AL al-Bayt University

Prince Hussein bin Abdullah Faculty of Information Technology

Computer Department

Modelling and Analysis of Software Reliability Phenomenon in Distributed Development Environment

نمذجة وتحليل ظاهرة عول البرمجيات في بيئة بناء موزعة

By

Nesreen T. Qallab

Supervisor

Dr. Omar Shatnawi

Submitted this Thesis to complement the Requirements for the Master's Degree of Science in Computer Science

Deanship of Graduated Studies

Al al_bayt University

May,2017

Committee Decision

This Thesis (Modelling and Analysis of Software Reliability Phenomenon in Distributed Development Environment) was Discussed and Approved on 21/5/2017

The Members Of Discussion Committee	**Signature**
Dr. Omar Shatnawi	.....
Dr. Saad Bani-Mohammad	.....
Dr.Akram Hamarsheh	.....
Dr.Ahmad Otoom	.....

Dedication

To my mother who put her life for me.

To pure soul of my father.

To soul of my daughter, who revolves around me, Sara..

To the candle of my life, my daughter, Mira.

To all my family For their encouragement and understanding during the period of my study.

Acknowledgment

First, I would like to thank God for his graces and guidance. Thanks also to my mentor Dr. Omar Shatnawi for his expert guidance and timely input throughout of my thesis. I would also like to thank him for his constant motivation and supporting during my graduate studies at Al al _Bayt university. Also, I would like to thank the members of the examination committee for their comments, which contributed to the improvement of thise thesis.

Finally, I would lik to thank my friends Eng.Raya Omoush, Eng Manal Al_Eassa Miss.Rasha Harahsheh, Miss.Fatima Hajjawi, Miss.Maryam Mosleh for their supporting and Encouragement during my study.

Nesreen Qallab

May,2017

Contents

Tables list

Figures list

Appendices list

Abstract

The present scenario of software development life-cycle has switched into a distributed environment because of the development of network technology and ever increased demand of sharing the resources to optimize the cost. In the software reliability engineering literature, few attempts have been made to model the fault testing and debugging process in a distributed development environment.

As the area of software fault-debugging in distributed development environment is not thoroughly investigated in current literature, even though it is estimated to have been one of the most expensive endeavor in the industry. This objective dictates developing a novel testing-effort dependent software-reliability modelling approach for distributed-systems developed under imperfect-debugging environment. Fault-debugging process and testing-effort expenditures are described by a non-homogenous Poisson process and testing-effort curve functions respectively. The resultant integrated modelling approach proves to be conducive for obtaining several models by following a single methodology and thus present a perspective investigation for studying of general models without making many assumptions.

To the best of our knowledge this is the first time that this kind of integration modelling approach has been carried out for distributed systems that describes the relationship among the calendar time, the testing-effort consumption, and fault- correction/debugging process under imperfect-debugging environment. Actual software reliability data cited in literature have been employed to demonstrate the applicability of the proposed integrated modelling approach. The results are fairly encouraging and plausible when compared with well-documented modeling-approach.

Key-Words: software engineering, software testing, imperfect debugging, testing-effort, Laplace test.

Chapter 1

Introduction and Overview

Developing large-scale distributed software system is generally a quite complex and time consuming process. Due to their development complexity, these systems are hardly ever "perfect" (Lavinia et al., 2011). They are developed using the classical software engineering activities (Shatnawi, 2013). However, the nature and complexity of their requirements have drastically changed and users all over the world have become much more demanding in terms of cost, schedule and quality. Several techniques available for investigating the cost/schedule of software; however, reliability is most important attributes of software quality (Musa et al., 1987).

1.1 Software Reliability

Software is an integral part of any computer system. The level of cost, schedule, and quality are the very important characteristics of software products. Nowadays several techniques exist for investigating the cost and schedule of software; however, reliability is the most important attribute of quality to measure software quality. The study of software reliability is important as it has direct affect the cost and time to delivery.

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).

The objective of software testing and debugging phase in the software-development-process is detecting and correcting faults, to make the software more reliable.

Software-reliability-models are utilized for assessing the degree of achievement of software-quality, deciding the time to software release for operational use, and evaluating the maintenance cost for faults undetected during the testing-phase (Yamada, 2014).

Software reliability quantities have generally been defined with respect to time. There are three kinds of time:

- the execution-time, the actual CPU time spent by the processor in executing the program,
- the calendar-time, the familiar time we experience, and
- the clock-time, the time from the beginning to the end of program execution.

Experimentation has shown that models based on execution time are superior to those based on calendar-time or clock-time.

The aim of software quality/reliability measurement and assessment (Yamada, 2014) is illustrated in Figure 1.1.

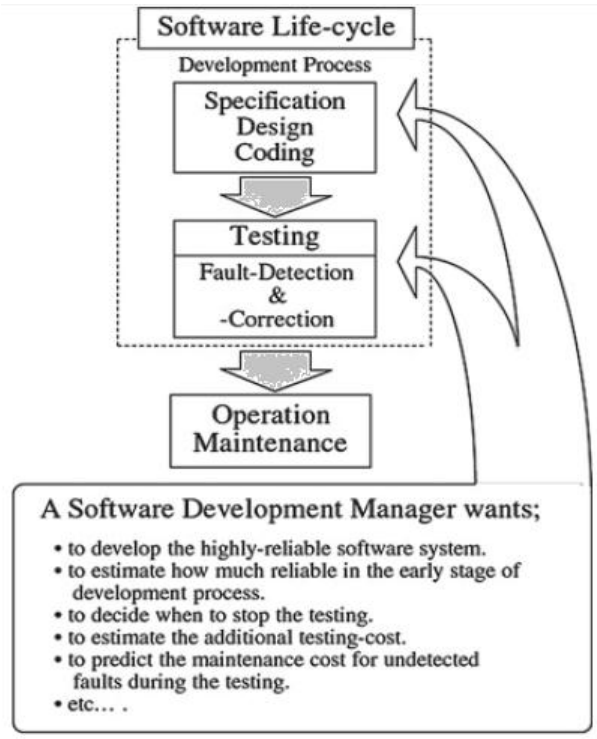

Fig. 1.1 Aim of software quality/reliability measurement

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). Software reliability measures can be used for system engineering, project management during development, and software management during operational use, and evaluation of software engineering technologies (Musa et al., 1987).

1.2 Software Reliability Modelling Types

Software reliability modelling has become one of the most important aspects in software reliability engineering. There are two main types of software reliability models: the deterministic and the probabilistic, more details can be found in (Kapur et al.; 2011; Pham, 2006). 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:

- **Error Seeding Models**
- **Failure Rate Models.**
- **Curve Fitting Models**.
- **Reliability Growth Models.**
- **Markov Structure Models**
- **Non-Homogeneous Poisson Process (NHPP) Models.**

1.3 Non-Homogeneous Poisson Process 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.

As a general class of well-developed stochastic process model in reliability engineering, non-homogeneous Poisson process models have been successfully used in studying hardware reliability problems. They are especially useful to describe software fault debugging-process or fault-correction phenomena which possess certain trends such as reliability-growth or deterioration. Therefore, an application of non-homogeneous Poisson process models to software reliability analysis is then easily implemented. Other important advantages of non-homogeneous Poisson process models which should be stressed here are that non-homogeneous Poisson process models are closed under superposition and time transformation. We can easily incorporate two or more existing non-homogeneous Poisson process models by summing up the corresponding mean value functions. It is worth mentioning that the non-homogeneous Poisson process 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.

The term ‘non-homogeneous’ implies that the characteristics of the probability distributions that describe the random variables representing the testing and debugging processes vary with time. This variation of failure intensity in time is to be expected since faults are being corrected and/or introduced in a program as time passes. Numerous non-homogeneous Poisson process based models have been formulated to assess software reliability (Kapur et al., 2011; Ahmad et al., 2010; Shatnawi, 2014, 2016; Idris, 2009).

1.4 Testing-Effort Modelling

Testing and debugging phase in the software-development-process aims at detecting and correcting faults, and hence making reliable software. The testing and debugging phase, which aims to improve the reliability of a software system, is the most costly, time-consuming phase among the four phases. About half of the resources consumed during the software development cycle are testing resources (Ohtera & Yamada, 1990; Shatnawi, 2013). Due to the increased size of the software, effective utilization of resource has become even more important than before (Wang et al., 2010).

In reality, no software manager/developer is going to spend-infinite-resources on testing/debugging software. Testing resources include execution time, man power etc that affects reliability. The function that describes how testing resources are distributed is usually referred to as testing effort function and it has been incorporated into software reliability modelling (Peng et al., 2014).

In software reliability literature, testing-effort curves (viz, as exponential, Rayleigh, Weibull, logistic etc) have been employed in the literature to measure testing resources (Yamada et al., 1985; Kuo et al., 2001; Huang et al., 2007; Kapur et al., 2008; Shatnawi 2013). The exponential and Rayleigh can be modelled as, "the testing-effort consumption rate is proportional to the testing resources available"

$$\frac{\partial}{\partial t}W_t = c(t) \cdot (d - W_t) \qquad (1.1)$$

Solving (1.1) under initial condition $W_{t=0} = 0$, yields

$$W_t = d \cdot \left(1 - \exp(\int_0^t c(x) \cdot dx)\right) \qquad (1.2)$$

***Case* 1:** If $c(t) = c$, the testing-effort expenditures follows an exponential curve:

$$W_t = d \cdot (1 - \exp(-c \cdot t)) \quad (1.3)$$

***Case* 2:** If $c(t) = c \cdot t$, the testing-effort expenditures follows a Rayleigh type curve: $W_t = d \cdot \left(1 - \exp(-\frac{c \cdot t^2}{2})\right)$ (1.4)

***Case* 3:** If $c(t) = c \cdot r \cdot t^{r-1}$, the testing-effort expenditures follows a Weibull function:

$$W_t = d \cdot (1 - \exp(-c \cdot t^r)) \quad (1.5)$$

***Case* 4:** If $c(t) = c \cdot \frac{W_t}{d}$, the testing-effort expenditures follows a logistic function:

$$W_t = \frac{d}{1 + r \cdot \exp(-c \cdot t)} \quad (1.6)$$

It is worth mentioning that Rayleigh and exponential testing-effort consumption curves are a special cases of the Weibull testing-effort consumption curve.

1.5 Challenges Facing Software Reliability

As software is created by error-prone humans, and there is no way to prevent programmes from making mistakes. Faults can be introduced during the software development-lifecycle. Therefore, it is impossible to guarantee a failure-free software system (Lyu, 2007). In software reliability engineering literature, fault-debugging is challenging, and least developed. Software fault-debugging process is the process of detecting, locating, and correcting faults in software (IEEE, 1990). Approximately 20% of all software faults take 80% of all the required effort to analyse, isolate and correct software faults (Boehm and Basili, 2001). Software-failure is estimated to cost American industries USD 60 billion per year (Tassey, 2002). Jones state that imperfect-debugging phenomenon

were discovered in most software-development-companies (Jones, 2008). Reusability is a key direction to improving software development productivity and quality (Shatnawi, 2013; 2017). Due to high demand on quality and productivity in social systems, measuring reliability of software systems in distributed development environment is major concern for software developers (Tamura et al., 2006).

1.6 Thesis objective

This study has attempted to develop an integrated modelling approach, so as to capture different reliability growth curves ranging from exponential to highly S-shaped and incorporates the effect of software fault- correction/debugging complexity with time-dependent variation in testing-effort consumption for distributed systems developed under imperfect-debugging environments. Such approach is very much suited also for object-oriented software development environment.

1.7 Structure of the Thesis

The following is a brief of the remaining Chapters:

Chapter 2 reviews some of the well-documented and established non-homogenous Poisson process based software reliability model for software quality/reliability measurement and assessment in a distributed development environment.

Chapter 3 proposes a newly developed quantitative technique for software quality/reliability measurement and assessment model.

Chapter 4 defines the technique that has been employed for parameter estimation and software reliability data analyses, and provides the comparison criteria used for validation/evaluation.

Chapter 5 presents the applications of the proposed integrated modelling approach to actual software reliability data through data analyses and model comparisons.

Chapter 6 concludes and identifies possible avenues for future research.

Chapter 2

Software Reliability Modelling in Distributed Development Environment: Literature Review

Software reliability 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.

All models discussed in this Chapter are based on non-homogeneous Poisson process because they can be easily applied in actual software development. Therefore, they are very useful in describing testing and debugging processes.

2.1 Software Reliability Modelling

The non-homogenous Poisson process based models that explain the software reliability-growth-phenomenon or fault-debugging-process in distributed development environment can be of two categories:

- **Time-dependent behavior of fault-correction process.** That is the number of software faults being corrected is proportional to the remaining faults.
    - **Yamada et al. (2000) Model**
    - **Kapur et al. (2009a) Model**
    - **Kapur et al. (2009b) Model**
- **Time-dependent variation in testing-effort consumption.** That is, the number of faults being corrected by the current testing-effort expenditures at any time is proportional to the remaining number of faults.

- **Kapur et al. (2004) Model**
- **Shatnawi (2013) Model**

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

- Fault-debugging-process follows non-homogenous Poisson process.
- Software reliability growth phenomenon in the re-used components is uniform (i.e., follows an exponential growth curve) while in the newly-developed component is not (i.e., follows an S-shaped growth curve).
- Fault-correction phenomena for re-used and newly-developed components has been modelled individually and is summed up to get the total fault-correction phenomenon of the software system.

The following are some of non-homogeneous Poisson process based software-reliability-models were proposed for distributed development environment.

2.1.1 Yamada et al. (2000) Model

This model was a pioneering attempt in the field of software reliability modeling and paved the way for measuring reliability in distributed development environment. The model incorporates the exponential software reliability growth model (Goel and Okumoto, 1979) and the delayed S-shaped software reliability growth model (Yamada et al., 1983).

2.1.2 Kapur et al. (2004) Model

The model describes the reliability-growth-phenomenon with respect to the testing-effort consumptions. The author incorporates a time-dependent fault removal rate in the newly developed software sub-system with respect to testing-effort. This can account for learning which increases with testing and debugging time.

2.1.3 Kapur et al. (2009a) Model

The unified framework describes the fault-correction process using unified modelling approach. Using this generalized approach, a wide range of models (existing as well as new) can be developed for different design environment.

2.1.4 Kapur et al. (2009b) Model

The model describes the software reliability growth phenomenon considering two types of imperfect-debugging-process. The first type of imperfect-debugging is where all detected errors are not removed completely resulting in the same fault content of the software. The second type, known as error-generation, describes the situation when each error removal attempt increases the fault content of the software. For newly developed component, it is assumed that removal process follows logistic growth curve due to the fact that learning of removal team grows as testing progresses.

2.1.5 Shatnawi (2013) Model

The model integrates testing-effort function into Yamada et al. (2000) model to get a better description of the software fault-correction process. To relax the pre-specified fault-content-weight or testing-weight parameters for each software

component that has been adopted in the aforementioned models. The author assumed that the ratio of fault-density and the amount of testing-effort expenditure in re-used to newly-developed modules is about 1 to 4, as reported "the defect rate for reused code is 0.9 defects per kilo line of code (KLOC), while the rate for newly developed software is 4.1 defects per KLOC in a study conducted at Hewlett-Packard (HP)" (Lim, 1994; Shatnawi, 2013).

2.2 Study Motivation

The aforementioned software reliability models are constructed considering the debugging scenarios as tabulated in Table 2.2.1. However, none of them provide insightful interpretations for both the testing-effort expenditure and imperfect-debugging phenomena during testing and debugging phase. A proposed solution is developing an integrated modelling approach.

Therefore, this study has attempted to develop an integrated modelling approach, so as to capture different reliability growth curves ranging from exponential to highly S-shaped and incorporates the effect of software fault-correction/debugging complexity with time-dependent variation in testing-effort consumption for distributed systems developed under imperfect-debugging environments. Such in approach is very much suited also for object-oriented software development environment. Because object-oriented based on client-server idea, therefor it is a distributed environment.

Table 2.1 Models under comparison

Modelling Approach	NHPP	Calendar-Time	Testing-Effort (CPU time)	Imperfect-Debugging
Yamada et al. (2000)	√	√		
Kapur et al. (2004)	√	√	√	
Kapur et al. (2009a)	√	√		
Kapur et al. (2009b)	√	√		√
Shatnawi (2013)	√	√	√	
Proposed	√	√	√	√

To the best of our knowledge this is the first time that this kind of non-homogenous Poisson process based integration modelling approach that describes the relationship among the calendar time, the testing-effort consumption, and fault- correction/debugging process under imperfect-debugging environment, has been studied for distributed systems.

2-3 Study Methodology

- **Step 1- Study software reliability data:** The models require that software reliability data be available. The first step in developing a model is to carefully study such data in order to gain an insight into the nature of the process being modeled. It is highly desirable to plot the data as a function of, say, calendar time, execution time, or number of test cases executed. The objective of such plots is to try to determine the appropriate variables to use in the model. Sometimes it is desirable to model several such combinations and then use the

- fitted models for answering a variety of questions about the failure process. Occasionally, it may be necessary to normalize the data to, for example, account for changes in system size during testing.

- **Step 2- Formulate a Reliability Model:** The next step is to construct an appropriate model based upon an understanding of the software technology, testing process, and development environment. The data and plots from Step 1 are likely to be very helpful in this process.

- **Step 3- Obtain Estimates of Model Parameters:** Different methods are generally required depending upon the nature of available data. The most commonly used one is the method of maximum likelihood because it has very good statistical properties. However, sometimes, the method of least squares or some other method may be preferred.

- **Step 4- Obtain the Fitted Model:** The fitted model is obtained by substituting the estimated values of the parameters in the developed model. At this stage, we have a fitted model based on the available failure data.

- **Step 5- Perform Goodness-of-Fit Test:** Before proceeding further, it is advisable to conduct suitable goodness-of-fit test to check the model fit. If the model fits, i.e., if it is a satisfactory descriptor of the observed failure process, we can move ahead. However, if the model does not fit, we have to collect additional data or seek a better, more appropriate model. There is no easy answer to either how much data to collect or how to look for a better model. Decisions on these issues are very much problem dependent and require a clear understanding of the models and the software development environment.

- **Step 6- Obtain Estimates of Performance Measures:** At this stage, we can compute various quantitative measures to assess the performance of the software system.
- **Step 7- Decision Making:** The ultimate objective of developing a model is to use it for making some decisions about the software system, e.g., whether to release the system or continue testing. Such decisions are made at this stage of the modeling process based on the information developed in the previous steps.

Chapter 3

Testing-Effort Dependent Software Reliability Modelling for Distributed Systems in Imperfect-Debugging Environment: A Proposed Integrated Approach

Distributed systems are being developed using the classical software engineering activities (Shatnawi, 2013). Debugging is one of the most challenging, and least developed areas of software engineering. Software developers spend about 35-50 percent of their time debugging software. The cost of debugging and testing is estimated for 50-75 percent of the total budget of software development projects, amounting to more than $100 billion annually (Chmiel and Loui, 2004; O'Dell, 2007). Software development still at nascent stage and has a long way to go for confirm success of projects. Therefore, it is impossible to guarantee a failure-free software system (Lyu, 2007) and absent of imperfect-debugging phenomenon in almost every project (Jones, 2008).

The software debugging-process aims at detecting and correcting faults in order to improve the software reliability, which can be modelled by a mathematical relationship called a software reliability model. As already stated in Chapter 1 these models are used to measure software reliability and can plot the reliability the trends that are used to forecast the number of fault-corrected as a function of time. Execution time based models are superior to those based on calendar-time or clock-time (Musa et al., 1987). These models show how software reliability improves as the faults are detected and corrected. The testing and debugging activities in perfect and imperfect-debugging environments (Lin, 2011) are depicted in Figure 3.1 and 3.2 respectively.

Fig. 3.1 Testing and perfect-debugging activities

Fig. 3.2 Testing and imperfect-debugging activities

The main objective of this study is to construct/develop a software reliability model based on more realistic assumptions depicting different phenomena during the testing and debugging phase in a distributed development environment

- The first step in achieving this objective is to identify the unrealistic assumptions which existing models are based on.
- The second step is to build flexible model which relax these assumptions.

The following are some of unrealistic assumptions

1. All faults are of the same type, complexity and have the same impact on the reliability growth.
2. Testing-effort employed to detect, locate and correct the faults has the same consumption pattern.
3. Pre-specified testing-weight parameters for each software component.

To address these unrealistic issues, a newly developed software reliability model for distributed system, through an integrated modelling approach incorporating fault-debugging complexity with time-dependent variation in testing-effort consumption under imperfect-debugging environment is proposed.

3.1 Assumption and Notations

The following are the assumptions adopted for formulation of the proposed integrated modelling approach:

1. Testing resource is not constantly allocated during software testing phase, which can largely influence the debugging-process.
2. The debugging-process consists of three stages namely, fault-detection, fault-location and fault-correction. That is, each time a failure is reported, an immediate or delayed-effort takes place to correct it. Accordingly, the faults are classified into three types: easy, medium, and hard, according to their debugging-correction complexity.
3. The ratio of fault density and the amount of testing-effort expenditure in reused to newly developed components is about 1 to 4 (Lim, 1994; Shatnawi, 2013).
4. The debugging-team may not be able to correct the fault and the fault may remain or get replaced. While the first phenomenon is known as imperfect-debugging, the second is called error-generation.
5. Newly developed component contains “medium & hard “ types of faults, while reused component contains only “easy “type of fault “.

The following notations are used for the mathematical formulation purpose:

m_{w_t} Expected number of faults debugged in time-dependent variation in testing-effort consumption $(0, W_t]$

i, j Subscripts that denotes the re-used and newly-developed components

m_i Expected number of faults debugged in re-used modules

m_j Expected number of faults debugged in newly-developed modules

W_t Amount of testing-effort consumed in the time interval $(0, t]$

$W_{i,j}$ Expected effort spent on modules debugging $W_i = W_t \cdot g_i$; $W_j = W_t \cdot g_j$

$w_{i,j}$ Current effort spent on modules debugging, that is, $W_t = \int w_x \cdot dx$

$g_{i,j}$ Proportion of effort spent on modules debugging $0 \leq g_i(g_j) \leq 0.2(0.8); \sum g_{i,j} = 1$

a Total number of faults lying dormant in software $\sum a_{i,j} = a$

a_i Initial fault-content in re-used modules $a_i = ah_i$

a_j Initial fault-content in newly developed modules $a_j = ah_j$

$h_{i,j}$ Proportion of fault-content in modules $0 \leq h_i(h_j) \leq 0.2(0.8); \sum h_{i,j} = 1$

p Probability of fault removal on a detection of a fault

α Rate at which faults may be introduced during the debugging-process

c_t Time dependent rate at which testing resources are consumed, with respect to remaining available resources

a, c, r Constant parameter in testing-effort functions

3.2 Formulation

Modelling the Imperfect Fault-Debugging-Process of '*i*' Reused Components. To model the fault correction process of '*i*' re-used components, the imperfect-debugging-model with testing-effort (Kapur et al., 2011; 2009) is selected. The selected model assumed that faults are of type '*easy to debug*, and their debugging-process is modeled as one-stage process. That is, once the failure is reported that fault that caused it, is corrected immediately without delay as illustrated in Figure 3.3. The model is given as

$$m_{w_{t_i}} = \frac{a_i}{1-\alpha_i} \cdot \left(1 - e^{-p_i \cdot b_i \cdot (1-\alpha_i) \cdot w_{t_i}}\right) \qquad (3.1)$$

The above mean-value-function in (**3.**1) represents the expected number of faults corrected.

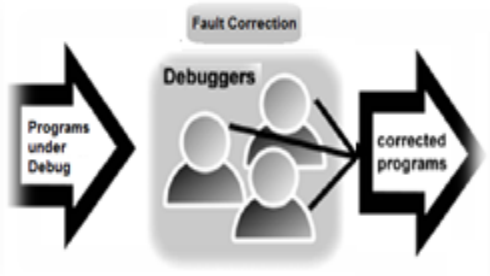

Fig. 3.3 Fault-debugging-process for 'easy to debug' type

Modelling the Imperfect Fault-Debugging-Process of '*j*' Newly Developed Components. To model the fault correction process of '*j*' newly developed components, the imperfect-debugging-model with testing-effort (Kapur et al., 2011; 2009) is adopted for the purpose. The adopted model assumed that faults are of two types: '*medium to debug* and '*hard to debug*', and their debugging-process is modeled as two-stage process and three-stage process receptively. That is, once the failure is reported that fault that caused it, is corrected with different time-delay.

For '*medium to debug'* faults, the imperfect debugging-process is modeled as a two-stage process—fault detection followed by correction as illust

rated in Figure 3.4. The mean-value-function for components containing '*medium to debug* faults is given as

$$m_{w_{t_j}} = \frac{a_j}{1-\alpha_j} \cdot \left(1 - \left(\left(1 + b_j \cdot w_{t_j}\right) \cdot e^{-b_j \cdot w_{t_j}}\right)^{p_j \cdot (1-\alpha_j)}\right) \qquad (3.2)$$

Fig. 3.4 Fault-debugging-process for 'medium to debug' type

For '*hard to debug'* faults, the imperfect-debugging-process is modeled as a three-stage process—fault detection, isolation followed by correction as illustrated in Figure 3.5. The mean-value-function for components containing '*hard to debug* faults is given as

$$m_{w_{t_j}} = \frac{a_j}{1-\alpha_j} \cdot \left(1 - \left(\left(1 + b_j \cdot w_{t_j} + {b_j}^2 \cdot \frac{{w_{t_j}}^2}{2}\right) \cdot e^{-b_j \cdot w_{t_j}}\right)^{p_j \cdot (1-\alpha_j)}\right)$$

(3.3)

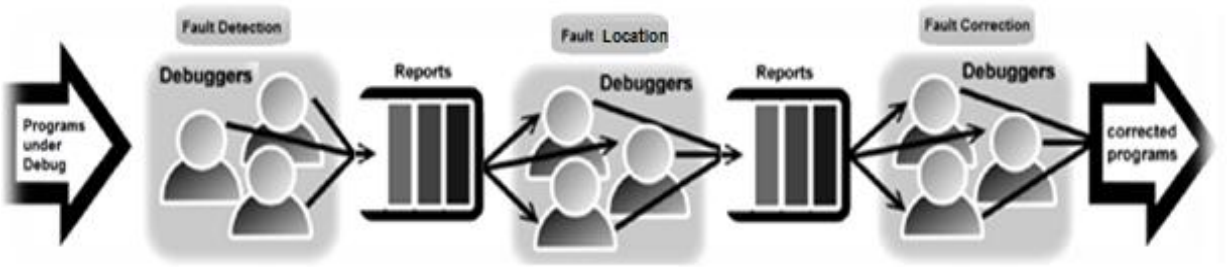

Fig. 3.5 Fault-debugging-process for 'hard to debug' type

The total imperfect-debugging of '*j*' newly developed components is the superposition of the sum of the two debugging-process with mean-value-functions given in (3.2) and (3.3) respectively, as

$$m_{w_{t_j}} = \frac{a_j}{1-\alpha_j} \cdot \left(1 - \left(\left(1 + b_j \cdot w_{t_j}\right) \cdot e^{-b_j \cdot w_{t_j}}\right)^{p_j \cdot (1-\alpha_j)}\right) + \frac{a_j}{1-\alpha_j} \cdot \left(1 - \left(\left(1 + b_j \cdot w_{t_j} + {b_j}^2 \cdot \frac{{w_{t_j}}^2}{2}\right) \cdot e^{-b_j \cdot w_{t_j}}\right)^{p_j \cdot (1-\alpha_j)}\right). \quad (3.4)$$

3.3 Modelling the Total Imperfect Fault-Debugging-Process

The proposed modelling approach for software developed in distributed environment is the superposition of the sum of the total debugging-process of '*i*' reused and '*j*' newly developed components with mean-value-function given in (3.1) and (3.4) respectively, as

$$
\begin{aligned}
m_{w_t} = \; & \sum_{i=1}^{n} m_{w_{t_i}} + \sum_{j=n+1}^{m} m_{w_{t_j}} \\
= \; & \sum_{i=1}^{n} \frac{a_i}{1-\alpha_i} \cdot \left(1 - e^{-p_i \cdot b_i \cdot (1-\alpha_i) \cdot w_{t_i}}\right) + \\
& \sum_{j=n+1}^{m} \frac{a_i}{1-\alpha_i} \cdot \left(\left(1 - \left(\left(1 + b_j \cdot w_{t_j}\right) \cdot e^{-b_j \cdot w_{t_j}}\right)^{p_j \cdot (1-\alpha_j)}\right) \right. \\
& + \\
& \left. \left(1 - \left(\left(1 + b_j \cdot w_{t_j} + {b_j}^2 \cdot \frac{{w_{t_j}}^2}{2}\right) \cdot e^{-b_j \cdot w_{t_j}}\right)^{p_j \cdot (1-\alpha_j)}\right)\right) \qquad (3.5)
\end{aligned}
$$

This proposed modelling approach given above in (3.5) is very interesting from various points of view. Besides its interpretation, it has the models (Goel and Okumoto, 1979; Yamada et al., 1992; Kapur et al. 1999; Yamada et al., 2000; Kapur et al. 2011; Shatnawi, 2013) as special cases. Thus, highlight it is flexibility and applicability.

Chapter 4

Model Validation and Comparison Criteria

To check the validity of the models under comparisons including the proposed modeling approach given previously in chapter (3) to describe reliability-growth, it has been tested on three software reliability datasets obtained from actual software-development-project. The first data-set was collected during 19 weeks of testing, 328 faults were detected (Ohba, 1984). The second data-set was collected during 38 weeks of testing, 231 faults were detected (Misra, 1983). The fourth data-set was collected during 35 months of testing, 1301 faults were detected (Brooks and Motely, 1980). These data-set were deliberately chosen from different testing environments where the growth curves range from exponential to highly S-shaped (for more details refer to Appendix).

For model validation and evaluation, we consider a simple case in which the software system composed of two re-used components and two newly-developed

$$m_{w_t} = \sum_{i=1}^{2} m_{w_{t_i}} + \sum_{j=3}^{4} m_{w_{t_j}}$$

$$= \sum_{i=1}^{2} \frac{a_i}{1-\alpha_i} \cdot \left(1 - e^{-p_i \cdot b_i \cdot (1-\alpha_i) \cdot w_{t_i}}\right) +$$

$$\sum_{j=3}^{4} \frac{a_j}{1-\alpha_j} \cdot \left(\left(1 - \left(\left(1 + b_j \cdot w_{t_j}\right) \cdot e^{-b_j \cdot w_{t_j}}\right)^{p_j \cdot (1-\alpha_j)}\right) +\right.$$

$$\left(1-\left(\left(1+b_j\cdot w_{t_j}+{b_j}^2\cdot\frac{{w_{t_j}}^2}{2}\right)\cdot e^{-b_j\cdot w_{t_j}}\right)^{p_j\cdot(1-\alpha_j)}\right) \tag{4.1}$$

where

$a_1 = a\cdot h_1;\ a_2 = a\cdot h_2 = a\cdot(.2-h_1); a_3 = a\cdot h_3;\ a_4 = a\cdot h_4 = a\cdot(.2-h_3);$

$\sum_{k=1}^{4} a_k = a;\ b_1 = b_2;\ b_3 = b_4;$

$W_{t(1)} = W_t\cdot g_1;\ W_{t(2)} = W_t\cdot g_2 = W_t\cdot(.8-g_1);$

$\sum_{k=1}^{4} W_{t(k)} = W_t$

4.1. Software Reliability Data Analysis Technique

Prior to employing software reliability modelling approach to software reliability data it is important to check out whether the reliability data shows growing behaviour with time, If the data does not show growing behaviour with time, then software reliability modelling should not be applied for estimation reliability of the system Kanoun et al. (1997) and they used Laplace test for this purpose. Let n_i represents the number of faults corrected in time $i(i = 1,2,3,\dots,k)$, then Laplace factor can be obtained as

$$u_k = \frac{\sum_{i=1}^{k}(i-1)n_i - \frac{k-1}{2}\sum_{i=1}^{k} n_i}{\sqrt{\frac{k^2-1}{2}\sum_{i=1}^{k} n_i}} \tag{4.2}$$

Negative values represent a reliability-growth, otherwise positive values suggest a reliability decline, and the range of values below positive 2 and above negative 2 represents stablility.

4.2 Model Validation and Evaluation

We evaluate the performance (i.e., goodness-of-fit) of the models under comparison using mean-square-fitting-error, Bias, Variation, and root-mean-square-prediction-error metrics. The smaller the metric value the better (Kapur et al., 2011; Shatnawi, 2016).

- **The mean-square-fitting-error** (MSE) or long-term predictions is defined by (Lyu, 1996) as

$$MSE = \frac{1}{k}\sum_{i=1}^{k}\left(\hat{m}_{t_i} - x_i\right)^2 \quad (4.3)$$

where $\hat{m}_{t_i}$ is the mean number of faults at time t_i estimated by a model, x_i is the expected number of faults corrected at time t_i, and k is the number of observations.

- **The Akaike Information Criterion** (AIC) is defined by Khoshogoftaar & Woodcock (1991) as

$$AIC = -2 \times log(\max of\ likelihood\ function) + 2 \times N \quad (4.4)$$

where *N* is the number of the parameters used in the model.

- ***Bias***. is given as,

$$Bias = \frac{1}{k}\sum_{i=1}^{k} PE_i \quad (4.5)$$

where $PE_i = observed_i - estimated_i$

- ***Variation*** is given as,

$$Variation = \sqrt{\frac{1}{k-1}\sum_{i=1}^{k}(PE_i - Bias)^2} \quad (4.6)$$

- ***Root-Mean-Square-Prediction-Error*** *(RMSPE)* is a way of measuring how good a model is over the actual data.

$$RMSPE = \sqrt{(Bias^2 + Variation^2)} \quad (4.7)$$

Other than these metrics used in comparing models. (Musa et al., 1987) have suggested the following attributes for choosing a model:

- **Capability.** The model should possess 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 should not be estimation should not be too complex and is easy to understand and apply even for persons without extensive mathematical background.

4.3 Parameter Estimation Techniques

To carry out the estimation part of software modelling, we employ the statistical-package-for-social-sciences (SPSS) based on the nonlinear-regression-technique.

Chapter 5

Data Analyses and Model Comparisons

For model validation and evaluation, we consider a simple case in which the software system composed of two reused software components and two newly developed. Three reliability data collected from actual software-development-project, have been analyzed and employed to show the applicability of the proposed modeling-approach. As these reliability data-set were extensively studied (Yamada et al., 2000; Tamura et al., 2006; Kapur et al., 2009a; Kapur et al., 2009b; Shatnawi, 2013), direct comparison with the work of other can be made. In this study, we treat these reliability data-set as they were observed from the testing phase after confirmation of the integration of all software components.

5.1 First-Software-Development-Project

The first software reliability data had been obtained during 19 weeks of testing/debugging of PL/I application program test data of size 1,317,000 lines of code (LOC). Over the course of 19 weeks, 47.65 CPU hours were consumed, and 328 software faults were reported (for more details see Appendix A).

Figure 5.1 traces the Laplace-trend-test. Prior to the 17th week, the trend test values indicate stable reliability. It is clearly seen reliability fluctuations for the 5th week and 6th week. However these fluctuations does not last for long time. Therefore, we should not pay attention to it. Stable reliability trend indicates that the corrective actions have no visible effect on reliability.

In such situation the testing and debugging team must introduce new test sets. However, after the 16th week, the trend become-stable. In such situation the system is used less or the reason behind this may also be due to unrecorded faults. Therefore, the testing and debugging team must take particular care (Kanoun et al., 1997).

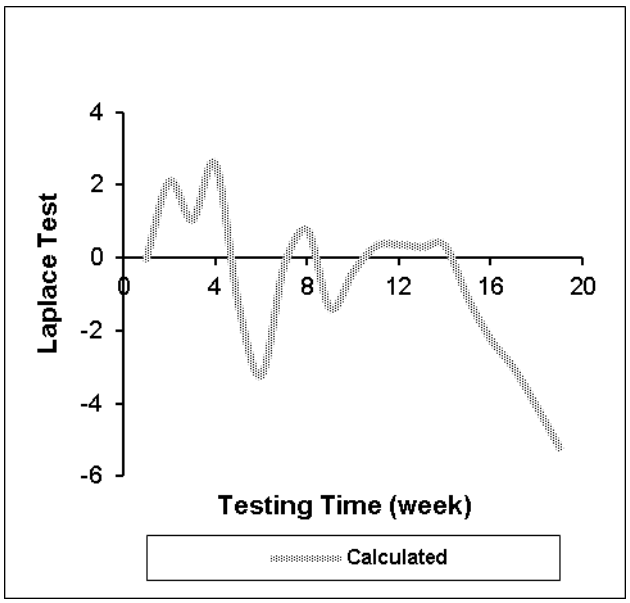

Fig. 5.1 Laplace-test-data-trend

The resultant parameter estimation and the goodness-of-fit metrics in terms of *MSE*, *AIC*, *Bias*, *Variation*, and *RMSPE* of the models under comparison are tabulated in Table 5.1. According to Table 5.1, we can see that the logistic function has lower *MSE*, *AIC*, *bias*, *variation*, and *RMSPE* metric values among the testing-effort functions under comparison. Therefore, the comparison criteria favour the logistic testing-effort function and, hence, adopted for further evaluation. It is worth mentioning that the exponential function fails to give any plausible estimation results.

Table 5.1 Parameter estimation and comparison criteria metrics results

Testing-Effort Functions Under Comparison	Parameter Estimation			Comparison Criteria				
	D	*C*	*r*	MSE	AIC	Bias	Variation	RMSPE
Exponential	*	*	—	*	*	*	*	*
Rayleigh	49.32	0.014	—	5.237	52.68	0.560	2.28	2.347
Weibull	799.22	0.002	1.115	15.65	45.67	3.362	4.12	5.318
Logistic	54.84	0.226	13.03	1.629	47.93	-0.062	1.310	1.311

* the function fails to give any plausible result
— the component is not part of the corresponding function

The fitting of the testing-effort functions under comparison to the actual non-cumulative and cumulative testing-effort are graphically illustrated in Figure 5.2, 5.3 respectively. From the Figure 5.3, we can observe that the logistic testing-effort function provides a better fit than the other functions under comparison. Therefore, the logistic testing-effort function provides more accurate description of resource consumption than other functions.

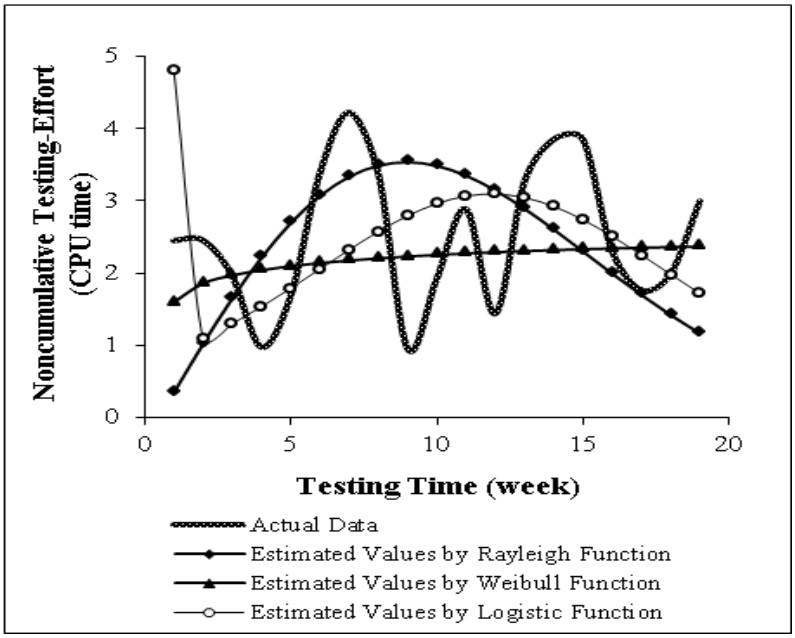

Fig. 5.2 Non-cumulative testing-effort curves

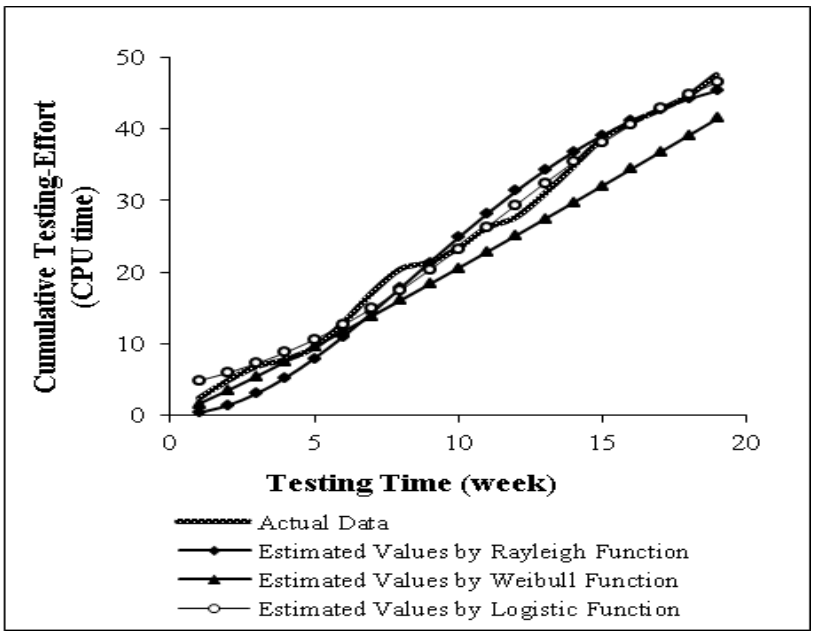

Fig. 5.3 Cumulative testing-effort curves

The resultant parameter estimation of the proposed modeling apparoch is tabulated in Table 5.2. According to the estimated values in Table 5.2, the probability of perfect-debugging '*p'* of the faults encountered in the re-used components is lower than that encountered in the newly developed components. According to the estimated values in Table 5.2, the error introduction or generated rate '□' the debugging-process doesn't introduce any error for re-used components but it is not the case for newly developed components.

It is calcualted that a total of 409 faults were detected in the 19 weeks including 108 faults were generated, and out of them, only 333 were perfectly debugged and corrected in the same debugging-period as shown in Table 5.3. Besides, our proposed modelling approach in Table 5.4reveals the number of faults detected and how many of them were corrected for each type respectively.

As the software system composed of four components two of them are re-used and the other two are newly developed. Tables 5.5 and 5.6, reveal very important results that can be of immense-help to the developer and decision maker such as the initial fault-content, amount of testing-effort expenditure, total number of fault-content included the introduced errors due to imperfect-debugging environment, number of fault introduced, and number fault corrected for each of these modules.

Table 5.2 Parameter estimation

Model	Parameter							Proportion Parameter			
	a	$b_{1,2}$	$p_{1,2}$	$\alpha_{1,2}$	$b_{3,4}$	$p_{3,4}$	$\alpha_{3,4}$	h_1	h_3	g_1	g_3
Proposed	300.64	.383	.501	0	.771	.988	.350	.200	.691	.142	.118

Table 5.3 Plausible Results

	PL/I application program				
	Initial Fault-Content a	*Effort Consumed* W_t	*Total Fault-Content* a_{W_t}**	*Fault Introduced* $a_{W_t} - a$	*Fault Corrected* m_{W_t}
Estimated	300.64	46.55	408.81	108.17	333.37
Reported*	328	47.65	—	—	—

* *refers to software reliability data (DS-I) in the Appendix*
— *the component is not given in DS-I*
** $a_{W_t} = a + \alpha \cdot m_{W_t}$ given in (Kapur et al., 2009, 2011)

Table 5.4 Fault Type Content Results

Model	PL/I application program		
	Easy Faults	*Medium Faults*	*Hard Faults*
Detected	60.128	298.27	50.42
Corrected	24.313	258.64	50.41

Table 5.5 Calculated Results for Re-used Components

Re-used Components									
component 1					component 2				
a	W_t	a_{W_t}	$a_{W_t} - a$	m_{W_t}	a	W_t	a_{W_t}	$a_{W_t} - a$	m_{W_t}
60.13	6.61	60.13	0	24.31	0	2.70	0	0	0

Table 5.6 Calculated Results for Newly-Developed Components

Newly-Developed Components									
component 3					component 4				
a	W_t	a_{W_t}	$a_{W_t} - a$	m_{W_t}	a	W_t	a_{W_t}	$a_{W_t} - a$	m_{W_t}
207.7	31.75	298.22	90.52	258.6	32.77	5.49	50.42	17.65	50.41

The fitting of the proposed modeling approach to the actual non-cumulative and cumulative software reliability data are graphically illustrated in Figure 5.4, 5.5 respectively. From Figure 5.5, we can observe that the estimated values (cumulative number of corrected faults , as a result of our propsed model) are very close to the actual software reliability data and therefore fits the data excellently well.

As faults are corrected, the fault-correction intensity which represents the fault-correction rate per fault per testing-effort weeks tends to drop and reliability tends to increase. The changeability of this rate shown in Figure 5.4 may be attributed to the imperfect-debugging phenomenon or fault debugging complexity. Resulting in a step increase in fault-correction intensity and a step decrease in

reliability. Therefore, we have in Figure 5.4 a step increase or decrease in fault-correction intensity. As fault-correction intensity is an alternative way of expressing reliability and software reliability is the inverse of fault-correction intensity. Therefore, both Figures commonly called reliability growth curves.

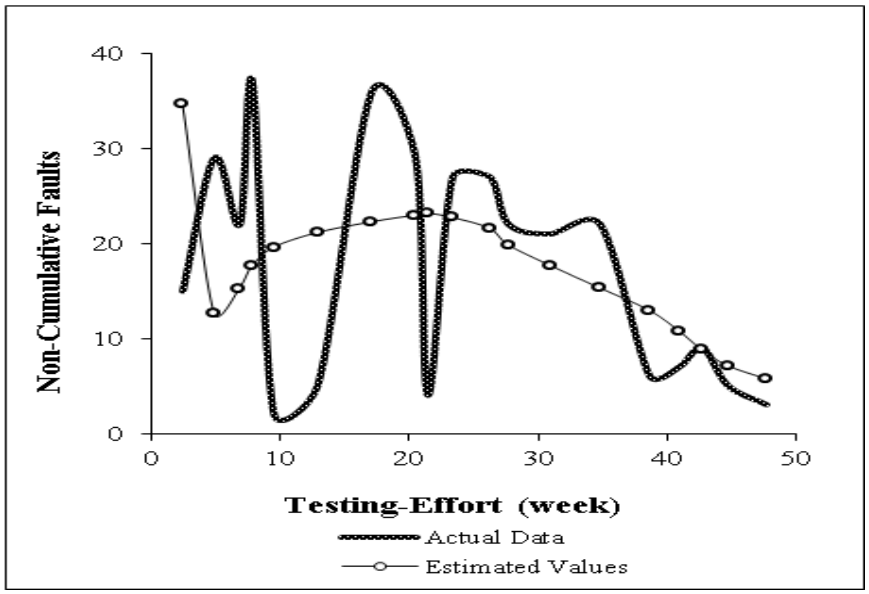

Fig. 5.4 Non-cumulative reliability data curves

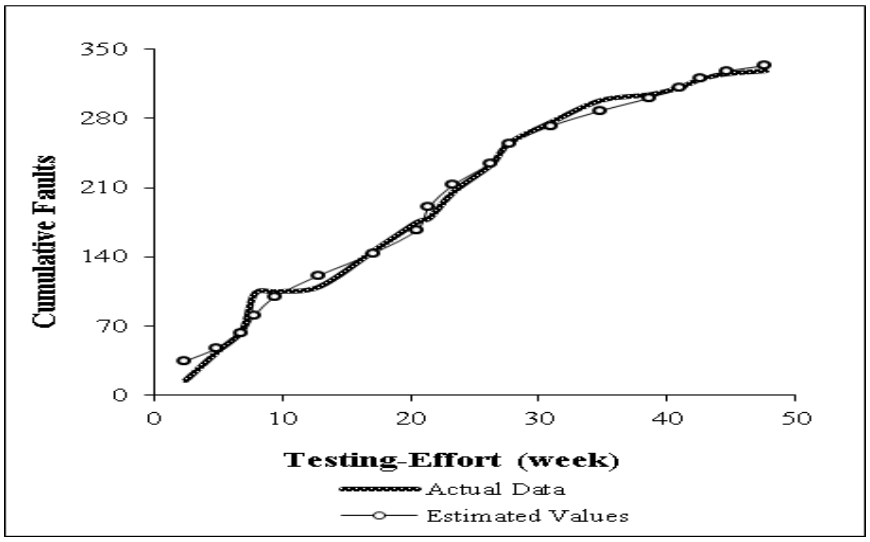

Fig. 5.5 Cumulative reliability data curves

The resultant goodness-of-fit metrics in terms of MSE, AIC, bias, variation and RMSPE of the proposed model compared with other existing models are given in Table 5.7. As given in Table 5.7, the overall values of MSE, bias, variance and RMSPE for the proposed model are the lowest. As results of comparison, we may conclude that the proposed modelling approach fits better than the other models under comparison for this actual software reliability data.

Table 5.7 Comparison criteria metric results

Models under Comparison	Comparison Criteria				
	MSE	AIC	Bias	Variation	RMSPE
Yamada et al. (2000)	124.94	209.67	-0.231	11.48	11.49
Tamura et al. (2006)	351.72	140.15	—	—	—
Goswami et al. (2007)	38263.9	—	174.89	8105.18	8107.06
Kapur et al. (2009a)	41.70	—	0.448	112.41	112.42
Kapur et al. (2009a)	48.00	—	0.329	77.16	77.16
Kapur et al. (2009b)	35.72	—	0.07	37.69	37.69
Shatnawi (2013)	92.51	241.15	-0.5362	9.866	9.881
Proposed	78.12	232.16	-0.1796	9.079	9.081

— *the metric is not measured by the corresponding model*

5.2 Second-Software-Development-Project

The Second software reliability data had been obtained during 38 weeks of testing and debugging of space shuttle software system. Over the course of 38 weeks, 2456.4 CPU hours were consumed, and 231 software faults were reported (for more details see Appendix *B*).

Figure 5.8 traces the Laplace-trend-test. The values of the trend test are completely negative from beginning. There are fluctuations, but this fluctuation does not drastically affect the reliability and the reliability growing behaviour. Reliability growth may result from a period during which the system is underutilized; it may also be caused by unrecorded faults. Therefore, the testing and debugging team must take particular care (Shatnawi, 2016).

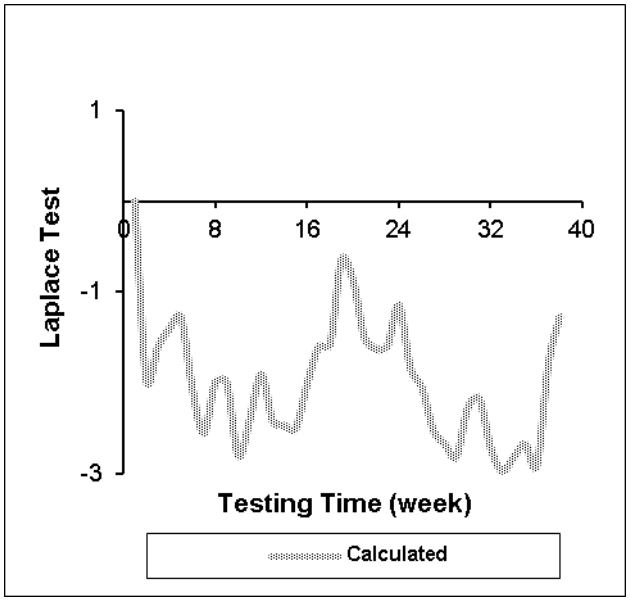

Fig. 5.6 Laplace test data trend

The resultant parameter estimation and the goodness-of-fit metrics in terms of *MSE*, *AIC*, *Bias*, *Variation* and *RMSPE* of the testing-effort functions under comparison are tabulated in Table 5.9 According to Table 5.9, we can see that the logistic function has lower MSE, AIC, bias, variation, and RMSPE metric values among the testing-effort functions under comparison. Therefore, the comparison criteria favour the logistic testing-effort function and, hence, adopted for further evaluation. It is worth mentioning that the exponential function fails to give any plausible estimation results.

Table 5.8 Parameter estimation and comparison criteria metrics results

Testing-Effort Functions Under Comparison	Parameter Estimation			Comparison Criteria				
	d	*c*	*R*	MSE	AIC	Bias	Variation	RMSPE
Exponential	*	*	—	*	*	*	*	*
Rayleigh	2241	.0040	—	23666.8	1626.2	53.5	146.16	155.66
Weibull	5063	.0084	1.1639	4225.18	654.3	10.4	65.026	65.85
Logistic	2836	.0985	10.49	8982.06	1110.4	-8.2	95.69	96.04

** the function fails to give any plausible result*
— the component is not part of the corresponding function

The fitting of the testing-effort functions under comparison to the actual non-cumulative and cumulative testing-effort are graphically illustrated in Figures 5.7 ,5.8 respectively. From Figure 5.8, we can observe that the Weibull testing-effort function provides overall a better fit than the other functions under comparison. Therefore, the Weibull function provides more accurate description of resource consumption than other functions.

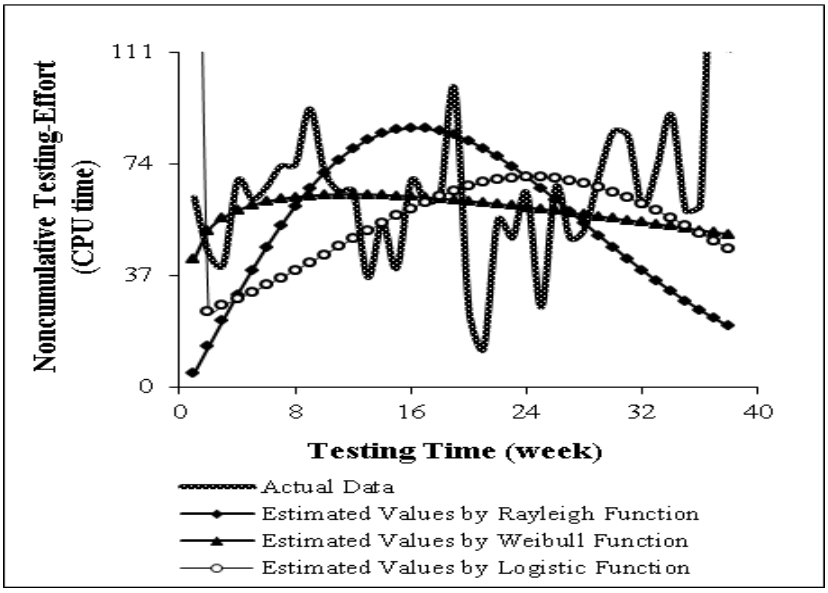

Fig. 5.7 Non-cumulative testing-effort

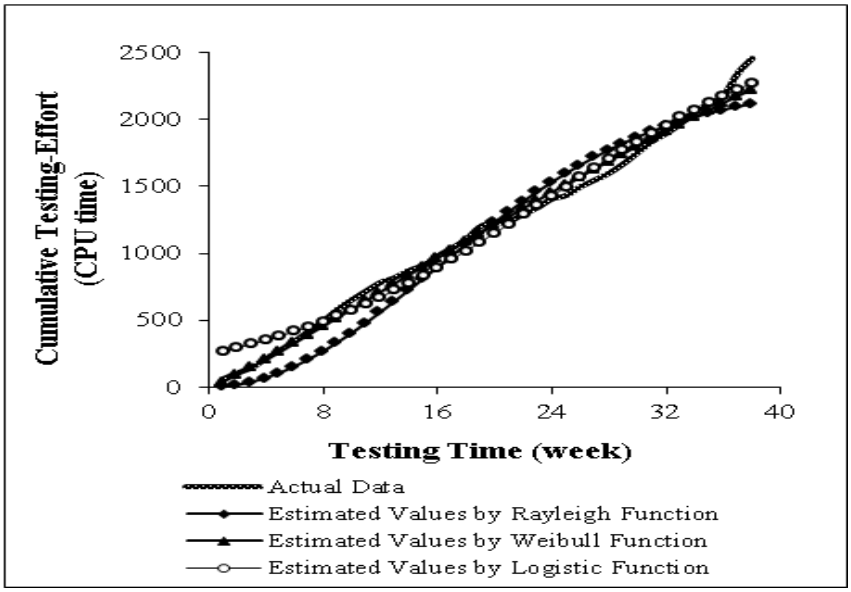

Fig. 5.8 Cumulative testing-effort curves

According to the estimated values in Table 5.9, the probability of perfect-debugging or debugging-efficiency '*p*' of the faults encountered in the re-used components is higher than that encountered in the newly developed components. According to the estimated values in Table 5.9, the error introduction or generated rate '□' per detected fault in the re-used components is higher than that in the newly developed components.

It is estimated that a total of 380 faults were detected in the 38 weeks months including 108 faults were generated, and out of them, only 218 were perfectly debugged and corrected in the same debugging period as shown in Table 5.10. Besides, our proposed modelling approach in Table 5.10, reveals the number of faults detected and how many of them were corrected for each type respectively.

Tables 5.11 and 5.12, reveal very important results that can be of immense-help to the software developer and decision maker such as the initial fault-content, amount of testing-effort expenditure, total number of fault-content included the introduced errors due to imperfect-debugging environment, number of fault introduced, and number of fault corrected for each of these modules.

Table 5.9 Parameter estimation

Model	Parameter							Proportion Parameter			
	A	$b_{1,2}$	$p_{1,2}$	$\alpha_{1,2}$	$b_{3,4}$	$p_{3,4}$	$\alpha_{3,4}$	h_1	h_3	g_1	g_3
Proposed	355.39	.077	.815	.288	.011	.545	.009	.185	.659	.049	.027

Table 5.10 Plausible Results

	Space Shuttle Software System				
	Initial Fault-Content a	*Effort Consumed* W_t	*Total Fault-Content* a_{W_t}**	*Fault Introduced* $a_{W_t} - a$	*Fault Corrected* m_{W_t}
Estimated	355.39	2226.57	380.48	25.08	217.56
Reported*	231	2456.40	—	—	—

* *refers to software reliability data (DS-II) in the Appendix*
— *the component is not given in DS-II*
** $a_{W_t} = a + \alpha \cdot m_{W_t}$ given in (Kapur et al., 2009, 2011)

Table 5.11 Fault Type Content Results

Model	Space Shuttle Software System		
	Easy Faults	*Medium Faults*	*Hard Faults*
Detected	94.95	234.29	51.24
Corrected	82.88	125	9.68

Table 5.12 Calculated Results for Re-used Components

Re-used Components									
component 1					component 2				
a	W_t	a_{W_t}	$a_{W_t} - a$	m_{W_t}	a	W_t	a_{W_t}	$a_{W_t} - a$	m_{W_t}
65.75	336.2	72.79	7.04	58.42	5.33	109.1	22.16	19.04	24.46

Table 5.13 Calculated Results for Newly-Developed Components

Newly-Developed Components									
component 3					component 4				
a	W_t	a_{W_t}	$a_{W_t} - a$	m_{W_t}	a	W_t	a_{W_t}	$a_{W_t} - a$	m_{W_t}
234.2	1721	234.3	0.087	125	50.11	60.12	51.24	1.125	9.68

The fitting of the proposed model to the actual non-cumulative and cumulative software reliability data are graphically illustrated in Figure 5.9 ,5.10 respectively. From Figure 5.10, we can observe that the estimated values are very close to the actual software reliability data and therefore fits the data excellently well.

The changeability of this rate shown in Figure 5.9 may be attributed to the imperfect-debugging phenomenon or fault debugging complexity. Resulting in a step increase in fault-correction intensity and a step decrease in reliability. Therefore, we have in Figure 5.9 a step increase or decrease in fault-correction intensity. As fault-correction intensity is an alternative way of expressing reliability and software reliability is the inverse of fault-correction intensity. Therefore, both Figures commonly called reliability growth curves.

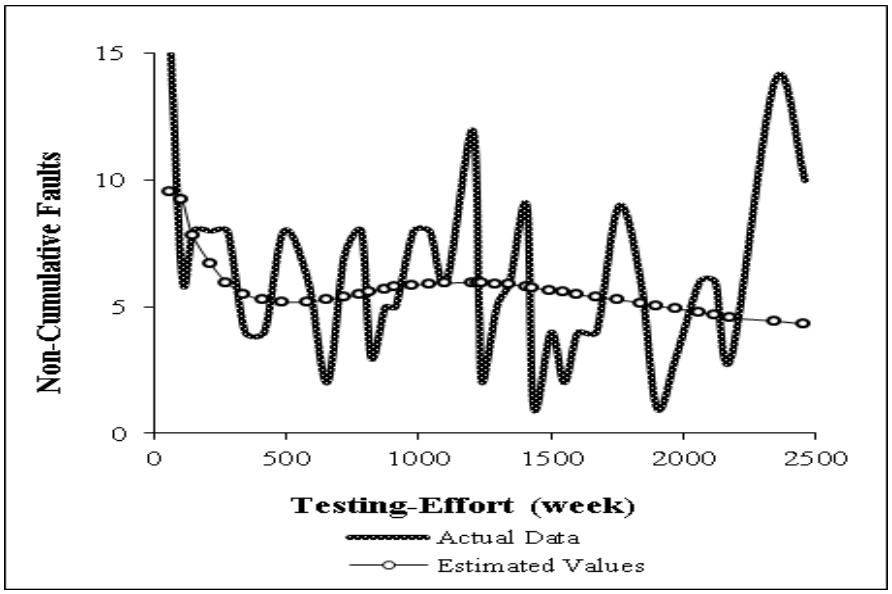

Fig. 5.9 Non-cumulative reliability data curves

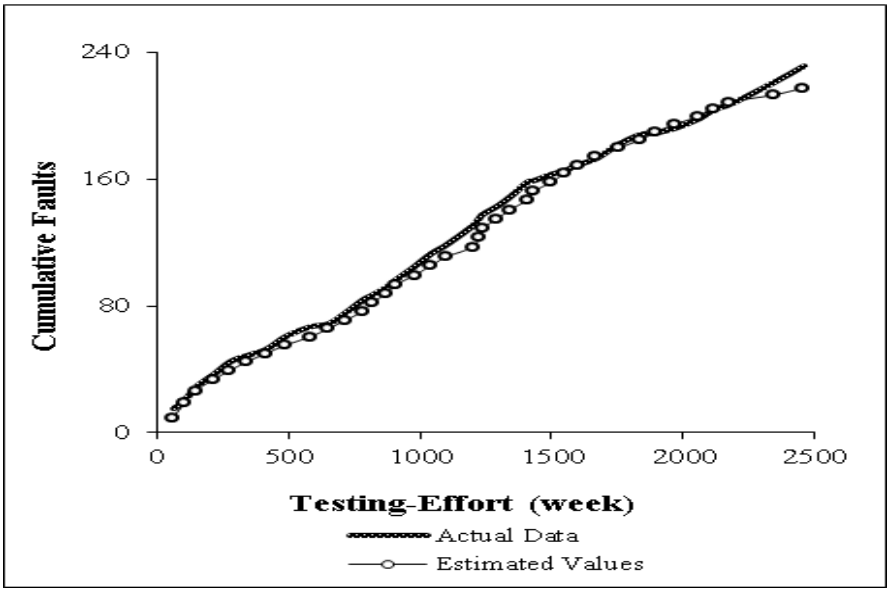

Fig. 5.10 Cumulative reliability data curves

The resultant goodness-of-fit metrics in terms of MSE, AIC, bias, variation and RMSPE of the proposed model compared with other existing models are given in Table 5.14. As given in Table 5.14, the values of MSE and AIC for the proposed model are the lowest.

Since lower values are better, the comparsion criteria favour the proposed modeling approach.

As a results , we may conclude that the proposed modelling approach fits better than the other models under comparison for this actual software reliability data.

Table 5.14 Comparison criteria metric results

Models under Comparison	Comparison Criteria				
	MSE	AIC	Bias	Variation	RMSPE
Yamada et al. (2000)	859.80	258.23	—	—	—
Kapur et al. (2004)	663.34	215.72	—	—	—
Proposed	40.65	203.16	4.838	4.208	6.413

— *the metric is not measured by the corresponding model*

5.3 Third Software Development Project

The third software reliability data had been obtained during 35 months of testing and debugging of various stages of formal test and integration of a Defense, Ground Based Radar software system of size 124K LOC. During the period, 1846.92 CPU hours were consumed, and 1301 software faults were reported (for more details see Appendix *C*).

Figure 5.11 traces the Laplace trend test. Trend test indicates reliability decay, which is expected and considered normal at the start of a new activity. Since the decay has lasted for short period, it may neglect it. However, during the period from the 7^{th} month till 20^{th} month, we have seen some fluctuations, but this

fluctuation has no effect on reliability's growing behavior. After that the behavior reliability grew monotonically. Such reliability growth that follows a reliability decline is usually accepted (Shatnawi, 2016).

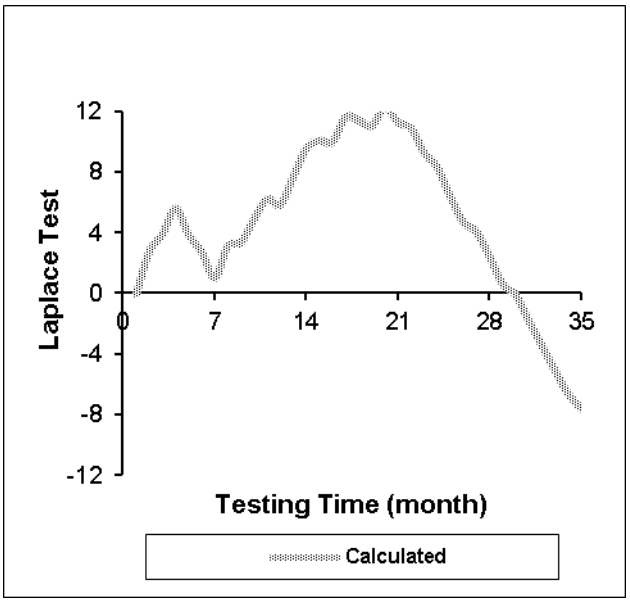

Fig. 5.11 Laplace test data trend

The parameter estimation and the goodness-of-fit metrics in terms of *MSE*, *AIC*, *Bias*, *Variation* and *RMSPE* of the testing-effort functions under comparison are tabulated in Table 5.3.1. According to Table 5.3.1, we can see that the Weibull function has lower MSE, AIC, bias, variation, and RMSPE metric values among the testing-effort functions under comparison. Therefore, the comparison criteria favour the Weibull function and, hence, adopted for further evaluation. It is worth mentioning that the exponential function fails to give any plausible estimation results.

Table 5.15 Parameter estimation and comparison criteria metrics results

Testing-Effort Functions Under Comparison	Parameter Estimation			Comparison Criteria				
	D	*C*	*r*	MSE	AIC	Bias	Variation	RMSPE
Exponential	*	*	—	*	*	*	*	*
Rayleigh	2873	.00173	—	663.99	294.8	-1.461	26.102	26.143
Weibull	2670	.00077	2.07	633.23	285.0	-2.037	25.448	25.529
Logistic	2067	.161	38.64	2179.8	372.2	-5.033	47.094	47.362

* *the function fails to give any plausible result*
— *the component is not part of the corresponding function*

The fitting of the testing-effort functions under comparison to the actual non-cumulative and cumulative testing-effort are graphically illustrated in Figures 5.12 ,5.13 respectively. From Figure 5.13, we can observe that the Weibull testing-effort function provides overall a better fit than the other functions under comparison. Therefore, the Weibull function provides more accurate description of resource consumption than other functions.

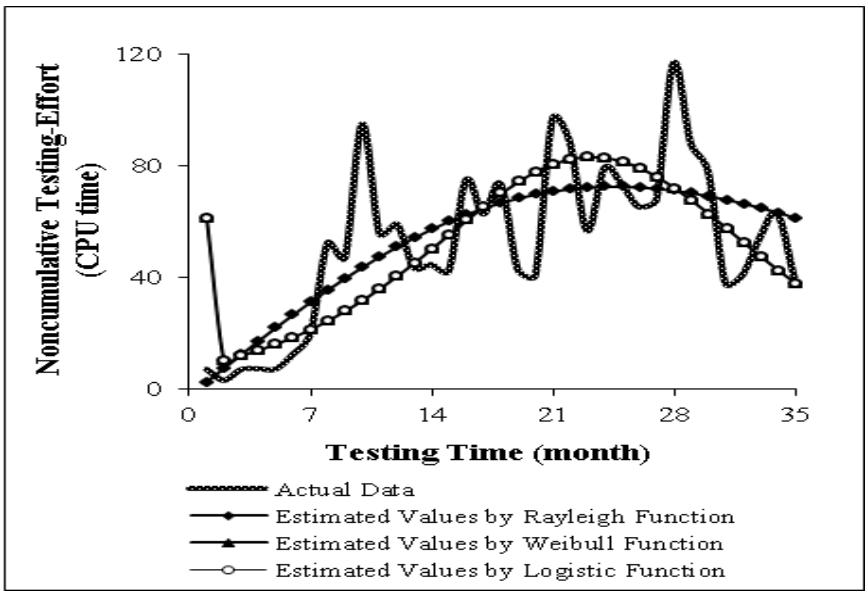

Fig. 5.12 Non-cumulative testing-effort curves

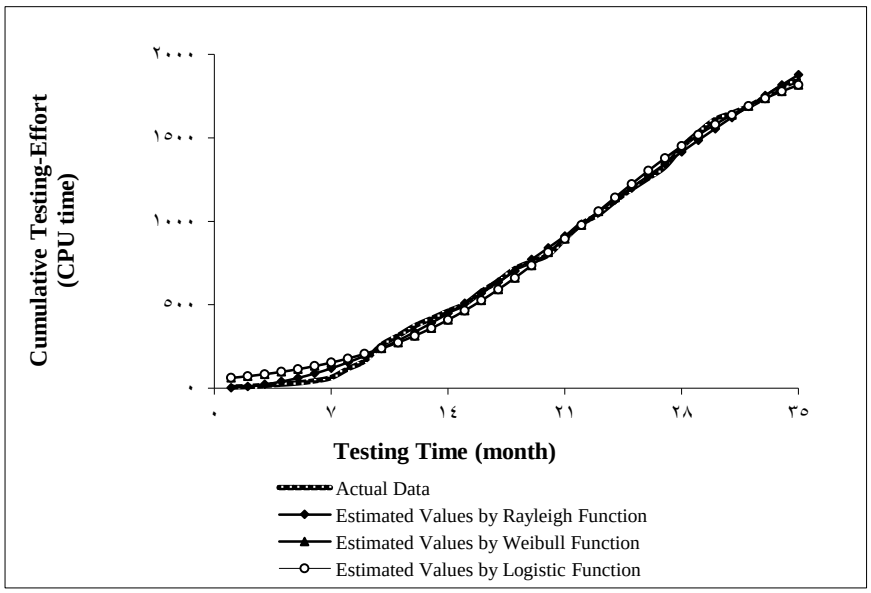

Fig. 5.13 Cumulative testing-effort curves

The resultant parameter estimation tabulated in Table 5.3.16. According to the estimated values in Table 5.16, the probability of perfect-debugging or debugging-efficiency '*p'* of the faults encountered in the re-used components is "certain" while that was not the case for the newly developed components. According to the estimated values in Table 5.16, the error introduction or generated rate '□' per detected fault in the newly developed components is higher than that in the re-sued components.

It is estimated that a total of 1356 faults were detected in the 35 months including 45 faults were generated, and out of them, only 1297 were perfectly debugged and corrected in the same debugging period as shown in Table 5.17. Besides, our proposed modelling approach in Table 5.18 reveals the number of faults detected and how many of them were corrected for each type respectively.

Tables 5.19 and 5.20, reveal very important results that can be of immense-help to the software developer and decision maker such as the initial fault-content, amount of testing-effort expenditure, total number of fault-content included the introduced errors due to imperfect-debugging environment, number of fault introduced, and number fault corrected for each of these modules.

Table 5.16 Parameter estimation

Model	Parameter							Proportion Parameter			
	A	$b_{1,2}$	$p_{1,2}$	$\alpha_{1,2}$	$b_{3,4}$	$p_{3,4}$	$\alpha_{3,4}$	h_1	h_3	g_1	g_3
Proposed	1311.32	.074	1	.001	.009	.833	.042	.167	.101	.197	.255

Table 5.17 Plausible Results

	Ground Based Radar Software System				
	Initial Fault-Content a	*Effort Consumed* W_t	*Total Fault-Content* a_{W_t}**	*Fault Introduced* $a_{W_t} - a$	*Fault Corrected* m_{W_t}
Estimated	1311.32	1873.65	1356.21	44.99	1297.34
Reported*	1301	1846.92	—	—	—

* *refers to software reliability data (DS-III) in the Appendix*

— *the component is not given in DS-II*

**$a_{W_t} = a + \alpha \cdot m_{W_t}$ given in (Kapur et al., 2009, 2011)

Table 5.18 Fault Type Content Results

Model	Ground Based Radar Software System		
	Easy Faults	*Medium Faults*	*Hard Faults*
Detected	262.50	956.18	137.54
Corrected	233.94	942.08	121.33

Table 5.19 Calculated Results for Re-used Components

Re-used Components									
component 1					component 2				
a	W_t	a_{W_t}	$a_{W_t} - a$	m_{W_t}	a	W_t	a_{W_t}	$a_{W_t} - a$	m_{W_t}
218.99	369.11	219.21	0.219	219.21	43.27	5.62	43.29	0.015	14.74

Table 5.20 Calculated Results for Newly-Developed Components

Newly-Developed Components									
component 3					component 4				
a	W_t	a_{W_t}	$a_{W_t} - a$	m_{W_t}	a	W_t	a_{W_t}	$a_{W_t} - a$	m_{W_t}
916.61	1021.14	956.18	39.57	942.08	132.44	477.78	137.54	5.096	121.33

The fitting of the proposed model to the actual non-cumulative and cumulative software reliability data are graphically illustrated in Figures 5.14 ,5.15 respectively. From Figure 5.15, we can observe that the estimated values are very close to the actual software reliability data and therefore fits the data excellently well.

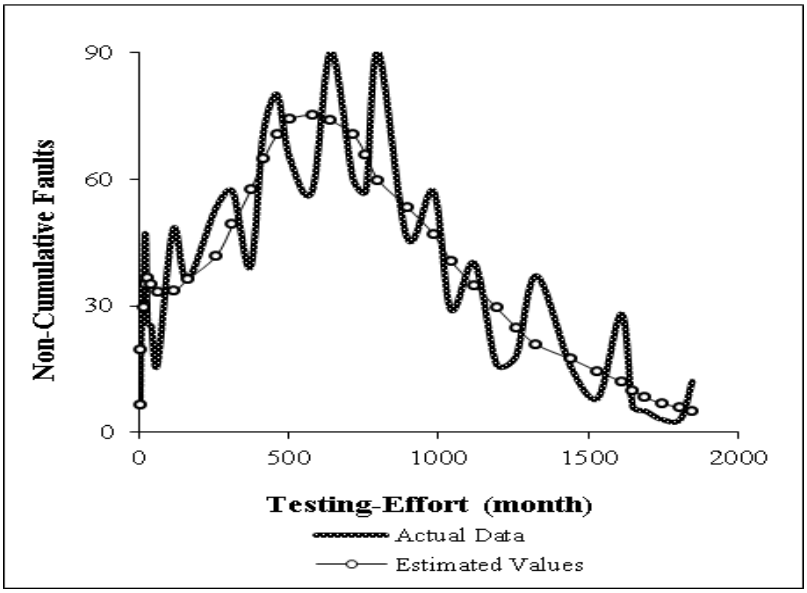

Fig. 5.14 Non-cumulative reliability data curves

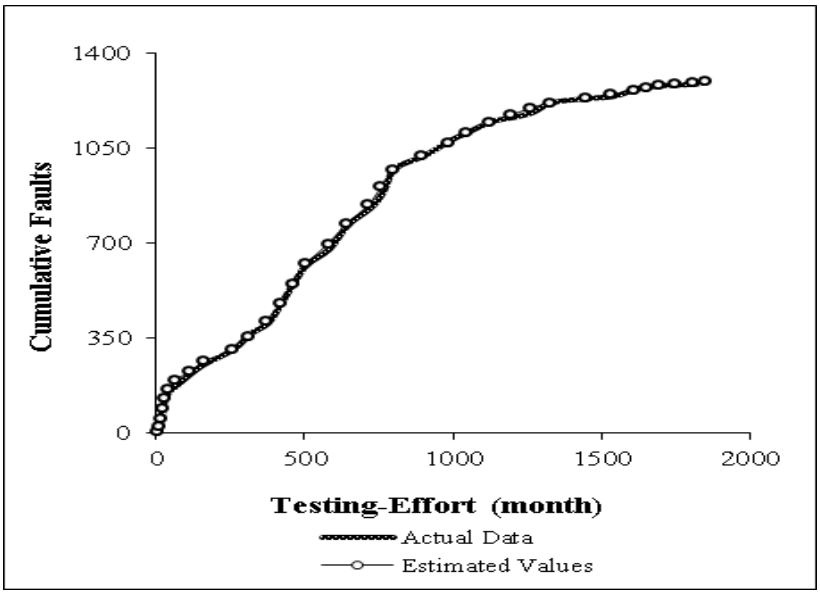

Fig. 5.15 Cumulative reliability data curves

The resultant goodness-of-fit metrics of the proposed model compared with other existing models are given in Table 5.21. As given in Table 5.21, the values of MSE, bias, variation and RMSPE for the proposed model are the lowest.

Since lower values are better, the comparsion criteria favour the proposed modeling approach.

Table 5.21 Comparison criteria metric results

Models under Comparison	Comparison Criteria				
	MSE	AIC	Bias	Variation	RMSPE
Yamada et al. (2000)	3095.19	—	8.262	3379.89	3379.91
Kapur et al. (2009a)	82.21	—	0.695	659.64	659.64
Kapur et al. (2009a)	137.95	—	0.457	1846.45	1846.45
Proposed	80.87	256.21	0.066	9.1239	9.1241

— *the metric is not measured by the corresponding model*

Chapter 6

Concluding Remarks and Future Work

The importance of modelling and analysis of software debugging-process or fault-correction phenomena in a distributed development environment has been well recognized and many studies have addressed this problem. The aim of most of these endeavors has been to develop analytical models for the fault correction phenomena in order to compute quantities of interest such as the number of faults corrected, effort consumption, number of faults introduced due to imperfect-debugging activities, number of remaining faults and the software reliability function.

In this thesis, we have explored the importance of testing-resource and imperfect-debugging phenomenon, through an integrated component-based modelling approach for distributed development environment. Therefore, this attempt, could be of immense-help to the developer in controlling and monitoring the testing-process closely and effectively allocating the resources to reduce the testing-cost and to meet the given reliability-requirements.

Therefore, this study provides a new insight into the development of software reliability modelling in distributed development environment. It has also demonstrated the integration of a set of existing non-homogenous Poisson process-based software-reliability-model. The resultant integrated component-based modelling approach has been validated and compared with other existing non-homogenous Poisson process based software reliability models by applying

them on three software-reliability-data. The results were very plausible and yields insightful interpretations for the resources expenditures during the testing and debugging phase.

Today is a period of transition for neural network technology. As neural network can be described in a mathematical form and they have a significant advantage over analytical models, because they require only software reliability data history as input and no assumptions. The extension of our integrated component-based modelling approach to demonstrate the applicability of the neural network approach to the modelling of software reliability in distributed development environment, is an ongoing challenge that stimulates more future research efforts.

References

Ahmad, N. Khan, MG. Rafi, LS. **A Study of Testing-Effort Dependent Inflection S-shaped Software Reliability Growth Models with Imperfect Debugging**. *International Journal of Quality & Reliability Management*, 27(1), 2010, pp. 89-110.

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

Boehm, B. and Basili, VR. **Software Defect Reduction Top 10 List**, *Computer*, 34(1), 2001, pp. 135–137.

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

Chmiel, R. and Loui, MC. **Debugging**. *SIGCSE Bulletin*, 2004, 36(1), pp. 17-21.

Goel, AL. 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.

Goswami, DN. Jha, PC. Johri, P. and Kapur, R. **Software Reliability Growth Model for Distributed Environment Incorporating two types of Imperfect Debugging.** *Proceedings of 3rd International Conference on Reliability and Safety Engineering*, 2007, pp. 308-319.

Huang, CY. Kuo, SY. and Lyu, MR. **An Assessment of Testing-Effort Dependent Software Reliability Growth Models**. *IEEE Transactions on Reliability*, 56(2), 2007, pp. 198-211.

Idris, K. **The PNZ Software Reliability Model Revisited**, M.Sc. Thesis, Al al-Bayt University, 2009 (unpublished).

IEEE Computer Society, ***IEEE Standard Glossary of Software Engineering Terminology***: IEEE Standard 610.12-1990.

Jones, C. **Applied software measurement: Global Analysis of Productivity and Quality**, McGraw-Hill, 3rd edition, 2008.

Kapur, PK. Johri P. and Singh Ompal. **Modeling Software Reliability Growth in Distributed Environment Using Unified Approach**. In Proceedings of the 3rd National Conference; INDIACom-2009 Computing For Nation Development, February 26 – 27, 2009a.

Kapur, PK. Khatri, SK. Johri P. and Singh O. **Incorporating Concept of Two Types of Imperfect Debugging for Developing Flexible Software Reliability Growth Model in Distributed Development Environment.** *(JTES) Delving: Journal of Technology and Engineering Sciences*, 1(1), 2009b, pp.9-19.

Kapur, PK. Goswami, DN. and Gupta, A. **Software Reliability Growth Model with Testing Effort Dependent Learning Function for Distributed Systems**. *International Journal of Reliability, Quality and Safety Engineering*, 11(4), 2004, pp. 365–377.

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

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

Kapur, PK. Pham, H. Anand, S. and Yadav, K. **A Unified Approach for Developing Software Reliability Growth Models in the Presence of Imperfect Debugging and Error Generation**. IEEE Transactions on Reliability, 60(1), 2011, pp. 331-340.

Kapur, PK. Pham, H. Gupta, A. and Jha, PC. **Software Reliability Assessment with OR Applications**. Springer-Verlag, 2011.

Kanoun, K. Kaaniche, M. and Laprie, J-C. **Qualitative and Quantitative Reliability Assessment**. *IEEE Software*, 14(2), 1997, pp. 77-87.

Khoshogoftaar, TM. and Woodcock, TG. **Software Reliability Model Selection: A Case Study**. *Proc. of the Int'l Symposium on Software Reliability Engineering*, 1991, pp. 183-191

Kuo SY, Huan CY, Lyu MR. **Framework for Modelling Software Reliability using various Testing-Effort and Fault-Detection Rates**. *IEEE Transactions on Reliability*, 50(3), 2011, pp. 310-320.

Lavinia, A, Dobre, C, Pop, F, and Cristea, V. **A Failure Detection System for Large Scale Distributed Systems**. *International Journal of Distributed Systems and Technologies*, 2(3), 2011, pp. 64–87

Lim, WC. **Effects of Reuse on Quality, Productivity and Economics**. *IEEE Transactions on Software*, 11(5), 1994, pp. 23–30.

Lin, C-T. **Analyzing the Effect of Imperfect Debugging on Software Fault Detection and Correction Processes via a Simulation Framework,** *Mathematical and Computer Modelling*, 54(11), 2011, pp. 3046–3064.

Lyu, MR. **Software Reliability Engineering: A Roadmap**, *Future of Software Engineering* (FOSE'07), IEEE Computer Society, Washington, DC, USA, 2007, pp. 153–170.

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

Misra, PN. **Software Reliability Analysis**. *IBM Systems Journal*, 22(3), 1983, pp.262–279.

Musa, JD. Software Reliability Engineering: More Reliable Software Faster and Cheaper, 2nd Edition, AuthorHouse, 2004.

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

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

O'Dell DH. The debugging Mindset. ACM Queue. 2007, 15(1), pp. 1-20

Ohba, M. **Software Reliability Analysis Models.** *IBM Journal of Research and Development*, 28(4), 1984, pp. 428–443.

Ohtera, H. and Yamada, S. **Optimal Allocation and Control Problems for Software-Testing Resource**. *IEEE Transactions on Reliability*, 39(2), 1990, pp. 171-176.

Pfleeger, SA. and Atlee, JM. **Software Engineering: Theory and Practice**. 3rd edition, Prentice-Hall, 2006.

Shatnawi, O. **Measuring Commercial Software Operational Reliability: an Interdisciplinary Modelling Approach**. *Eksploatacja i Niezawodno □□ - Maintenance and Reliability*, 16(4), 2014, pp.585–594.

Peng, R. Li, YF. Zhang, WJ. Hu, QP. **Testing Effort Dependent Software Reliability Model for Imperfect Debugging Process considering both Detection and Correction**. *Reliability Engineering and System Safety*, 126, 2014, pp. 37–43

Pham, H. **System Software Reliability**. Springer, 2006.

Shatnawi, O. **An Integrated Software Reliability Modelling Approach to Imperfect Fault Debugging Activities**, *International Journal of Systems Assurance Engineering and Management*, (communicated, 2017).

Shatnawi, O. **An Integrated Framework for Developing Discrete-Time Modelling in Software Reliability Engineering**, *Quality and Reliability Engineering International*, 32(8), 2016, pp. 2925.2943.

Shatnawi, O. **A Software Reliability Model for Distributed Systems.** *Al Manarah Journal for Research and Studies*, 13(6), 2007, pp. 201-214.

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

Shatnawi, O. **Testing-Effort Dependent Software Reliability Model for Distributed Systems**. *International Journal of Distributed Systems and Technologies*, 4(2), 2013, pp. 1-14.

Tassey, G. **The Economic Impacts of Inadequate Infrastructure for Software Testing**, *Technical Report RTI Project Number 7007.011*, **National Institute of Standards and Technology**, Gaithersburg, MD, USA, 2002.

Tamura, Y. Yamada, S. and Kimura, M. **A Reliability Assessment Tool for Distributed Software Development Environment based on Java and J/Link**. *European Journal of Operational Research*. 175(1), 2006, pp. 435–445.

Wang, Z. Tang, K. and Yao, X. M**ulti-objective Approaches to Optimal Testing Resource Allocation in Modular Software Systems**. *IEEE Transactions on Reliability*, 59(3), 2010, 563–575.

Yamada, S. Ohba, M. and Osaki, S. **S-shaped Reliability Growth Modelling for Software Error Detection**, *IEEE Transactions on Reliability*, 32(5), 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.

Yamada, S. **Software Reliability Modeling: Fundamentals and Applications**. Springer, 2014.(book)

Yamada, S. Ohtera, H. and Narihisa, H. **Software Reliability Growth Models with Testing-Effort**. *IEEE Transactions on Reliability*, 35(1), 1985, pp.19-23.

Yamada, S. Tamura, Y. and Kimura, M. **A Software Reliability Growth Model for a Distributed Development Environment.** *Electronics and Communications* in Japan-Part 3, 83(12), 2000, pp. 1–8.

Zhao, X. Wang, T. Liu, E. and Clapworthy, GJ. **Web Services in Distributed Information Systems: Availability, Performance and Composition**. *International Journal of Distributed Systems and Technologies*, 1(1), 2010, pp. 1–16.

الملخص

تحولت دورة تطوير البرمجيات الحالية إلى بيئة بناء موزعة بسبب تطور تكنولوجيا الشبكات وزيادة الطلب على الموارد المشتركة لتخفيض التكلفة.

قليل من الدراسات السابقة في هندسة عول البرمجيات التي حاولت نمذجة احصاء عدد الأخطاء في البرمجية وتصحيحها في بيئة التطوير الموزعة ويمكن أن يعزى السبب إلى التعقيد الذي ينطوي عليه وضع نظم موزعة على نطاق واسع. كما أن تصحيح الأخطاء في البرمجيات في بيئة بناء موزعة لم يتم تناوله بدقة في الدراسات لحالية، على الرغم من أنه من الاهداف الأساسية في هذه الصناعات.

هذا الهدف دفعنا إلى وضع نهج جديد يعتمد على الجهد المبذول في اختبار موثوقية البر.مجيات المطورة في بيئة موزعة وغير مثالية التصحيح.(لا يتم تصحيح الاخطاء كليا" فيها) وفي هذه الدراسة تم وصف عمليتي تصحيح الأخطاء والجهد المستهلك في عملية الاختبار من خلال عملية بواسون غير المتجانسة (**NHPP**) و اقترانات اختبار الجهد المبذول (**Testing_effort Functions**)

منهج النمذجة الذي تم بناؤه في دراستنا هذه يمكن أن يكون مواتيا للحصول على عدة نماذج باتباع منهجية واحدة وبالتالي يمكن ان يكون تحقق نظري لدراسة النماذج العامة دون اتخاذ فرضيات عديدة . ولذلك، فإنه يوفر منصة مشتركة متكاملة لنماذج اختبار عول البرمجيات في بيئات تصحيح أخطاء البرمجيات المثالية وغير المثالية.

على حد علمنا هذه المرة الأولى التي يتم فيها بناء مثل هذا النهج للأنظمة الموزعة ،والذي يصف العلاقة بين الوقت /التقويم، والجهد المستهلك في الاختبار، وعملية تصحيح الأخطاء في بيئات تصحيح غير مثالية . ويعتبر منهجا" مناسبا" جدا لتطوير

(object-oriented systems)

ولإثبات قابلية تطبيق منهج النمذجة المقترح تم استخدام بيانات حقيقية أجريت عليها مشاريع نمذ جة ظاهرة عول البرمجيات في الدراسات السا بقة . و كانت نتائج دراستنا مشجعة بالمقارنة مع نتائج نماذج آخرى تم بناؤها في بيئة مماثلة.

Appendices

Appendix A

Dataset I: collected during 19 weeks of testing, 47.65 CPU hours were consumed, and 328 software faults were corrected during debugging (Ohba, 1984).

Test time (month)	Execution time (CPU hour)	Detected faults	Cumulative execution time (CPU hour)	Cumulative detected faults
1	2.45	15	2.45	15
2	2.45	29	4.9	44
3	1.96	22	6.86	66
4	0.98	37	7.84	103
5	1.68	2	9.52	105
6	3.37	5	12.89	110
7	4.21	36	17.1	146
8	3.37	29	20.47	175
9	0.96	4	21.43	179
10	1.92	27	23.35	206
11	2.88	27	26.23	233
12	1.44	22	27.67	255
13	3.26	21	30.93	276
14	3.84	22	34.77	298
15	3.84	6	38.61	304
16	2.3	7	40.91	311
17	1.76	9	42.67	320
18	1.99	5	44.66	325
19	2.99	3	47.65	328

Appendix B

Dataset II: collected during 38 weeks of testing, 2456.4 CPU hours were consumed, and 231 software faults were detected during debugging (Misra, 1983).

Test time (month)	Execution time (CPU hour)	Detected faults	Cumulative execution time (CPU hour)	Cumulative detected faults
1	62.5	15	62.5	15
2	44	6	106.5	21
3	40	8	146.5	29
4	68	8	214.5	37
5	62	8	276.5	45
6	66	4	342.5	49
7	73	4	415.5	53
8	73.5	8	489	61
9	92	6	581	67
10	71.4	2	652.4	69
11	64.5	7	716.9	76
12	64.7	8	781.6	84
13	36	3	817.6	87
14	54	5	871.6	92
15	39	5	910.6	97
16	68	8	978.6	105
17	61	8	1039.6	113
18	62.6	6	1102.2	119
19	98.7	12	1200.9	131
20	25	5	1225.9	136
21	12	2	1237.9	138
22	55	5	1292.9	143
23	49	6	1341.9	149
24	64	9	1405.9	158
25	26	1	1431.9	159
26	66	4	1497.9	163
27	49	2	1546.9	165
28	52	4	1598.9	169
29	70	4	1668.9	173
30	84.5	9	1753.4	182
31	83	6	1836.4	188
32	60	1	1896.4	189
33	72.5	3	1968.9	192
34	90	6	2058.9	198
35	58	6	2116.9	204

36	60	3	2176.9	207
37	168	14	2344.9	221
38	111.5	10	2456.4	231

Appendix C

C – Dataset III: collected during 35 months of testing, 1846.92 CPU hours were consumed, and 1301 software faults were detected during debugging (Brooks and Motely, 1980).

Test time (month)	Execution time (CPU hour)	Detected faults	Cumulative execution time (CPU hour)	Cumulative detected faults
1	7.25	7	7.25	7
2	3.17	22	10.42	29
3	7.08	32	17.5	61
4	7.33	47	24.83	108
5	7.25	26	32.08	134
6	12.58	25	44.66	159
7	19.92	16	64.58	175
8	52.5	48	117.08	223
9	47.18	36	164.26	259ض
10	95.1	53	259.36	312
11	55.75	57	315.11	369
12	59.25	39	374.36	408
13	43.58	71	417.94	479
14	44.75	80	462.69	559
15	42.33	65	505.02	624
16	75	57	580.02	681
17	62.83	90	642.85	771
18	73.58	60	716.43	831
19	42.75	57	759.18	888
20	40.67	90	799.85	978
21	96.75	46	896.6	1024
22	88.58	57	985.18	1081
23	56.75	29	1041.93	1110
24	79.25	40	1121.18	1150
25	73.5	16	1194.68	1166
26	65.33	18	1260.01	1184
27	67.83	37	1327.84	1221
28	116.92	15	1444.76	1236
29	88.08	8	1532.84	1244
30	78.08	28	1610.92	1272
31	37.92	6	1648.84	1278
32	41.08	5	1689.92	1283
33	54.5	3	1744.42	1286
34	63	3	1807.42	1289
35	39.5	12	1846.92	1301