \tag{2.17}$$

The above mean value function given in Equation (2.17) is exponential in nature and does not provide a good fit to the S-shaped growth curves that generally occur in Software Reliability. But the model is popular due to its simplicity. Next, we briefly discuss below some S-shaped SRGMs.

2.2.2 Delayed S-shaped Model, (Yamada et al., 1983)

Fault detection/removal process in this model is assumed to be a two-phase process consisting of failure occurrence and it is eventual removal by isolation. It takes into account the time taken to isolate and remove a fault and so it is important that the data to be used here should be that of fault isolation. It is further assumed that the number of faults isolated at any time is proportional to the current number of faults not isolated. Failure occurrence rate and fault isolation rate per fault are assumed to be same and equal to b. Thus

$$\begin{aligned} \frac{d}{dt} m_f(t) &= b\left(a - m_f(t)\right) \\ \frac{d}{dt} m(t) &= b\left(m_f(t) - m(t)\right) \end{aligned} \tag{2.18}$$

Solving the above system of equations (2.18) with initial conditions $m_f(t=0)=0$ and $m(t=0)=0$,

$$m(t) = a\left(1 - (1+bt)\exp(-bt)\right) \tag{2.19}$$

Alternately the model can also be formulated as one stage process directly as follows

$$\frac{d}{dt} m(t) = b(t)\left(a - m(t)\right), \quad \text{where } b(t) = \frac{b^2 t}{1+bt} \tag{2.20}$$

It is observed that $b(t) \to b$ as $t \to \infty$. This model was specifically developed to account for lag in the failure observation and its subsequent removal. This kind of derivation is peculiar to software reliability only.

2.2.3 Inflection S-shaped Models

Inflection Shaped Model, (Ohba, 1984)

The model attributes S-shapedness to the mutual dependency between software faults. Other than assumption 3 it is also assumed that the software contains two types of faults, namely mutually dependent and mutually independent. The mutually

independent faults are those located on different execution paths of the software, therefore they are equally likely to be detected and removed. The mutually dependent faults are those faults located on the same execution path. According to the order of the software execution, some faults in the execution path will not be removed until their preceding faults are removed.

The ratio r is called the inflection parameter ($0<r\leq 1$). If all faults in the software are mutually independent ($r=1$) then the faults are randomly removed and the growth curve is exponential.

According to the assumptions of the model, the fault removal intensity can be written as

$$\frac{d}{dt}m(t) = b(t)\big(a - m(t)\big) \tag{2.21}$$

where $b(t)$ represents the fault removal rate at time t and is defined as

$$b(t) = b\phi(t) \tag{2.22}$$

where b represents the fault removal rate in the steady state and $\varphi(t)$ represents the inflection function and is defined as

$$\varphi(t) = r + (1-r)\frac{m(t)}{a} \tag{2.23}$$

where $\varphi(t=0)=r$ and $\varphi(t=\infty)=1$

Solving Equation (2.21) under the initial condition $m(t=0)=0$ we get

$$m(t) = a\frac{1-\exp(-bt)}{1+\frac{1-r}{r}\exp(-bt)} \tag{2.24}$$

If $r=1$, the model reduces to the G-O model. For different values of r different growth curves can be obtained and in that sense it is flexible.

Alternately the model can also be formulated as one stage process directly as follows

$$\frac{d}{dt}m(t) = b(t)\big(a - m(t)\big), \quad \text{where } b(t) = \frac{b}{1+\frac{1-r}{r}\exp(-bt)} = \frac{b}{1+\beta\exp(-bt)} \tag{2.25}$$

where $\beta = \dfrac{1-r}{r}$

It is observed that $b(t)\rightarrow b$ as $t\rightarrow\infty$.

Inflection Shaped Model, (Bittanti et al., 1988)

The fault removal rates are different during the early and late stages of software testing depending upon the nature of faults contained in the software. The rate may decrease

sharply during testing due to reduction in latent faults. On the contrary it can also happen that the removal of faults increases the skill of the testing team leading to more efficient testing and higher failure reports and fault removals (often observed when testing has been done for certain duration). They exploited this change in fault removal rate and termed as the Fault Exposure Coefficient (FEC) for their SRGM.

The FEC is given as a function of faults removed as follows

$$K(t) = k_i + (k_f - k_i)\frac{m(t)}{a} \tag{2.26}$$

where $K(t=0)=k_i$ and $K(t=\infty)=k_f$

According to the values of k_i and k_f one can distinguish between the following cases:

1. Constant FEC: $k_i = k_f$
2. Increasing FEC: $k_i < k_f$
3. Decreasing FEC: $k_i > k_f$
4. Vanishing FEC: $k_f = 0$, $k_i > 0$

The fault removal intensity is given as

$$\frac{d}{dt}m(t) = K(t)[a - m(t)] \tag{2.27}$$

Solving Equation (2.27) with the usual initial condition we get

$$m(t) = a\frac{1-\exp(-k_f t)}{1+\frac{k_f - k_i}{k_i}\exp(-k_f t)} \tag{2.28}$$

which is similar to Equation (2.24). Again, the structure of the model is flexible. The shape of the growth curve is determined by the parameters k_i and k_f and can be both exponential and S-shaped for the four cases discussed above.

Alternately the model can also be formulated as one stage process directly as follows

$$\frac{d}{dt}m(t) = b(t)\big(a - m(t)\big), \quad \text{where } b(t) = \frac{k_f}{1+\frac{k_f - k_i}{k_i}\exp(-k_f t)} = \frac{b}{1+\beta\exp(-bt)} \tag{2.29}$$

where $\beta = \dfrac{k_f - k_i}{k_i}$

It is observed that $b(t) \to b$ as $t \to \infty$.

Inflection Shaped Model, (Kapur and Garg, 1992)

This model is based upon the following additional assumption: On a failure observation, the fault removal process also removes portion of the remaining faults, without their causing any failures. Based on the assumption the fault removal intensity can be written as

$$\frac{d}{dt}m(t) = u[a - m(t)] + v\frac{m(t)}{a}[a - m(t)] \tag{2.30}$$

Solving Equation (2.30) with the usual initial condition, we have

$$m(t) = a\frac{1-\exp(-(u+v)t)}{1+\frac{v}{u}\exp(-(u+v)t)} \tag{2.31}$$

which is similar to Equations (2.24) and (2.28), though they have been derived under different assumptions. Curves for $m(t)$ can be exponential or S-shaped depending upon the values of u and v.

Alternately the model can also be formulated as one stage process directly as follows

$$\frac{d}{dt}m(t) = b(t)(a - m(t)), \quad \text{where } b(t) = \frac{(p+q)}{1+\frac{q}{p}\exp(-(p+q)t)} = \frac{b}{1+\beta\exp(-bt)} \tag{2.32}$$

It is observed that $b(t) \to b$ as $t \to \infty$.

2.2.4 Imperfect debugging model, (Kapur and Garg, 1990)

In all the preceding models it has been assumed that on a failure, the error causing the failure is removed with certainty. In reality this may always not be true. Let p the probability of effort removal on a failure. Then

$$\frac{d}{dt}m(t) = bp(a - m(t)) \tag{2.33}$$

Solving the differential Equation (2.33) with the initial condition $m(t=0)=0$ gives

$$m(t) = a(1 - \exp(-bpt)) \tag{2.34}$$

when $p=1$, the model reduces to the G-O model.

2.2.5 Fault Generation Model, (Yamada et al., 1992)

In general it is considered to be unrealistic in software reliability modelling to assume that the faults detected by software testing are perfectly removed without introducing new faults. Software reliability assessment models with imperfect debugging by

assuming that new faults are sometimes introduced when the faults originally latent in a software system are corrected and removed during the testing phase is presented below. It is assumed that the fault detection rate is proportional to the sum of the numbers of faults remaining originally in the system and faults introduced by imperfect debugging

$$\frac{d}{dt}m(t) = b\big(a(t) - m(t)\big) \tag{2.35}$$

where

$$a(t) = a(1+\alpha t)$$

Solving the differential Eq. (2.35) under the boundary condition $m(t=0)=0$, we get

$$m(t) = a\left[\left(1-e^{-bt}\right)\left(1-\frac{\alpha}{b}\right)+\alpha t\right] \tag{2.36}$$

when $\alpha=0$, the model reduces to the Equation (2.17).

2.2.6 PNZ Model, (Pham et al., 1999)

A general software reliability model is used to derive a model that integrates imperfect debugging with the learning phenomenon. Learning occurs if testing appears to improve dynamically in efficiency as one progresses through a testing phase. Learning usually manifests itself as a changing fault-detection rate. Published models and empirical data suggest that efficiency growth due to learning can follow many growth-curves, from linear to that described by the logistic function**.**

The expected cumulative number of faults removed in the time interval $(t, t+\Delta t)$ is proportional to the sum of the numbers of faults remaining originally in the system and faults introduced by imperfect debugging; satisfies the following differential equation:

$$\frac{d}{dt}m(t) = b(t)\big(a(t) - m(t)\big) \tag{2.37}$$

where

$$b(t) = \frac{b}{1+\beta\exp(-bt)}$$
$$a(t) = a(1+\alpha t)$$

Both $a(t)$ and $b(t)$ are time-dependent functions. An increasing $a(t)$ implies an increasing total number of faults, and thus reflects fault generation. Whereas, $b(t)$ is an S-shaped curve that can capture the learning process of the software testers.

Solving the differential Eq. (2.37) under the boundary condition $m(t=0)=0$, we get

$$m(t) = \frac{a}{1+\beta e^{-bt}}\left[\left(1-e^{-bt}\right)\left(1-\frac{\alpha}{b}\right)+\alpha t\right] \quad (2.38)$$

According to the values of parameters of the PNZ model given in Equation (2.38), we can distinguish between the following cases:

1) When the test skills of the test-team does not improve during testing (i.e., β=0). In this case, the PNZ model reduces to fault generation model (Yamada et al., 1992) given in Equation (2.36).
2) When the test skills of the test-team does not improve during testing (i.e., β=0), and the no faults introduced during debugging (i.e, α=0). In this case, the PNZ model reduces to exponential model (Goel and Okumoto, 1979) given in Equation (2.17).

Chapter 3

Proposed Model

In general, among various SRGMs, two most important factors affect reliability: the number of initial faults and the fault removal rate. The number of initial faults is the number of faults in the software at the beginning of the test. This number is usually a representative measure of software reliability. Knowing the number of residual faults can help to determine whether the software is suitable for customers to use or not, and how much more testing resources are required. It can provide an estimate of the number of failures that will eventually be encountered by the customers. The fault removal rate, on the other hand, is used to measure the effectiveness of fault removal by test techniques and test cases. In the literature (Goel and Okumoto, 1979), (Yamada et al., 1983), (Lyu, 1996), (Musa et al., 1987), (Kapur el al., 1999), most researchers assume a constant fault removal rate per fault in deriving their SRGM. That is, they assume that all faults have equal probability of being removed during the software testing process, and the rate remains constant. In reality, the fault removal rate strongly depends on the skill of test teams, program size and software testability.

Through real data experiments and analyzes on several software development projects (Bittanti et al., 1988), (Pham et al., 1999), (Kuo et al., 2001), (Kapur et al, 2006), (Shatnawi, 2009(a)), (Shatnawi, 2009(b)), it has been observed the fault removal rate has three possible trends as time progresses: increasing, decreasing or constants. It decreases when the software has been used and tested repeatedly, showing reliability growth. It can also increase if the testing techniques/requirements are changed, or new faults are introduced due to new software features or imperfect debugging.

The learning-process of software developers has also been studied (Yamada et al., 1983), (Ohba, 1984), (Bittanti et al., 1988), (Kapur and Garg, 1992), (Pham, 2000) (Kapur et al, 2006), (Shatnawi, 2009(a)), (Shatnawi, 2009(b)). The learning is closely related to the changes in the efficiency of testing during a testing phase. The idea is that in organizations that have advanced software-process, testers might be allowed to improve dynamically their testing process as they learn more about the product. This could result in a fault removal rate which increases monotonically over the testing period. However, there are some pitfalls too. What all researchers appear to agree upon

is that in practice there often is an apparent growth in fault removal ability as testing progresses (Pham et al., 1999).

3.1 Model Development

The PNZ model is revisited and some research directions are further discussed. Based on data analyses and model comparisons the authors of the PNZ model claimed that this is the best descriptive and predictive model (Pham et al., 1999), (Pham, 2000).

However, the authors of the PNZ model assumed that on a failure, the fault causing the failure is removed with certainty. In reality this may always not be true. In other words, during debugging process, the testing team may not be able to remove the fault perfectly on the detection of the failure. This phenomenon is known as imperfect debugging.

The objective is to extend the scope of the PNZ model to address this issue. Thus, making it depicts the real-life situation more realistically.

Model Assumptions

1. Fault removal phenomenon is modelled by NHPP.
2. Software is subject to failures during execution caused by faults remaining in the software.
3. Overall fault-content is linearly time-dependent, which includes initial fault-content and the number of faults introduced.
4. Fault removal rate is a S-shaped curve that can capture the learning-process of the software testers, and this function is affected by the probability of perfect debugging.
5. Faults can be introduced during the debugging process, i.e., fault generation.
6. Debugging process may not lead to the complete removal of the faults, i.e., the debugging process is imperfect**.**

Model Formulation

The expected cumulative number of faults removed in the time interval (t, $t+\Delta t$) is proportional to the sum of the numbers of faults remaining originally in the system and faults introduced by imperfect debugging; satisfies the following differential equation:

$$\frac{d}{dt}m(t) = b(t)\big(a(t) - m(t)\big) \tag{3.1}$$

where

$$b(t) = \frac{bp}{1 + \beta \exp(-bpt)}$$
$$a(t) = a(1 + \alpha t)$$

Both $a(t)$ and $b(t)$ are time-dependent functions. An increasing $a(t)$ implies an increasing total number of faults, and thus reflects fault generation. Whereas, $b(t)$ is an S-shaped curve that can capture the learning process of the software testers, and this function is affected by the probability of fault removal on a failure.

Solving the differential Eq. (3.1) under the boundary condition $m(t=0)=0$, we get

$$m(t) = \frac{a}{1 + \beta e^{-bpt}} \left[\left(1 - e^{-bpt}\right) \left(1 - \frac{\alpha}{bp}\right) + \alpha t \right] \tag{3.2}$$

This proposed model given above in Eq. (3.2) is very interesting from various points of view. Besides its interpretation as a general flexible S-shaped fault removal model, this model has the models (Goel and Okumoto, 1979), (Ohba, 1984), (Bittanti et al., 1988), (Kapur and Garg, 1990), (Yamada et al., 1992), (Pham et al., 1999) as special cases. Thus, it is able to model both cases of strictly decreasing failure intensity and the case of increasing-and-decreasing failure intensity. Neither the exponential model nor the ordinary delayed S-shaped model can do both.

According to the values of parameters of the proposed model given in Equation (3.2), we can distinguish between the following cases:

1) When the debugging process is perfect (i.e., p=1). In this case, the proposed model reduces to PNZ model (Pham et al., 1999) given in Equation (2.38).
2) When the test skills of the test-team does not improve during testing (i.e., β=0) and the debugging process is perfect (i.e., p=1). In this case, the proposed model reduces to fault generation model (Yamada et al., 1992) given in Equation (2.36).
3) When the test skills of the test-team does not improve during testing (i.e., β=0), and the no faults introduced during debugging (i.e, α=0). In this case, the proposed model reduces to imperfect debugging model (Kapur and Garg, 1990) given in Equation (2.34).
4) When the test skills of the test-team does not improve during testing (i.e., β=0), the debugging process is perfect (i.e., p=1), and the no faults introduced during debugging (i.e, α=0). In this case, the proposed model reduces to exponential model (Goel and Okumoto, 1979) given in Equation (2.17).

3.2 Discrete Version of the Proposed Model

The utility of discrete time NHPP based SRGMs cannot be undermined. As the software reliability data are discrete, these models many times provide better fit than their continuous time counterparts. Hence in spite of difficulties in terms of mathematical complexity, discrete models are proposed regularly.

The assumptions, which are with respect to time in the continuous case, can be reinterpreted in terms of number of test cases. The test case can be any duration of time viz. hour, day, week, month etc. The expected cumulative number of faults removed between the n^{th} and $(n+1)^{th}$ test cases is proportional to the number of faults remaining after the execution n^{th} test run, satisfies the following difference equation:

$$m(n+1)-m(n)=b(n+1)\big(a(n)-m(n)\big) \tag{3.3}$$

where

$$b(n+1)=\frac{bp}{1+\beta(1-bp)^{n+1}}$$

$$a(n)=a(1+\alpha n)$$

Solving, using the method of probability generation function (PGF) with initial condition $m(n=0)=0$, after tedious algebraic manipulations, one can get the closed form solution as given below:

$$m(n)=\frac{a}{1+\beta(1-bp)^{n}}\left[\left(1-(1-bp)^{n}\right)\left(1-\frac{\alpha}{bp}\right)+\alpha n\right] \tag{3.4}$$

This discrete version of the proposed model given above in Eq. (3.4) is very interesting from various points of view. Besides its interpretation as a general flexible S-shaped fault removal model, this model has the models (Yamada and Osaki, 1979), (Ohba, 1984), (Kapur et al., 1999), (Kapur et al., 2006), (Kapur et al., 2008), (Shatnawi, 2009(a)), as special cases. Thus, it is able to model both cases of strictly decreasing failure intensity and the case of increasing-and-decreasing failure intensity. Neither the exponential model nor the ordinary delayed S-shaped model can do both.

Chapter 4

Model Validation and Comparison Criteria

To check the validity of the models under comparisons including the proposed model given in Equation (3.2) to describe the software reliability growth, it has been tested on four actual software reliability datasets (DS) collected from real software development projects. The first datasets (DS-I) was cited from (Brooks and Motley, 1980), the fault data set is for a radar system of size 124 KLOC tested for 35 months in which 1301 faults were removed. The second datasets (DS-II), the software was tested for 38 weeks during which 2456.4 computer hours where used and 231 faults where removed, (Pham, 2000). The third datasets (DS-III), software for monitoring real-time control system consist of about 200 modules having on average 1,000 lines of a high level language such as FORTRAN. Since the test data are recorded daily, the test operation performed in a day are regarded to be a test instance, the data was collected during 111 days of testing, 481 faults were detected (Pham, 2000). The fourth datasets (DS-IV) was collected during 15 month of testing, 1138 faults were detected (Brooks and Motley, 1980). The datasets were deliberately chosen from different testing environments where the growth curves range from exponential to highly S-shaped.

4.1 Model Validation

The performance of an SRGM is judged by its ability to fit the past software failure occurrence / fault detection data (goodness of fit) and to predict satisfactorily the future behavior of the software failure occurrence / fault detection process from present and past failure occurrence / fault detection data (predictive validity) (Musa et al., 1987), (Kapur et al., 1999).

Other than these metrics used in comparing SRGMs. (Musa et al., 1987) have suggested the following attributes for choosing an SRGM:

- **Capability.** The model should posses the ability to estimate with satisfactory accuracy metrics needed by the software managers.
- **Quality of Assumptions.** The model assumptions should be plausible and must depict the testing environment.

- **Applicability.** A model can be judged as the better one if it can be applied across software products of different sizes, structures, platforms and functionalities.
- **Simplicity.** The data required for an ideal SRGM should be simple and inexpensive to collect. The parameters estimation should not be too complex and is easy to understand and apply even for persons without extensive mathematical background.

4.2 Comparison Criteria

The Goodness of Fit Criteria

- The Sum of Squared Error (SSE)**.** SSE measures the distance of a model estimate value from the actual data, as follows

$$SSE = \sum_{i=1}^{k} \left(\hat{m}(t_i) - x_i\right)^2 \tag{4.1}$$

where k is the number of observations, $\hat{m}(t_i)$ is the estimated cumulative number of faults by time t_i obtained from the fitted mean value function and x_i is the total number of faults removed by time t_i. Lower value of SSE indicates less fitting error, thus better goodness of fit.

- The Akaike Information Criterion (AIC)**.** This criterion was first proposed as SRGM model selection tool by (Khoshogoftaar & Woodcock, 1991) and defined as

$$AIC = -2 \times \log(Max.\ \ of\ \ \ Likelihood\ \ \ function) + 2 \times N \tag{4.2}$$

where N is the number of the parameters used in the model. Lower value of AIC indicates more confidence in the model thus a better fit and predictive validity.

- Root Mean Square Prediction Error (RMSPE):

$$RMSPE = \sqrt{\left(Bias^2 + Variation^2\right)} \tag{4.3}$$

where *Bias* is the difference between the observation and prediction of number of failures at any instant of time *i* is known as PEi.(prediction error). The average of PEs is known as bias. The standard deviation of prediction error is known as *variation*.

- Coefficient of Multiple Determination (R^2)**.** This Goodness-of-fit measure can be used to investigate whether a significant trend exists in the observed failure intensity. We define this coefficient as the ratio of the Sum of Squares (SS) resulting from the trend model to that from a constant model subtracted from 1, that is

$$R^2 = 1 - \frac{residual \quad SS}{corrected \quad SS} \tag{4.4}$$

R^2 measures the percentage of the total variation about the mean accounted for by the fitted curve. It ranges in value from 0 to 1. Small values indicate that the model does not fit the data well. The larger, the better the model explains the variation in the data.

In other words, we evaluate the performance of the models under comparison using SSE, AIC, RMSPE, and R^2 metrics. For SSE, AIC and RMSPE, the smaller the metric value the better the model fits relative to other models run on the same dataset (DS). For R^2, the larger the metric value the better.

The Predictive validity Criterion

Predictive validity is defined as the capability of the SRGM to determine the future fault/failure behavior from present and past fault/failure behavior (i.e., data). This capability is significant only when failure behavior is changing. This criterion was proposed by (Musa et al., 1987). Assume that we have observed x_k failures by the end of test time t_k. We use the failure data up to time $t_e(\leq t_k)$ to estimate the parameters of $m(t)$. Substituting the estimates of the parameters in the mean value function yields the estimate of the number of failures $\hat{m}(t_k)$ by t_k. The estimate is compared with the actually observed number x_k. This procedure is repeated for various values of t_e. We can visually check the predictive validity by plotting the Relative Prediction Error (RPE) ratio $\left((\hat{m}(t_k) - x_k)/x_k\right)$ against the normalized test time (t_e/t_k) (i.e., testing progress ratio). The RPE ratio will approach zero as t_e approaches t_k. If the RPE value is negative/positive the model is said to underestimate/overestimate the future failure phenomenon. A value close to zero for RPE indicates more accurate prediction, thus more confidence in the model and better predictive validity. The value of RPE is said to be acceptable if it is within ±10 percent (Kapur et al., 1999).

4.3 Software Reliability Evaluation Measures

Let $[N(t); t \geq 0]$ denotes a discrete counting process representing the cumulative number of failures experienced or fault removed up to time t, then it can normally be modeled as an NHPP with the superposed mean value function $m(t)$. The NHPP model with $m(t)$ is formulated by

$$\Pr\{N(t) = x_f\} = \frac{[m(t)]^{x_f}}{x_f!} \exp[-m(t)], x_f \geq 0 \tag{4.5}$$

Based on the NHPP model with $m(t)$, the following quantitative reliability measures can be derived.

Expected Number of Remaining Faults

Let W(t) denotes the number of faults remaining in the software at time t, then we have

$$W(t) = N(\infty) - N(t) \tag{4.6}$$

The expected value of $W(t)$ is given by

$$E\{W(t)\} = m(\infty) - m(t) = H(t) \tag{4.7}$$

which is equivalent to the variance of $W(t)$, where $m(\infty)$ represents the expected number of faults to be eventually removed.

Software Reliability

The probability of no failures occurred (i.e., no faults removed) in the interval time $(t, t+t_o]$ where t_0 is the mission time, given that x_o failure (fault) has occurred (removed) by time t, is given by

$$R(t_o \mid t) = \exp\{m(t) - m(t + t_o)\}, t_o \geq 0 \tag{4.8}$$

which means a reliability function in time t, independent of x_o. This is called a conditional reliability function.

4.4 Parameter Estimation Techniques

Parameter estimation is of primary importance in software reliability estimation and prediction. Once the analytical solution $m(t)$ is known for a given model, the parameters in the solution need to be determined. Parameter estimation is achieved by applying either the estimation method of Least Squares or the estimation method of Maximum Likelihood. The Maximum Likelihood Estimation (MLE) method is the most important and widely used estimation technique.

Therefore, we adopted the MLE method to estimate the unknown parameters (a,p,b,α,β) of the models under comparison. Since all the data sets used are given in the form of pairs (t_i,x_i)(i=1,2,...,k), where x_i is the cumulative number of faults removed by time t_i ($0<t_1<t_2<\ldots<t_k$) and t_i is the accumulated time spent to remove x_i faults.

The Likelihood function L for the unknown parameters with the mean value function $m(t)$ is given as

$$L(a,p,b,\alpha,\beta\,|\,(t_i,x_i))=\prod_{i=1}^{k}\frac{[m(t_i)-m(t_{i-1})]^{x_i-x_{i-1}}}{(x_i-x_{i-1})!}\exp\left(-\left(m(t_i)-m(t_{i-1})\right)\right) \quad (4.9)$$

Taking natural logarithm of equation (16) we get

$$\ln\ L=\sum_{i=1}^{k}(x_i-x_{i-1})\ln[m(t_i)-m(t_{i-1})]-\{m(t_i)-m(t_{i-1})\}-\sum_{i=1}^{k}\ln[(x_i-x_{i-1})!] \quad (4.10)$$

The MLE of the SRGM parameters can be obtained to by maximizing L in Equations (4.9) or (4.10).

For faster and accurate calculations, the statistical package for social sciences (SPSS) has been utilized for the purpose. To estimate the parameters:

1- We opening the SPSS, in the first column we write the time and in the second column we write the cumulative number of faults.

2 - Click analyze, regression, and then chose nonlinear.

3 – In the Nonlinear Regression window, we chose the dependent variable the cumulative number of faults, we write the model in the Model Expression box, and we click parameters to write the name and corresponding starting value to each parameters and then click add

4 – Finally we click continue, then ok, and in the output window we find the parameters values under the label parameter estimate.

Chapter 5

Data Analyses and Model Comparisons

5.1 First Software Development Project

The results of the parameter estimation and the goodness-of-fit metrics in terms of *SSE*, *AIC*, *RMSPE*, and R^2 of the models under comparison are given in Table 5.1.

According to the estimated values of (α) the debugging process is perfect and no fault introduced. It is clearly seen that the proposed model is the best among the models under comparison in terms of *SSE*, *AIC*, *RMSPE* and R^2.

Table 5.1: Parameters Estimations (for DS-I)

Models under Comparison	Parameters Estimation					Comparison Criteria			
	a	*b*	*p*	α	β	SSE	AIC	RMSPE	R^2
Exponential (Goel & Okumoto, 1979)	*	*	—	—	—	*	*	*	*
Delayed S-shaped (Yamada et al., 1983)	1689.4	.090	—	—	—	95014.89	504.62	52.85	.987
Imperfect Debugging (Kapur & Garg, 1992)	*	*	*	—	—	*	*	*	*
Fault Generation (Yamada et al., 1992)	*	*	—	*	—	*	*	*	*
Inflection S-shaped (Ohba, 1984)	1331.5	.201	—	—	20.18	7212.75	338.10	14.56	.999
PNZ (Pham et al., 1999)	1327.0	.204	—	0	21.4°	7421.15	338.55	14.76	.999
Proposed	1331.1	.208	.966	0	20.16	7181.33	338.09	14.53	.999

* indicates the model fails to give any plausible result

– indicates the parameter is not part of the corresponding model

The fitting of the proposed model to DS-I is graphically illustrated in figure 5.1.1 given below. It is clearly seen that the proposed model fits DS-I excellently.

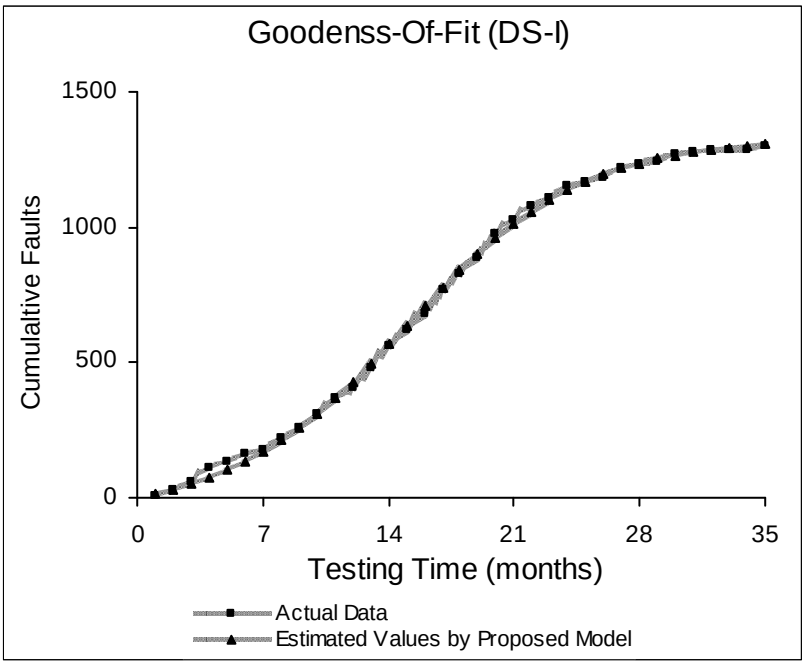

Figure 5.1.1. Faults identification curve for (DS-I)

DS-I are truncated into different proportions and used to estimate the parameters of the proposed model. For each truncation, one value of *RPE* ratio is obtained and is graphically illustrated in figure 5.1.2 given below. It is clearly seen that 60% of the total test time is sufficient to predict the future of the fault removal process reasonably well.

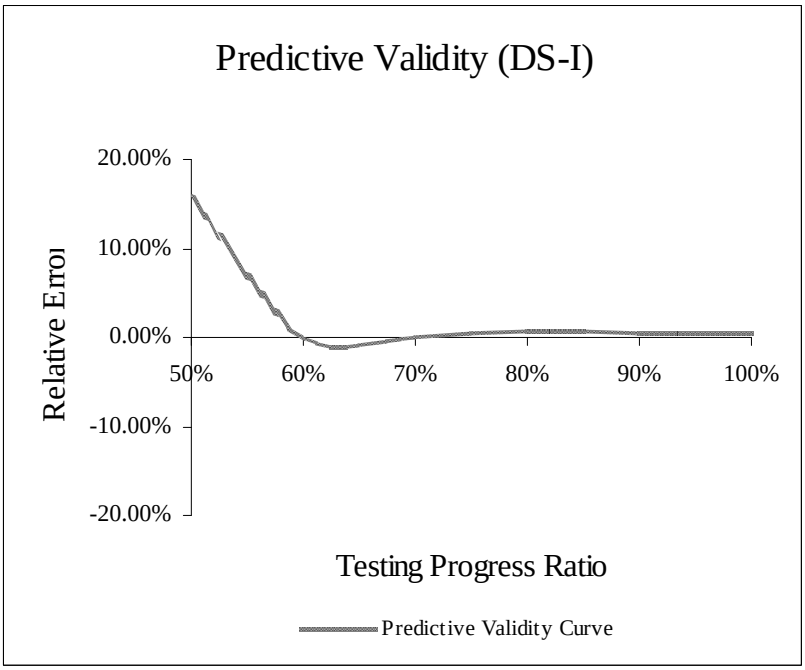

Figure 5.1.2 Predictive validity curve for (DS-I)

The fitting of the proposed model to actual remaining cumulative number of faults for DS-I is graphically illustrated in figure 5.1.3. It is clearly seen that the proposed model fits the actual data well.

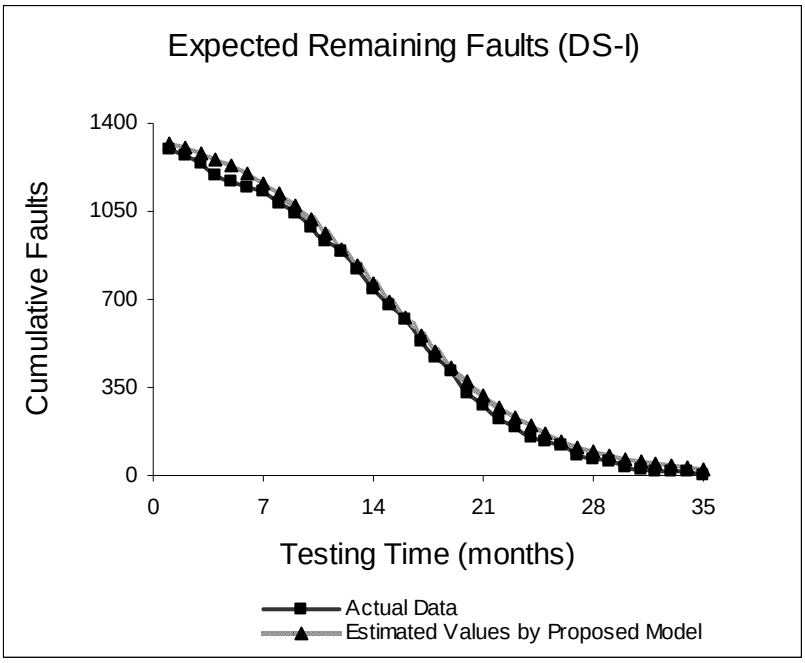

Figure 5.1.3. Remaining faults curve for (DS-I)

Figure 5.1.4 illustrate the software reliability growth for DS-I. It is observed that the reliability is improving during testing.

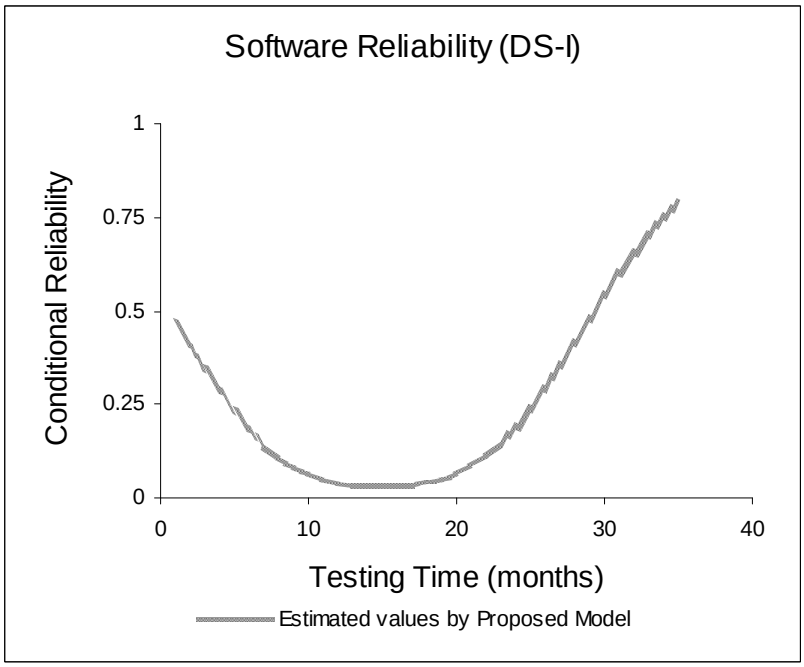

Figure 5.1.4. Reliability curve for (DS-I)

5.2 Second Software Development Project

The results of the parameter estimation and the goodness-of-fit metrics in terms of *SSE*, *AIC*, *RMSPE*, and R^2 of the models under comparison are given in Table 5.2.

According to the estimated values of (α) the debugging process was not perfect and the total number of faults introduced by the 38^{th} weeks of testing is ($a(t=38)$-a), i.e., (252-56=196). It is clearly seen that the proposed model is the best among the models under comparison in terms of *SSE*, *AIC*, *RMSPE* and R^2.

Table 5.2: Parameters Estimations (for DS-II)

Models under Comparison	Parameters Estimation					Comparison Criteria			
	a	b	p	α	β	SSE	AIC	RMSPE	R^2
Exponential (Goel & Okumoto, 1979)	475.50	.016	—	—	—	764.43	203.49	4.54	.995
Delayed S-shaped (Yamada et al., 1983)	230.36	.101	—	—	—	4800.43	291.17	11.38	.966
Imperfect Debugging (Kapur & Garg, 1992)	475.50	.038	.426	—	—	764.58	205.50	4.54	.996
Fault Generation (Yamada et al., 1992)	56.01	.176	—	.092	—	618.21	200.20	4.09	.996
Inflection S-shaped (Ohba, 1984)	465.44	.017	—	—	.027	832.47	203.64	4.73	.995
PNZ (Pham et al., 1999)	60.10	.160	—	.085	0	590.07	198.97	3.99	.996
Proposed	56.05	.176	1	.092	0	587.13	198.48	3.98	.996

— indicates the parameter is not part of the corresponding model

The fitting of the proposed model to DS-II is graphically illustrated in figure 5.2.1 given below. It is clearly seen that the proposed model fits DS-II excellently.

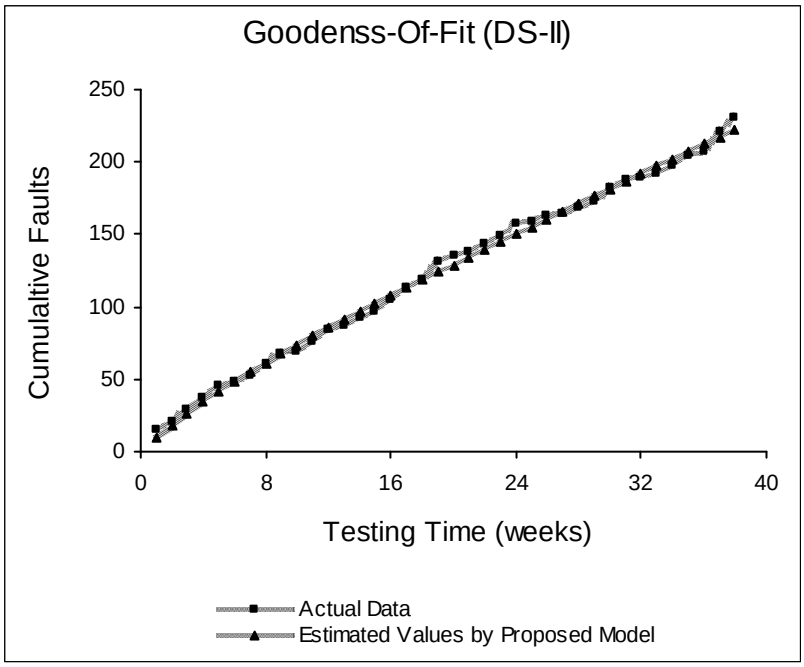

Figure 5.2.1 Faults identification curve for (DS-II)

DS-II are truncated into different proportions and used to estimate the parameters of the proposed model. For each truncation, one value of *RPE* ratio is obtained and is graphically illustrated in figure 5.2.2 given below. It is clearly seen that 50% of the total test time is sufficient to predict the future of the fault removal process reasonably well.

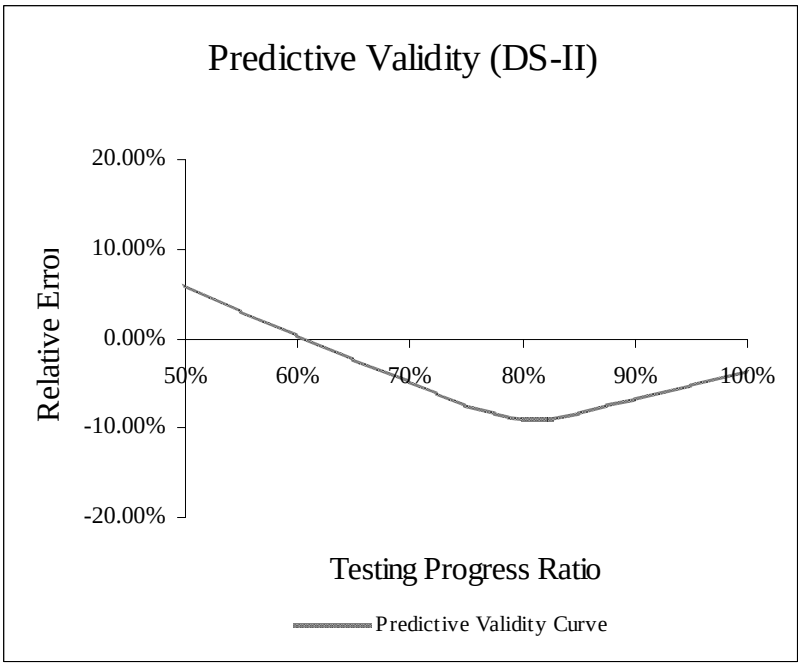

Figure 5.2.2 Predictive validity curve for (DS-II)

The fitting of the proposed model to actual remaining cumulative number of faults for DS-II is graphically illustrated in figure 5.2.3. It is clearly seen that the proposed model fits the actual data well.

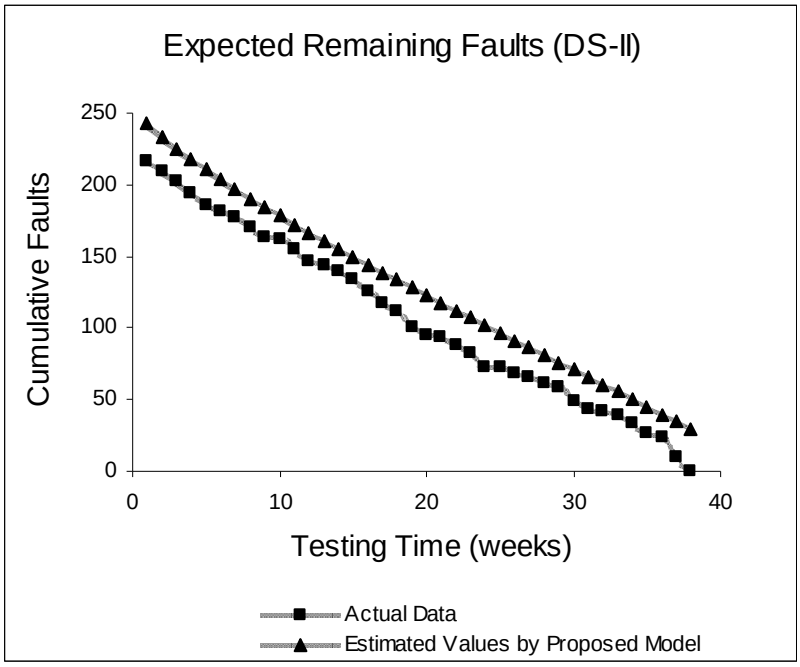

Figure 5.2.3. Remaining faults curve for (DS-II)

Figure 5.2.4 illustrate the software reliability growth for DS-II. It is observed that the reliability is improving during testing.

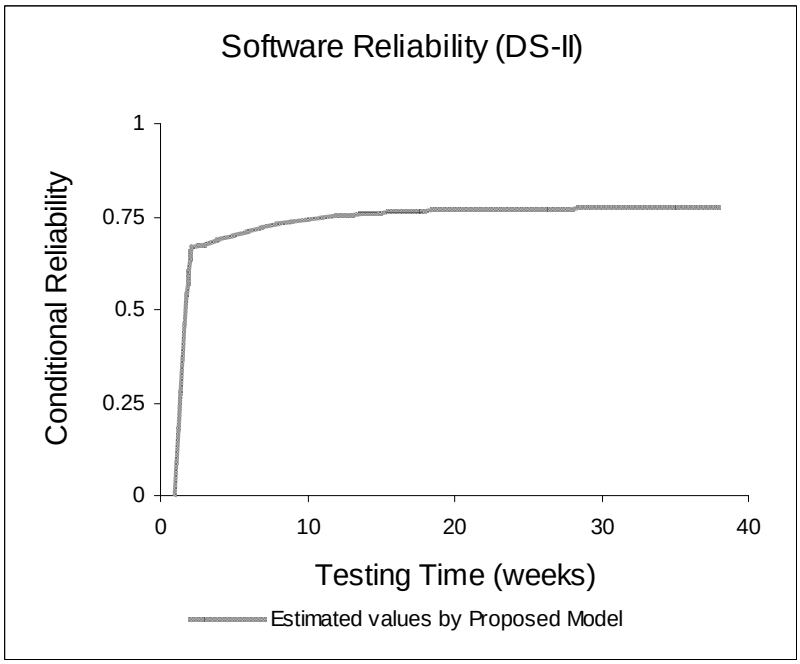

Figure 5.2.4. Reliability curve for (DS-II)

5.3 Third Software Development Project

The results of the parameter estimation and the goodness-of-fit metrics in terms of *SSE*, *AIC*, *RMSPE*, and R^2 of the models under comparison are given in Table 5.3.

According to the estimated values of (α) the debugging process was perfect and no fault introduced. It is clearly seen that the proposed model is the best among the models under comparison except for the metric *SSE* the Inflection S-shaped and PNZ model provide slightly better results.

Table 5.3: Parameters Estimations (for DS-III)

Models under Comparison	Parameters Estimation					Comparison Criteria			
	a	*b*	*p*	α	β	SSE	AIC	RMSPE	R^2
Exponential (Goel & Okumoto, 1979)	538.10	.026	—	—	—	87704.21	733.19	28.23	.965
Delayed S-shaped (Yamada et al., 1983)	488.10	.066	—	—	—	36194.02	645.18	18.14	.985
Imperfect Debugging (Kapur & Garg, 1992)	538.10	.027	.952	—	—	87711.52	735.31	28.24	.965
Fault Generation (Yamada et al., 1992)	465.40	.017	—	.027	—	87959.11	732.48	28.27	.965
Inflection S-shaped (Ohba, 1984)	484.57	.066	—	—	3.65	32411.55	642.53	17.17	.987
PNZ (Pham et al., 1999)	484.57	.067	—	0	3.65	32430.53	642.53	17.17	.987
Proposed	485.18	.067	.998	0	3.66	32439.31	642.46	17.17	.987

— indicates the parameter is not part of the corresponding model

The fitting of the proposed model to DS-III is graphically illustrated in figure 5.3.1 given below. It is clearly seen that the proposed model fits DS-III excellently.

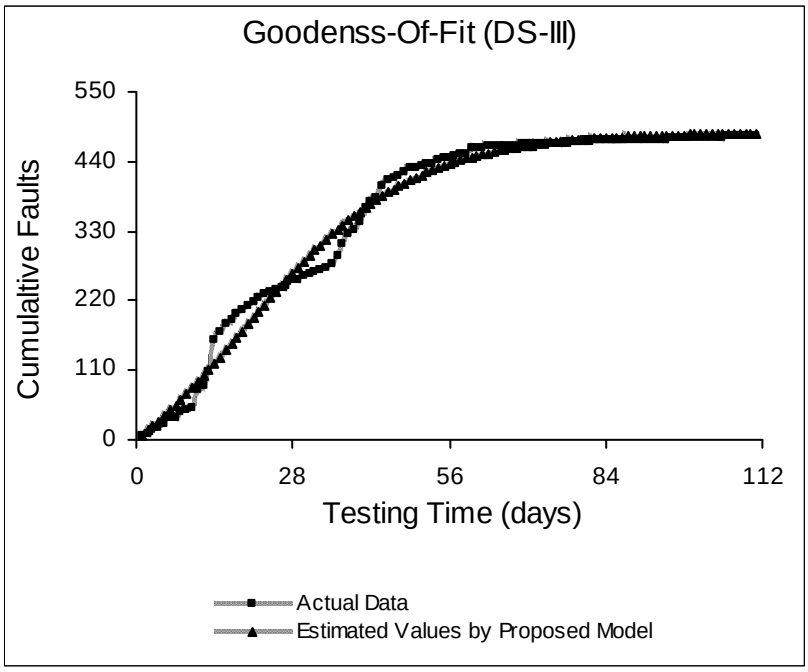

Figure 5.3.1. Faults identification curve for (DS-III)

DS-III are truncated into different proportions and used to estimate the parameters of the proposed model. For each truncation, one value of *RPE* ratio is obtained and is graphically illustrated in figure 5.3.2 given below. It is clearly seen that 60% of the total test time is sufficient to predict the future of the fault removal process reasonably well.

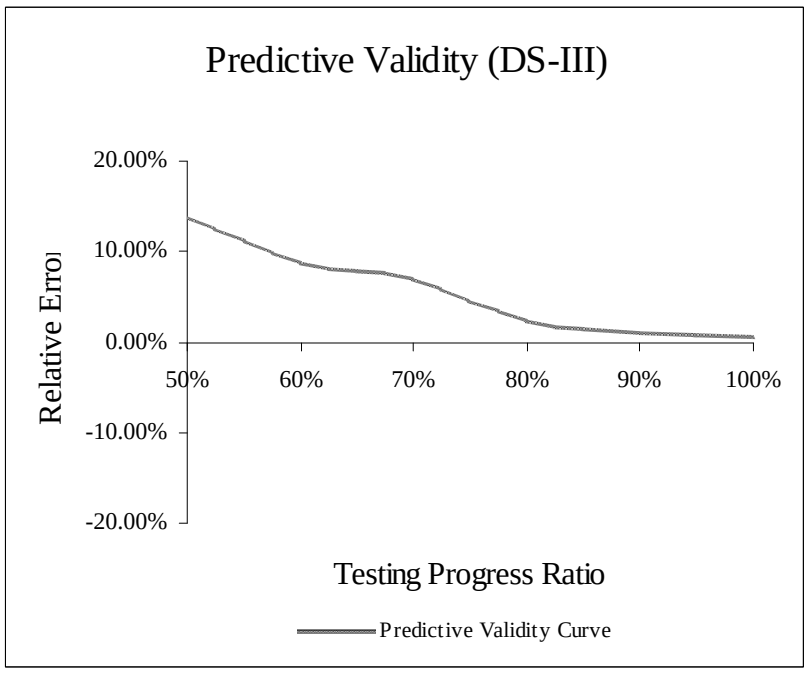

Figure 5.3.2 Predictive validity curve for (DS-III)

The fitting of the proposed model to actual remaining cumulative number of faults for DS-III is graphically illustrated in figure 5.3.3. It is clearly seen that the proposed model fits the actual data reasonably well.

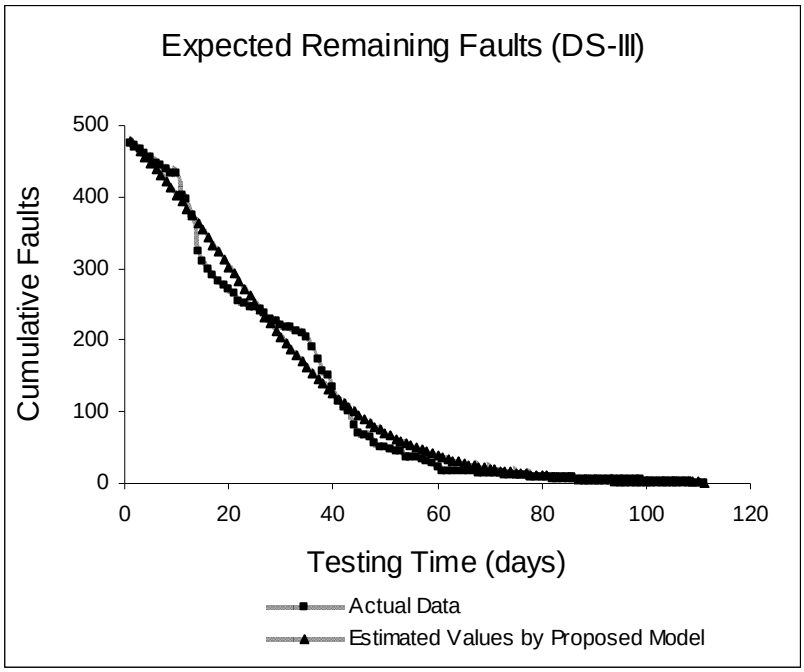

Figure 5.3.3. Remaining faults curve for (DS-III)

Figure 5.3.4 illustrate the software reliability growth for DS-III. It is observed that the reliability is improving during testing.

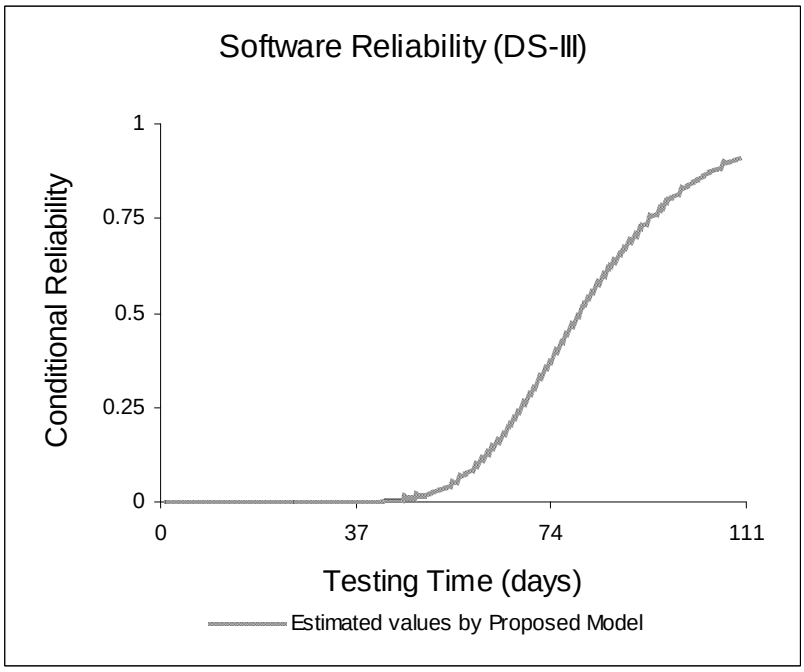

Figure 5.3.4. Reliability curve for (DS-III)

5.4 Fourth Software Development Project

The results of the parameter estimation and the goodness-of-fit metrics in terms of *SSE*, *AIC*, *RMSPE*, and R^2 of the models under comparison are given in Table 5.4.

According to the estimated values of (α) the debugging process was not perfect and the total number of faults introduced by the 15^{th} months of testing is ($a(t=15)$-a), (1224-452=772). It is clearly seen that the proposed model is the best among the models under comparison in terms of *SSE*, *AIC*, *RMSPE* and R^2.

Table 5.4: Parameters Estimations (for DS-IV)

Models under Comparison	Parameters Estimation					Comparison Criteria			
	a	*b*	*p*	α	β	SSE	AIC	RMSPE	R^2
Exponential (Goel & Okumoto, 1979)	1267.18	.138	—	—	—	22006.24	286.08	39.63	.981
Delayed S-shaped (Yamada et al., 1983)	1058.61	.413	—	—	—	104914.5	723.13	86.42	.97
Imperfect Debugging (Kapur & Garg, 1992)	1267.18	.246	.559	—	—	21999.38	287.91	39.61	.981
Fault Generation (Yamada et al., 1992)	536.06	.472	—	.090	—	5734.96	246.69	20.24	.99
Inflection S-shaped (Ohba, 1984)	1267.18	.138	—	—	1	22006.24	288.08	39.63	.981
PNZ (Pham et al., 1999)	451.53	.891	—	.114	1.087	5427.24	242.36	19.69	.99
Proposed	451.53	.894	.997	.114	1.087	5426.85	242.35	19.69	.99

— indicates the parameter is not part of the corresponding model

The fitting of the proposed model to DS-IV is graphically illustrated in figure 5.4.1 given below. It is clearly seen that the proposed model fits DS-IV excellently.

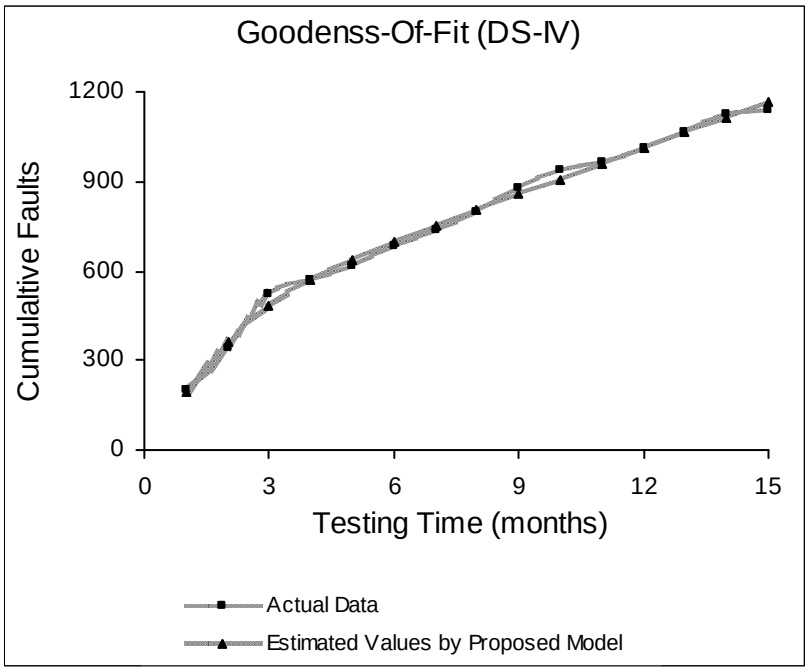

Figure 5.4.1. Faults identification curve for (DS-IV)

DS-IV are truncated into different proportions and used to estimate the parameters of the proposed model. For each truncation, one value of *RPE* ratio is obtained and is graphically illustrated in figure 5.4.2 given below. It is clearly seen that 50% of the total test time is sufficient to predict the future of the fault removal process reasonably well.

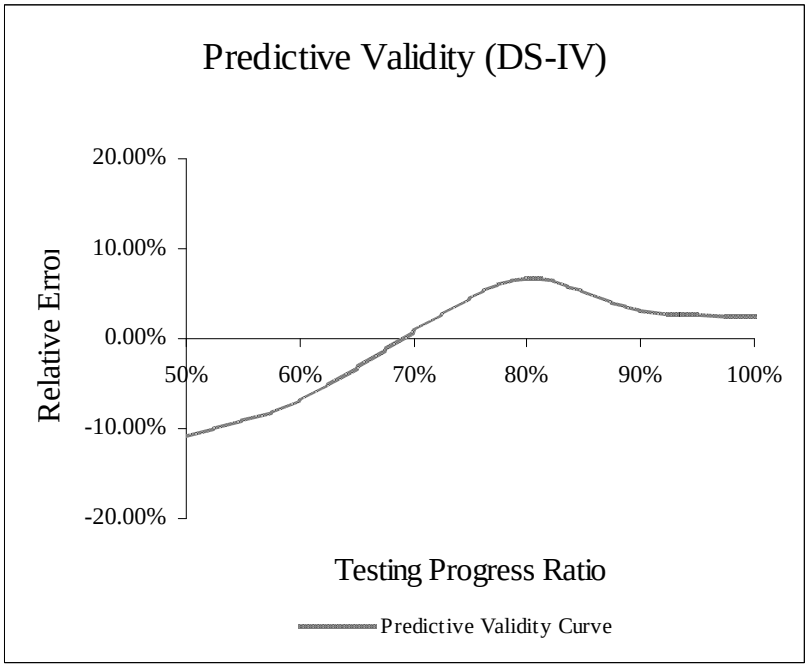

Figure 5.4.2 Predictive validity curve for (DS-IV)

The fitting of the proposed model to actual remaining cumulative number of faults for DS-IV is graphically illustrated in Figure 5.4.3. It is clearly seen that the proposed model fits the actual data reasonably well.

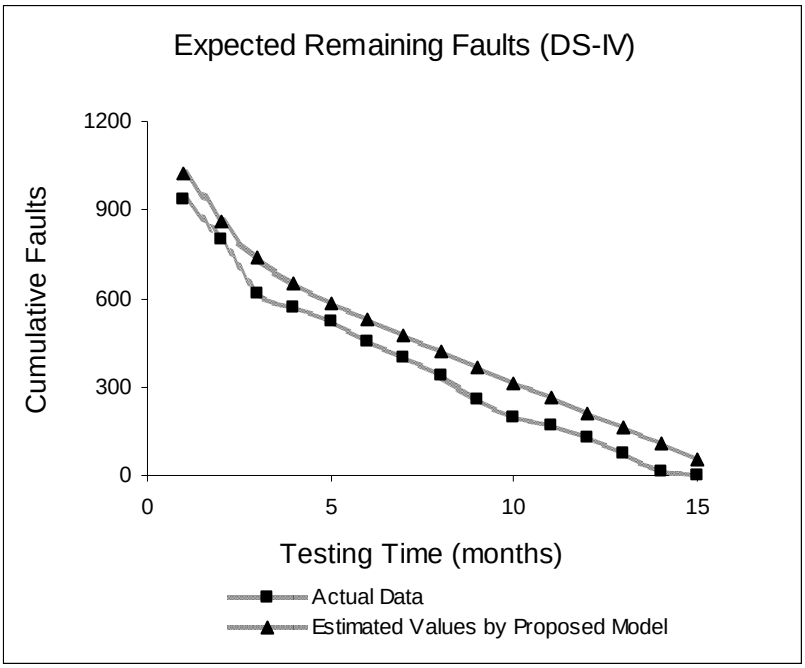

Figure 5.4.3. Remaining faults curve for (DS-IV)

Figure 5.4.4 illustrate the software reliability growth for DS-IV. It is observed that the reliability is improving during testing.

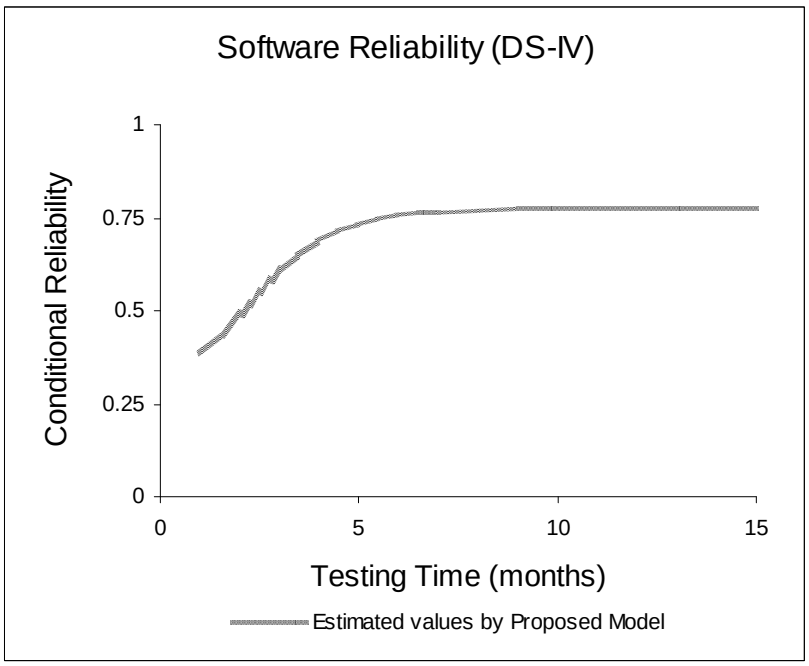

Figure 5.4.4. Reliability curve for (DS-IV)

Chapter 6

Conclusions

In this thesis, a newly developed continuous SRGM with two types of imperfect debugging and learning process of the testing team as testing progresses has been presented. The first type, known as fault generation, describes the situation when each fault removal attempt increases the fault content of the software. The second type, less damaging, is the case of imperfect debugging where all detected faults are not removed completely. Here the numbers of removal attempts are more than actual fault content but imperfect debugging does not change the content of faults in the software. The concept of learning has been incorporated in the fault removal rate to show the gain in experience and improvement in the testing efficiency of the team as the testing grows. To model learning, fault removal rate has been taken as logistic function.

The model has been validated and compared with the mentioned NHPP models by applying them on four fault removal datasets. The results of the proposed model are encouraging in terms of provides improved goodness of fit criteria, predictive validity criterion, and software reliability evaluation measures for software reliability data due to its applicability and flexibility.

Software reliability evaluation measures can provide engineers with insightful information about software development and testing effort, and help project managers make the best decisions in allocating testing effort. Hence, we conclude that the proposed model not only fit the past well but also predict the future reasonably well.

Finally, the proposed model provides a large scope for further extensions and generalizations. For example, incorporation of testing-effort, classification of software faults during testing phase, these are being brought out in a future work.

References

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

Bittanti, S. Blozern, P. Pedrotti, E. Pozzi, M. and Scattolini, A. **A Flexible Modelling Approach in Software Reliability Growth**, In: Goos G and Hartmanis I (Eds.), Software Reliability Modeling and Identification, Springer-Verlag, 1988, pp. 101-140.

Brooks WD and Motley RW, **Analysis of Discrete Software Reliability Models**, Technical Report (RADC-TR-80-84), Rome Air Development Center: New York, 1980.

Goel, A.L. and Okumoto, K. **Time Dependent Error Detection Rate Model for Software Reliability and other Performance Measures**, IEEE Transactions on Reliability, 28(3), 1979, pp. 206-211.

Huang, C.Y. Kuo, S.Y. and Lyu, M.R. **An Assessment of Testing-Effort Dependent Software Reliability Growth Models**, IEEE Transactions on Reliability, 56(2), 2007, pp. 198-211.

Kapur, P.K. and Garg, R.B. **A Software Reliability Growth Model for an Error Removal Phenomenon**, Software Engineering Journal, 7(4), 1992, pp. 291-294.

Kapur, P.K. and Garg, R.B. **Optimal Software Release Policies for Software Reliability Growth Model under Imperfect Debugging**, Recherche Operationnelle / Operations Research (RAIRO), 24, 1990, pp. 295-305.

Kapur, P.K. Bardhan, A.K. and Shatnawi, Omar. **Why Software Reliability Growth Modelling should Define Errors of Different Severity**, Quality Control and Applied Statistics, 49(6), 2004, pp. 699-702.

Kapur, P.K. Garg, R.B and Kumar, S. **Contributions to Hardware and Software Reliability**, World Scientific, 1999.

Kapur, P.K. Jha, P.C. and Singh, V.B. **On the Development of Discrete Software Reliability Growth Models**, In: KB. Misra, (ed.) *Handbook of Performability Engineering*, Springer, 2008, pp. 1239-1255.

Kapur, PK. Singh, O. Shatnawi, Omar. And Gupter, A. **A Discrete NHPP Model for Software Reliability Growth with Imperfect Fault Debugging and Fault Generation,** International Journal of Performability Engineering, 2(4), 2006, pp. 351-368.

Kapur, P.K. Shatnawi, Omar. Aggarwal, A.G. and Kumar, R. **Unified Framework for Developing Testing Effort Dependent Software Reliability Growth Models**, WSEAS Transactions on Systems, 8(4), 2009, pp. 521-531.

Kenny, G.Q. **Estimating Defects in Commercial Software during Operational Use**, IEEE Transactions on Reliability, 42(1), 1993, pp. 107-115.

Khoshogoftaar, T.M. and Woodcock, T.G. **Software Reliability Model Selection: A Case Study**, Proc. of the International Symposium on Software Reliability Engineering, 1991, pp. 183-191

Kuo, S.N. Huang, C.Y, and Lyu, M.R. **Framework for Modelling Software Reliability, using various Testing-Efforts and Fault-Detection Rates**, IEEE Transactions on Reliability, R-50(3), 2001, pp. 310-320.

Lyu, M. R. **Handbook of Software Reliability Engineering**, McGraw-Hill, 1996.

Musa, J.D. Iannino, A. and Okumoto, K. **Software Reliability: Measurement, Prediction, Applications**, McGraw-Hill, 1987.

Musa, J.D. **Software Reliability Engineering: More Reliable Faster and Cheaper**, 2nd edition, McGraw-Hill, 2004.

Ohba, M. **Software Reliability Analysis Models**, IBM Journal of Research and Development, 28, 1984, pp. 428-443.

Pfleeger, S.A. and Atlee, J.M. **Software Engineering: Theory and Practice**, 3rd edition, Prentice-Hall, 2006.

Pham, H. Nordmann, L. and Zhang, X. **A General Imperfect Software-Debugging Model with S-shaped Fault Detection Rate**, IEEE Transactions on Reliability, 48(2), 1999, pp. 169-175.

Pham, H. **Software Reliability**, Springer-Verlag, 2000.

Pressman R.S. **Software Engineering: A Practitioner's Approach**, 5th Edition, McGraw-Hill, 2001.

Shatnawi, Omar. **Measuring Software-Operational Reliability: An Interdisciplinary Modelling Approach**, Proc. of the 18th IFIP World Computer Congress–Student Forum (WCC'2004), Toulouse, France, 2004, pp. 165-176.

Shatnawi, Omar(a). **Discrete Time Modelling In Software Reliability Engineering: A Unified Approach,** International Journal of Computer Systems Science and Engineering, 24(6), 2009, pp. 71-77.

Shatnawi, Omar(b). **Discrete Time NHPP Models for Software Reliability Growth Phenomenon**, International Arab Journal of Information Technology, 6(2), 2009, pp. 124-131.

Shatnawi, Omar. Kapur, P.K. **A Generalized Software Fault Classification Model**, WSEAS Transactions on Computers, 7(9), 2008, pp. 1375-1384.

Yamada, S. and Osaki, S. **Discrete Software Reliability Growth Models**, Applied Stochastic Models and Data Analysis, Vol. 1, 1985, pp. 65-77.

Yamada, S. Ohtera, H. and Narihisa, H. **Software Reliability Growth Models with Testing Effort**, IEEE Transactions on Reliability, R-35, 1986, pp. 19-23.

Yamada, S. Ohba, M. and Osaki, S. **S-shaped Reliability Growth Modelling for Software Error Detection**, IEEE Transactions on Reliability, R-32, 1983, pp. 475-478.

Yamada, S. Tokuno, K. and Osaki, S. **Imperfect Debugging Models with Fault Introduction Rate for Software Reliability Assessment**, International Journal of System Science, 23(12), 1992, pp.2253-2264.

Xie M , **Software Reliability Modelling**, World Scientific, 1991.

Zhang X, and Pham H, **Comparisons of Nonhomogenous Poisson Process Software Reliability Models and its Applications**, International Journal of System Science, 31(9), 2000, pp.1115-1123.

Appendixes

1 - Dataset I: collected during 35 months of testing, 1301 faults were detected

testing time (months)	defects found
1	7
2	29
3	61
4	108
5	134
6	159
7	175
8	223
9	259
10	312
11	369
12	408
13	479
14	559
15	624
16	681
17	771
18	831
19	888
20	978
21	1024
22	1081
23	1110
24	1150
25	1166
26	1184
27	1221
28	1236
29	1244
30	1272
31	1278
32	1283
33	1286
34	1289
35	1301

2 – Dataset II: collected during 38 weeks of testing, 231 faults were detected

testing time (weeks)	defects found
1	15
2	21
3	29
4	37
5	45
6	49
7	53
8	61
9	67
10	69
11	76
12	84
13	87
14	92
15	97
16	105
17	113
18	119
19	131
20	136
21	138
22	143
23	149
24	158
25	159
26	163
27	165
28	169
29	173
30	182
31	188
32	189
33	192
34	198
35	204
36	207
37	221
38	231

3 – Dataset III: collected during 111 days of testing, 481 faults were detected

testing time (days)	defects found
1	5
2	10
3	15
4	20
5	26
6	34
7	36
8	43
9	47
10	49
11	80
12	84
13	108
14	157
15	171
16	183
17	191
18	200
19	204
20	211
21	217
22	226
23	230
24	234
25	236
26	239
27	243
28	252
29	254
30	259
31	263
32	264
33	268
34	271
35	277
36	290
37	309
38	324
39	331
40	346
41	367
42	375
43	381
44	401
45	411

46	414
47	417
48	425
49	430
50	431
51	433
52	435
53	437
54	444
55	446
56	446
57	448
58	451
59	453
60	460
61	463
62	463
63	464
64	464
65	465
66	465
67	465
68	466
69	467
70	467
71	467
72	468
73	469
74	469
75	469
76	469
77	470
78	472
79	472
80	473
81	473
82	473
83	473
84	473
85	473
86	473
87	475
88	475
89	475
90	475
91	475
92	475
93	475

94	475
95	475
96	476
97	476
98	476
99	476
100	477
101	477
102	477
103	478
104	478
105	478
106	479
107	479
108	479
109	480
110	480
111	481

4 – Dataset IV: collected during 15 month of testing, 1138 faults were detected

testing time (months)	defects found
1	203
2	339
3	522
4	569
5	615
6	686
7	740
8	797
9	877
10	941
11	968
12	1010
13	1065
14	1127
15	1138

ملخص

في السيناريو الحالي ، أنظمة الحاسوب لا غنى عنها بالنسبة للمجتمع وضرورة وأهمية الحاسوب تتزايد بشكل سريع. لتلبية هذا الطلب المتزايد ، تعقيد منتجات البرمجيات لبناء مثل هذه الأنظمة الحاسوبية ، عزز إلى حد كبير. أثناء تطوير مثل هذه الأنظمة البرمجية المعقدة، العديد مِن حالات فشل البرامج قد تَحدث. للحد من هذه العيوب، يتطلب الإختبار الشامل للبرامج، لذا أنظمة البرامج التي يعول عليها جداً يتم تطويرها. على مدى العقود الثلاثة الماضية، كانت هناك عدة محاولات لنمذجة العمليات المرتبطة بحالات فشل البرامج على أساس الإفتراضات الأساسية المختلفة لكيفية إختبار البرامج. هذه النماذج هي التي تعرف مجتمعة بنماذج نمو عول البرمجيات. من المهم أن نلاحظ أنه بسبب تعقيد تصميم البرمجيات، فإنه ليس من المتوقع أن يكون نموذج وحيد يمكن أن تدمج جميع العوامل التي يعتقد بأنها تؤثر على عول البرمجيات.

في هذه الأطروحة، نعرض كيف نبدأ بالفرضيات البسيطة جداً، نماذج نمو عول البرمجيات من نوع عملية poisson غير المتجانس للزمن المستمر، جعلها أكثر واقعية بشكل تدريجي بدمج التصحيح الناقص مع تضمين عملية التعلم في التصحيح وإدخال أخطاء جديدة، إن تطبيق النموذج المعمم الناتج أثبت خلال عدة مجموعات بيانات عول البرمجيات الحقيقية التي تم الحصول عليها من مشاريع تطوير البرمجيات المختلفة. النموذج المعمم المقترح تم التأكد منه مقابل النماذج الأخرى ضمنها النماذج الموجودة (مثل، PNZ، Inflection S-shaped، Imperfect، Delayed s-shaped، Exponential) نتيجة لذلك يتأكد تطبيقه. مجموعات بيانات عول البرمجيات تم إختيارها عمداً من بيئات الأختبار المختلفة حيث منحنيات النمو تتراوح من أسي تماماً إلى هيئة S جداً. النتائج مشجعة إلى حد لا بأس به من ناحية جودة الملائمة، صلاحية التنبوء، ومقاييس تقييم عول البرمجيات.

المساهمة الرئيسية لهذه الأطروحة هي تقديمها مفهوم نوعان من التصحيح الناقص خلال ظاهرة إزالة خطأ البرمجيات مع معدل إزالة الخطأ اللوجستية. معظم نماذج نمو عول البرمجيات التي نوقشت في السابق نادراً ما يفرق بين ملاحظة الفشل وعمليات إزالة الخطأ. في بيئة تطوير البرامج الحقيقية، عدد حالات الفشل الملاحظ ليس من الضروري أن يكون مماثلاً لعدد الأخطاء المزالة. إذا كان عدد حالات الفشل الملاحظ أكثر من عدد الأخطاء المزالة يكون لدينا حالة التصحيح الناقص. بسبب تعقيد نظام البرمجيات والفهم الناقص لمتطلبات البرامج، المواصفات والتركيب، فريق الإختبار قد لا يستطيع إزالة الخطأ تماماً عند إكتشاف الفشل

والخطأ الأصلي قد يبقى أو يستبدل بخطأ أخر. في حين أن الظاهرة الأولى تعرف بالتـــصحيح الناقص، تسمى الثانية توليد الخطأ. في حالة التصحيح الناقص محتوى خطأ البرمجيات لا يتغير، لكن فقط بسبب الفهم الناقص للبرامج، الخطأ المكتشف لم يتم إزالتها بالكامل. ولكن فـــي حالـــة توليد الخطأ محتوى الخطأ يزداد كما التقدم في الإختبار والإزالة ينتج أخطاء جديدة عند إزالـــة خطأ قديم. لتشكيل التعلم، معدل إزالة الخطأ تؤخذ كوظيفة لوجستية.