International Journal of Reliability and Applications
Vol. 3, No. 2, pp. 81-90, 2002

An Effective Stopping Rule for Software Reliability Testing

Bok Sik Yoon*
Department of Science
Hongik University, Seoul 121-791, Korea

Abstract. The importance of the reliability of software is growing more and more as more complicated digital computer systems are used for real-time control applications. To provide more reliable software, the testing period should be long enough, but not unnecessarily too long. In this study, we suggest a simple but effective stopping rule which can provide just proper amount of testing time. We take unique features of software into consideration and adopt non-homogeneous Poisson process model and Bayesian approach. A numerical example is given to demonstrate the validity of our stopping rule.

Key Words : *Software Reliability, Stopping Rule, Bayesian Approach.*

1. INTRODUCTION

As digital computer systems are applied more and more for the control of complicated large-scale systems, the relative role of software becomes important more and more in the system reliability. The failure of software in the nuclear power plant control system can cause severe safety problems and the error in the space shuttle system control software may delay the project. That is, the importance of the reliability of software is growing more and more as more complicated digital computer systems are used for real-time control applications.

Since early 1970's the importance of software reliability has been recognized, and coniderable efforts have been made for this topic. The researchers soon understood that concepts and techniques used for hardware reliability cannot be applied successfully for software directly. Software is considered to have some unique features, which make the direct application of the techniques used for the hardware reliability analysis undesirable and force us to develop its own model. Due to the increase of interests and efforts for software reliability modeling, we currently have several models which can be applied to analyze software reliability as surveyed in [14] - [16].

Up to now there were considerable amount of research efforts for the stopping time problem in reliability testing[5,13]. The question of when to stop testing the software and release it for its intended use may occur in every software development. The objective of this study is to find a simple stopping rule which can be used conveniently in practical situations. Since the stopping problem is a

* *E-mail address : bsyoon@wow. hongik.ac.kr*

type of decision problem, it will be desirable to approach on the basis of the life-time cost[10]. However, in this study the life-time cost minimization is not considered to simplify the stopping rule. Also, in this study continuous execution time is chosen as a time domain because of the following two reasons: First, the real-world software problem usually occurs in large scale systems such as nuclear power control system, space shuttle control system or utility program, where lots of different inputs are intermittently received. Second, we can get simpler models using continuous time.

To derive a reasonable stopping rule for software reliability testing, special features of software and its reliability should be understood first. Based on them, we can select proper models and measures to be used for stopping rule. In section 2, we will briefly discuss some features of both software and basic software reliability models. In section 3, Goel-Okumoto model, which is selected to demonstrate the stopping rule, is introduced. Also, the parameter estimation is performed according to Bayesian approach. In section 4, a stopping rule is introduced and in section 5, a numerical example is given. The conclusion is followed in section 6.

2. SPECIAL FEATURES OF SOFTWARE RELIABILITY AND MODELING

2.1 Special features of software reliability

In this subsection, we will see what makes the difference between software and hardware, and then in the following subsection we will scan several models developed for software reliability analysis briefly.

Special features which distinguish software reliability models from hardware reliability model are:

(1) Software does not age with time. Once it becomes perfect, it will never fail and possibly the probability of no error can be greater than zero in small substructures. Thus, the percentile of failure distribution, operational reliability, or failure rates is more intuitive than MTTF[10].

(2) The failure occurs due to human errors in design and logic. Software failure occurs only when some random inputs reveal unexpected errors. This brings us to the shock model[2,3,9]. Hidden bugs may be detected and diagnosed during the debugging stage and accordingly one of the main concern of software reliability is the reliability growth in the 'burn-in' period[9,12]. But, it is different from hardware burn-in period in the sense that we cannot be sure of the correctness of the debugging and the failure rate will not increase after the burn-in period.

(3) Every software is unique. Because there are no replicated items like semiconductors, we cannot have repetitive data. This suggests us to follow Bayesian approach.

(4) We cannot use redundant components to increase system reliability. But, we may use the concept of components through the structural design of programming [10]. Also, a fault tolerant design based on diversity concept is possible[11].

Based on above features, we can grasp some desirable properties of software reliability models. Actually, Langberg and Singpurwalla(1985) tried to generalize the software reliability model using shock model and Bayesian approach[1,3,9,12]. In the following section, we will scan Langberg and Singpurwalla (L-S) model and three other historic models which can be generalized by L-S model. More detailed analysis of these models can be found in [9]. Also, other software reliability models are appeared in [6], [14], [15].

2.2 Basic software reliability models

(1) Jelinski-Moranda(J-M) model (1972) [8]

The form of J-M model is

$$f_{T_i}(t:N,\Lambda)=(N-i-1)\Lambda\exp(-\Lambda(N-i-1)t),\quad t\geq 0 \tag{1}$$

where N=the number of bugs in the program, Λ=failure rate of each bug, T_i =the elapsed time between i^{th} and $(i+1)^{st}$ failures, $i=1,\ldots,N$. That is, T_i's are iid with desity (1). In [8], N and Λ are estimated by maximum likelihood method.

(2) Littlewood-Verrall(L-V) model (1973) [10]

Littlewood and Verrall(1973) apply Bayesian inference for their model having the following form.

$$f_{T_i}(t:\Lambda_i)\ =\Lambda_i e^{-\Lambda_i t}, \tag{2}$$

where Λ_i has a Gamma(α,ϕ_i) prior.

By setting ϕ_i according to programmer's confidence about the correctness of i^{th} debugging, we can include the incomplete debugging case. If we want steadily decreasing failure rate, we can set ϕ_i as

$$\phi_i=\beta_0+\beta_1 i$$

(3) Goel-Okumoto(G-O) model (1979) [6]

This model will be discussed in detail in section 3. G-O model has the following form.

$$P(N(t)=y)=\frac{m(t)^y}{y!}e^{-m(t)}, \tag{3}$$

where $N(t)$= the number of failure up to time t, $m(t)=E(N(t))$. That is, $\{N(t),t\geq 0\}$ is considered to be an NHPP(non-homogeneous Poisson process).

(4) Langberg and Singpurwalla's generalization(1985) [9]

They starts with the shock model such as

$$F_{T_i}(t:\omega,N,N^*)=1-\exp(-\omega(N-i+1)t/N^*), \tag{4}$$

where ω= intensity function, N^*= Total number of inputs, N= Number of inputs which cause failure. If we set $\Lambda=\frac{\omega}{N^*}$, we get exactly same form as (1), that is,

$$F_{T_i}(t:N,\Lambda)=1-\exp(-\Lambda(N-i+1)t) \tag{5}$$

We can see that if we treat N and Λ as unknown constants(parameters), we get J-M model, and if we give gamma prior to Λ, assuming N known we get L-V model. Also, we assume Λ known and N to have Poisson prior we can derive G-O model.

These models reflect more or less the unique properties of software. Especially, failure rates in all of them are decreasing in time and at each debugging. Now, comparing the pattern of failure rate variation in each model, we can see that they all have DFRs in time, but the decreasing patterns are different. In J-M model the failure rate between two successive failures remains constant. In L-V model and G-O model it decreases both in time and at each debugging, but only L-V model can reflect the case of incomplete debugging, even though this can be made only by human judgement.

3. GOEL-OKUMOTO MODEL AND BAYESIAN INFERENCE

3.1 Goel-Okumoto model

In this study, G-O model is selected to demonstrate our stopping rule because of its intuitive appeal, mathematical simpleness and practical applicability. Let $N(t)$ be the cumulative number of failures by time t as in section 2. Assume that the counting process $\{N(t), t\geq 0\}$ has the property of independent increments, that is, the number of software failures during nonoverlapping time intervals do not affect each other. And assume $m(t)=E(N(t))$ is bounded and non-decreasing, and $m(0)=0$, $m(t)\to a$ as $t\to\infty$, where a is the exact number of errors(bugs) in the software. Assume further the following (6)-(8).

$$m(t+\triangle t)-m(t)=b(a-m(t))\triangle t+o(\triangle t) \tag{6}$$

$$P(N(t+\triangle t)-N(t)=1)=\lambda(t)\triangle t+o(\triangle t) \tag{7}$$

$$P(N(t+\triangle t)>1)=o(\triangle t) \tag{8}$$

From (6), we get

$$m'(t)=ab-bm(t) \tag{9}$$

and combining boundary conditions, we have

$$m(t)=a(1-\exp(-bt)). \tag{10}$$

or

$$m'(t)\equiv\lambda(t)=abe^{-bt}. \tag{11}$$

Under the assumption of independent increments, (7) and (8), $\{N(t), t\geq 0\}$ becomes an NHPP(Nonhomogeneous Poisson Process) with mean $m(t)$ and intensity $\lambda(t)$, i.e.

$$P(N(t)=y)=\frac{m(t)^y}{y!}e^{-m(t)}, \quad y=0,1,2,\ldots \tag{12}$$

Goel and Okumoto[4] used maximum likelihood estimation to estimate a and b. The likelihood function at the k^{th} failure time becomes

$$\begin{aligned} L(a,b:T_1,\ldots,T_k) &= L(a,b:S_1,\ldots,S_k) \\ &= \prod_{i=1}^{k} m'(S_i)\exp(-(m(S_i)-m(S_{i-1})) \\ &= (ab)^k\exp(-b\sum_{i=1}^{k}S_i)\exp(-a(1-e^{-bS_k})), \end{aligned} \tag{13}$$

where $S_0=0$, $S_i=\sum_{j=1}^{i}T_j$ = i^{th} failure time, $i=1,\ldots,k$. After taking log, we have

$$\log L = k(\log a + \log b) - b\sum_{i=1}^{k} S_i - a(1 - e^{-bS_k}).$$

From $\frac{\partial \log L}{\partial a} = 0$ and $\frac{\partial \log L}{\partial b} = 0$ we have

$$\frac{k}{a} = 1 - e^{-bS_k}, \tag{14}$$

$$\frac{k}{b} = \sum_{i=1}^{k} S_k + aS_k e^{-bS_k}. \tag{15}$$

By solving (14) and (15) numerically we can get MLE's for a and b. But the solution may not be unique. From the following Hessian H of $\log L$, we can see this fact.

$$H = \begin{bmatrix} -k/a^2 & -S_k e^{-bS_k} \\ -S_k e^{-bS_k} & -k/b^2 + aS_k^2 e^{-bS_k} \end{bmatrix}. \tag{16}$$

More specifically, $H_{11} < 0$ but the determinant of H is

$$|H| = \frac{k^2 - ab^2 S_k e^{-bS_k}(k + ae^{-bS_k})}{a^2 b^2}, \tag{17}$$

where $a > 0$, $0 < b < 1$, the sign of which depends on k and S_k.

3.2 Estimation by Bayesian approach

The derivation of G-O model in the last section was made by a probabilistic intuitive approach. Now we follow Bayesian approach to estimate the model parameters. The reason is two-fold: First, usually there are not sufficient number of data for reliable statistical estimation as mentioned in section 2. Second, MLE's are not convenient to compute as we can see in section 3.1. But in applying Bayesian method, some assumptions are needed to simplify the estimation procedure because the G-O model has two parameters.

We can see from (6) each parameter has its practical meaning:

$a =$ the expected number of total errors in the software

and

$$b = \frac{E(\#\ \textit{of errors detected during}\ (t, t+\Delta t))}{E(\#\ \textit{of errors remaining at}\ t)\Delta t}$$

= occurrence rate of an error.

From this, we assume reasonably a and b are independent and b, the occurrence rate of an error, is constant and a has a gamma prior

$$f(a : \alpha, \beta) = \beta^{\alpha} a^{\alpha - 1} e^{-\beta a} / \Gamma(\alpha).$$

Then,

$$P(a : b, t_1, \dots, t_k) \propto e^{-ca} a^k e^{-\beta a} a^{\alpha - 1} = e^{-(c+\beta)a} a^{\alpha + k - 1},$$

where $c = 1 - e^{-bS_k}$, from which we notice that the posterior of a is Gamma ($\alpha + k, \beta + c$). Now the Bayes estimator under squared error loss is

$$E(a : data, b) = \frac{\alpha + k}{\beta + c}. \tag{18}$$

Now, if b is constant but it may change as the testing goes on, the following procedure can be proposed.

Step 1. Find estimator $\hat{a} = E(a : \hat{b}, data)$ with previous $\hat{b}$.

Step 2. Update $\hat{b}$ using the equation $\frac{d\log L}{db} = 0$ with fixed $\hat{a}$.

Note that in step 2, the equation

$$\frac{d\log L}{db} = \frac{k}{b} - \sum_{i=1}^{k} S_i - \hat{a} S_k e^{-bS_k} = 0 \tag{19}$$

may have more than one solution and we need to check the negativity of

$$\frac{d^2\log L}{db^2} = \frac{-k}{b^2} + \hat{a} S_k^2 e^{-bS_k}$$

at the solution of (19) to confirm its maximality.

Note that our approach here is quite intuitive and empirical and different from the formal Bayesian approach[1].

4. A STOPPING RULE

As mentioned in section 2, the desirable measure in software reliability is the failure rate or operational reliability. In this section, a simple stopping rule which can be implemented conveniently in the practical situation is developed. Based on G-O model, we first calculate various measures which can be used for stopping criteria.

Let $N^c(t)$ = the number of errors remaining at time t. Then, since from (12) $N(\infty)$ follows Poisson distribution with mean a independently of $N(t)$ and $N^c(t) = N(\infty) - N(t)$, we have

$$P(N^c(t) = x : a, b, N(t) = y) = \frac{a^{x+y}}{(x+y)!} e^{-a} \tag{20}$$

and

$$E(N^c(t) : a, b, N(t) = y) = a - y. \tag{21}$$

We also have the conditional reliability at $S_k = s_k$ as

$$F^c(x : s_k, a, b) = \exp(-a(e^{-bs_k} - e^{-b(s_k+x)})), \tag{22}$$

and the conditional failure rate at $S_k = s_k$ as

$$\lambda(x : s_k, a, b) = \frac{f(x : s_k, a, b)}{F^c(x : s_k, a, b)} = abe^{-b(s_k+x)}. \tag{23}$$

We now develop a stopping rule based only on operational reliability. To do this, we need to provide two criteria apriori: the guaranteed period and the desired reliability. Since it is possible to set up the mission period during which the current version of software is to perform its duty until more advanced one replaces its role, the guaranteed period seems to be more meaningful.

Let the guaranteed period be T and assume that P(No failure during T) $\geq R$ is required, i.e. the desired reliability = R. If the kth failure time is given, we get the reliability function at S_k as (22). Now, if no failure is observed during the testing period t, we have

$$P(T_{k+1} > T : T_{k+1} > t, S_k = s, a, b) = \frac{P(T_{k+1} > T : S_k = s, a, b)}{P(T_{k+1} > t : S_k = s, a, b)}$$

$$= \frac{F^c(T : s, a, b)}{F^c(t : s, a, b)} \tag{24}$$

and we want this value is greater than or equal to R. That is,

$$\frac{\exp(-a(e^{-bs_k} - e^{-b(s_k+T)}))}{\exp(-a(e^{-bs_k} - e^{-b(s_k+t)}))} \geq R,$$

from which we can derive

$$t \geq t^* \equiv \frac{\log(e^{-bT} - (\log R / ae^{-bs}))}{-b}. \tag{25}$$

Using (25), we can develop the following stopping rule. Note that (24) increases at each failure debugging and in the elapsed time between two successive failures.

[Stopping rule]

Step 1. Continue test.
If time to next failure $\geq t^*$, then stop with the guaranteed reliability R. Else go to step 2.

Step 2. At the next, say k^{th}, failure, update $\hat{a}$(and $\hat{b}$, if desirable) using
$\hat{a} = \dfrac{a+k}{\beta+1-e^{-bs_k}}$ (and by the procedure proposed in 3.2 for $\hat{b}$)
If $F^c(T : s_k, \widehat{a}, \hat{b}) \geq R$, then stop with reliability $F^c(T : s_k, \widehat{a}, \hat{b})$. Else go to step 1.

If the guaranteed period is very long, it may take too much time until stopping the test. In this case we can revise the step 2 of the above stopping rule by adding the criterion $E(N^c(t) : a, b, N(t) = k) = a - k \leq 0$, which was devised to prevent the expected number of remaining bugs at the time of the k^{th} failure from being negative. The revised step will be the following.

Revised Step 2. At the next, say k^{th}, failure, update $\hat{a}$(and $\hat{b}$, if desirable) as in the original step 2. If $F^c(T : s_k, \widehat{a}, \hat{b}) \geq R$ or $\hat{a} - k \leq 0$, then stop with reliability $F^c(T : s_k, \widehat{a}, \hat{b})$. Else go to step 1.

5. A NUMERICAL EXAMPLE

To demonstrate how the stopping rule in section 4 works, the data given in [5] is taken. The data originally from the U.S. Navy Fleet Computer Programming center is the record of the errors occurred during the development of software for the real-time, multicomputer complex which forms the core of the Naval Tactical Data Systems. The data is shown in table 1.

The result of application of the stopping rule is summarized in table 2.

Table 1. NTDS DATA

Error No. k	Time Between Errors t_k, days	Cumulative Time $S_k = \sum_{i=1}^{k} t_i$, days
Production (checkout) phase		
1	9	9
2	12	21
3	11	32
4	4	36
5	7	43
6	2	45
7	5	50
8	8	58
9	5	63
10	7	70
11	1	71
12	6	77
13	1	78
14	9	87
15	4	91
16	1	92
17	3	95
18	3	98
19	6	104
20	1	105
21	11	116
22	33	149
23	7	156
24	91	247
25	2	249
26	1	250
Test Phase		
27	87	337
28	47	384
29	12	396
30	9	405
31	135	540
User Phase		
32	258	798
Test Phase		
33	16	814
34	35	849

Table 2. Resulted Stopping Times

Guarantee Period(T)	Rel.(R)	t^*	k	Stopping Time & Step	
100	.95	96.1	31	501.1	Step 1
	.90	92.2	31	497.2	Step 1
200	.95	187.5	32	727.5	Step 1
	.90	175.1	32	715.1	Step 1
300	.95	230.5	*need more test		
	.90	176.5			
500	.95	348.7			
	.90	260.0			

** Initial prior of a is $X^2(30)$. i.e. $\widehat{a}_0 = 30$.
b is set to 0.0005.

From table 2, we can see our stopping rule gives us reasonable stopping times. For example, based on table 2, if we stop at the 31st failure, we cannot guarantee 300 or more hours of failure-free operation. This is consistent with the given data as we can see at the 32nd failure.

6. CONCLUDING REMARKS

We can develop various stopping rules based on reasonable software reliability measures, like reliability, failure rate and expected number of remaining errors in the software. The stopping rule in section 4, is only one example. Though this stopping rule can be applied to any model introduced in section 2, the application was made only to G-O model in this study. It will be interesting to compare the stopping times under different models with the same data.

If we consider life time costs including debugging cost, execution cost during testing and failure cost after release, we may get more practical stopping rule. But in this case because of the difficulty in the estimation it seems to be very hard to get a simple stopping rule.

REFERENCES

[1] Achcar, J.A., D.K. Dey and M. Niverthi (1998). A Bayesian approach using nonhomogeneous Poisson processes for software reliability models, in *Frontiers in Reliability* ed. A.P. Basu et al., World Scientific.

[2] Barlow, R.E. and F. Proshan (1981). *Statistical Theory of Reliability and Life Testing*, To Begin With, Silversprings.

[3] Chatterjeea, S., R.B. Misrab, and S.S. Alama (1998). A generalised shock model for software reliability, *Computers & Electrical Engineering*, **24**(5), 363-368.

[4] Esaki, K. and M. Takahashi (1999). A model for program error prediction based on testing characteristics and its evaluation, *International Journal of Reliability, Quality and Safety Engineering*, **6**(1), 7-18.

[5] Forman, E.H. and N.D. Singpurwalla (1977). An empirical stopping rule for debugging and testing computer software, *J. Amer. Statis. Assoc.* **72**, 750-757.

[6] Goel, A.L. and K. Okumoto (1979). Time-dependent error-dependent rate model for software reliability and other performance measures, *IEEE Trans. Reliability*, R-28, 206-211.

[7] Finkelstein, M.S. (1999). A point-process stochastic model for software reliability, *Reliability Engineering & System Safety*, **63**(1), 67-71.

[8] Jelinski, Z and P. Moranda (1972). Software reliability research, in *Statistical Computer Performance Evaluation*, (W. Freaberger, ed.), 465-484, Academic Press, New York.

[9] Langberg, N. and N.D. Singpurwalla (1985). Unification of some software reliability models via the Bayesian approach, *SIAM Journal on Scientific and Statistical Computation*, **6**(3), 781-790.

[10] Littlewood, B. (1979). How to measure software reliability and how not to, *IEEE Trans. Reliability*, R-28, 103-110.

[11] Littlewood, B., P. Popov and L. Strigini (2001). Modelling software design diversity: a review, *ACM Computing Surveys* (to appear).

[12] Littlewood, B. and J.L. Verrall (1973). A Bayesian reliability growth model for computer software, *J. Roy. Stat. Soc.* **22**, 332-346.

[13] Littlewood, B. and D. Wright (1997). Some conservative stopping rules for the operational testing of safety-critical software, *IEEE Trans. Software Engineering*, **23**(11), 673-683.

[14] Lyu, M. (1996). *Handbook of Software Reliability Engineering*, McGraw-Hill, NY.

[15] Musa, J.D., A. Iannino and K. Okumoto (1987). *Software Reliability*, McGraw-Hill, New York.

[16] Pham, H. and X. Zhang (1997). An NHPP software reliability model and its comparison, *International Journal of Reliability, Quality and Safety Engineering*, **4**(3), 269-282.