Optimal Software Release Policy Based on Cost and Reliability with Testing Efficiency

Chin-Yu Huang[1], Sy-Yen Kuo[1], and Michael R. Lyu[2]
[1]Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan
sykuo@cc.ee.ntu.edu.tw
[2]Computer Science & Engineering Department
The Chinese University of Hong Kong
Shatin, Hong Kong
lyu@cse.cuhk.edu.hk

Abstract

In this paper, we study the optimal software release problem considering cost, reliability and testing efficiency. We first propose a generalized logistic testing-effort function that can be used to describe the actual consumption of resources during the software development process. We then address the problem of how to decide when to stop testing and when to release software for use. In addressing the optimal release time, we consider cost and reliability factors. Moreover, we introduce the concept of testing efficiency, and describe how reliability growth models can be adjusted to incorporate this new parameter. Theoretical results are shown and numerical illustrations are presented.

1. Introduction

For a large-scale or international software company, successful development of a software system depends on its software components. Therefore, the reliability of a large software system needs to be modeled/analyzed during the software development process. The future failure behavior of a software system is predicted by studying and modeling its past failure behavior[1-3]. It is very important to ensure the quality of the underlying software systems in the sense that they perform their functions correctly. Recently, we [4-5] proposed a new software reliability growth model which incorporates the concept of logistic testing-effort function into an NHPP model to get a better description on the software fault phenomenon. In this paper, we will extend the logistic testing-effort function into a more generalized form and show that the generalized logistic testing-effort function has the advantage of relating the work profile more directly to the natural structure of software development through experiments on real data sets. In practice, if we want to detect more additional faults, it is advisable to introduce new tools/techniques, which are fundamentally different from the methods currently in use. The advantage of these methods is that they can design/ propose several testing programs or automated testing tools to meet the client's technical requirements, schedule, and budget. If software companies can afford a bigger budget for testing and debugging, a project manager can maximize the software reliability. Hence the cost trade-off of new techniques/tools can be considered in the software cost model and viewed as the investment required to improve the long-term competitiveness. Therefore, in this paper, in addition to modeling the software fault-detection process, we will discuss the optimal release problem based on cost and reliability considering testing-effort and efficiency.

2. Testing-effort function and software reliability modeling

2.1. Review of SRGM with logistic testing-effort function

If we let the expected number of faults be $N(t)$, with mean value function as $m(t)$, then an SRGM based on NHPP can be formulated as a Poisson process:

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

Furthermore, if the number of faults detected by the current testing-effort expenditures is proportional to the number of remaining faults, then we have the following differential equation:

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

where $m(t)$ is the expected mean number of faults detected in time $(0, t]$, $w(t)$ is the current testing-effort consumption at time t, a is the expected number of initial faults, and $r>0$ is the error detection rate per unit testing-effort at time t.

Solving the above differential equation, we have

$$m(t)=a(1-\exp[-r(W(t)-W(0))])=a(1-\exp[-rW^*(t)]) \tag{2}$$

Recently, we [4-5] proposed a *Logistic* testing-effort

0-7695-0368-3/99 $10.00 © 1999 IEEE

function to describe the possible test effort patterns. The current testing-effort consumption is

$$w(t) = \frac{NA\alpha \times \exp[-\alpha t]}{(1 + A\exp[-\alpha t])^2} = \frac{NA\alpha}{(\exp[\frac{\alpha t}{2}] + A\exp[-\frac{\alpha t}{2}])^2} \tag{3}$$

where N is the total amount of testing effort to be eventually consumed, α is the consumption rate of testing-effort expenditures, and A is a constant.
The cumulative testing effort consumption of *Logistic* testing-effort function in time (0, t] is

$$W(t) = \frac{N}{1 + A\exp[-\alpha t]} \text{ and } W(t) = \int_0^t w(t)dt \tag{4}$$

2.2. A generalized logistic testing-effort function

From the previous studies in [4-5], we know that the *Logistic* testing-effort function (i.e. the Parr model [8]) is based on a description of the actual software development process and can be used to describe the work profile of software development. In addition, this function can also be used to consider and evaluate the effects of possible improvements on software development methodology, such as top-down design, stepwise refinement or structured programming. Therefore, if we relax some assumptions when deriving the original Parr model and take into account the structured development effort, we get a generalized *Logistic* testing-effort function

$$W_k(t) = \frac{N \times \sqrt[\kappa]{(r+1)/\beta}}{\sqrt[\kappa]{1 + A\exp[-\alpha\kappa t]}} \tag{5}$$

where κ is the structuring index and its value is large for modeling well-structured software development efforts and β is a constant.
When κ=1, the above equation becomes

$$W_k(t) = \frac{N}{1 + A\exp[-\alpha t]} \times \frac{2}{\beta} \tag{6}$$

If β is viewed as a normalized constant and we have β=2, the above equation is equal to Eq. (4) [5]. Similarly, if β=κ+1, we get a more generalized and plain solution for describing the cumulative testing effort consumption in time (0, t]: $W_k(t) = \frac{N}{\sqrt[\kappa]{1 + A\exp[-\alpha\kappa t]}}$ (7)

3. New tools/techniques for increased efficiency of software testing

It is well known that when the software coding is completed, the testing phase comes next and it is a necessary but expensive process. Once all the detectable faults are removed from a new software, the testing team will need to determine when to stop testing and make a software risk evaluation. If the results meet their requirement specifications and the related criteria are also satisfied, the team will adorn and announce that this software product is ready for releasing/selling. Therefore, adequately adjusting some specific parameters of a SRGM and adopting the corresponding actions in the proper time interval can greatly help us to speedup obtaining the desired solution. In fact, several existing approaches can satisfy our requirements. For example, we have discussed the applications of testing-effort control and management problem in our previous studies [4]. The methods we proposed can easily control the modified consumption rate of testing-effort expenditures and could detect more faults in the prescribed time interval. It means that the developers/ testers will devote all their available knowledge/energy to complete such tasks without additional resources. Alternative to controlling the testing-effort expenditures, we believe that new testing schemes will help achieving a given operational quality at a specific time. That is, through some new techniques/tools, we can detect/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 the extra software development cost. In practice, if we want to detect more potential faults, we may introduce new techniques/tools that are not yet used, or bring in experts to make a radical software risk analysis. In addition, there are newly proposed automated testing tools/techniques for increasing test coverage and can be used to replace traditional manual software testing regularly. The benefits to software developers/testers include increased software quality, reduced testing costs, improved release time to market, repeatable test steps, and improved testing productivity. These techniques can make software testing and correction easier, detect more bugs, save more time, and reduce expenses significantly [3]. Altogether, we hope that the experts, automated testing tools or techniques could greatly help us in detecting additional faults that are difficult to find during regular testing and usage, in identifying and correcting faults more cost effectively, and in assisting clients to improve their software development processes. To conclude, we introduce a gain parameter (GP) to describe the behavior or characteristics of automated testing techniques/tools and incorporate it into the mean value function [11]. Therefore, the modified mean value function is depicted in the following:

$$m_e(t) = a(1 - \exp[-r\sigma W_\kappa^*(t)]), t \geq T_s$$
$$m_e(t) = a(1 - \exp[-rW_\kappa^*(t)]), t < T_s < T \tag{8}$$

where σ is the gain parameter (GP) and T_s is the starting time of adopting new techniques/tools.
Eq. (8) means that if σ increases, $m_e(t)$ increases. Thus the gain of employing new automated techniques/tools is

$$\frac{m_e(t)}{m(t)} = (1+P) \begin{cases} P > 0, & t \geq T_s \\ P = 0, & t < T_s \end{cases} \tag{9}$$

where P is the additional fraction of faults detected by using new automated tools or techniques during testing. Substituting Eq. (8) into Eq. (9) and rearranging the above equations, we obtain the estimated value of gain parameter:

$$\sigma = \frac{-1}{r \times \left(W_\kappa(t) - W_\kappa(0)\right)} \ln\left((1+P) \times \exp[-rW^*_\kappa(t)] - P\right) \tag{10}$$

In fact, we can interpret the gain parameter from different views. If we make a premise that the main goal of automated techniques/tools is to test/debug software with less testing effort, then the gain parameter σ 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 same testing effort, introducing new automated tools/methods should help us in detecting/removing more additional faults which are hard to detect without these new methods. Actually the most important thing is how to provide enough information about these approaches to the test team. Before adopting these automated techniques/tools, we should get the quantitative information from the industrial data relative to the 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. Certainly, the methods' past performance in aiding the reliability growth should be considered in determining whether they will be successful again or not [11]. The distribution of σ can be estimated by performing various simulations based on actual data sets. Additionally, the test teams' capacity of exercising these techniques/tools and the related environmental profiles also play an important role in achieving the desired goals. Here, we illustrate the new parameter with one real numerical example. This data set is from Ohba [10]. If we are given the data sets whose parameters for test-effort function are in Table 1, then we can analyze the distribution of the gain parameter in various ways as follows.

Table 1: Parameters of generalized logistic testing-effort function and mean value function

N	A	α	κ	a	r
54.8364	13.0334	0.226337	1	394.076	0.0427223
52.0072	40.6042	0.188809	1.5	384.707	0.0450374
50.2178	115.228	0.170001	2	377.157	0.0478156
48.7768	429.673	0.158042	2.63	369.029	0.0509553
48.1368	833.105	0.151344	3	367.829	0.0519053
48.1693	2188.22	0.144234	3.5	412.871	0.0399382
47.8507	5709.29	0.139933	4	414.426	0.0398619
47.6561	14839.3	0.136507	4.5	416.114	0.0397324

From [10], we know that when the testing is completed (about the 19th week), the number of faults detected is about 328. If we want to increase additional 0.01, 0.03, 0.05, 0.07, 0.09, 0.1, 0.11 and 0.12 fraction of detected faults respectively, we must ensure that the gain parameter σ will be corresponding to growth as plotted in Fig. 1. That is, the performance and the related assistance of new tools/ methods should fit the growth curve as time progresses. If the trend fits, it means that through these new methods, we can adjust the consumption of testing-effort expenditures or raise the fault detection rate. In fact, in this data set, we find that at the end of testing, P=0.01, 0.02,..., 0.09, 0.1 or even 0.11, but P=0.12 is hard. The reason is that the value of 1-$m(t)/a$ controls the maximum possible value of P. That is, we should consider whether or not introducing new techniques/tools to detect additional faults only when the precondition: (1-$m(t)/a$)>P is satisfied. Finally, these tools or techniques we discussed have a big impact on software testing and reliability. In fact, they can provide the developers or test teams with feedback of useful information on the testing process for improvement as well as scheduling. However, we know that with more efficient testing, more time can be spent to make this software more reliable. Besides, introducing these new methods also requires that the original design of software be modified to get the best performance. Thereafter, the continued use of these tools/techniques can improve the software design.

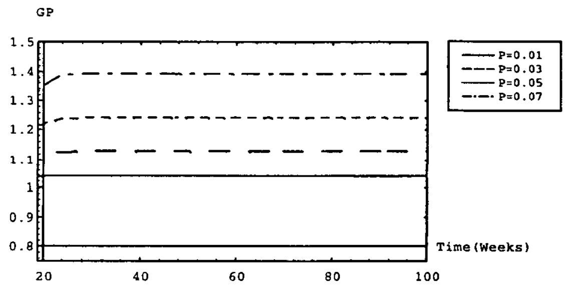

Figure 1: GP graph for the data set (κ=2.63326).

4. Optimal software release time

When the software testing is completed, software product is ready for release to users. However, proper timing is very important. If the reliability of the software does not meet the manager's goal, the developers or testers may request external help in testing. An optimal release policy for the proposed model based on such considerations is studied here.

4.1. Software release time based on cost criterion

Okumoto and Goel [6] were the first to discuss the software optimal release policy from the cost-benefit viewpoint. Using the total software cost evaluated by cost criterion, the cost of testing-effort expenditures during software development phase and the cost of correcting errors before and after release are:

$$C1(T) = C_1 m(T) + C_2[m(T_{LC}) - m(T)] + C_3 \times \int_0^T w(x)dx$$

Generally, in order to detect additional faults during testing, the test teams/debuggers may use new automated tools or techniques if they are available. Hence the cost trade-off of tools should be considered in software cost model. But they thereby save some of the greater expense of correcting errors during operation. By summing up the above cost factors, the modified software cost model is

$$C2(T) = C_0(T) + C_1 \times (1+P) \times m(T) + C_2[m(T_{LC}) - (1+p) \times m(T)] + C_3 \times \int_0^T w(x)dx \quad (11)$$

where $C_0(T)$ is the cost function of including automated tools/techniques to detect an additional fraction P of faults during testing.

In fact, the cost of a new tool or 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. By differentiating Eq. (11) with respect to T and let $C_1(1+P)= C_1^*$ and $C_2(1+P)= C_2^*$, we have

$$\frac{d}{dT}C2(T) = \frac{d}{dT}C_0(T) + C_1^* arw(T)\exp[-rW^*(T)] - C_2^* arw(T) \times \exp[-rW^*(T)] + C_3 \times w(T) \quad (12)$$

However, the assumption that $C_0(T)$ is a constant may not be realistic in many situations. In addition, the assumption may lead to ill-defined testing. Therefore, we relax this assumption and explore the results. Here we propose two possibilities for $C_0(T)$ in order to interpret the cost consumption: (1) $C_0(T)$ is a constant [5], and (2) $C_0(T)$ is proportional to the expenditures of testing-effort. The second assumption considers that C_0(T) is a convex non-decreasing function of testing time t and initially is zero, while the time in progress increases linearly or non-linearly. For example, if in addition to introducing new automated tools/ techniques, we also adopt effective approaches from senior mentors in a professional software consultant company. Normally the testers can solve general problems by themselves when faults occur. But if some problems are so difficult to solve, they must ask the consultants to get more proper solutions. Therefore, we can conceive that such extra cost may include travel expenses to clients, charges for the phone support or root cause analysis of software faults, etc [3]. Sometimes they also provide the services for code inspections, diagramming, unit testing, and test planning. Generally, if using good automated approaches for software fault detection or useful/powerful supports during software development becomes available, the testers can detect more faults. But the cost will be higher. Therefore, if $C_0(T) = C_{01} + C_0 \times (\int_{Ts}^T w(t)dt)^m$, $T \geq T_s$ and $C_0(T)= 0$, $T<T_s$, then we get

$$\frac{d}{dT}C2(T) = w(t) \times [ar(C_1^* - C_2^*) \times \exp[-rW^*(t)] + C_3 + C_0 \times m \times (\int_{Ts}^T w(t)dt)^{m-1}].$$

Since $w(t)>0$ for $0 \leq T < \infty$, therefore, $\frac{d}{dT}C2(T)=0$ if $P(T) \equiv$

$$\{ar(C_2^* - C_1^*)\exp[-rW^*(t)] - C_0 m(\int_{Ts}^T w(t)dt)^{m-1}\} = C_3 \quad (13)$$

The left-side in Eq. (13) is monotonically decreasing function of T. Therefore, if $ar \times (C_2^* - C_1^*)\exp[-r(W(T_s) - W(0))] > C_3$ and $P(T_{LC})<C_3$, it means that there exists a finite and unique solution T_0 satisfying Eq. (13) which can be solved by numerical methods. It is noted that $\frac{d}{dT}C2(T) <0$ for $0 \leq T_s \leq T < T_0$ and $\frac{d}{dT}C2(T) >0$ for $T>T_0$. Thus, $T=T_0$ minimizes $C2(T)$ for $T_0 <T_{LC}$.

4.1.1. Numerical Example 1. Here we illustrate how to minimize the software cost in which the new automated tools/techniques are introduced during testing. From the previous estimated parameters for the data set in Table 1, we get N=48.7768, A=429.673, α=0.158042, κ=2.63326, a=369.029, r=0.0509553, C_{01}=\$1000, C_1=\$10 per error, C_2=\$50 per error, C_0=\$10, T_s=19, C_3=\$100 per unit testing-effort expenditures, and T_{LC}=100 weeks. The numerical example for the relationship between the cost optimal release time and P is given in Table 2. From Table 2, we find that the bigger the P, the larger the optimal release time and the smaller the total expected software cost. The reason is that if we have better testing performance, we can detect more latent or undetected faults through additional techniques/tools. Therefore, we can really shorten the testing time and release this software earlier. Here, we observe some facts as follows:

(1) When P is relatively small (such as 0.01, 0.02, or 0.03,..., 0.06), the total expected cost is larger than the expected value of traditional cost model (i.e. 4719.66). The reason is due to C_{01}, i.e. the basic cost of adopting new automated techniques/tools.

(2) As P increases, the optimal release time T^* increases but the total expected software cost $C(T^*)$ decreases since we can detect more faults and reduce the cost of correcting the errors during operational phase.

(3) Even under the same P and with different cost functions, the larger the cost, the smaller the optimal release time. But there is insignificant difference in estimating the total expected software cost.

Table 2: Relationship between the cost optimal release time T_o*, $C(T_o*)$, and P based on the cost function $C_0(T) = 1000 + 10 \times (\int_{19}^{100} w(t)dt)^{1.2}$

P	T_0*	$C(T_0*)$	P	T_0*	$C(T_0*)$
0.01	19.6006	5573.46	0.07	20.7684	4615.04
0.02	19.7501	5414.07	0.08	20.0436	4454.81
0.03	19.9157	5254.54	0.09	21.3539	4294.43
0.04	20.0983	5094.88	0.10	21.7086	4133.91
0.05	20.2998	4935.07	0.11	22.1225	3973.24
0.06	20.5221	4775.13			

4.2. Software release time based on reliability criterion

In general, the software release time problem is also associated with the reliability of a software system. Hence, if we know that the software reliability has reached an acceptable reliability level, we can determine the right time to release this software. Software reliability is defined as the probability that a software failure doesn't occur in (T, T+Δt] given that the most recent failure occurred at T [1-2, 5, 6, 12]. Therefore,

$$R(\Delta t \mid T) = \exp[-(m_e(T+\Delta t) - m_e(T))] \text{ , } \Delta t, T \geq 0 \quad (14)$$

In addition, we also define the second measure of software reliability for the proposed model, i.e., the ratio of the cumulative number of detected faults at time T to the expected number of initial faults.

$$R_2(T) \equiv m_e(T)/a \qquad (15)$$

We can solve this equation and obtain a unique T_l satisfying $R_2(T_l)=R_0$. It is noted that the larger the value of $R_2(T)$, the higher the software reliability.

4.2.1. Numerical Example 2. Tables 3 shows the relationships between the reliability optimal release time T_l*, Δt and P based on R_0= 0.95. From Tables 3, we find that as P increases, the optimal release time T_l* increases. The reason is as follows. From Eq. (14), we know that $R(\Delta t|T)$ denotes the conditional reliability function that the software will still operate after T+Δt given that it has not failed after time T [5-6]. In addition, from Eq. (8), we know $m_e(t)$=(1+P)$m(t)$. That is, $m_e(T+\Delta t) - m_e(T) = (1+P) \times (m(T+\Delta t) - m(T)) \geq (m(T+\Delta t) - m(T))$. Therefore, $-(m_e(T+\Delta t) - m_e(T)) \leq -(m(T+\Delta t) - m(T))$ and $\exp[-m_e(T+\Delta t) - m_e(T))] \leq \exp[-(m(T+\Delta t) - m(T))]$.
Hence, in Table 3, the reason why the optimal release time T_l*=22.6945 under P=0.01, Δt=0.1, and R_0=0.95 is slightly larger thn the optimal release time (without introducing any extra automated tools during testing) T_l*=22.6702 under Δt=0.1, R_0=0.95 is obvious. On the other hand, through using these new tools, we may detect some extra faults in (T, T+Δt] and these faults are potentially hard to detect/locate in (T, T+Δt] if these automated tools are not available. In fact, if the faults are hard to locate/detect after a long period of testing, the tester may treat the software system as a reliable/stable system. This phenomenon could occur in practice and it is significant because the ability or knowledge of testers/developers is limited. In this case (without introducing any extra automated tool), the target reliability $R(\Delta t|T_l)$=R_0 may be achieved at time T_l, the reliability optimal release time. In fact, through new automated techniques/tools, it is probably easier to find these extra latent faults in the interval (T_l, T_l+Δt]. Therefore, the reliability optimal release time will be delayed till the reliability goal is reached.

Table 3: Relationship between the reliability optimal release time T_1* and P based on the first measure of software reliability R_o=0.95.

P	T_l* (Δt=0.1)	T_l* (Δt=0.2)	P	T_l* (Δt=0.1)	T_l* (Δt=0.2)
0.01	22.6945	24.3281	0.07	22.8351	24.4677
0.02	22.7185	24.3519	0.08	22.8578	24.4902
0.03	22.7423	24.3755	0.09	22.8803	24.5125
0.04	22.7659	24.3989	0.10	22.9025	24.5346
0.05	22.7892	24.422	0.11	22.9245	24.5565
0.06	22.8123	24.445			

4.3. Software release time based on cost-reliability criterion considering efficiency

From subsection 4.2, we can easily get 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 minimizing C(T), subject to $R(\Delta t|T) \geq R_0$ where 0<R_0<1., i.e.,

T* = optimal software release time = ***max***(T_0 ,T_l)

where T_0= finite and unique solution T satisfying Eq. (13), and T_l= finite and unique T satisfying Eq. (14), or Eq. (15). Combining the cost and reliability requirements and considering the efficiency, we have the following theorem.

Theorem 1:

Assume $C_0(T) = C_{01} + C_0 \times (\int_{Ts}^{T} w(t)dt)^m$, C_{0l}>0, C_0>0, C_l>0, C_2>0, C_3>0, and C_2>C_l, we have

(1) if $ar \times (C_2^* - C_1^*)\exp[-r(W(T_s) - W(0))] > C_3$ and $P(T_{LC})$ <C_3, T^* = ***max***(T_0, T_l) 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 $ar \times (C_2^* - C_1^*)\exp[-r(W(T_s) - W(0))] < C_3$, T^*=T_l 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 $P(T_{LC})>C_3$, $T^* \geq T_l$ for $R(\Delta t|T_s)<R_0<1$ or $T^*\geq T_s$ for $0<R_0\leq R(\Delta t|T_s)$.

From the above theorem, we can easily determine the optimal software release time based on the cost and reliability requirements considering efficiency. Table 5 shows the cost-reliability optimal release time under different R_0, Δt, and cost function. Similarly, Table 6 shows the relationship between the optimal release time T_l* based on the target reliability $R_2(T)$ and P. From these tables and following the same arguments in Theorem 1, we can obtain the optimal software release time based on the cost and reliability criteria.

Table 5: Relationship between the reliability optimal release time T* and P with R_0=0.95 and cost function $C_0(T) = 1000+10\times(\int_{19}^{100} w(t)dt)^{1.2}$

P	*T** (Δt=0.1)	*T** (Δt=0.2)	*P*	*T** (Δt=0.1)	*T** (Δt=0.2)
0.01	22.6945	24.3281	0.07	22.8351	24.4677
0.02	22.7185	24.3519	0.08	22.8578	24.4902
0.03	22.7423	24.3755	0.09	22.8803	24.5125
0.04	22.7659	24.3989	0.10	22.9025	24.5346
0.05	22.7892	24.422	0.11	22.9245	24.5565
0.06	22.8123	24.445			

Table 6: Relationship between the cost optimal release time T_l* based on reliability $R_2(T)$ and P

P	$R_2(T)$	T_l*	*P*	$R_2(T)$	T_l*
0.01	0.900	23.8117	0.07	0.955	26.4809
0.02	0.910	25.4759	0.08	0.962	23.4478
0.03	0.915	21.8179	0.09	0.972	24.6907
0.04	0.925	22.4203	0.10	0.982	27.2990
0.05	0.930	20.7667	0.11	0.991	27.6343
0.06	0.945	24.3389			

5. Conclusions

In this paper we present an SRGM with generalized testing-effort function. It is a much more realistic model and more suitable for describing the software fault detection/removal process. Furthermore, we also discussed the effects of introducing new tools/techniques for increased software testing efficiency, and studied the related optimal software release time problem from the cost-reliability viewpoint. The procedure for determining the optimal release time *T** has been developed and the optimal release time has been shown to be finite. In practice, sometimes it is difficult for us to locate the faults that have caused the failure based on the test data reported in the test log and test anomaly documents. Therefore, it is advisable to introduce new tools/techniques, which are fundamentally different from the methods in use. In addition, there are still many more potential cost functions. We will present further investigations on describing the mathematical properties of the SRGM with generalized testing-effort function in the future.

Acknowledgment

We would like to express our gratitude for the support of the National Science Council, Taiwan, R.O.C., under Grant NSC 87-TPC-E-002-017. Partial support of this work (by Michael R. Lyu) is provided by a UGC direct grant at CUHK. Referees' insightful comments and suggestions are also highly appreciated.

References

[1] S. Yamada, J. Hishitani, and S. Osaki, "Software Reliability Growth Model with Weibull Testing Effort: A Model and Application," *IEEE Trans. on Reliability*, Vol. R-42, pp. 100-105, 1993.

[2] S. Yamada and S. Osaki, " Cost-Reliability Optimal Release Policies for Software Systems," *IEEE Trans. on Reliability*, Vol. 34, No. 5, pp. 422-424, 1985.

[3] J. D. Musa (1998). *Software Reliability* Engineering: *More Reliable Software, Faster Development and Testing.* McGraw-Hill.

[4] C. Y. Huang, J. H. Lo and S. Y. Kuo, "A Pragmatic Study of Parametric Decomposition Models for Estimating Software Reliability Growth," *Proceedings of the 9th International Symposium on Software Reliability Engineering* (ISSRE'98), pp. 111-123, Nov. 4-7. 1998, Paderborn, Germany.

[5] C. Y. Huang, S. Y. Kuo and I. Y. Chen, "Analysis of a Software Reliability Growth Model with Logistic Testing-Effort Function," *Proceedings of the 8th International Symposium on Software Reliability Engineering* (ISSRE'97), pp. 378-388, Nov. 1997, Albuquerque, New Mexico. U.S.A.

[6] K. Okumoto and A. L. Goel, "Optimum Release Time for Software Systems Based on Reliability and Cost Criteria," *Journal of Systems and Software*, Vol. 1, pp. 315-318, 1980.

[7] M. R. Lyu (1996). *Handbook of Software* Reliability *Engineering.* McGraw Hill.

[8] F. N. Parr, "An Alternative to the Rayleigh Curve for *Software* Development Effort," *IEEE Trans. on Software Engineering*, SE-6, pp. 291-296, 1980.

[9] M. Lipow, "Prediction of Software Failures," *Journal of Systems and Software*, Vol. 1, pp. 71-75, 1979.

[10] M. Ohba, " Software Reliability Analysis Models," *IBM J. Res. Develop.*, Vol. 28, No. 4, pp. 428-443, July 1984.

[11] J. Farquhar and A. Mosleh, "An Approach to Quantifying Reliability-Growth Effectiveness," *Proceedings Annual Reliability and Maintainability Symposium*, pp. 166-173, 1995.

[12] P. K. Kapur and R. B. Garg, "Cost-Reliability *Optimum* Release Policies for a Software System under Penalty Cost," *Int. J. of Systems Science*, Vol. 20, pp. 2547-2562, 1989.

[13] M. R. Lyu and A. Nikora, "Using Software Reliability Models More Effectively," *IEEE Software*, pp. 43-52, 1992.