Release Policy, Change-Point Concept, and Effort Control through Discrete-Time Imperfect Software Reliability Modelling

Omar Shatnawi
Computer Science Department
Al al-Bayt University
Mafraq 25113, Jordan

P.K. Kapur
Center for Interdisciplinary Research, Amity University
Nodia 201313, India

Mohd Taib Shatnawi
Al-Huson University College
Al-Balqa' Applied University
Irbid 21110, Jordan

ABSTRACT

Nonhomogeneous Poisson process based software reliability models play an important role in developing software systems and enhancing the performance of computer software. As software reliability grows on the basis of the execution of computer test runs. Nonhomogeneous Poisson process type of discrete-time software reliability models, or difference equations, is more realistic and often provides better fit than their continuous-time counterparts. Since discrete-time model conserves the properties of the continuous-time model, the estimation of its parameter would be simpler and more accurate. In this paper, we explore the importance of testing resource and imperfect debugging phenomenon consideration in software reliability growth modeling. The resultant model is very useful for the reliability analysis as the measure of reliability is computed considering the distribution of testing-effort, influence of the testing efficiency and the changes of the testing process. Using the resultant model, testing-effort control, change-point concept and optimal release policy have also been investigated. Therefore, this paper thus provides a new insight into development of discrete-time modelling in software reliability engineering, that could be of immense help to the software project manager in monitoring and controlling the testing process closely and effectively allocating the resources in order to reduce the testing cost and to meet the given reliability requirements.

General Terms

Software Reliability Engineering.

Keywords

Software reliability, software testing, imperfect debugging, nonhomogeneous Poisson process, change-point, effort control, software release policy.

1. INTRODUCTION

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, it is very important to ensure the quality or reliability of the underlying software in the sense that it performs its functions, i.e., designed and built for. Software development process is often called software development life cycle, 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 life cycle 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. Software failure is estimated to cost American industries USD 60 billion every year [1]. It can be discovered before or after system delivery, during testing, or during operation and maintenance. Testing and debugging 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 [2]. Several methods exist for studying the cost and schedule of software; however, reliability is the only measure of software quality. Software reliability is the probability that a software product will function failure-free for a specified period of time in a specified environment [3].

Mathematical modelling based on stochastic and statistics theories are useful to describe the software fault debugging phenomenon and to evaluate the reliability quantitatively [4]. Software reliability growth analysis based on statistical correlations of real faults detection data collected during testing, is, one of the approaches to conducting statistical reliability assessment. Because of the complexity of the factors influencing the debugging process, the quantities associated with reliability are random variables, and hence reliability models are based on the stochastic process. Nonhomogeneous Poisson process forms one of the main classes of the existing software reliability models, due to its mathematical tractability and wide applicability. They are useful in describing fault removal process, providing trends such as reliability growth and fault content. Software reliability models consider the debugging process as a counting process characterized by the value function of a nonhomogeneous Poisson process.

Software reliability models based on nonhomogeneous Poisson process are generally classified into two groups. The first group contains models, which use the execution time or calendar time. Such models are called continuous-time models. The second group contains models, which use the test cases as a unit of fault removal period. Such models 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, the test case 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. Therefore, discrete-time models are more realistic and often provide better fit than their continuous-time counterparts [2-9].

The achieved software reliability during testing phase is highly related to the amount of development resources, that is, testing-effort, spent during debugging process. A testing-effort function describes the distribution or consumption pattern of testing resources during the testing period. Hence it is very important to track the software reliability growth with respect to the testing-effort expenditure [10-20].

When the testing is in its late stage and the product release date is approaching, an assessment is done to review the progress of testing and requirement for the additional efforts is worked out to meet the pre-specified reliability targets. However, through software reliability models one can describe how to increase the fault removal by accelerating the testing-effort intensity [2, 3, 21-32]. In the testing-effort control problem we estimate the requirement for additional efforts for the aspiration level to achieve.

A number of discrete-time nonhomogeneous Poisson process based software reliability models in the software reliability engineering literature, assume diverse testing and debugging environments. However, most of them are based upon constant or monotonically increasing fault removal rate. In practice, as the testing grows, so does the skill and efficiency of the testers. With the introduction of new testing strategies and new test cases, there comes a change in fault removal rate. The time point where the change in removal curve appears is termed as change-point [23]. The concept of change point is relatively recent in the software reliability modeling. A number of reasons are associated for modeling under change-point concept such as changes in the testing and debugging environment, testing strategy, software complexity and size, fault density, skill, and motivation and constitution of the testing and debugging team. Modeling using the change-point concept provides answers to the number of questions related to the changing scenarios during testing phase [2].

Reliability, scheduled delivery and cost are the three main quality attributes for almost all software. The primary objective of the software developer's to attain them at their best values, then only they can obtain long-term profits and make a brand image in the market for longer survival.

Software users demand faster deliveries, cheaper software and quality product, whereas software developers aim at minimizing their development cost, maximizing the profit margins and meeting the competitive requirements. The resulting situation calls for tradeoffs between conflicting objectives of software users' requirements with the developers. As a course of best alternative the developer management must determine optimally when to stop testing and release the software system to the user focusing on the users' requirements, simultaneously satisfying their own objectives. Such a problem is known as software release time decision problem in the literature of software reliability engineering [2]. Most of the software release time decision problems formulated considering cost, reliability or failure intensity and number of faults removed require the exact computation of cost function coefficients, amount of available resources, reliability or failure intensity aspiration levels, etc. The values of these quantities besides some static factors depend on a number of factors, which are non-deterministic in nature. However, in the actual software development, the manager must spend and control the testing resources with a view to minimizing the total software cost and satisfying reliability requirements rather than only minimizing the cost [4]. By the application of software release time problem managers are able to predict the delivery date of the software with a predetermined level of quality measure to be attained by that time.

The remainder of this paper is organized as follows. Section 2 presents how beginning with very simple assumptions, nonhomogeneous Poisson process type of discrete-time software reliability models, are gradually made more realistic with the incorporation of imperfect debugging and testing effort expenditures. The resultant model adopts the number of test cases as a unit of fault removal period, which is countable and more appropriate measure than machine time or calendar time used in continuous-time software reliability models. Section 3 provides the testing effort trade-off with respect to aspiration level for the debugging process. Section 4 incorporates the change point in fault removal growth phenomenon. Optimal software release policies for software reliability growth phenomenon under imperfect debugging are investigated in Section 5.

2. SOFTWARE RELIABILITY MODELLING

Software reliability modeling is an important and fast developing field of research. Its importance is due to the increased dependency on computer software systems in our daily life and to the fact that the software system cannot be made error free. The most important software development problem is building software to customer demands are that it be more reliable, built faster, and built cheaper. The scientific disciple software reliability engineering concerns at scheduling and systematizing the software development process and have a control over the various stages of software development using its tools, methods and process to engineer quality software. Major roles of software reliability engineering lies in assuring, controlling and measuring the software reliability, the key attribute of software quality during the testing and operational phases of software development life cycle and locating the time point of an appropriate balance between the cost of testing and fixing bugs during operational use. The tools of software reliability engineering known as software reliability models are used successfully to evaluate software quantitatively, develop test cases, schedule status, resource optimization, to count the number of faults remaining in the software and estimate and predict the software reliability during testing and operational environment [2].

2.1 Nonhomogeneous Poisson Process Models

Nonhomogeneous Poisson processes are an important class of software reliability models. A discrete counting process $(N_n)_{n\geq 0}$ is said to be nonhomogeneous Poisson process with mean value function m_n, i.e., expected cumulative number of faults detected by n test cases, if it satisfies the following conditions:

- There are no failures experienced or fault detected at $n = 0$, i.e., $N_{n=0} = 0$.
- The counting process has independent increments, i.e., for any collection of the numbers of test cases $n_1, n_2, \dots, n_k$ where $(0 < n_1 < n_2 < \cdots < n_k)$, the k random variables $(N_{n_1}, N_{n_2} - N_{n_1}, \dots, N_{n_k} - N_{n_{k-1}})$ 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

$$\mathbb{Pr}\left[N_{n_i} - N_{n_j} = x\right] = \frac{(\lambda_n)^x}{x!} e^{-\lambda_n}, \; x = 0,1,2 \dots \tag{1}$$

Therefore, if we define the expected value number of faults N_n whose mean value function is known as m_n, then an software reliability model based on nonhomogeneous Poisson process can be formulated as a Poisson process

$$\mathbb{Pr}[N_n = x] = \frac{(m_n)^x}{x!} \cdot e^{-(m_n)},\ x = 0,1,2 \ldots \qquad (2)$$

The intensity function λ_n (or the mean value function m_n) is the basic building block of all the nonhomogeneous Poisson process models existing in the software reliability engineering literature.

2.2 Model Development

Earlier software reliability modelling approaches were developed based on calendar time or execution time as the unit of fault-detection/removal period and either assume that the consumption rate of testing resources is constant, or do not explicitly consider the testing effort and its effectiveness. Later, many models were proposed describing the relationship among the testing time (calendar time), testing-effort expenditure and the number of software faults detected. However, very limited approaches were proposed in discrete time owning to the complexity of exact form solution of the mean value function [25-29].

One of the pioneering attempts was the discrete software reliability growth model with testing effort due to Kapur et al. [30]. The model describes the relationship among the executed number of test runs, testing-effort expenditure and the number of software faults detected in an ideal debugging environment.

Under the basic assumption that the expected cumulative number of faults detected between the n^{th} and the $(n+1)^{th}$ test cases is proportional to the number of faults remaining after the execution of the n^{th} test run, satisfies the following difference equation

$$\lambda_n = \frac{m_{n+1} - m_n}{\delta} = b(a - m_n) \qquad (3)$$

here a is a fault content and b constant fault removal per remaining fault per test case.

To solve this difference equation we employ the method of probability generating function.

Multiply both sides by z^n and summing over n from 0 to ∞ , we get:

$$\sum_{n=0}^{\infty} z^n m_{n+1} - \sum_{n=0}^{\infty} z^n m_n = a\delta b \sum_{n=0}^{\infty} z^n - \delta b \sum_{n=0}^{\infty} z^n m_n$$

$$\frac{1}{z}\sum_{n=0}^{\infty} z^{n+1} m_{n+1} - \sum_{n=0}^{\infty} z^n m_{(n)} = a\delta b \sum_{n=0}^{\infty} z^n - \delta b \sum_{n=0}^{\infty} z^n m_n$$

$$\sum_{n=0}^{\infty} z^{n+1} m_n - \sum_{n=0}^{\infty} z^{n+1} m_n = a\delta b \sum_{n=0}^{\infty} z^{n+1} - \delta b \sum_{n=0}^{\infty} z^{n+1} m_n$$

$$\sum_{n=0}^{\infty} z^{n+1} m_{n+1} = a\delta b \sum_{n=0}^{\infty} z^{n+1} + (1-\delta b)\sum_{n=0}^{\infty} z^{n+1} m_n$$

$$(z^1 m_1 + z^2 m_2 + z^3 m_3 + \cdots) = a\delta b(z^1 + z^2 + z^3 + \cdots) + (1-\delta b)(z^1 m_0 + z^2 m_1 + z^3 m_2 + \cdots) \qquad (4)$$

Comparing the coefficients of like powers of z on both sides and under the boundary condition $m_{n=0} = 0$, we get

$$m_1 = a\delta b + (1-\delta b)m_0 = a\delta b + 0 = a(1-(1-\delta b)^1)$$

$$m_2 = a\delta b + (1-\delta b)m_1 = a\delta b + (1-\delta b)a(1-(1-\delta b)^1) = a(1-(1-\delta b)^2)$$

$$m_3 = a\delta b + (1-\delta b)m_2 = a\delta b + (1-\delta b)a(1-(1-\delta b)^2) = a(1-(1-\delta b)^3) \qquad (5)$$

By mathematical induction the closed form solution is given by:

$$m_n = a(1-(1-\delta b)^n) \qquad (6)$$

and hence, the failure intensity can be obtained as follows

$$\lambda_n = ab(1-\delta b)^n \qquad (7)$$

However, the assumption of perfect debugging makes software models mathematically simple, but on the other hand, far from reality. In fact, Imperfect debugging of faults were discovered in almost every company [33]. Therefore, in actual software development environment, the debugging team may not be able to remove the fault perfectly and the detected fault may get replaced by another fault or may remain. While the first phenomenon is known as fault generation, the second is called imperfect fault debugging. In case of fault generation the fault content increases as the testing progresses and debugging results in introduction of new faults while removing old ones But in case of imperfect fault debugging the fault content is not changed, but just because of incomplete understanding of the software, the detected fault is not removed completely [31, 32, 34]. Later, they modified the model to incorporate the first phenomenon of the imperfect debugging [8]. However, the fact that both types of imperfect debugging may occurs simultaneously cannot be ignored. Therefore, we further extend the model to integrate the effect of both phenomena of imperfect debugging.

Assuming that fault removal rate per additional fault removed $b\ (0 < b < 1)$ is reduced by the probability of perfect debugging $p\ (0 < p \le 1)$ and a constant proportion of removed faults are generated $\alpha\ (0 \le \alpha < 1)$ while removal, the difference equation describing the imperfect debugging phenomenon can be modelled as

$$\lambda_n = \frac{m_{n+1} - m_n}{\delta} = bp(a - (1-\alpha)m_n) \qquad (8)$$

Solving the above difference equation, following using the probability generating function method, one can get the approximate solution

$$m_n = \frac{a(1-\beta_n)}{(1-\alpha)} \qquad (9)$$

here $\beta_n = \left(1 - \delta pb(1-\alpha)\right)^n$

and hence, the failure intensity can be obtained as follows

$$\lambda_n = apb\beta_n \qquad (10)$$

The equivalent continuous-time model corresponding to the Equation (9) is given by

$$m_t = \frac{a(1-\beta_t)}{(1-\alpha)} \qquad (11)$$

here $\beta_t = \exp(-bp(1-\alpha)t)$

which can be derived as a limiting case of discrete-time model substituting $t = n\delta$, $\lim_{x\to 0}(1+x)^{1/x} = exp$, and taking limit $\delta \to 0$.

2.3 Testing Effort Expenditure Model

Let W_n denotes the debugging effort expenditure in test cases $(0, n_i]$, β denotes the consumption rate of the debugging effort expenditure, and α denotes the amount of debugging effort to be eventually consumed. Under the basic assumption that the expected cumulative debugging effort expenditures between the n^{th} and the $(n+1)^{th}$ test cases to be proportional to the remaining amount of testing effort expenditure, satisfies the following difference equation

$$w_{n+1} = \frac{W_{n+1}-W_n}{\delta} = \beta_{n+1}(\alpha - W_n) \qquad (12)$$

when $\beta_{n+1} = \beta$, we have a discrete exponential curve

$$W_n = \alpha(1-(1-\delta\beta)^n) \qquad (13)$$

and hence, the current testing effort expenditure after the execution of the n^{th} test case,, can be obtained as follows

$$w_n = \alpha\beta(1-\delta\beta)^n \qquad (14)$$

when $\beta_{n+1} = \frac{\beta}{\alpha}W_{n+1}$, we have a discrete logistic curve

$$W_n = \frac{\alpha}{1+m(1-\delta\beta)^n} \qquad (15)$$

here m is defined as $W_{n=0} = \frac{\alpha}{1+m}$

and hence, the current testing effort expenditure after the execution of the n^{th} test case, can be obtained as follows

$$w_n = \frac{\alpha m\beta\,(1-\delta\beta)^n}{(1+m(1-\delta\beta)^n)^2} \qquad (16)$$

when $\beta_{n+1} = \beta(n+1)$, we have a discrete exponential curve

$$W_n = \alpha(1-\prod_{i=0}^{n}(1-i\delta\beta)) \qquad (17)$$

and hence, the current testing effort expenditure after the execution of the n^{th} test case,, can be obtained as follows

$$w_n = \alpha\delta\beta n\left(\prod_{i=0}^{n-1}(1-i\delta\beta)\right) \qquad (18)$$

2.4 Model Formulation

To incorporate the testing effort expenditures into software reliability modelling, we assuming that the mean number of faults detected between the n^{th} and the $(n+1)^{th}$ test cases by the current testing-effort is proportional to the mean number of faults not detected, this model can be expressed by the following difference equation

$$\frac{m_{n+1}-m_n}{\delta} = w_n bp(a-(1-\alpha)m_n) \qquad (19)$$

Solving the above difference Equation (19), following using the probability generating function method, and after tedious algebraic manipulations, one can get the approximate solution

$$m_n = \frac{a(1-\beta_{w_n})}{(1-\alpha)} \qquad (20)$$

here $\beta_{w_n} = \prod_{i=0}^{n}(1-\delta pb(1-\alpha)w_i)$

and hence, the failure intensity can be obtained as follows

$$\lambda_n = a\delta bpw_n\beta_{w_{n-1}} \qquad (21)$$

here $\beta_{w_{n-1}} = \prod_{i=0}^{n-1}(1-\delta pb(1-\alpha)w_i)$

This modelling approach clearly illustrates the benefit of formulating discrete-time imperfect debugging models with respect to the amount testing effort expenditures per executed test run. Therefore, the above resultant model m_n, can depict more accurate utilization of test resources as well as reliability prediction at the time of software release.

The equivalent continuous-time model corresponding to the Equation (20) is given by

$$m_t = \frac{a(1-\beta_{w_t})}{(1-\alpha)} \qquad (22)$$

here $\beta_{w_t} = \exp(-bp(1-\alpha)w_t)$

which can be derived as a limiting case of discrete-time model substituting $t = n\delta$, $\lim_{x\to 0}(1+x)^{1/x} = exp$, and taking limit $\delta \to 0$.

2.5 Parameter Estimation

Parameters estimation is of primary concern in software reliability measurement. Software reliability data can be collected in the form of testing effort $W_j\,(W_1 < W_2 < W_3 < \cdots < W_k)$ consumed in test cases $(0, n_i]$, where $i = 1,2,\ldots,k$ in which $m_j\,(0 < m_1 < m_2 < \cdots < m_k)$ faults are detected. Then the parameters (α, β, m) in the discrete testing-effort function are estimated by the method of least squares as follows

$$\begin{aligned} &minimize\ \sum_{i=1}^{k}\left(W_i - \widehat{W}\right)^2 \\ &subject\ to \qquad \widehat{W}_k = W_k \end{aligned} \qquad (23)$$

where $\widehat{W}_j = W_j$ implies that the estimated value of the testing effort is equal to the actual value.

Using these estimated parameter values, we estimate the parameters in the imperfect software reliability model by the method of maximum likelihood estimation. It is one of the most popular and useful statistical method for fitting a mathematical model to some data. The Likelihood function L for the unknown parameters with the mean value function m_n takes on the form

$$L\left(a,b,p,\alpha \middle| (W_i, x_i)\right) = \prod_{i=1}^{k}\frac{\left(m_{n_i}-m_{n_i-1}\right)^{x_i-x_{i-1}}}{(x_i-x_{i-1})!}exp\left(-\left(m_{n_i}-m_{n_i-1}\right)\right) \qquad (24)$$

Taking natural logarithm, we get

$$\ln L = \sum_{i=1}^{k}(x_i - x_{i-1})\ln\left(m_{n_i}-m_{n_i-1}\right) - \left(m_{n_i} - m_{n_i-1}\right)\sum_{i=1}^{k}\ln(x_i - x_{i-1}) \qquad (25)$$

The MLE of the unknown parameters can be obtained by maximizing the likelihood function subject to the parameters constraints. Based on the estimation, we can estimate the current behavior of the testing process, predict the future behavior of the testing process, and make decisions regarding resource allocation and release time.

3. TESTING-EFFORT CONTROL AND MANAGEMENT

The management of a software development project has time schedules for testing and release of software. During testing, often the management is not satisfied with the progress of the debugging and the growth of the reliability curve. Then there arises need for employing additional testing-effort in terms of new techniques, testing tools, more manpower so as to remove more faults than what could be possibly achieved with the current level of debugging efforts in a pre-specified time interval. Therefore, we suggest a debugging effort trade-off with respect to aspiration level for the debugging process. This analysis gives an insight into the current level of progress in debugging and later on helps in the estimation of extra efforts/cost required to achieve the aspiration level [2, 21-23].

Assume that the software has been testing up to the execution of the n_1 test run, and is to be released after the execution of the n_2 test run, where $n_2 > n_1$. The collected software reliability data in test cases $(0, n_i]$, can be employed to estimate the parameter of the selected software reliability model. According to the estimated results, the number of faults that expected to be detected after the execution of the n_2 test run

$$m_{n_2} = \frac{a\left(1-\prod_{i=0}^{n_2}(1-\delta d w_i)\right)}{(1-\alpha)} \quad (26)$$

where $d = bp(1-\alpha)$

here the number of faults that are expected to be detected in test cases $(n_1, n_2]$ is the difference $\left(m_{n_2} - m_{n_1}\right)$.

The number of test cases required can be calculated, by taking the natural logarithm of the above expression, we have

$$ln\left(1 - \frac{m_{n_2}(1-\alpha)}{a}\right) = \sum_{i=0}^{n_2} ln(1-\delta d w_i) \quad (27)$$

here the value of n_2 can be found iteratively.

However, the management may not be satisfied with the results obtained at the time of release. One major reason for this dissatisfaction could be that the reliability level achieved is not matching the management's aspiration from the debugging process. In order to consider the level of reliability that we may achieve from debugging using the specified amount of testing resources, we define it in terms the desirable number of faults to be detected $\widehat{m_N}$. If $\widehat{m_N} > m_{n_2}$, then the fault detection rate has to be increased. Accordingly we calculate the effort that is needed to detect $\left(\widehat{m_N} - m_{n_2}\right)$ faults in $(n_1, n_2]$, as follows

$$\widehat{m_N} - m_{n_2} = \frac{a-(1-\alpha)m_{n_1}}{1-\alpha}\left(1 - \prod_{k=1}^{n_2-n_1}(1-\delta d w_k)\right) \quad (28)$$

The amount of additional resources needed, can be obtained after some algebraic simplifications, as

$$\sum_{k=1}^{n_2-n_1} w_k = \frac{-1}{\delta d} ln\left(1 - \frac{(m_N - m_{n_1})(1-\alpha)}{a-(1-\alpha)m_{n_1}}\right) \quad (29)$$

4. CHANGE POINT CONCEPT

As the testing progresses, the testing team gains experience and with the employment of new tools and techniques, the fault detection rate gets changed. This change can also be caused by shift in testing strategy, defect density, introduction of new test cases, and induction of skilled personnel in team or simply by the increase in efficiency of present team. The point of time where the change in fault detection rate is observed can be termed as change-point [23]. Very few attempts have been made to incorporate the change-point in fault removal growth phenomenon. The work in this area started with Zhao [35] who introduced the change-point analysis in Hardware and Software reliability. Some pioneering work has been done in the area by Shyur [36]; Chang [37]; Wang [38].

Most of the research efforts are made in continuous time. Of late, Kapur et al. [2] incorporated the change-point concept in discrete-time [39]. In this section an attempt has been made to discuss the discrete time imperfect debugging model incorporating change-point concept, based on the following assumptions:

- the fault detection rate,

$$b = \begin{cases} b_1, & when\ 0 \le n \le \tau \\ b_2, & when\ \quad n > \tau \end{cases} \quad (30)$$

here $b_1$and b_2 are the fault detection rates before and after the change-point, and τ represents the test case number from whose execution onward change in the fault detection rate is observed.

- the probability of fault removal on a fault detection,

$$p = \begin{cases} p_1, & when\ 0 \le n \le \tau \\ p_2, & when\ \quad n > \tau \end{cases} \quad (31)$$

here $p_1$and p_2 are the probability of perfect debugging before and after the change-point

- the fault introduction rate,

$$\alpha = \begin{cases} \alpha_1, & when\ 0 \le n \le \tau \\ \alpha_2, & when\ \quad n > \tau \end{cases} \quad (32)$$

here $p_1$and p_2 are the fault introduction rate before and after the change-point

The solution for m_n under the initial condition $m_{n=0} = 0$, and $n = \tau,\ m_n = m_\tau$ is

$$m_n = \begin{cases} \frac{a(1-\beta_1{}^n)}{(1-\alpha_1)}, & when\ 0 \le n \le \tau \\ \frac{a(1-\beta_1{}^\tau \beta_2{}^{n-\tau}) + m_\tau(\alpha_1-\alpha_2)}{(1-\alpha_2)}, & when\ n > \tau \end{cases} \quad (33)$$

here $\beta_i = 1 - \delta p_i b_i (1-\alpha_i),\ i = 1,2$

and hence,

$$\lambda_n = \begin{cases} a p_1 b_1 \beta_1{}^n, & when\ 0 \le n \le \tau \\ a p_2 b_2 \beta_1{}^n \beta_2{}^{n-\tau}, & when\ n > \tau \end{cases} \quad (34)$$

The study of reliability models with change-point reveals that a great improvement in the accuracy of evaluation of software reliability is achieved with the use of change-point models as it considers the more realistic situations of the testing process. The presented model describes the difference of testing environments before and after the change-point employing different fault detection rates, probability of perfect debugging, and fault introduction rate while the perfect debugging model had ignored such differences completely.

The equivalent continuous-time model corresponding to the Equation (33) is given by

$$m_t = \begin{cases} \frac{a(1-\beta_t)}{(1-\alpha_1)}, & when\ 0 \le t \le \tau \\ \frac{a(1-\beta_\tau \beta_{t-\tau}) + m_\tau(\alpha_1-\alpha_2)}{(1-\alpha_2)}, & when\ t > \tau \end{cases} \quad (35)$$

here $\beta_t = \exp(-b_1 p_1 (1-\alpha_1)t)$, $\beta_\tau = \exp(-b_1 p_1 (1-\alpha_1)\tau)$, and $\beta_{t-\tau} = \exp\left(-b_2 p_2 (1-\alpha_2)(t-\tau)\right)$

which can be derived as a limiting case of discrete-time model substituting $t = n\delta$, $\lim_{x\to 0}(1+x)^{1/x} = exp$, and taking limit $\delta \to 0$.

5. RELEASE POLICY

The software reliability models developed to estimate and predict software reliability can be used to formulate an optimization model for software release time decision. Different policies were formulated based on nonhomogeneous Poisson process models considering different aspects of the software release time. The pioneering attempt was due to Okumoto and Goel [40]. They have investigated unconstrained problem with either cost minimization or reliability maximization objective. Later, Yamada and Osaki [41] discussed release time problems with cost minimization objective under reliability aspiration constraint and reliability maximization objective under cost constraint. Xie and Yang [42] investigated the optimal release time policy based on imperfect fault debugging software reliability model [43, 44]. They claimed that the cost of testing is a function of perfect debugging probability, since the testing cost parameter depends on the testing team composition and testing strategy used [2]. Further, Kapur et al. [29] modified the model, integrating the effect of both imperfect fault debugging and fault generation with a separate cost of repairing a fault due to perfect and imperfect fault debugging during testing and operational phases. Most of the research efforts are made in continuous time. Of late, Kapur et al. [21, 28] analyzed the release time problem in discrete time. Recently, Kapur et al. [45] proposed an optimization problem of determining the optimal time of software release based on goals set by the management in terms of cost, reliability and failure intensity etc. subject to the system constraints.

5.1 Cost Criterion

Kapur et al. [28] formulated the discrete-time version of the Okumoto and Goel [40] cost function for the total expected cost incurred during software life-cycle, when the software released after N test cases is given as

$$C_N = C_0 + C_1 \cdot m_N + C_2\left(m_{N_{lc}} - m_N\right) + C_3 \cdot N \qquad (36)$$

Here C_0 is the set-up cost for software testing, $C_1(C_2)$ is the cost of fixing a fault before (after) release of the software, C_3 is the cost per test case per unit effort expenditure, N_{lc} is the software life cycle length and W_N is the cumulative effort expenditure at the N^{th} test case.

It is a well-established fact that the increase in fault content of software due to fault generation has a direct effect on the software cost similar to the effect due to imperfect debugging.

Therefore, the cost of testing $C_{3_{p,\alpha}}$, should possess the following two properties [2]:

- $C_{3_{p,\alpha}}$ is a monotonous increasing function of p and $1-\alpha$
- when $p \to 1$ and $\alpha \to 1$, $C_{3_{p,\alpha}} \to \infty$

The second property implies that perfect debugging is impossible in practice or the cost of achieving it is extremely high. A simple function that meets the above two properties above is given by

$$C_{3_{p,\alpha}} = \frac{C_3}{1-p(1-\alpha)} \qquad (37)$$

Obviously, all these assumptions and considerations make the software cost model more realistic.
In the release policies discussed so far, the testing cost is increasing with N. If N becomes infinitely large, so does the testing costs. In reality, no software developer will spend infinite resources on testing the software. In this section, we discuss a release policy for a discrete-time version of the continuous-time model [29] with a finite limit on testing effort expenditures as follows

$$\begin{aligned} minimze \quad C_{N_{p,\alpha}} &= C_0 + \left(C_1 p + C_1^*(1-p)\right)m_N + \\ &\left(C_2 p + C_2^*(1-p)\right)\left(m_{N_{lc}} - m_N\right) + \\ &\frac{C_3}{1-p(1-\alpha)}W_N \end{aligned} \qquad (38)$$

Comparing the cost when the software is released after execution of the $(N+1)$ and N test runs and equating it to zero, yields

$$C_{(N+1)_{p,\alpha}} - C_{N_{p,\alpha}} = \left(C_1 p + C_1^*(1-p)\right)(m_{N+1} - m_N) - \left(C_2 p + C_2^*(1-p)\right)(m_{N+1} - m_N) + \frac{C_3}{1-p(1-\alpha)}(W_{N+1} - W_N) = 0$$

Or, (39)

$$\begin{aligned} C_{(N+1)_{p,\alpha}} - C_{N_{p,\alpha}} &= \left(C_1 p + C_1^*(1-p)\right)\lambda_N \\ &- \left(C_2 p + C_2^*(1-p)\right)\lambda_N + C_{3_{p,\alpha}} w_N = 0 \end{aligned}$$

and hence,

$$\lambda_N = \frac{C_{3_{p,\alpha}}}{\left(p(C_2 - C_1) + (1-p)(C_2^* - C_1^*)\right)\left(1-p(1-\alpha)\right)} w_N \qquad (40)$$

The functional form of the discrete fault detection intensity at the $\boldsymbol{N^{th}}$ test case is given as

$$\lambda_N = m_{N+1} - m_N = a\delta bpw_N\left(\prod_{i=0}^{N-1}(1 - \delta dw_i)\right) \qquad (41)$$

$\lambda_{N=0} = ab$, $\lambda_{N=\infty} = 1$, $0 < \lambda_N < 1$, and λ_N is a deceasing function in time.

Theorem I: Assuming $C_2 > C_1 > 0, C_2^* > C_1^* > 0$, $C_{3_{p,\alpha}} > 0$, and $N \geq 0$.

1. If $ab > \frac{C_{3_{p,\alpha}}}{p(C_2 - C_1) + (1-p)(C_2^* - C_1^*)}$, then $C_{N+1_{p,\alpha}} - C_{N_{p,\alpha}} < 0$ for $0 < N < N_0$, and hence, there exists a finite and unique $N = N_0 (> 0)$ minimizing $C_{N_{p,\alpha}}$.
2. If $ab \leq \frac{C_{3_{p,\alpha}}}{p(C_2 - C_1) + (1-p)(C_2^* - C_1^*)}$, then $C_{N+1_{p,\alpha}} - C_{N_{p,\alpha}} > 0$ for $N > 0$, and hence, $C_{N_{p,\alpha}}$ is minimum for $N = 0$.

5.2 Reliability Criterion

Determining release time with only cost minimization objective becomes purely a developer-oriented policy for software release. Such a decision may not truly prove to be optimization of release time. Release time decision is related to the marketing activities of the software development. In the era of customer oriented marketing, deciding release time by minimizing the cost of testing and debugging incurred during testing and operational phases may completely ignore the customer requirement of developing software with high reliability. In view of this, the policy of reliability maximization [40] at the release time can give a reasonably affirmative solution. Such a policy for any of the nonhomogeneous Poisson process type of discrete-time software reliability model can be formulated as

$$Maximize \quad R_{x|N} = e^{m_N - m_{N+x}} \qquad (42)$$

This policy may require to test the software for an infinite time as reliability is defined as the probability that a software failure does not occur between the N^{th} and the $(N+1)^{th}$ test cases, given that the last failure occurrence time $N \geq 0 (x \geq 0)$ is an increasing function with N. But this is not the solution we are looking for as software cannot be tested for infinite time. After a certain time of testing the time required to detect an additional fault increases exponentially which in turn also increases the cost of testing. Consider the case of any firm; no one neither possesses unlimited amount of resources to dispose on testing nor can they continue testing for infinite time. For such a policy we can specify a target level of reliability $\hat{R}$, and release our software at the time point where that level is achieved, irrespective of the cost incurred.

It can be easily verified that $R_{x|N}$ is increasing in N with

$$R_{x|0} = e^{-m_x}, \text{ and } R_{x|\infty} = 1$$

Thus, if $R_{x|0} < \hat{R}$, there exists $N = N_1 (> 0)$, such that $R_{x|N_1} = \hat{R}$. Hence the optimal release time policy based on achieving a desired level of reliability R_0 can be determined by the following theorem.

Theorem II: Assuming $N_{lc} > N$

1. If $R_{x|0} < \hat{R}$, then $N_1 \leq \hat{N} < N_{lc}$
2. If $R_{x|0} \geq \hat{R}$, then $0 \leq \hat{N} < N_{lc}$.

5.3 Cost-Reliability Criterion

Both of the former policies considered only one of the aspects of release time; considering any one of them ignores the other. Reliability being a key measure of quality should be considered keeping in mind the customer's requirement; on the other hand, resources are always limited so that they must be spent judiciously. It is important to have a tradeoff between software cost and reliability. Yamada and Osaki [41] formulated constrained release time problems which minimize the expected software development cost subject to reliability not less than a predefined reliability level $\hat{R}$ or maximize reliability subject to cost not exceeding a predefined budget $\hat{B}$.

$$\begin{array}{ll} Minimize & C_{N_{p,\alpha}} \\ Subjcet\ to & R_{x|N} \geq \hat{R} \\ & C_{N_{p,\alpha}} \leq \hat{B}, N > 0 \end{array}$$

Or, (43)

$$\begin{array}{ll} Maximize & R_{x|N} \\ Subjcet\ to & C_{N_{p,\alpha}} \leq \hat{B} \\ & R_{x|N} \geq \hat{R}, N > 0 \end{array}$$

The optimal release time for the above optimization problems can be obtained combining the results of Theorems I and II for the imperfect software reliability model with testing effort according to the Theorems III and IV, respectively.

Theorem III: Assuming $N_{lc} > N_0$ and $N_{lc} > N_1$, then release time is determined based on the following observation, where N_0, N_1 are as defined in theorems I and II,

1. If $ab > \frac{C_{3_{p,\alpha}}}{p(C_2-C_1)+(1-p)(C_2^*-C_1^*)}$ and $R_{x|0} \geq \hat{R}$, then $\hat{N} = N_0$.
2. If $ab > \frac{C_{3_{p,\alpha}}}{p(C_2-C_1)+(1-p)(C_2^*-C_1^*)}$ and $R_{x|0} < \hat{R}$, then $\hat{N} = Max_{(N_0,N_1)}$.
3. If $ab \leq \frac{C_{3_{p,\alpha}}}{p(C_2-C_1)+(1-p)(C_2^*-C_1^*)}$ and $R_{x|0} \geq \hat{R}$, then $\hat{N} = 0$.
4. If $ab \leq \frac{C_{3_{p,\alpha}}}{p(C_2-C_1)+(1-p)(C_2^*-C_1^*)}$ and $R_{x|0} < \hat{R}$, then $\hat{N} = N_1$.

Theorem IV: Assuming, $N_0 < N_{lc} < N_1$, $N_{lc} > N_A$ and $N_{lc} > N_B$

1. If $ab \leq \frac{C_{3_{p,\alpha}}}{p(C_2-C_1)+(1-p)(C_2^*-C_1^*)}$ and $C_{N_{0(p,\alpha)}} > C_{N_{B(p,\alpha)}}$, or
2. If $ab \leq \frac{C_{3_{p,\alpha}}}{p(C_2-C_1)+(1-p)(C_2^*-C_1^*)}$ and $C_{N_{0(p,\alpha)}} < C_{N_{B(p,\alpha)}}$, then the budget constraint is met for all $N(0 \leq N < N_A)$, where $C_{N_{A(p,\alpha)}} < C_{N_{B(p,\alpha)}}$, then $\hat{N} = N_A$.
3. If $ab > \frac{C_{3_{p,\alpha}}}{p(C_2-C_1)+(1-p)(C_2^*-C_1^*)}$ and $C_{N_{0(p,\alpha)}} > C_{N_{B(p,\alpha)}}$, then more budget is required in order to release the software to meet the above objective.
4. If $ab > \frac{C_{3_{p,\alpha}}}{p(C_2-C_1)+(1-p)(C_2^*-C_1^*)}$ and $C_{N_{0(p,\alpha)}} < C_{N_{B(p,\alpha)}}$, then the budget constraint is met for all $N(0 \leq N \leq N_A)$, where $C_{N_{B(p,\alpha)}|N_B<N_0} = C_{N_{B(p,\alpha)}}$ and $C_{N_{A(p,\alpha)}|N_A>N_0} = C_{N_{B(p,\alpha)}}$ and then $\hat{N} = N_A$.

For computing the release time of any software project using any of the above policies first of all the practitioners require the software failure data. Using the collected data we first determine the unknown parameters of the model taken into consideration. Now after obtaining the parameters of the cost and/or reliability function and bounds on the budget or reliability based on the above theorems, we can determine the release time [2].

As mentioned earlier reliability, scheduled delivery and cost are the three main quality attributes for almost all software. By determining the releases time of the software optimally taking into consideration the various constraints and aspects of the software enables to best achieve these objectives.

6. CONCLUSION

Research activities in software reliability engineering have been conducted and a number of nonhomogeneous Poisson process type of discrete-time software reliability models have been proposed to assess the software reliability during testing phase. Thus, as the testing and debugging process plays a very important role in determining the remaining fault-content, number of changes and estimate the time of each change, software release time, and allocation of testing resources, software reliability models must consider the effect of testing efficiency. In software reliability engineering literature, the efficiency of testing and debugging is incorporated as an imperfect debugging software reliability models. To be effectively practical, two types of imperfect debugging phenomena are addressed. Incorporating the imperfect fault debugging phenomenon in software reliability modelling is very important to the reliability measurement as it is related to the efficiency of the testing and debugging teams.

The objective of our endeavor is to describe the development and formulation of discrete-time modelling considering different aspects of the testing environment that affect the debugging process and to provide answers to the number of questions related to the changing scenarios during testing phase. Thus, this study gives a thorough overview of the current state of reliability evaluation and provides several bright ideas about how to improve it.

7. ACKNOWLEDGMENTS

The authors are thankful to the anonymous referee for very useful comments which resulted in substantial improvement of the manuscript.

8. REFERENCES

[1] Tassey, G. 2002. 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.

[2] Kapur, P.K., Pham, H., Gupta, A. and Jha, P.C. 2011. Software Reliability Assessment with OR Applications. Springer-Verlag, London.

[3] Musa, J.D., Iannino, A. and Okumoto, K. 1978. Software Reliability. McGraw-Hill, New York.

[4] Yamada, S. 2014. Software Reliability Modeling: Fundamentals and Applications, Springer, Japan.

[5] Yamada, S. and Osaki S. 1985. Discrete software reliability growth models. Applied Stochastic Models and Data Analysis, vol. 1, no. 1, pp. 65–77.

[6] Shatnawi, O. 2009. Discrete time NHPP models for software reliability growth phenomenon. International Arab Journal of Information Technology, vol. 6, no. 2, pp. 124–131.

[7] Shatnawi, O. 2009. Discrete time modelling in software reliability engineering: A unified approach. International Journal of Computer Systems Science and Engineering, vol 24, no. 6, pp. 391–398.

[8] Kapur, P.K., Aggarwal, A.G., Shatnawi, O. and Kumar, R. 2010. On the development of unified scheme for discrete software reliability growth modeling.

International Journal of Reliability, Quality and Safety Engineering, vol. 17, no. 3, pp. 245–260.

[9] Shatnawi, O. An integrated framework for developing discrete-time modelling in software reliability engineering. Quality and Reliability Engineering International, 2016, in press.

[10] Putnam, L.H. 1978. A general empirical solution to the macro software sizing and estimating problem. IEEE Transactions on Software Engineering, vol. 4, no. 4, pp. 345–361.

[11] Yamada, S., Ohtera, H. and Narihisa, H. 1986. Software reliability growth models with testing effort. IEEE Transactions on Reliability, vol. 35, no. 1, pp. 19–23.

[12] Yamada, S., Hishitani, J. and Osaki, S. 1991. Test-effort dependent software reliability measurement. International Journal of Systems Science, vol. 22, no. 1, pp. 73–83.

[13] Yamada, S., Hishitani, J. and Osaki, S. 1993. Software reliability growth model with Weibull testing effort: a model and application. IEEE Transactions on Reliability, vol. 42, no. 1, pp. 100–106.

[14] Bokhari, M.U. and Ahmad, N. 2006. Analysis of software reliability growth models: the case of log-logistic test-effort function. In: Proceedings 7th IASTED International Conference on Modeling and Simulation, Montreal, QC, Canada, pp. 540–545.

[15] Kapur P.K., Goswami, D.N. and Gupta, A. 2004. A software reliability growth model with testing effort dependent learning function for distributed systems. International Journal Reliability, Quality and Safety Engineering, vol. 11, no. 4, pp. 365–377.

[16] Kuo, S.Y, Huang, C.Y. and Lyu, M.R. 2001. Framework for modeling software reliability, using various testing-efforts and fault-detection rates. IEEE Transactions on Reliability, vol. 50, no. 3, pp. 310–320.

[17] Huang, C.Y. 2005. Performance analysis of software reliability growth models with testing effort and change-point. Journal of Systems and Software, vol. 76, no. 2, pp. 181–194.

[18] Huang, C.Y. 2005. Cost reliability optimal release policy for software reliability models incorporating improvements in testing efficiency. Journal of Systems and Software, vol. 77, no. 2, pp. 139–155.

[19] Huang, C.Y. and Lyu, M.R. 2005. Optimal release time for software systems considering cost, testing-effort and test efficiency. IEEE Transactions on Reliability, vol. 54, no. 4, pp. 583–591.

[20] Shatnawi, O. 2013. Testing-effort dependent software reliability model for distributed systems. International Journal of Distributed Systems and Technologies, vol. 4, no. 2, pp. 1-14.

[21] Kapur, P.K., Shatnawi, O., Jha, P.C. and Bardhan, A.K. 2004. Software testing-effort control and release time problem. In: Proceedings of the International Conference of the Association of Asia-Pacific OR Societies within IFORS, New Delhi, pp. 247-256.

[22] Kapur, P.K. and Bardhan, A.K. 2002. Testing effort control through software reliability modelling. International Journal of Modelling and Simulation, vol. 22, no. 1, pp. 90–96.

[23] Kapur, P.K., Gupta, A., Shatnawi, O. and Yadavalli, V.S.S. 2006. Testing-effort control problem using flexible software reliability growth model with change-point. International Journal of Performability Engineering, vol. 2, no. 3, pp. 245–262.

[24] Rommelfanger, H.J. 2004. The advantages of fuzzy optimization models in practical use. Fuzzy Optimization and Decision Making, vol. 3, no. 4, pp. 295–309

[25] Lyu, M.R. 1996. Handbook of Software Reliability Engineering. McGraw-Hill, New York.

[26] Xie, M. 1991. Software Reliability Modelling. World Scientific, Singapore.

[27] Pham, H. 2000. Software Reliability. Springer-Verlag, New York.

[28] Kapur, P.K., Garg, B. and Kumar, S. 1999. Contributions to Hardware and Software Reliability. World Scientific, Singapore.

[29] Kapur, P.K., Gupta, D., Gupta, A. and Jha, P.C. 2008. Effect of introduction of fault and imperfect debugging on release time. Journal of Ratio Mathematica, vol. 18, pp. 62–90.

[30] Kapur, P.K., Xie, M., Garg, R.B. and Jha, A.K. 1994. A discrete software reliability growth model with testing effort. In: Proceedings 1st International Conference on Software Testing, Reliability and Quality Assurance, 21–22 December 1994, New Delhi, pp 16–20.

[31] Kapur, P.K., Shatnawi, O. and Yadavalli, V.S.S. 2006. A discrete NHPP model for software reliability growth with imperfect fault debugging and fault generation. International Journal of Performability Engineering, vol. 2, no. 4, pp. 351–368.

[32] Shatnawi, O. 2014. Measuring commercial software operational reliability: an interdisciplinary modelling approach. Eksploatacja i Niezawodność - Maintenance and Reliability, vol. 16, no. 4, pp. 585–594.

[33] Jones C. Applied Software Measurement. 3rd ed. McGraw-Hill, 2008.

[34] Kapur, P.K, Pham, H., Anand, S. and Yadav K. 2011. A unified approach for developing software reliability growth models in the presence of imperfect debugging and error generation. IEEE Transactions on Reliability, vol. 60, no. 1, pp. 331–340.

[35] Zhao, M. 1993. Change-point problems in software and hardware reliability, Communications in Statistics-Theory and Methods, vol. 22, no. 3, pp. 757–768.

[36] Shyur, H.J. 2003. A stochastic software reliability model with imperfect-debugging and change-point, Journal of Systems and Software, vol. 66, no. 2, pp. 135-141.

[37] Chang, Y.P. 2001. Estimation of parameters for nonhomogeneous Poisson process software reliability with change-point model, Communications in Statistics-Simulation and Computation, vol. 30, no. 3, pp. 623-635.

[38] Wang, Z. and Wang J. 2005. Parameter estimation of some NHPP software reliability models with change-point, Communications in Statistics- Simulation and Computation, vol. 34, no. 1, pp. 121-134.

[39] Goswami, D.N., Khatri, S.K. and Kapur, R. 2007. Discrete software reliability growth modeling for errors of different severity incorporating change-point concept. International Journal of Automation and Computing, vol. 4, vol. 4, pp. 395–405.

[40] Okumoto, K. and Goel, A.L. 1980. Optimum release time for software systems based on reliability and cost criteria. Journal of Systems and Software, vol. 1, pp. 315–318.

[41] Yamada, S. and Osaki, S .1987. Optimal software release policies with simultaneous cost and reliability requirements. European Journal of Operational Research, vol 31, no. 1, pp. 46–51.

[42] Xie, M. and Yang, B. 2003. A study of the effect of imperfect debugging on software development cost. IEEE Transactions on Software Engineering, vol. 29, no. 5, pp. 471–473.

[43] Ohba, M. and Chou, X.M. 1989. Does imperfect debugging effect software reliability growth. In: Proceedings 11th international conference of software engineering, pp. 237–244.

[44] Kapur, P.K. and Garg, R.B. 1990. Optimal software release policies for software reliability growth models under imperfect debugging. Operations Research / Recherché Operationanelle, vol. 24, no. 3, pp. 295–305.

[45] Kapur, P.K., Khatri, S.K., Tickoo, A. and Shatnawi, O. 2014. Release time determination depending on number of test runs using multi attribute utility theory. International Journal of Systems Assurance Engineering and Management, vol. 5, no. 2, pp. 186–194.