Cost-reliability-optimal release policy for software reliability models incorporating improvements in testing efficiency

Chin-Yu Huang *

Department of Computer Science, National Tsing Hua University, No. 101, Section 2, Kuang Fu Road, Hsinchu 30055, Taiwan, Republic of China

Received 2 August 2004; received in revised form 27 September 2004; accepted 21 October 2004
Available online 26 November 2004

Abstract

Over the past 30 years, many software reliability growth models (SRGMs) have been proposed for estimation of reliability growth of products during software development processes. One of the most important applications of SRGMs is to determine the software release time. Most software developers and managers always want to know the date on which the desired reliability goal will be met. In this paper, we first review a SRGM with generalized logistic testing-effort function and the proposed generalized logistic testing-effort function can be used to describe the actual consumption of resources during the software development process. Secondly, if software developers want to detect more faults in practice, it is advisable to introduce new test techniques, tools, or consultants, etc. Consequently, here we propose a software cost model that can be used to formulate realistic total software cost projects and discuss the optimal release policy based on cost and reliability considering testing effort and efficiency. Some theorems and several numerical illustrations are also presented. Based on the proposed models and methods, we can specifically address the problem of how to decide when to stop testing and when to release software for use.

© 2004 Elsevier Inc. All rights reserved.

Keywords: Software cost; Software testing; Software reliability; Release criteria; Estimation methods

1. Introduction

With the steadily growing power and reliability of hardware, software has been identified as a major stumbling block in achieving desired levels of system dependability. It is very important to ensure the quality of the underlying software systems in the sense that they perform their functions correctly. Although extensive research has been done in the area of hardware reliability, the growing importance of software dictates that the focus shift to software reliability. In general, reliability is a better criterion because it directly correlates to the user experience with the product and inversely correlates with field failure costs. Actually, software reliability is similar to hardware reliability since both can be described by probability distributions. In general, software reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment (AIAA, 1993; Lyu, 1996). Accurately modeling software reliability and predicting its possible trends are essential to determining overall product's reliability.

During the past 30 years, a number of Software Reliability Growth Models (SRGMs) were proposed (Xie, 1991; Lyu, 1996; RAC, 1997; Pham, 2000; Grottke, 2001). SRGMs are applicable to the late stages of testing. They can provide very useful information about how to improve reliability. Some important metrics, such as the number of initial faults, failure intensity, reliability within a specific time period, number of remaining faults, mean time between failures (MTBF), and mean time to failure (MTTF), can be easily determined
through SRGMs. For instance, Schneidewind software reliability model was used by NASA Space Shuttle Program in the prediction of software reliability, by the Naval Surface Warfare Center for Trident and Tomahawk software reliability prediction, and by the Marine Corps Tactical Systems Support Activity for software reliability assessment (Keller and Schneidewind, 1997). Moreover, Gana and Huang (1997) also reported that the use of SRGMs has greatly enhanced the project’s ability to manage and improved the reliability of the global 5ESS-2000 switch products to out-perform significantly the downtime objective established by Bellcore for all regional Bell operating companies. On the other hand, Triantafyllos and Vassiliadis (1996) employed SRGMs to predict the faults of the microcode development of the IBM4381 and the IBM 9370 families of computers. They concluded that some reliability models were very successful in predicting the faults during the final phases of the development. However, choosing a good model that can be used to explain the current and past failure behavior most adequately is very important. From our studies, we find that many authors considered a Non-homogeneous Poisson Process (NHPP) as a stochastic process to describe the fault process.

Actually, most SRGMs use calendar time as the unit of fault detection and removal period. Very few SRGMs use the human power, number of test case runs, or CPU time as the unit (Yamada et al., 1986, 1990, 1991, 1993; Yamada, 2000). Musa’s basic execution time model was the first one to explicitly require that the time measurements are in actual CPU time utilized in executing the application under test (Musa et al., 1987; Musa, 1999; Grottke, 2001). Recently, we (Huang et al., 1998, 1999a,b; Kuo et al., 2001; Huang and Kuo, 2002; Huang et al., 2003; Huang, in press) proposed a SRGM that incorporates the concept of logistic testing-effort function (TEF) into an NHPP model to get a better description on the software fault phenomenon. The logistic TEF has the advantage of relating the work profile more directly to the natural structure of software development. It can be used to pertinently describe the resource consumption during the software development process and get a conspicuous improvement in modeling the distribution of testing-effort expenditures. In this paper, we first give a brief review of SRGM with generalized logistic TEF. The proposed model has a fairly accurate prediction capability.

On the other hand, if we want to detect more additional faults in practice, it is advisable to introduce new test technologies: tools and test harnesses for developing and executing all kinds of manual and automated tests. These test technologies are different from the methods currently we use and they can help software developers get their product done quicker and more reliably. The benefit of these methods is that they can design (or propose) several testing programs (or automated testing tools) to test software for satisfying the client’s technical requirements, schedule, and budget. In addition, these methods can help engineers how to assessed software process by analyzing problem reports.
from earlier projects, to improve project planning with reliability receiving more precise and equal consideration, to achieved a fixed-percentage increase in testing efficiency (or a fixed-percentage reduction in problem reports after release), or to highlight hard-to-find problems in the code semantics and structure, etc. (Huensch et al., 1990; Huber, 1999; Musa, 1999). Hence, the cost trade-off of new test techniques can be considered in software cost model and viewed as the investment required improving the long-term competitiveness.

Altogether, we wish that these new test techniques could greatly help us in detecting more faults that are difficult to find during regular testing and usage, in identifying and correcting faults most cost effectively, and in assisting clients to improve their software development processes. Thus, the fault detection rate may not smooth and can be changed at some time moment called change-point (Zhao, 1993; Chatman, 1995; Chang, 2001; Kwang, 2001; Shyur, 2003; Zou, 2003; Huang, in press).

To conclude, we will introduce a gain parameter (GP) to describe the behavior or characteristics of new test techniques and also incorporate the concept of change-point into the software reliability modeling.

In addition to modeling the software fault detection process, we also address the problem faced by most software managers, namely, how to decide when to stop testing and release software. This is a problem of decision-making under uncertainty and involves a tradeoff between reliability and cost. If we know that the software reliability of this computer system has reached an acceptable reliability level, then we can determine the right time to release this software. Actually, one of the most important applications of SRGMs is to determine the software release time (Okumoto and Goel, 1980; Yamada et al., 1984; Hou et al., 1996; Singpurwalla and Wilson, 1999; Kapur et al., 1999; Dohi, 2000; Zheng, 2002; Westland, 2002; Rinsaka and Dohi, 2004). Here we propose a new software cost model that can be used to formulate realistic total software cost projects discuss the optimal release policy based on cost and reliability considering testing-effort and efficiency. The cost model considers the testing cost, the debugging cost during testing phase, and the extra cost due to introduce new test techniques, etc.

The rest of this paper is organized as follows. We will give a brief review of the SRGM with a generalized logistic TEF in Section 2. Section 3 first discusses the problem of testing-effort control and management. Moreover, we introduce the concept of testing efficiency improvement obtained by new test techniques during testing in this section. The optimal software release time problem based on minimizing cost subject to achieving a given level of reliability considering the extra cost of introducing new techniques during testing is discussed in Section 4. Finally, Section 5 concludes the paper.

2. SRGM with generalized logistic testing-effort function

2.1. Model description

In this section, an NHPP model with TEF is present. The following assumptions are made for software reliability modeling (Yamada and Osaki, 1985; Yamada et al., 1986, 1993; Yamada, 2000; Kapur et al., 1999; Kuo et al., 2001; Huang and Kuo, 2002; Huang et al., 2003):

1. The fault removal process follows the NHPP.
2. The software system is subject to failures at random times caused by the manifestation of remaining faults in the system.
3. The mean number of faults detected in the time interval \((t, t + \Delta t)\) by the current testing-effort expenditures is proportional to the mean number of remaining faults in the system.
4. The proportionality is a constant over time.
5. The consumption curve of testing effort is modeled by a generalized logistic TEF.
6. Each time a failure occurs, the fault that caused it is immediately and perfectly removed and no new faults are introduced. Moreover, correction of faults takes only negligible time and a detected fault is removed with certainty.

If we define the expected value number of faults, \(N(t)\), whose mean value function (MVF) is known as \(m(t)\), then an SRGM based on NHPP can be formulated as a Poisson process:

\[
P_r[N(t) = n] = \frac{(m(t))^n \exp\left[-m(t)\right]}{n!}, \quad n = 0, 1, 2, \ldots \tag{1}
\]

Furthermore, if the number of faults detected by the current testing-effort expenditures is proportional to the number of remaining faults, then we obtain the following differential equation (Yamada and Osaki, 1985; Yamada et al., 1986, 1993; Huang and Kuo, 2002):

\[
\frac{dm(t)}{dt} \times \frac{1}{w(t)} = r \times [a - m(t)], \tag{2}
\]

where \(m(t)\) is the expected mean number of faults detected in time \((0, t]\) and \(m(0) = 0\), \(w(t)\) is the current testing-effort consumption at time \(t\) (such as volume of test cases, human power, and CPU time, and so on), \(a\) is the expected number of initial faults, and \(r\) is the fault detection rate per unit testing-effort at testing time \(t\) that satisfies \(r > 0\).

Solving the above differential equation, we have

\[
m(t) = a(1 - \exp[-r(W(t) - W(0))]) = a(1 - \exp[-rW'(t)]). \tag{3}
\]
Eq. (3) is an NHPP model with MVF considering the testing-effort consumption. The consumed testing-effort indicates how effective the faults are detected in the software and can be modeled by different distributions. Yamada et al. (1986, 1990, 1993) found that the testing-effort could be described by a Weibull-type distribution:

\[ W(t) = N(1 - \exp[-\beta^m]), \]

where \( \beta \) is the scale parameter and \( m \) is the shape parameter.

Actually, for the Weibull-type curves, when \( m = 1 \) or \( m = 2 \), we obtain the exponential or the Rayleigh curve respectively and therefore, they are special cases of the Weibull testing-effort function. However, when \( m = 3, 4, \) or \( 5 \), we can see that these testing-effort curves may have an apparent peak phenomenon during the software development process. That is, a peak work rate will occur. This phenomenon seems not so realistic because it is not commonly used to interpret the actual software development/test process (Huang and Kuo, 2002). Thus, the Weibull function may not be suitable for modeling the testing-effort consumption curve although it can be made to fit or approximate many distributions and represents flexible testing-effort by controlling the shape parameter \( m \). Recently, we (Huang et al., 1998, 1999a,b; Kuo et al., 2001; Huang and Kuo, 2002; Huang et al., 2003) proposed a generalized logistic TEF:

\[ W(t) = \frac{N}{\sqrt{1 + A \exp[-2\kappa t]}}, \]

where \( N \) is the total amount of testing-effort eventually consumed, \( \kappa \) is the consumption rate of testing-effort expenditures, \( A \) is a constant parameter, and \( \kappa \) is a structuring index whose value is larger for better structured software development efforts.

It is also noted that the current testing-effort consumption is

\[ w(t) = \frac{dW(t)}{dt} = \frac{1}{2\kappa} \times \left( \exp \left[ \frac{2\kappa^2 t}{K + 1} \right] + A \exp \left[ \frac{-2\kappa t}{K + 1} \right] \right)^{-\frac{1}{2}} \]

and \( w(t) \) reaches its maximum value at time

\[ t_{max} = \frac{1}{2\kappa} \times \ln \frac{A}{\kappa}. \]

In addition, the failure intensity at testing time \( t \) is

\[ \lambda(t) = \frac{dm(t)}{dt} = arw(t) \exp[-rW^*(t)]. \]

The expected number of faults to be detected eventually is

\[ m(\infty) = a \times \left( 1 - \exp \left[ -rN \left( 1 - \frac{1}{\sqrt{1 + A}} \right) \right] \right). \]

It means that the expected number of undetected faults if a test would have been applied for an infinite amount of time is

\[
\begin{align*}
& a - m(\infty) = a \times \exp \left[ -rN \left( 1 - \frac{1}{\sqrt{1 + A}} \right) \right] \\
& \quad \geq a \times \exp[-rN], \quad \text{if } A \gg 1.
\end{align*}
\]

That is, not all the original faults in a software system can be fully detected with a finite testing effort since effort to be eventually consumed during the testing phase is limited to \( N \). Furthermore, the software reliability, i.e., the probability that no failures occur in \((t, t + \Delta t)\) given that the last failure occurred at testing time \( t \), is (Xie, 1991; Lyu, 1996; Kapur et al., 1999; Pham, 2000):

\[
R(\Delta t) = \exp[-(m(t + \Delta t) - m(t))].
\]

2.2. Parameter estimation

In order to validate the proposed model, experiments on real software failure data will be performed. Two most popular estimation techniques are maximum likelihood estimation (MLE) and least squares estimation (LSE) (Xie, 1991; Pham, 2000). The maximum likelihood technique estimates parameters by solving a set of simultaneous equations and is better in deriving confidence intervals. But the equation sets may be very complex and usually must be solved numerically. On the other hand, the method of least squares minimizes the sum of squares of the deviations between what we actually observe and what we expect. Sometimes LSE is preferred because it produces unbiased results (Triantafyllos and Vassiliadis, 1996).

For example, using the method of LSE, the evaluation formula \( S_1(N, A, \kappa) \) of Eq. (5) with \( \kappa = 1 \) is depicted as follows:

\[
\text{Minimize } S_1(N, A, \kappa) = \sum_{i=1}^{n} \left( W_i - W(t_i) \right)^2,
\]

where \( W_i \) is the cumulative testing-effort actually consumed in time \((0, t_i)\) and \( W(t) \) is the cumulative testing effort estimated by the generalized logistic TEF in Eq. (5).

Differentiating \( S_1 \) with respect to \( N, A, \) and \( \kappa \), setting the partial derivatives to zero and rearranging these terms, we can solve this type of nonlinear least square problems. Thus we obtain

\[
\frac{\partial S_1}{\partial N} = \sum_{i=1}^{n} 2 \left( W_i - \frac{N}{1 + A \exp[-2\kappa t]} \right) \frac{1}{1 + A \exp[-2\kappa t]} = 0.
\]

Thus, the least squares estimator \( N \) is given by solving the above equation:
\[ N = \sum^n_{i=1} \left( \frac{W_i}{1 + A \exp[-x]} \right)^2. \]  
(11)

Next, we have
\[ \frac{\partial \mathcal{S}_1}{\partial A} = \sum^n_{i=1} 2 \left( W_i - \frac{N}{1 + A \exp[-x]} \right) \frac{N \exp[-x]}{(1 + A \exp[-x])^2} = 0, \]
(12)
and
\[ \frac{\partial \mathcal{S}_1}{\partial x} = \sum^n_{i=1} 2 \left( W_i - \frac{N}{1 + A \exp[-x]} \right) \frac{N A t \exp[-x]}{(1 + A \exp[-x])^2} = 0. \]
(13)

The other parameters \( A \) and \( x \) can also be obtained by substituting the least squares estimator \( N \) into Eqs. (12) and (13).

Similarly, if the MVF is described in Eq. (3), then the evaluation formula \( S2(a, r) \) can be obtained as

\[ \text{Minimize } S2(a, r) = \sum^n_{i=1} [m_i - m(t_i)]^2, \]
(14)

where \( m_i \) is the cumulative number of detected faults in a given time interval \((0,t_i]\) and \( m(t_i) \) is the expected number of software faults. Differentiating \( S2 \) with respect to \( a \) and \( r \), setting the partial derivatives to zero, and rearranging these terms, we can solve this type of non-linear least square problems.

On the other hand, the likelihood function for the parameters \( a \) and \( r \) in the NHPP model with \( m(t) \) in Eq. (3), is given by

\[ L \equiv P_r \{ N(t_1) = m_1, N(t_2) = m_2, \ldots, N(t_n) = m_n \} = \prod^n_{i=1} \frac{(m(t_i) - m(t_{i-1}))!}{(m_i - m_{i-1})!} \exp[-(m(t_i) - m(t_{i-1}))], \]
(15)

where \( m_0 \equiv 0 \) for \( t_0 \equiv 0 \).

Therefore, taking logarithm of the likelihood function in Eq. (15), we have

\[ \ln L = \sum^n_{i=1} (m_i - m_{i-1}) \ln[m(t_i) - m(t_{i-1})] \]
\[ - \sum^n_{i=1} (m(t_i) - m(t_{i-1})) - \sum^n_{i=1} \ln[(m_i - m_{i-1})!]. \]

From Eq. (3), we know that \( m(t_i) - m(t_{i-1}) = a(\exp[-r W^*(t_{i-1})]) - \exp[-r W^*(t_i)] \). Thus,

\[ \ln L = \sum^n_{i=1} (m_i - m_{i-1}) \ln a + \sum^n_{i=1} (m_i - m_{i-1}) \]
\[ \times \ln[\exp[-r W^*(t_{i-1})] - \exp[-r W^*(t_i)]] \]
\[ - a(1 - \exp[-r W^*(t_0)]) - \sum^n_{i=1} \ln[(m_i - m_{i-1})!]. \]
(16)

Consequently, the maximum likelihood estimates \( a \) and \( r \) can be obtained by solving \( \frac{\partial \ln L}{\partial a} = \frac{\partial \ln L}{\partial r} = 0 \), i.e.,
\[ a = \frac{\sum^n_{i=1} (m_i - m_{i-1})}{\sum^n_{i=1} \exp[-r W^*(t_0)]} = \frac{m_n}{1 - \exp[-r W^*(t_0)]}, \]
(17)
and
\[ a \times (W^*(t_0)) \times \exp[-r W^*(t_0)] = \sum^n_{i=1} (m_i - m_{i-1}) \times (W^*(t_0)) \times \exp[-r W^*(t_{i-1})]. \]
(18)

Therefore, parameters \( a \) and \( r \) can be solved by numerical methods.

2.3. Numerical example

This section evaluates the performance of the proposed model. The selected data set is from the paper by Ohba (1984) for a PL/I database application software system consisting of approximately 1,317,000 lines of code. During 19 weeks, 47.65 CPU hours were consumed and about 328 software faults were removed. Besides, the total cumulative number of detected faults after a long time of testing was 358 (Kapur and Younes, 1995). This value can be used as an additional comparison criterion.

2.3.1. Comparison criteria

The comparison criteria we use to compare various models’ performance are described as follows:

1. The goodness-of-fit criterion. The Mean Square of Fitting Error (MSE) is defined as:
\[ \frac{\sum_{i=1}^k [m(t_i) - m_0]^2}{k}. \]
(19)

2. The predictive validity criterion. The Relative Error (RE) is defined as:
\[ \frac{m(t_q) - q}{q}. \]
(20)

Assuming we have observed \( q \) failures by the end of test time \( t_q \), we use the failure data up to time \( t_e \) \((t_e \leq t_q)\) to estimate the parameters of \( m(t) \). Substituting the estimates of these parameters in the mean value function yields the estimate of the number of failures \( m(t_q) \) by \( t_q \). The estimate is compared with the actual number \( q \). The procedure is repeated for various values of \( t_q \). We can check the predictive validity by plotting the relative error for different values of \( t_q \) (Musa et al., 1987).

3. The Accuracy of Estimation (AE) is defined as:
\[ \frac{M_d - a}{M_a}, \]
(21)
where $M_a$ is the actual cumulative number of detected faults after the test, and $a$ is the estimated number of initial faults by SRGM.

### 2.3.2. Data analysis

In the first place, parameters of all selected models are estimated and the related mean value functions are obtained. Here we estimate the parameters $N$, $A$, $x$, and $\kappa$ in the generalized logistic TEF by using the methods of MLE and LSE. The values of estimated parameters are listed in Table 1. As seen from Table 1, the proposed model estimates $\kappa = 2.63326$. Fig. 1 graphically illustrates the comparisons between the observed failure data and the data estimated by the generalized logistic TEF. Using the estimated generalized logistic TEF, the other parameters $a$, $r$ in Eq. (3) can be solved numerically by the method of MLE or LSE. Secondly, all the selected models are compared with each other based on objective criteria. Table 2 lists the comparisons of different SRGMs for this data set we investigated. Moreover, since parameters are usually estimated based on limited amount of data, the estimated parameters and hence the model may not be accurate (Xie, 2000). Thus, Figs. 2 and 3 show the MVFs and the 90% bounds for the proposed model. Moreover, using the Fisher information matrix, the variance of parameters $a$ and $r$ can be estimated and shown in Table 3. Table 4 also shows the confidence limits of parameters for the proposed model. Finally, we compute the relative error in prediction for this data set and the results are plotted in Fig. 4. Numbers closer to zero imply more accurate prediction (Musa et al., 1987). It is obvious that our model gives a better fit to the observed data and predicts the future behavior well.

#### Table 1

<table>
<thead>
<tr>
<th>Model</th>
<th>$N$</th>
<th>$A$</th>
<th>$x$</th>
<th>$\kappa$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eq. (5)</td>
<td>54.8364</td>
<td>13.0334</td>
<td>0.2263</td>
<td>1</td>
</tr>
<tr>
<td>Eq. (5)</td>
<td>48.7768</td>
<td>429.673</td>
<td>0.1580</td>
<td>2.63326</td>
</tr>
</tbody>
</table>

#### Table 2

Comparison results of different SRGMs

<table>
<thead>
<tr>
<th>Model</th>
<th>$a$</th>
<th>$r$</th>
<th>AE (%)</th>
<th>MSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eq. (3) with $\kappa = 1$</td>
<td>394.08</td>
<td>0.0427223</td>
<td>10.06</td>
<td>118.59</td>
</tr>
<tr>
<td>Eq. (3) with $\kappa = 2.63326$</td>
<td>369.03</td>
<td>0.509553</td>
<td>3.08</td>
<td>110.65</td>
</tr>
<tr>
<td>Goel and Okumoto Model</td>
<td>513.15</td>
<td>0.583653</td>
<td>43.34</td>
<td>222.09</td>
</tr>
<tr>
<td>Yamada Delayed S-Shaped Model</td>
<td>374.05</td>
<td>0.1976510</td>
<td>14.48</td>
<td>168.67</td>
</tr>
</tbody>
</table>

#### Table 3

Values of $\text{var}(a)$ and $\text{var}(r)$ for the proposed model

<table>
<thead>
<tr>
<th>Model</th>
<th>$\text{var}(a)$</th>
<th>$\text{var}(r)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposed model (Eq. (3) with $\kappa = 1$)</td>
<td>732.148</td>
<td>0.0000243015</td>
</tr>
<tr>
<td>Proposed model (Eq. (3) with $\kappa = 2.63326$)</td>
<td>527.903</td>
<td>0.0000257265</td>
</tr>
</tbody>
</table>
We see that $R_{\text{risk}}(t)$ can be used to evaluate the risk caused by the remaining fault patterns that still exist after the testing phase is completed. We can further define Operational Quality Index (OQI) as the degree of how software is free of remaining faults:

$$Q(t) = (1 - R_{\text{risk}}(t)) \times 100\%.$$  \hspace{1cm} (23)

Actually, the quality index is very important for widely distributed commercial software. In order to reduce risk and achieve a given operational quality at a specified time, we can use the SRGM to estimate and control the required testing-effort. But the major problem is how to estimate the number of extra faults that have to be found. Here let us consider the following scenario:

(1) Due to economic and beneficial considerations, software testing and debugging will eventually be terminated at a specified time point, $T_2$.

(2) Based on the SRGM selected by software developers or test teams, the expected number of initial faults, $a$, in this software system is estimated at time $T_1$ ($0 < T_1 < T_2$).

(3) By applying the estimated parameters into the selected SRGM, the test teams can predict the cumulative number of faults at time $T_2$ and the $R_{\text{risk}}(T_2)$. The estimated value of $R_{\text{risk}}(T_2)$ may have already satisfied the developers’ desired goal. If not, in order to meet the requirements, the developers must detect extra faults $\delta m^*(T_2 - T_1)$ during the time interval $T_2 - T_1$.

Based on the above scenario, if we let $R_{\text{risk}}^r(T_2) \equiv 1 - \frac{m^*(T_2)}{m(\infty)} = 1 - \frac{a'}{a}$ and $a^*(a^* > m(T_2))$ is the target number of faults to be detected at time $T_2$, then

$$a' = m(T_1) + \delta m^*(T_2 - T_1), \quad a' \geq m(T_2)$$

$$= a(1 - \exp[-r(W(T_1) + W^*(T_2 - T_1) - W(0))])$$

$$= a(1 - \exp[-rW^*(T_2 - T_1)])$$

$$\times \exp[-r(W(T_1) - W(0))].$$  \hspace{1cm} (24)

It is noted that $m(T_1)$ is the cumulative number of faults detected at time $T_1$, and $\delta m^*(T_2 - T_1)$ is the number of extra faults which need to be detected in order to reach the desired goal at time $T_2$ ($\delta m^*(T_2 - T_1) > \delta m(T_2 - T_1)$).

Rearranging the above equation, we have

$$a' - m(T_1) = \delta m^*(T_2 - T_1) = (a - m(T_1))$$

$$\times (1 - \exp[-rW^*(T_2 - T_1)])$$

$$\Rightarrow a - a^* = (a - m(T_1))$$

$$\times \exp[-rW^*(T_2 - T_1)].$$  \hspace{1cm} (25)
Here

\[ W^*(T_2 - T_1) = \int_{T_1}^{T_2} w(t) \, dt = W^*(T_2) - W^*(T_1) = \frac{N}{\sqrt{1 + A \exp[-x^* \kappa T_2]}} - \frac{N}{\sqrt{1 + A \exp[-x^* \kappa T_1]}}. \] (26)

Substituting Eq. (26) into Eq. (25), we get

\[ \frac{N}{\sqrt{1 + A \exp[-x^* \kappa T_2]}} - \frac{N}{\sqrt{1 + A \exp[-x^* \kappa T_1]}} = -\frac{1}{r} \ln \left( 1 - \frac{a^* - m(T_1)}{a - m(T_1)} \right). \] (27)

The modified testing-effort function \( W^*(T_2 - T_1) \) during the time interval \((T_1, T_2)\) can be controlled by using \( x^* \), the modified consumption rate of testing-effort expenditures which satisfies Eq. (26). This can be solved numerically. For instance, here we use Eq. (3) as the MVF for illustration and also let \( \kappa = 1 \). If \( T_1 = 19, T_2 = 30 \), and the desired operational quality index \( Q(30) \) is 97\% which is larger than the original case \( Q(30) = 88.25\% \), using Eqs. (24)-(26), the modified expenditure rate \( x^* \) is estimated as 0.0996481. This means that \( x^* \) can be used to satisfy Eqs. (25) and (26) during time interval \((T_1, T_2)\) in order to achieve the desired operational quality. Fig. 5 shows the modified TEF versus time.

3.2. New approach to increase software testing efficiency and its impact on software reliability improvement

We have discussed the applications of testing-effort control and management problem in the previous subsection. The method we proposed can easily control the modified consumption rate of testing-effort expenditures and detect more faults in the prescribed time interval. It means that the software developers or testers will devote their all-available knowledge to complete such tasks without additional resources. However, alternative to controlling the testing-effort expenditures, we believe that new testing schemes will help us to achieve a given operational quality at a specified time. In other words, through introducing experienced consultants or new (advanced) test techniques, we can detect and remove more additional faults, i.e., these faults may or may not cause any failure or they are not easily exposed during the test phase, although these new methods will increase software development costs.

Rivers (1998) thought that there is a consistent improvement in the efficiency of testing during a testing phase. One can get more efficient feed-forward improvement if the software developers or testers learn and adjust their testing during an actual testing phase, and re-use that experience in the follow-on testing phases. He conceived that the efficiency of the fault detection process can be degraded or enhanced for a number of reasons, such as the mapping between defects and defect-sensitive constructs is not necessarily one to one, so there may be more constructs that detect a particular defect, etc. Moreover, Hou et al. (1994, 1996) applied exponential and S-shaped learning curves to the HGDMs (Hyper-Geometric Distribution Models) to account for fault detection variability. Malaia et al. (1993, 2002) also analyzed the fault exposure ratio and used it to explain the fluctuation of per-fault detectability while developing their coverage-based SRGMs (Musa et al., 1987; Rivers, 1998).

Actually, testing efficiency is an indirect measure and it can be obtained by NF/TE, where NF is the number of faults detected and TE is the testing-effort. NF is an absolute scale measure, while TE is on the ratio scale. It is noted that absolute is stronger than ratio scale (Fenton and Pfleeger, 1997). Consequently, if we want to increase testing efficiency, the number of faults detected is the key factor. On the other hand, Musa (1999) ever defined testing efficiency in terms of failure intensity and time. He thought that the testing efficiency is a normalized quantity, expressing percentage changes in failure intensity rather than absolute change. He suggested increasing testing efficiency by using run categories. Moreover, he also advised to use experimental designs such as orthogonal arrays to select run categories, given the situation where the input variables that define the run orthogonal arrays are limited in their degree of interaction (Musa, 1999).

Consequently, if the software managers wish to detect more potential faults in practice, they may introduce new test techniques, which are not yet used, or bring in consultants to make a radical software risk analysis, or experienced consultants. The benefits to software developers and testers include increased software quality, reduced testing costs, improved release time to the next phase, repeatable test steps, and improved testing productivity. That is, these techniques can make software testing and correction easier, detect more bugs,
save more time, and reduce much expense. For instance, experienced consultants frequently play an important role in technology transfer. The role of the consultants is demanding since they must work rapidly and efficiently to be effective. Moreover, the consultants usually have only a short time to learn the needs of the project and the appropriate background, develop appropriate recommendations, and help implement them (Musa, 1999; Tian, 1999).

To conclude, we wish that the consultants or new techniques could greatly help software developers or testers in detecting more faults that are difficult to find during regular testing and usage, in identifying and correcting faults most cost effectively, and in assisting clients to improve their software development processes. In this case, the fault detection rate may not smooth and can be changed at some time moment called change-point (Zhao, 1993; Chang, 2001; Kwang, 2001; Shyur, 2003; Zou, 2003; Huang, in press). Here we introduce a gain parameter (GP) to describe the behavior or characteristics of new techniques and also incorporate the concept of change-point into software reliability growth modeling. Let us consider a modified MVF with change-point and this function can be depicted as follows:

\[
m_{\text{modified}}(t) = a(1 - \exp[-rW^*(t)])I(t < T_S) + a(1 - \exp[-\sigma rW^*(t)])I(t \geq T_S),
\]

where \(\sigma\) is the gain parameter, \(T_S\) is time of introducing new techniques called change-point, and \(I(t < T_S)\) and \(I(t \geq T_S)\) are the corresponding indicators:

\[
I(t < T_S) = \begin{cases} 0, & t \geq T_S, \\ 1, & t < T_S, \end{cases} \\
I(t \geq T_S) = \begin{cases} 1, & t \geq T_S, \\ 0, & t < T_S. \end{cases}
\]

On the other hand, the software managers always just want to understand the impact of new introduced techniques on fault detection process in practice. Sometimes they only want to know how percentage is increased and feel that this way is more simple and realistic. In general, they could set an objective and then tell the software developers or testers about the desired goal (please refer to Fig. 6). To examine this case, we will discuss the following example.

Using Ohba’s data set described in Section 2, Fig. 6 shows the effect of more efficient testing on the number of detected faults with time when we decide to introduce new test techniques. The upper bold dot-dash lines are desired goal set by the project managers. The managers decide that the nineteenth week is the best time of introducing new techniques and the software developers or testers can obtain external helps thereafter. It is obvious that more efficient software testing will increase the fault detection rate and thus we can detect more faults in software. Consequently, the gain-effect of employing new test techniques can be expressed by

\[
\frac{m_{\text{modified}}(t)}{m(t)} = (1 + P) \quad \text{and} \quad \begin{cases} P > 0, & t \geq T_S, \\ P = 0, & t < T_S, \end{cases}
\]

where \(P\) is the additional fraction of faults detected by using new or advanced techniques during testing. That is, we expect that the number of detected faults can have a fixed percentage increment after change point when we introduced new test techniques.

Substituting Eq. (30) into Eq. (28) and rearranging the above equations, the estimated value of GP is given by

\[
\sigma = 1 \quad \text{for} \quad t < T_S \quad \text{and} \quad \sigma = -\frac{1}{rW^*(t)} \ln \left( (1 + P) \times \exp[-rW^*(t)] - P \right)
\]

for \(t \geq T_S\).

Actually, we can interpret the gain parameter from different views. If we make a premise that the main goal of new test techniques is to test and debug software with less testing effort, then \(\sigma\) and the testing-effort are inversely proportional to each other. That is, they have a joint effect on the software development process. On the other hand, under the same testing effort expenditures, introducing new test techniques should greatly help us in detecting and removing more additional faults which are hard to detect without introducing these new methods. That is, they can help us get our product done quicker and more reliably. For instance, at the end of testing (19th week), the estimated cumulative number of failures detected is about 324 for Ohba’s data set. But the software managers do not satisfy the result and decide to introduce new techniques to detect more faults with 5 percent increment thereafter (as shown in Fig. 7(e), the upper dot-dash line is the expected MVF and the other is the original estimated MVF). In this case, the corresponding distribution of GP is depicted.
in Fig. 8(e). The other cases are also provided in Fig. 7(a)–(j) and Fig. 8(a)–(j).

However, the most important thing is how to provide enough information about these techniques to the software development teams and this is the chief concern for the project managers. Therefore, before adopting these new test techniques, we should get the quantitative information from the industrial data relative to these methods’ past performance applied in other instances (i.e. the previous experience in software testing), or qualitative information from the subjective valuation of methods’ attributes (Musa, 1999; Westland, 2002). Certainly, the methods’ past performance in aiding the reliability growth should be considered in determining whether they will be successful again or not. Finally, developing software systems is generally characterized by business pressure to minimize development cost and time to market. However, it is often stated that delivering high-quality software does not necessarily mean that development costs will increase. This is only partially true. For example, striving for higher reliability through investing in appraisal techniques or tools will be paid back up by an obvious decrease in the costs of finding and fixing faults. But as with any other business critical activity, incremental, sustained improvements in software testing process will lead significant overall benefits (Huber, 1999).

4. Optimal software release policy

When the software testing is completed, software product is ready to release to users. However, proper (and exact) timing is very important. If the reliability of the software does not meet the project manager’s goal, software developers or testers may introduce additional test techniques or experienced consultants to aid in testing. An optimal release policy for the proposed model based on such considerations is studied in the following.

4.1. Software release time based on cost criterion

Okumoto and Goel (1980) firstly discussed the software optimal release policy from the cost-benefit viewpoint. Using the total software cost evaluated by cost criterion, the costs of testing-effort expenditures during software development phase and the costs of correcting errors before and after release are given by (Yamada
Fig. 8. Growth curve of GP versus time: (a) $p = 0.01$, (b) $p = 0.02$, (c) $p = 0.03$, (d) $p = 0.04$, (e) $p = 0.05$, (f) $p = 0.06$, (g) $p = 0.07$, (h) $p = 0.08$, (i) $p = 0.09$, (j) $p = 0.10$.

where $T_{LC}$ is the software life-cycle length, $C_1$ is the cost of correcting an error during testing, $C_2$ is the cost of correcting an error during operation, $C_2 > C_1$, and $C_3$ is the cost of testing per unit testing-effort expenditures (Boehm, 1981; Westland, 2002).

By summing up above stated cost factors, the modified software cost model can be shown as follows:

$$C_2(T) = C_0(T) + C_1(1 + P)m(T) + C_2[m(T_{LC}) - m(T)] + C_3 \int_0^T w(x) \, dx, \quad (32)$$

where $C_0(T)$ is the cost function of introducing new test techniques to detect an additional fraction $P$ of faults during testing.

Actually, the cost of a new test technique $C_0(T)$ may not be a constant during the testing phase of software development process. Moreover, in order to determine the testing cost $C_0(T)$, the most general cost estimating technique is to use the parametric methods if there are some meaningful data available (Boehm, 1981). Under the cost-benefit considerations, the new test techniques will pay for themselves if $C_1(T) - C_2(T) \geq 0$. That is, $C_0(T) \leq P \times m(T) \times (C_2 - C_1)$. This condition can be used to decide whether the new test techniques would be effective or not.

Moreover, it is worthwhile to note that the saving in development cost is given by

$$\Delta C = C_1(T) - C_2(T). \quad (34)$$

Thus the percentage saving is

$$\frac{\Delta C}{C_1(T)} = \frac{C_1(T) - C_2(T)}{C_1(T)} = 1 - \frac{C_2(T)}{C_1(T)}. \quad (35)$$

By differentiating Eq. (33) with respect to $T$ and let $C_1(1 + P) = C'_1$ and $C_2(1 + P) = C'_2$ we have
If we assume that $C_0(T)$ is a constant, then this assumption may not be realistic in many situations. In addition, this assumption may lead to ill-defined testing. Therefore, we can reasonably relax this assumption and explore the results. In the following, we will propose two possibilities for $C_0(T)$ in order to interpret the cost consumption.

**Case study 1:** $C_0(T)$ is a constant. For most small-scale software projects, it is generally enough for the project managers to purchase common types of test techniques if they decide to detect more faults after testers report the testing results to management. Therefore, the extra cost is normally fixed. On the other hand, some companies may need to create their own test tools due to certain project requirements. Sometimes creating these tools is not expensive since they only need to hire some engineers or consider outsourcing.

**Case study 2:** $C_0(T)$ is proportional to the expenditures of testing-effort. Just as mentioned above, it is enough for the project managers to simply purchase general automated tools if they want to detect more faults and the scale of the projects is not large. Sometimes test tool vendors offer various types of technical support, such as software upgrades, patches, or maintenance supports, etc. with different fees. Consequently, the cost of introducing these test techniques may not be constant and it can be linearly increasing in part of the software cost estimation.

### 4.1.1. $C_0(T)$ is a constant

$C_0(T) = C_0$, $T \geq T_S$; $C_0(T) = 0$, $T < T_S$, where $T_S$ is the start time of adopting new testing techniques.

\[
\frac{dC_2(T)}{dT} = w(T) \times [-ar(C_2 - C_1) \exp[-rW^*(T)] + C_3].
\]

Since $w(t) > 0$ for $0 < T < \infty$, $\frac{dC_2(T)}{dT} = 0$ if $ar(C_2 - C_1) \exp[-rW^*(T)] = C_3$.  

The left-side in Eq. (38) is monotonically decreasing function of $T$. Here we note that if $ar(C_2 - C_1) \exp[-rW^*(T_S)] \leq C_3$, then $ar(C_2 - C_1) \exp[-rW^*(T_{LC})] < C_3$ for $T_S < T < T_{LC}$. Thus the optimal software release time $T^* = T_S$ since $\frac{dC_2(T)}{dT} > 0$ for $T_S < T < T_{LC}$. On the other hand, if $ar(C_2 - C_1) \times \exp[-rW^*(T_S)] > C_3$ and $ar(C_2 - C_1) \exp[-rW^*(T_{LC})] < C_3$, there exists a finite and unique solution $T_0$ satisfying Eq. (38):

\[
T_0 = \frac{1}{\alpha} \times \ln \left( \frac{A \Theta^*}{N^* - \Theta^*} \right) \text{minimizes } C_2(T),
\]

where $\Theta = \frac{1}{\alpha} \left( \ln \left[ ar(C_2 - C_1) \right] \right) + \frac{N}{\alpha}$ and since $\frac{dC_2(T)}{dT} < 0$ for $T_S < T < T_0$. Since $\frac{dC_2(T)}{dT} > 0$ for $T_0 < T < T_{LC}$.

If $ar(C_2 - C_1) \exp[-rW^*(T_{LC})] > C_3$, then $ar(C_2 - C_1) \exp[-rW^*(T)] > C_3$ for $T_S < T < T_{LC}$. Therefore, the optimal software release time $T^* = T_{LC}$ since $\frac{dC_2(T)}{dT} < 0$ for $T_S < T < T_{LC}$.

### 4.1.2.

$C_0(T) = C_0 + C_0 \int_{T_S}^{T} w(t) dt$, $T \geq T_S$; $C_0(T) = 0$, $T < T_S$, where $T_S$ is the start time of adopting new testing techniques and $C_0$ is the nonnegative real number that indicates the basic cost of adopting new test techniques.

\[
\frac{dC_2(T)}{dT} = w(T) \times [ar(C_2 - C_1) \exp[-rW^*(T)] - C_2 \exp[-rW^*(T)] + C_3].
\]

Similarly, since $w(t) > 0$ for $0 \leq T < \infty$, $\frac{dC_2(T)}{dT} = 0$ if $ar(C_2 - C_1) \exp[-rW^*(T)] = (C_3 + C_0)$.  

We can see that the left-side in Eq. (41) is monotonically decreasing function of $T$. Thus if $ar(C_2 - C_1) \exp[-rW^*(T)] > (C_3 + C_0)$ and $ar(C_2 - C_1) \exp[-rW^*(T_{LC})] < (C_3 + C_0)$, there exists a finite and unique solution $T_0$ satisfying Eq. (41).

\[
T_0 = \frac{1}{\alpha} \times \ln \left( \frac{A \Theta^*}{N^* - \Theta^*} \right) \text{minimizes } C_2(T),
\]

where $\Theta = \frac{1}{\alpha} \left( \ln \left[ ar(C_2 - C_1) \right] \right) + \frac{N}{\alpha}$.

### 4.1.3.

$C_0(T) = C_0 + C_0 \left( \int_{T_S}^{T} w(t) dt \right)^m$, $T \geq T_S$; $C_0(T) = 0$, $T < T_S$.

\[
\frac{dC_2(T)}{dT} = C_0 w(T) + C_1 w(T) \times \exp[-rW^*(T)] - C_2 \exp[-rW^*(T)] + C_3.
\]

Because $w(t) > 0$ for $0 \leq T < \infty$, $\frac{dC_2(T)}{dT} = 0$ if $P(T) \equiv \left[ ar(C_2 - C_1) \exp[-rW^*(T)] - C_0 \right] \left( \int_{T_S}^{T} w(t) dt \right)^{m-1} = C_3$.

The left-side in Eq. (43) is monotonically decreasing function of $T$. Consequently, if $ar(C_2 - C_1) \times
based on C0(0) = 1000. From Eq. (8), we can easily compute the software reliability optimal release time. If we know that the software reliability of this computer system has reached an acceptable reliability level, then the percentage saving is 0.125, i.e., the cost saving obtained from introducing new techniques to reduce resources is 12.5 percent. Similarly, the relationship between the optimal release time and P based on different cost function is shown in Table 6.

### 4.1.4. Numerical examples

In this section we will illustrate how to minimize the software cost in which the new test techniques are introduced during testing. In addition to the previous estimated parameters in Tables 1 and 2, here we also let C01 = 1000, C1 = $10 per error, C2 = $50 per error, C3 = $100 per week, T0 = 100 weeks. The numerical examples on the relationship between the optimal release time and P are given in Table 5. In addition, compared with the estimated values of traditional software cost model (i.e., Eq. (19)), we have T* = 24.2828 and C(T*) = 4719.66. As seen from Table 5, we note that T* = 24.2839 and C(T*) = 4130.91 when P = 0.10. It means that C2(T) is smaller than C1(T) and both optimal release times are nearly equal. Therefore, the assumption C1(T) − C2(T) ≥ 0 is satisfied. In this case, the percentage saving is 0.125, i.e., the cost saving obtained from introducing new techniques to reduce resources is 12.5 percent. Similarly, the relationship between the optimal release time and P based on different cost function is shown in Table 6.

### 4.2. Software release time based on reliability criterion

In general, the software release time problem is also associated with the reliability of software system. Hence, if we know that the software reliability of this computer system has reached an acceptable reliability level, then we can determine the right time to release this software (Singpurwalla and Wilson, 1999; Yamada, 2000; Pham, 2000). From Eq. (8), we can easily compute the software reliability. It is noted that R(T) is increasing in T. Therefore, using Eq. (8), we can easily get the required testing time needed to reach the reliability objective R0 or decide whether R0 is reached or not at a specified time interval. If R(TS) < R0, there exists a unique T1 > TS satisfying R(T1) = R0. Thus by solving Eq. (8), we can determine the testing time needed to reach a desired reliability.

On the other hand, we can also define the second measure of software reliability for the proposed model, i.e., the ratio of the cumulative number of detected faults at the time T to the expected number of initial faults (Hou et al., 1996; Huang et al., 1999b).

$$R_2(T) = \frac{m_{\text{modified}}(T)}{a}.$$  

We can solve this equation and obtain a unique T1 satisfying R2(T1) = R0. It is noted that the larger the value of R2(T), the higher the software reliability is. Similarly, using the estimated parameters for the real data set in Tables 1, 7 and 8 show the relationships between the reliability optimal release time T1, Δt and P based on R0 = 0.9 and 0.95. Furthermore, Tables 9 and 10 depict the relationship between the cost optimal release time T0 and P based on two different cost functions and the corresponding software reliability R2(T). On the other hand, from Tables 7 and 8, we can see that as P

<table>
<thead>
<tr>
<th>P</th>
<th>Optimal release time T0</th>
<th>Total expected cost C(T0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>19.738</td>
<td>5574.05</td>
</tr>
<tr>
<td>0.02</td>
<td>20.002</td>
<td>5414.50</td>
</tr>
<tr>
<td>0.03</td>
<td>20.289</td>
<td>5254.74</td>
</tr>
<tr>
<td>0.04</td>
<td>20.607</td>
<td>5094.77</td>
</tr>
<tr>
<td>0.05</td>
<td>20.965</td>
<td>4934.60</td>
</tr>
<tr>
<td>0.06</td>
<td>21.975</td>
<td>4774.24</td>
</tr>
<tr>
<td>0.07</td>
<td>21.854</td>
<td>4613.69</td>
</tr>
<tr>
<td>0.08</td>
<td>22.446</td>
<td>4452.94</td>
</tr>
<tr>
<td>0.09</td>
<td>22.203</td>
<td>4292.02</td>
</tr>
<tr>
<td>0.10</td>
<td>24.284</td>
<td>4130.91</td>
</tr>
<tr>
<td>0.11</td>
<td>26.111</td>
<td>3969.62</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th>Optimal release time T1 (Δt = 0.1)</th>
<th>Total expected cost C(T0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>20.927</td>
<td>22.580</td>
</tr>
<tr>
<td>0.02</td>
<td>20.952</td>
<td>22.604</td>
</tr>
<tr>
<td>0.03</td>
<td>20.976</td>
<td>23.541</td>
</tr>
<tr>
<td>0.04</td>
<td>20.999</td>
<td>23.588</td>
</tr>
<tr>
<td>0.05</td>
<td>21.024</td>
<td>23.611</td>
</tr>
<tr>
<td>0.06</td>
<td>21.047</td>
<td>23.636</td>
</tr>
<tr>
<td>0.07</td>
<td>21.070</td>
<td>23.657</td>
</tr>
<tr>
<td>0.08</td>
<td>21.093</td>
<td>23.679</td>
</tr>
<tr>
<td>0.09</td>
<td>21.116</td>
<td>23.702</td>
</tr>
<tr>
<td>0.10</td>
<td>21.139</td>
<td>23.724</td>
</tr>
<tr>
<td>0.11</td>
<td>21.161</td>
<td>23.746</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th>Optimal release time T1 (Δt = 0.2)</th>
<th>Total expected cost C(T0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>20.927</td>
<td>22.580</td>
</tr>
<tr>
<td>0.02</td>
<td>20.952</td>
<td>22.604</td>
</tr>
<tr>
<td>0.03</td>
<td>20.976</td>
<td>23.541</td>
</tr>
<tr>
<td>0.04</td>
<td>20.999</td>
<td>23.588</td>
</tr>
<tr>
<td>0.05</td>
<td>21.024</td>
<td>23.611</td>
</tr>
<tr>
<td>0.06</td>
<td>21.047</td>
<td>23.636</td>
</tr>
<tr>
<td>0.07</td>
<td>21.070</td>
<td>23.657</td>
</tr>
<tr>
<td>0.08</td>
<td>21.093</td>
<td>23.679</td>
</tr>
<tr>
<td>0.09</td>
<td>21.116</td>
<td>23.702</td>
</tr>
<tr>
<td>0.10</td>
<td>21.139</td>
<td>23.724</td>
</tr>
<tr>
<td>0.11</td>
<td>21.161</td>
<td>23.746</td>
</tr>
</tbody>
</table>
Table 8
Relationship between the reliability optimal release time $T_1$ and $P$ based on the first measure of software reliability $R_0 = 0.95$

<table>
<thead>
<tr>
<th>$P$</th>
<th>Optimal release time $T_1$ ($\Delta t = 0.1$)</th>
<th>Optimal release time $T_1$ ($\Delta t = 0.2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>22.695</td>
<td>24.328</td>
</tr>
<tr>
<td>0.02</td>
<td>22.719</td>
<td>24.352</td>
</tr>
<tr>
<td>0.03</td>
<td>22.742</td>
<td>24.376</td>
</tr>
<tr>
<td>0.04</td>
<td>22.766</td>
<td>24.399</td>
</tr>
<tr>
<td>0.05</td>
<td>22.789</td>
<td>24.422</td>
</tr>
<tr>
<td>0.06</td>
<td>22.812</td>
<td>24.445</td>
</tr>
<tr>
<td>0.07</td>
<td>22.835</td>
<td>24.468</td>
</tr>
<tr>
<td>0.08</td>
<td>22.858</td>
<td>24.490</td>
</tr>
<tr>
<td>0.09</td>
<td>22.880</td>
<td>24.513</td>
</tr>
<tr>
<td>0.10</td>
<td>22.903</td>
<td>24.535</td>
</tr>
<tr>
<td>0.11</td>
<td>22.925</td>
<td>24.557</td>
</tr>
</tbody>
</table>

Table 9
Relationship between the cost optimal release time $T_0$, the second measure of software reliability $R_0(T)$, and $P$ based on $C_s(T) = 1000 + 10 \int_{0}^{100} w(t) dt$

<table>
<thead>
<tr>
<th>$P$</th>
<th>Cost optimal release time $T_0$</th>
<th>$R_0(T)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>19.738</td>
<td>0.890825</td>
</tr>
<tr>
<td>0.02</td>
<td>20.002</td>
<td>0.900818</td>
</tr>
<tr>
<td>0.03</td>
<td>20.289</td>
<td>0.910616</td>
</tr>
<tr>
<td>0.04</td>
<td>20.607</td>
<td>0.920616</td>
</tr>
<tr>
<td>0.05</td>
<td>20.965</td>
<td>0.930616</td>
</tr>
<tr>
<td>0.06</td>
<td>21.975</td>
<td>0.941974</td>
</tr>
<tr>
<td>0.07</td>
<td>21.854</td>
<td>0.950560</td>
</tr>
<tr>
<td>0.08</td>
<td>22.447</td>
<td>0.960646</td>
</tr>
<tr>
<td>0.09</td>
<td>23.203</td>
<td>0.970618</td>
</tr>
<tr>
<td>0.10</td>
<td>24.284</td>
<td>0.980617</td>
</tr>
<tr>
<td>0.11</td>
<td>26.111</td>
<td>0.990570</td>
</tr>
</tbody>
</table>

Table 10
Relationship between the cost optimal release time $T_0$, the second measure of software reliability $R_0(T)$, and $P$ based on $C_s(T) = 1000 + 10 \int_{0}^{100} w(t) dt$

<table>
<thead>
<tr>
<th>$P$</th>
<th>Cost optimal release time $T_0$</th>
<th>$R_0(T)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>19.602</td>
<td>0.889954</td>
</tr>
<tr>
<td>0.02</td>
<td>19.750</td>
<td>0.899490</td>
</tr>
<tr>
<td>0.03</td>
<td>19.916</td>
<td>0.909071</td>
</tr>
<tr>
<td>0.04</td>
<td>20.098</td>
<td>0.918690</td>
</tr>
<tr>
<td>0.05</td>
<td>20.299</td>
<td>0.928341</td>
</tr>
<tr>
<td>0.06</td>
<td>20.522</td>
<td>0.938019</td>
</tr>
<tr>
<td>0.07</td>
<td>20.768</td>
<td>0.947720</td>
</tr>
<tr>
<td>0.08</td>
<td>20.044</td>
<td>0.953784</td>
</tr>
<tr>
<td>0.09</td>
<td>21.354</td>
<td>0.967182</td>
</tr>
<tr>
<td>0.10</td>
<td>21.709</td>
<td>0.976937</td>
</tr>
<tr>
<td>0.11</td>
<td>22.123</td>
<td>0.986707</td>
</tr>
</tbody>
</table>

increases, the optimal release time $T_1$ increases. This phenomenon can be examined as follows.

From Eq. (8), we know that $R(\Delta t | T)$ denotes the conditional reliability function that the software will still operate after $T + \Delta t$ given that it has not failed after time $T$. In addition, from Eq. (28), we know $m_{modified}(t) = (1 + P) m(t)$ for $t > T_S$. Consequently, when $t > T_S$, we can see $m_{modified}(T + \Delta t) - m_{modified}(T) = (1 + P) (m(T + \Delta t) - m(T))$. Moreover, since $-m_{modified}(T + \Delta t) - m_{modified}(T)$, thus we have $\exp[-m_{modified}(T + \Delta t) - m_{modified}(T)] \leq \exp[-m(T + \Delta t) - m(T)]$.

As seen from Table 5, that’s the reason why the optimal release time $T_1 = 20.9273$ under $P = 0.01$, $\Delta t = 0.1$, and $R_0 = 0.9$ is slightly larger than the optimal release time (without introducing any new test techniques during testing) $T_1 = 20.9027$ under $\Delta t = 0.1$ and $R_0 = 0.9$.

Using these new test techniques, we may detect some extra faults in $(T, T + \Delta t)$ and these faults are potentially hard to detect. However, if software faults are hard to locate and detect after a long time testing, the developers or tester may treat the software system as a reliable (or stable) system. The phenomenon usually occurs in practice since the existing technology or knowledge of software developers is limited. On the other hand, if the company can afford to introduce consultants or new test techniques, the latent faults may quickly be detected in the interval $(T_1, T_1 + \Delta t)$ and the reliability optimal release time will be postponed till the goal of reliability is reached. Hence, software developers may consider Eq. (44) as the second measure of software reliability and it can aid in decision making.

4.3. Software release time based on cost-reliability criterion

From Section 4.2, we can obtain the required testing time needed to reach the reliability objective $R_0$. Here our goal is to minimize the total software cost to achieve the desired software reliability and then the optimal software release time is obtained. Therefore, the optimal release policy problem can be formulated as minimize $C(T)$ and subject to $R(\Delta t | T) \geq R_0$ where $0 < R_0 < 1$. That is,

$$T^* = \text{cost-reliability-optimal software release time} = \max(T_0, T_1).$$

It is noted that $T_0$ is the finite and unique solution $T$ satisfying Eq. (39), (42), or (43), and $T_1$ is the time needed to reach the reliability objective $R_0$. Combining the cost and reliability requirements and considering the testing-efficiency improvement, we have the following three theorems.

Theorem 1. Assume $C_s(T) = C_0$ (constant), $C_0 > 0$, $C_1 > 0$, $C_2 > 0$, $C_3 > 0$, and $C_4 > C_1$, we have

1. If $ar(C_2 - C_1) \exp[-rW^*(T_S)] > C_3$ and $ar(C_1 - C_1) \exp[-rW^*(T_LC)] > C_3$, $T^* = \max(T_0, T_1)$ for $R(\Delta t | T_S) < R_0 < 1$ or $T^* = T_0$ for $0 < R_0 \leq R(\Delta t | T_S)$.

2. If $ar(C_2 - C_1) \exp[-rW^*(T_S)] < C_3$, $T^* = T_1$ for $R(\Delta t | T_S) < R_0 < 1$ or $T^* = T_S$ for $0 < R_0 \leq R(\Delta t | T_S)$. 
(3) If \( ar(C'_2 - C'_1) \exp[-rW^*(T_{LC})] > C_3 \), \( T^* \geq T_0 \) for \( R(\Delta T)S < R_0 < 1 \) or \( T^* \geq T_S \) for \( 0 < R_0 \leq R(\Delta T)S \).

**Theorem 2.** Assume \( C_0(T) = C_{01} + C_0 \int_{T}^{T} w(t) \, dt \), \( C_{01}, C_0 > 0 \), \( C_1 > 0, C_2 > 0, C_3 > 0, \) and \( C_2 > C_1 \), we have

1. If \( ar(C'_2 - C'_1) \exp[-rW^*(T_S)] > (C_3 + C_0) \) and \( ar(C'_2 - C'_1) \exp[-rW^*(T_{LC})] < (C_1 + C_0) \), \( T^* = \max(T_0, T_1) \) for \( R(\Delta T)S < R_0 < 1 \) or \( T^* = T_0 \) for \( 0 < R_0 \leq R(\Delta T)S \).
2. If \( ar(C'_2 - C'_1) \exp[-rW^*(T_S)] < (C_3 + C_0) \), \( T^* = T_1 \) for \( R(\Delta T)S < R_0 < 1 \) or \( T^* = T_S \) for \( 0 < R_0 \leq R(\Delta T)S \).
3. If \( P(T_{LC}) > C_3 \), \( T^* \geq T_1 \) for \( R(\Delta T)S < R_0 < 1 \) or \( T^* \geq T_S \) for \( 0 < R_0 \leq R(\Delta T)S \).

**Theorem 3.** Assume \( C_0(T) = C_{01} + C_0 \left( \int_{T}^{T} w(t) \, dt \right)^m \), \( C_{01}, C_0 > 0 \), \( C_1 > 0, C_2 > 0, C_3 > 0, \) and \( C_2 > C_1 \), we have

1. If \( ar(C'_2 - C'_1) \exp[-rW^*(T_S)] > C_3 \) and \( P(T_{LC}) < C_3 \), \( T^* = \max(T_0, T_1) \) for \( R(\Delta T)S < R_0 < 1 \) or \( T^* = T_0 \) for \( 0 < R_0 \leq R(\Delta T)S \).
2. If \( ar(C'_2 - C'_1) \exp[-rW^*(T_S)] < C_3 \), \( T^* = T_1 \) for \( R(\Delta T)S < R_0 < 1 \) or \( T^* = T_S \) for \( 0 < R_0 \leq R(\Delta T)S \).
3. If \( P(T_{LC}) > C_3 \), \( T^* \geq T_1 \) for \( R(\Delta T)S < R_0 < 1 \) or \( T^* \geq T_S \) for \( 0 < R_0 \leq R(\Delta T)S \).

From the above three theorems, we can obtain the optimal software release policy based on cost and reliability criteria. Tables 11–14 show the cost-reliability optimal release time under different \( R_0, \Delta t \), and cost functions.

<table>
<thead>
<tr>
<th>( P )</th>
<th>Optimal release time ( T^* (\Delta t = 0.1) )</th>
<th>Optimal release time ( T^* (\Delta t = 0.2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>22.695</td>
<td>24.328</td>
</tr>
<tr>
<td>0.02</td>
<td>22.719</td>
<td>24.352</td>
</tr>
<tr>
<td>0.03</td>
<td>22.742</td>
<td>24.376</td>
</tr>
<tr>
<td>0.04</td>
<td>22.766</td>
<td>24.399</td>
</tr>
<tr>
<td>0.05</td>
<td>22.789</td>
<td>24.422</td>
</tr>
<tr>
<td>0.06</td>
<td>22.812</td>
<td>24.445</td>
</tr>
<tr>
<td>0.07</td>
<td>22.835</td>
<td>24.468</td>
</tr>
<tr>
<td>0.08</td>
<td>22.858</td>
<td>24.490</td>
</tr>
<tr>
<td>0.09</td>
<td>23.203</td>
<td>24.513</td>
</tr>
<tr>
<td>0.10</td>
<td>24.284</td>
<td>24.535</td>
</tr>
<tr>
<td>0.11</td>
<td>26.111</td>
<td>26.111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( P )</th>
<th>Optimal release time ( T^* (\Delta t = 0.1) )</th>
<th>Optimal release time ( T^* (\Delta t = 0.2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>20.927</td>
<td>22.580</td>
</tr>
<tr>
<td>0.02</td>
<td>20.952</td>
<td>22.604</td>
</tr>
<tr>
<td>0.03</td>
<td>20.976</td>
<td>22.674</td>
</tr>
<tr>
<td>0.04</td>
<td>20.999</td>
<td>22.744</td>
</tr>
<tr>
<td>0.05</td>
<td>21.024</td>
<td>22.842</td>
</tr>
<tr>
<td>0.06</td>
<td>21.047</td>
<td>22.864</td>
</tr>
<tr>
<td>0.07</td>
<td>21.070</td>
<td>22.882</td>
</tr>
<tr>
<td>0.08</td>
<td>21.093</td>
<td>22.895</td>
</tr>
<tr>
<td>0.09</td>
<td>21.354</td>
<td>23.702</td>
</tr>
<tr>
<td>0.10</td>
<td>21.709</td>
<td>23.724</td>
</tr>
<tr>
<td>0.11</td>
<td>22.123</td>
<td>23.746</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( P )</th>
<th>Optimal release time ( T^* (\Delta t = 0.1) )</th>
<th>Optimal release time ( T^* (\Delta t = 0.2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>22.695</td>
<td>24.328</td>
</tr>
<tr>
<td>0.02</td>
<td>22.719</td>
<td>24.352</td>
</tr>
<tr>
<td>0.03</td>
<td>22.742</td>
<td>24.376</td>
</tr>
<tr>
<td>0.04</td>
<td>22.766</td>
<td>24.399</td>
</tr>
<tr>
<td>0.05</td>
<td>22.789</td>
<td>24.422</td>
</tr>
<tr>
<td>0.06</td>
<td>22.812</td>
<td>24.445</td>
</tr>
<tr>
<td>0.07</td>
<td>22.835</td>
<td>24.468</td>
</tr>
<tr>
<td>0.08</td>
<td>22.858</td>
<td>24.490</td>
</tr>
<tr>
<td>0.09</td>
<td>22.880</td>
<td>24.513</td>
</tr>
<tr>
<td>0.10</td>
<td>22.903</td>
<td>24.535</td>
</tr>
<tr>
<td>0.11</td>
<td>22.925</td>
<td>24.557</td>
</tr>
</tbody>
</table>

5. Conclusions

In this paper, we first present a SRGM with generalized logistic TEF. It is a much more realistic model and more suitable for describing the software fault detection and removal process. On the other hand, in practice, sometimes it is difficult for software developers to locate the faults that have caused the failure based on the test.
data and the data were reported in the test log and test anomaly documents. Sometimes software managers may require that the developers have to detect more faults due to schedule pressure. In this case, we discuss the problem of testing-effort control and management. Adequately adjusting some specific parameters of a SRGM and adopting the corresponding actions in the proper time interval can greatly help us to speedup getting the desired solution sometimes. Besides, it is advisable to introduce new test techniques, which are fundamentally different from the methods in use. These test techniques can help developers get their product done quicker and more reliably. Thus, we further study the effects of introducing consultants or new test techniques for increasing software testing-efficiency. Finally, we discuss the optimal release policy based on cost and reliability considering testing effort and efficiency. The procedure for determining the optimal release time has been developed in detail and the optimal release time has been shown to be finite.

Acknowledgment

This research was supported by the National Science Council, Taiwan, under Grant NSC 93-2213-E-007-088.

References


Chin-Yu Huang is currently an Assistant Professor in the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan. He received the M.S. (1994) and the Ph.D. (2000) in Electrical Engineering from National Taiwan University, Taipei. He was with the Bank of Taiwan from 1994 to 1999 and was a senior software engineer at Taiwan Semiconductor Manufacturing Company from 1999 to 2000. Prior to joining NTHU in 2003, he was a division chief in the department of planning and design, the Central Bank of China, Taipei. His research interests are software reliability engineering, software testing, software safety and security, software quality assurance, software metrics, and software fault-tolerance, etc.