1. Introduction

As hardware is becoming more and more cheap and powerful, software is becoming more and more complex and important in computer-based information system. Software reliability is probably the most important of the characteristics inherent in the concept "software quality." What is software reliability? We concerns it with how well the software functions to meet the requirements of the customer. The IEEE defines it as : **software reliability is the probability of failure-free operations of a computer program for a specified time in a specified environment** [1] There are several relative basic concepts:

Failure: It is the departure of the external results of program operation from requirements.

"Failure" is something dynamic. The program has to be executing for a failure to occur

Fault: It is the defect in the program that, when executed under particular conditions, causes a failure. The cause of the failure or the internal error is said to be a fault. It is also referred as a bug.

"Fault" is a property of the program rather than a property of its execution or behavior A fault is created when a programmer makes an error.

Time: Reliability quantities are defined with respect to time, although it would be possible to define them with respect to other variables. We are concerned with three kinds of time: the *execution time* for a software is the CPU time that is actually spent by the computer in executing the software; the *calendar time* is the time people normally experience in terms of years, months, weeks, days, etc.; and the *clock time* is the elapsed time from start to end of computer execution in running the software.

There are four general ways of characterizing failure occurrences in time:

1). time of failure,
2). time interval between failures,
3). cumulative failures experienced up to a given time,
4). failures experienced in a time interval.

Failure functions: When a time basis is determined, failures can be expressed in several ways: the *cumulative failure function*, the *failure intensity function*, the *failure rate function*. The *cumulative failure function* (also called the mean value function) denotes the average cumulative failures associated with each point of time. The *failure intensity function* represents the rate of change of the cumulative failure function. The *failure rate function* is defined as the probability that a failure per unit time occurs in the interval $[t, t+\Delta t]$, given that a failure has not occurred before t. In this paper we shall use failure rate functions to simulate the software execution and get the cumulative failures of it. Appendix B of [2] provides the mathematics of these functions in details.

Failure data collection: Two types of failure data, namely failure-count data and time-between-failures data, can be collected for the purpose of software reliability measurement. They are illustrated in Table 1.1 and 1.2.

TABLE 1.1 Failure-Count Data

Times (days)	Failures in the period	Cumulative failures
8	4	4
16	4	8
24	3	11
32	5	16
40	3	19
48	2	21
56	1	22
64	1	23
72	1	24

TABLE 1.2 Time-Between-Failures Data

Failure number	Failure interval(days)	Failure times(days)
1	0.5	0.5
2	1.2	1.7
3	2.8	4.5
4	2.7	7.2
5	2.8	10.0
6	3.0	13.0
7	1.8	14.8
8	0.9	15.7
9	1.4	17.1
10	3.5	20.6
11	3.4	24.0
12	1.2	25.2
13	0.9	26.1
14	1.7	27.8
15	1.4	29.2
16	2.7	31.9
17	3.2	35.1
18	2.5	37.6
19	2.0	39.6
20	4.5	44.1
21	3.5	47.6
22	5.2	52.8

In this paper we shall use the first kind of data.

2. Software reliability modeling

2.1. General Information

One particular aspect of SRE(Software Reliability Engineering) that has received the most attention is software reliability modeling. There are many models have been proposed by researchers since 1970s. In this section we give a general classification about software reliability model, and describe three models, they are: Goel-Okumoto model, S-shaped model and Jelinski-Moranda model. The former two belong NHPP (Nonhomogeneous Poisson Process) type model, the latter is a markov kind model. We will give the simulate results and comparison of a practical switching system software for the three models.

A software reliability model describes software failures as a random process, which is characterized in either times of failures or the number of failures at fixed times. We denote by T_i and T_i' the random variables representing times to the *i*th failures and between the (i-1)th and *i*th failures, respectively. The realizations of T_i and T_i' will be denoted by t_i and t_i', respectively. Time can be specified in either calendar time, the actual chronological period, or execution time, the processor (CPU) time accumulated. Sometimes clock time is used as an approximation to execution time. Let N(t) be a random process representing the number of failures experienced by time t. The realization of this random process will be denoted n(t). Then $\mu(t)$, the mean value function, is defined as $\mu(t)=E[N(t)]$, which represents the expected number of failures at time t. The function $\mu(t)$ will be nondecreasing and is assumed to be a continuous and differentiable function of time t. The failure intensity function of the M(t) process is the instantaneous rate of change of the expected number of failures with respect to time. It is defined by

$$\lambda(t) = \frac{d\mu(t)}{dt}$$

2.2. Nonhomogeneous Poisson Process (NHPP) Models

2.2-1. General assumptions

As a general class of well-developed stochastic process models in reliability engineering, NHPP models have been successfully used in studying hardware reliability problems. NHPP models are especially useful to describe failure processess which possess certain trends such as reliability growth or deterioration.

Application of NHPP models to software reliability analysis is then easily implemented. The cumulative number of software failures up to time t, N(t), can be described by a NHPP and many existing software reliability models also belong to this class.

For the counting process $\{N(t), t\geq 0\}$ modelled by NHPP, N(t) follows a Poisson distribution with parameter m(t), that is, the probability that N(t) is a given integer n is expressed by

$$P\{N(t) = n\} = \frac{[m(t)]^n}{n!} e^{-m(t)}, n = 0,1,2,...$$

In the above m(t) is called the mean value function. The function m(t) describes The expected cumulative number of failures in [0,t). Hence, m(t) is a very useful descriptive measure of the failure behaviour.

The underlying assumptions of the NHPP are:

(1) N(0)=0,
(2) {N(t),t≥0} has independent increments,
(3) P{[N(t+h)-N(t)]=1}=λ(t)+o(h),
(4) P{[N(t+h)-N(t)≥2}=o(h).

In the above o(h) denotes a quantity which tends to zero for small h. The function λ(t) which is called the instantaneous failure intensity is defined as:

$$\lambda(t) = \lim_{\Delta t \to 0_+} \frac{P\{N(t+\Delta t) - N(t)\}}{\Delta t}.$$

Given λ(t), the mean value function μ(t)=E[N(t)] satisfies

$$\mu(t) = \int_0^t \lambda(s) ds$$

Inversely, knowing μ(t) , the failure intensity at time t can be obtained as

$$\lambda(t) = \frac{d\mu(t)}{dt}$$

Generally, by using different nondecreasing functions μ(t), we get different NHPP models. In the most simple case for which λ(t) is constant, the NHPP become a homogeneous Poisson process which has a mean value function as μ(t)=αt.

Due to the great variability of the mean value functions, NHPP models have been widely studied in the literature. The most well-known NHPP model is the model studied in [3] which later on, has been further generalized and modified by many researchers in order to improve the goodness-of-fit to real software failure data.

2.2-2. The Goel-Okumoto (GO) model

The general assumptions of the GO-model are:
(1) The cumulative number of faults detected at time t follows a Poisson distribution.
(2) All faults are independent and have the same chance of being detected.
(3) All detected faults are removed immediately and no new faults are introduced.

Specially, the GO-model assumes that the failure process is modeled by an NHPP model with mean value function μ(t) given by

$$\mu(t) = a(1 - e^{-bt}), a > 0, b > 0;$$

Where a and b are parameters to be determined by using collected failure data.
Note that for this model we have

μ(∞)=a and μ(0)=0.

Since μ(∞) is the expected number of faults which will eventually be detected, the parameter a is then the final number of faults that can be detected by the testing

process. The quantity b which is a constant of proportionality, can be interpreted as the failure occurrence rate per fault.

The intensity function λ(t) defined as the derivative of μ(t) is then

$$\lambda(t) = \frac{d\mu(t)}{dt} = abe^{-bt}$$

$$E\{\bar{N}(t)\} = E\{N(\infty) - N(t)\},$$

The expected number of remaining faults at time t, may then be calculated as follows

$$E\{\bar{N}(t) = \mu(\infty) - \mu(t) = a - a(1 - e^{-bt}) = ae^{-bt}.$$

Hence, the expected number of remaining faults E[N(t)] is an exponentially Decreasing function of t, as Figure 2.1 shown.

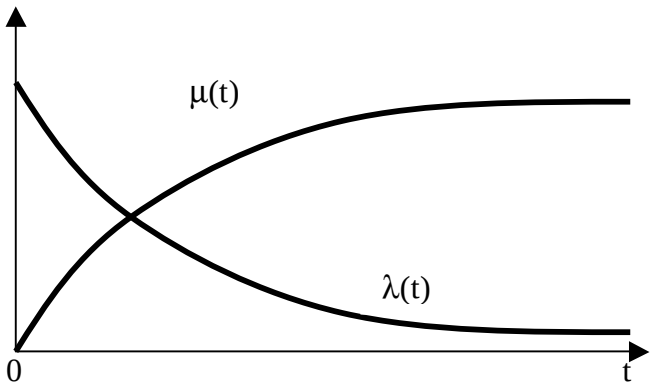


Figure 2.1. The shape of the intensity function and the mean value function Of the GO-model.

2.2-3. S-shaped NHPP model

The mean value function of the GO-model is exponential-shaped. Based on the experience, it is observed that the curve of the cumulative number of faults is often S-shaped relatively to the exponential-shaped mean value function which means that the curve crosses the exponential curve from below and the crossing occurs once and only once. A number of generalizations or modifications are thus proposed by Japanese researchers, see [4] and [5] where some heuristic reasons of the occurrence of S-shapedness are also provided.

Generally, the S-shapedness can be explained by the fact that faults are neither independent nor of the same size. At the beginning of the testing, some faults are by other faults "covered" and before these faults are detected and removed, the covered faults can not be detected. Hence, removing a detected fault at the beginning does not decrease the failure intensity very much since the same test data will still lead to a failure caused by other "covered" faults. In a later phase, large faults are already removed and the remaining faults have small size so that the fault detection

rate is of moderate size. Also because there are not many faults left in the software, the coverage has no significant effect at the end of testing phase. Another reason of the S-shaped behavior is that: The software reliability testing usually involves a learning process by which people become familiar with the software and the test tools. Their skills improve gradually and test effectiveness also increases. Hence, there is a tendency that the cumulative number of failures have a S-shaped form which implies that the increase of the reliability at the beginning is only of modest size due to the low test effectiveness, then the reliability increase quickly and later the improvement slows down again since most of the faults are already removed.
Figure 2.2 is the comparison of S-shaped model and GO-model

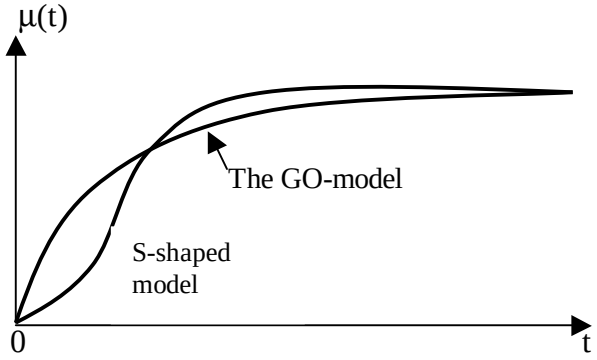


Figure 2.2. The S-shaped failure intensity function

Several different S-shaped NHPP models have been proposed in the existing literature, especially by Japanese researchers. The most well-known ones are the delayed S-shaped NHPP model and the inflected S-shaped NHPP model.

The mean value function of the delayed S-shaped NHPP model is

$$\mu(t)=a[1-(1+bt)e^{-bt}]; \quad b>0.$$

This is a two parameter S-shaped curve with parameter a denoting the number of Faults to be detected and b corresponding to a failure detection rate. The corresponding failure rate function of this delayed S-shaped NHPP model is

$$\lambda(t) = \frac{d\mu(t)}{dt} = ab(1+bt)e^{-bt} - abe^{-bt} = ab^2te^{-bt}.$$

The expected number of remaining faults E[N(t)] at time t is then

$$\mu(\infty)-\mu(t)=a-a[1-(1+bt)e^{-bt}]=a(1+bt)e^{-bt}.$$

Another general model of this kind is called inflected S-shaped model, it was proposed in [6] by Schagen in 1987. In this model the expected number of faults detected by time t is modeled the following mean value function

$$\mu(t) = \alpha[1 - e^{-\lambda_1 t} - \frac{\lambda_2(e^{-\lambda_1 t} - e^{\lambda_2 t})}{\lambda_2 - \lambda_1}].$$

In the above, α, λ_1 and λ_2 are the model parameters. It can be seen that if $\lambda_1=\lambda_2$, Then we have the limiting case

$$\mu(t)=\alpha[1-(1+\lambda_1 t)e^{-\lambda_1 t}],$$

which is the same as that for the delayed NHPP model with a=α and b=λ_1.

The mean value function of the inflected S-shaped NHPP model is

$$\mu(t) = \frac{a(1-e^{-bt})}{1+ce^{-bt}}; b>0, c>0$$

In the above a is again the total number of faults to be detected while b and c are called the failure detection rate and the inflection factor, respectively. The intensity function of this inflected S-shaped NHPP model can easily be derived as follows,

$$\lambda(t) = \frac{d\mu(t)}{dt} = \frac{abe^{-bt}(1+ce^{-bt}) + abce^{-bt}(1-e^{-bt})}{(1+ce^{-bt})^2}$$

$$= \frac{abe^{-bt} + abce^{-bt}}{(1+ce^{-bt})^2} = \frac{ab(1+c)e^{-bt}}{(1+ce^{-bt})^2}.$$

Similarly, as for the delayed S-shaped NHPP model, we may obtain the expected number of remaining faults E[N(t)] at time t as follows

$$\mu(\infty) - \mu(t) = a - \frac{a(1-e^{-bt})}{1+ce^{-bt}} = \frac{a(1+c)e^{-bt}}{1+ce^{-bt}}.$$

Some heuristic arguments for this mean value function are presented in [4]. Using this mean value function, the parameters can be estimated from maximum likely method.

2.3 Markov Models

2.3-1 General assumptions

Markov processes which are a general class of stochastic processes have been widely used and studied in reliability analyses. Many software reliability models also belong to this category. A Markov process is characterized by its state space together with the transition probabilities between these states.

A stochastic process {X(t),t≥0} is said to be a Markov process if its future Development depends only on the present state of the process, that is

$P[X(t) \geq x(t)|X(t_1) \geq x_1,\ldots,X(t_n) \geq x_n] = P[X(t) \geq x(t)|X(t_n) \geq x_n]$, for all $t_1 < t_2 \ldots < t$.

The above property is generally called the Markov property which has the following simple explanation. Given the present state of the process, its future behavior is independent of the past history of the process. This is the most important feature of a Markov process and although this needs not always be the case, it is a realistic simplification in many practical situations.

If the state space is discrete, a Markov process is also called the Markov chain. Define p_{ij} is the transition probability of the process between state i and state j, that is $p_{ij}(t+s) = P[X(t+s)=j|X(s)=i]$, s,t>0.

In general p_{ij} may depend on t as well as on s. If all p_{ij} , i,j>0, are independent of t, the Markov chain is called time-homogeneous.

The most famous result of a homogeneous continuous-time Markov chain is that it satisfies the so-called Kolmogorov equations, that is

$$p_{ij}(t+s) = \sum_{k}' p_{jk}(s)\, p_{kj}(t). \qquad s,t > 0$$

The theory of Markov processes is well developed. The initial condition of the process together with the transition probabilities completely determines the stochastic behavior of the Markov process. Knowing the transition probabilities, the probability that the process is in a certain state can be obtained by solving the Kolmogorov equations, and other reliability measures can also be calculated. However, in order to get a mathematically tractable software reliability model, some further assumptions usually have to be made.

Generally, each sojourn time interval for a Markov process, i.e. the time between two events, has an exponential distribution with the parameter dependent on the state being visited. Another property is that times between transitions are conditionally independent of each other given the successive states being visited. These properties together with the fact that the successive states visited form a Markov chain clarify the structure of a Markov process. Other standard results can be found in many elementary texts on stochastic processes. Also in many books on reliability Markov process models are discussed.

The process {N(t), t≥0} where N(t) is the number of events in a Markov process, such as the number of detected faults in a software context, is called a Markov counting process is the birth-death process for which a so-called birth increases the size of the process by one and a death decreases the size by one.

Figure 1.3 illustrates a realization of a Markov counting process

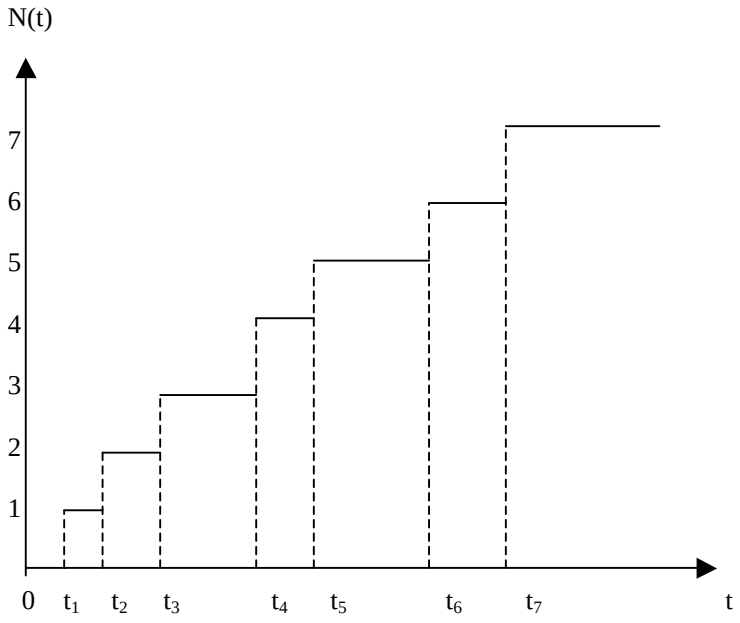


Figure 2.3. Arealization of a Markov counting process $\{N(t), t \geq 0\}$.

Markov models are very useful in studying software fault-removal processes, especially, they are useful during the software testing phase which is the most important one of software development. It is in this phase that software faults are detected and removed. The state of the process at time t is here the number of remaining faults at that time. The fault-removal process can usually be described by a so-called pure death process since the number of remaining faults is a decreasing function of time provided that no new faults due to incorrect debugging, the so-called birth-death processes can then be used in studying software reliability during the testing phase.

2.3-2. The Jelinski-Moranda (JM) model

The best-known software reliabilty model originally developed by Jelinski and Moranda is also a Markov process model. It is one of the earliest models and has strongly influenced many later models which are in fact modifications of this simple model.

Model assumptions and some properties, the underlying assumptions of the JM-model are:

(1) the number of initial software faults is an unknown but fixed constant;
(2) a detected fault is removed immediately and no new fault is introduced;
(3) times between failures are independent, exponentially distributed random quantities;
(4) all remaining software faults contribute the same amount to the software failure intensity.

Denote by N_0 the number of software faults in the software before the testing starts. By the assumption (3) and (4), the initial failure intensity is then equal to $N_0\phi$, where ϕ is a constant of proportionality denoting the failure intensity contributed by each fault. It follows from assumption (2) that, after a new fault is detected and removed, the number of remaining faults is decreased by one. Hence after kth failure, there are (N_0-k) faults left, and the failure intensity decreases to $\phi(N_0-k)$.

Denote by T_I, $i=1,2,\ldots, N_0$, the time between the (i-1):th and the i:th failures, T_i is thus the i:th failure-free time interval. By the assumptions, T_i's are then exponentially distributed random variables with parameter

$$\lambda(i)=\phi[N_0-(i-1)],\ i=1,2,\ldots,N_0.$$

The distribution of T_i is given by

$$P(T_i<t_i)=\phi(N_0-i+1)\exp\{-\phi(N_0-i+1)t_i\},\ i=1,2,\ldots,N_0.$$

The main property of the JM-model is that the failure intensity is constant between the detection of two consecutive failures. This is quite reasonable if the software is unchanged and the testing is random and homogeneous. A plot of the failure intensity function versus the cumulative time is displayed in Figure 2.4. In Figure 2.5 a plot of $\lambda(i)$ versus the number of detected faults i can be also found.

It should be pointed out here that this simple model has an order statistic explanation. Successive failure times constitute order statistics of N_0 independent random variables From an exponential distribution with parameter ϕ. General order statistic models have been studied by many researchers, see e.g. [7],[8],[9]

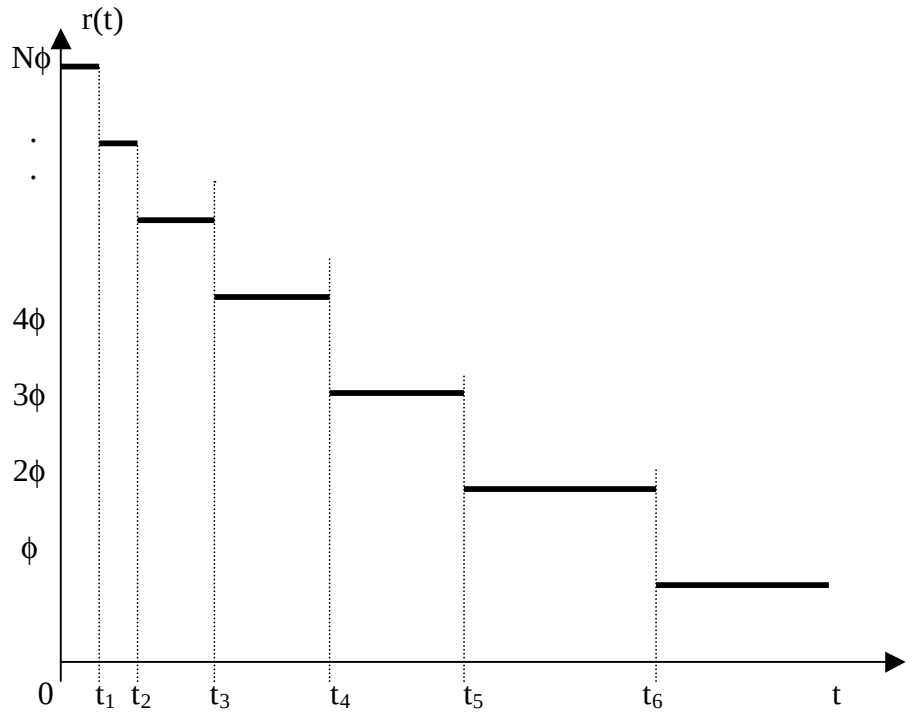


Figure 2.4. A realization of failure intensity as a function of time.

For each removal of a fault, the failure intensity decreases by ϕ.

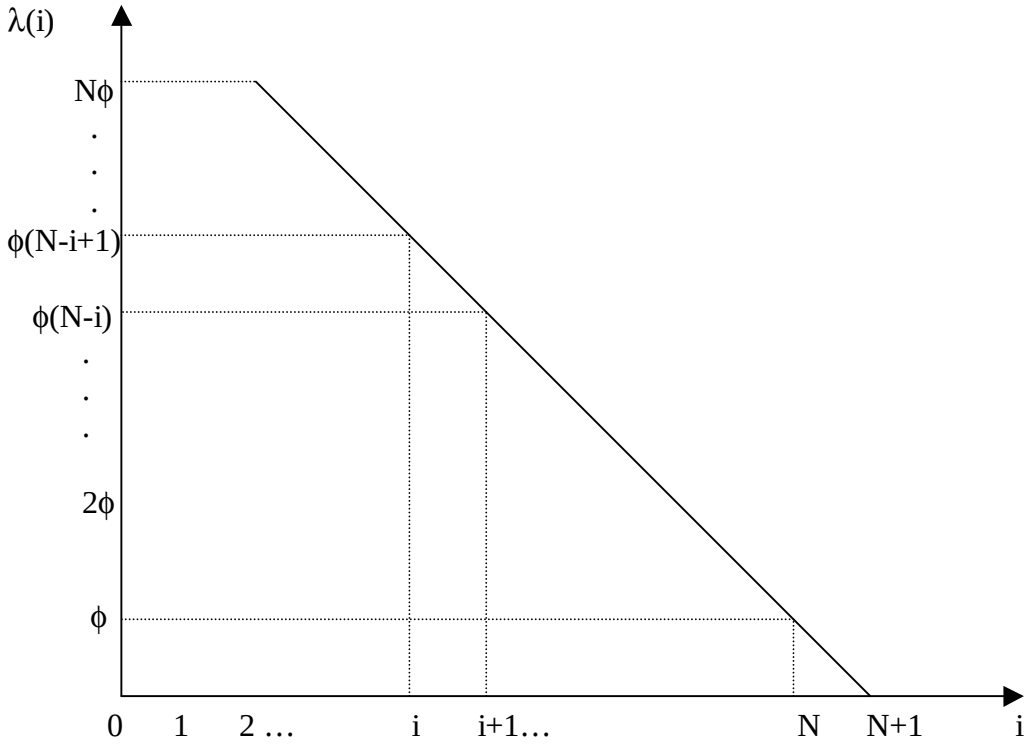


Figure 2.5. The failure intensity versus the number of removed faults

2.4 Estimation of model parameters

There are two practical methods of parameters estimation for software reliability models. They are maximum-likelihood (ML) and least-squares (LS). The most important and widely used formal estimation technique is ML, therefor, we give its details and an example here. About the least-squares approach, there are detailed description in [10] Page 351-364.

The foundation of the maximum-likelihood method is the *likelihood function.* The function is defined as the joint density of the observed data, $L(\beta; Y_D)$. This, In turn, is considered to be a function of the unknown set of w+1 parameters, β. Here Y_D represents a set of observations.

Figure 2.5 illustrates a typical likelihood function when there is only one unknown Parameter (denoted β_k). In this example a small value of $L(\beta_k; Y_D)$ for a particular β_k could be interpreted to mean that observing Y_D is rare event. It is reasonable to prefer the value of β_k which makes $L(\beta_k; Y_D)$ a maximum. That is, we would choose β_k so that the observed data are more probable than for any other choice. Formalizing the notion, we have the definition of maximum likelihood estimators. For each data set, let $\boldsymbol{\beta}$ be the values of the parameters that make $L(\beta_k; Y_D)$ as large as possible. These maximizing values will, of course, be functions of the data. The fuctions take on are known as the maximum likelihood estimates. Figure 2.5 shows the maximum likelihood estimate of β_k as $\boldsymbol{\beta}_k$.

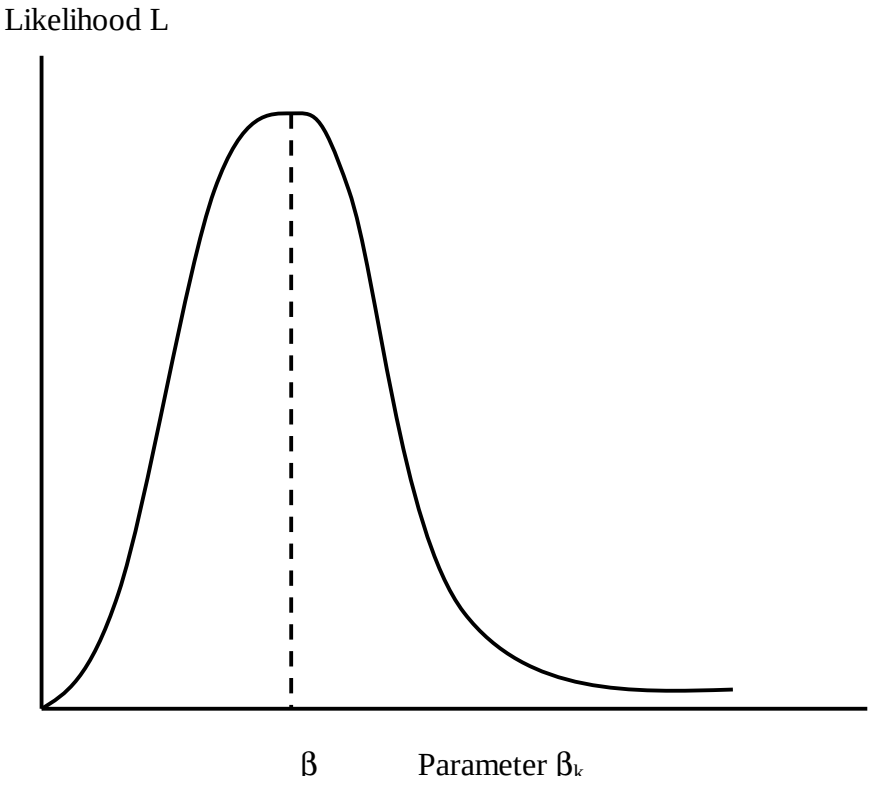


Figure 2.5 Typical likelihood function with only one unknown parameter.

Maximum likelihood estimates can be obtained by solving the simultaneous equations (one for each β_k)

$$\frac{\partial L(\beta; Y_D)}{\partial \beta_k} = 0, k = 0,..., w.$$

In practice it is customary and often more convenient to work with ln $L(\beta_k; Y_D)$ instead of $L(\beta_k; Y_D)$, the derivatives of both vanishing together. Thus, we will find our estimates by solving the simultaneous equations (which are called the maximum likelihood equations)

$$\frac{\partial \ln L(\beta; Y_D)}{\partial \beta_k} = 0, k = 0,..., w.$$

Generally the maximum likelihood equations are highly complicated and a numerical solution will be possible only with a computer. An excellent iterative procedure for carrying out the solution is given in Appendix D of [10].

Now we give the JM-model parameter estimation as an example

The parameters of the JM-model may easily be estimated by using the method of maximum likelihood. Let t_i denote the observed i:th failure-free time interval during the testing phase, that is t_i is the observed time between the (i-1):th and the i:th failure The number of faults detected is denoted here by n which will be called the sample size. Suppose that the failure data set $t=\{t_1, t_2, ..., t_n; n>0\}'$ is given, the parameters ϕ and N_0 in the JM-model can easily be estimated by maximizing the likelihood function. The likelihood function of the parameters N_0 and ϕ is given by

$$L(t_1, t_2, ..., t_n; N_0, \phi) = \prod_{i=1}^{n} \phi(N_0 - i + 1) \exp\{-\phi(N_0 - i + 1)t_i\}$$

$$= \phi^n \{\prod_{i=1}^{n} (N_0 - i + 1)\} \exp\{-\phi \sum_{i=1}^{n} (N_0 - i + 1)t_i\}.$$

The natural logarithm of the above likelihood function is

$$\ln L = \ln\left[\phi^n \left\{\prod_{i=1}^{n} (N_0 - i + 1)\right\} \exp\left\{-\phi \sum_{i=1}^{n} (N_0 - i + 1)t_i\right\}\right]$$

$$= \ln \phi^n + \ln\left\{\prod_{n=1}^{n} (N_0 - i + 1)\right\} - \phi \sum_{i=1}^{n} (N_0 - i + 1)t_i$$

$$= n \ln \phi + \sum_{i=1}^{n} \ln(N_0 - i + 1) - \phi \sum_{i=1}^{n} (N_0 - i + 1)t_i.$$

By taking the partial derivatives of this log-likelihood function with respect to N_0 and ϕ, respectively, and equating them to zero, we get the following likelihood

equations

$$\frac{\partial \ln L}{\partial N_0} = \sum_{i=1}^{n} \frac{1}{N_0 - i + 1} - \sum_{i=1}^{n} \phi t_i = 0$$

$$\frac{\partial \ln L}{\partial \phi} = \frac{n}{\phi} - \sum_{i=1}^{n} (N_0 - i + 1) t_i = 0.$$

Usually numerical procedures have to be used solve these two equations. However, the equation system can be simplified as follows. By solving ϕ from the second equation above we get

$$\phi = n \left\{ \sum_{i=1}^{n} (N_0 - i + 1) t_i \right\}^{-1},$$

and by inserting this into the first equation, we obtain an equation independent of ϕ,

$$\frac{1}{N_0} + \frac{1}{N_0 - 1} + \ldots + \frac{1}{N_0 - n + 1} = \frac{n \sum_{i=1}^{n} t_i}{\sum_{i=1}^{n} (N_0 - i + 1) t_i}.$$

The estimate of N_0 can then be obtained by solving this equation. Inserting the estimated N_0 into the expression of ϕ, we may then get the maximum likelihood estimate of ϕ.

2.5. General model characteristics and limitations

Random Process

A software reliability model, as previously noted, usually has the form of a random process that describes the behavior of failures with time. This is because both the human error process that introduces defects into code and run selection process that determines which code is being executed at any time are dependent on an enormous number of time-varying variables. The use of a random process model is appropriate for such a situation. Specification of the model generally includes specification of a function of time such as the mean value function (expected number of failures) or failure intensity. The parameters of the function are dependent on repair activity and program change and properties of the software product and the development process. Properties of the product include size, complexity, and structure. Properties of the development process include, among others, software engineering technologies and tools used and level of experience of personnel. The "time" involved in the characterization of the models is a cumulative time. The origin may be arbitrarily set. It is frequently the start of system test.

Software reliability models almost always assume that failures are independent of each other. They may do this through assuming that failure times are independent of each other or by making the Poisson process assumption of independent increments.

This condition would appear to be met for most situations. Failures are the result of two processes: the introduction of faults and their activation through selection of the input states. Since both of these processes are random, the chance of influence on one failure by another is small. Influence would require two conditions. One fault would have to affect the introduction of another during development. Further, an input state that results in failure for the first fault would have to cause the selection of an input state that results in failure for the second fault. The independence conclusion is supported by a study of correlograms of failure data from 15 projects [11]. No significant correlation was found.

It is possible that the selection of runs could be planned or manipulated during test. The tester has at least partial control of the environment, and this could make random selection of input states a poor model. However, a random process is still a reasonable model of failure behavior. The introduction of faults into code and the relationship between input state and code executed are usually both sufficiently complex process to make deterministic prediction of failure impractical. In other words, we can not predict which input states are more likely to yield failures. Consequently, a deterministic selection of input states will not have a deterministic effect on reliability. Of course, if the relative frequencies of selection have changed, then the operational profile has changed and that will affect the reliability.

There is one case in which manipulation of the characteristics of the random process can occur. It requires that:

(1) The relationship of program segments executed with respect to input state is Disjoint.
(2) There are clear differences in fault density between different program segments.

"Disjoint" means that each input state maps to a different set of program segments executed and there are no program segments executed in common by different input states. The clear differences may occur when some segments may be tested code from previous programs and some may be newly written. In this situation, it is possible to manipulate reliability figures to somewhat higher or lower figures by biasing the selection of input states. We select input states that exercise code that has either high or low fault density. Note that the essential character of failures as a random process is unchanged. We can not predict when the next failure will occur, even if we can manipulate average behavior. One example of manipulation would be to select input states deterministically in such a way that no code segments are reexecuted. If fault density is the same for all segments, the observed failure intensity would tend to be constant. Fault repair would show no effect on failure intensity. In reality, failure intensity based on random selection of input states would be decreasing.

With and without repair

Software reliability models must cover two situations, the situation of programs that are being repaired when failures occur and the situation of programs that are not. These situations can occur in either the test or the operational phase, but "no repair" is usually associated with the latter. Thus, as far as the program is concerned, the failure intensity is constant for the duration of the release. Hence, the failure process is conveniently modeled by a homogeneous Poisson process. This implies that the failure intervals are exponentially distributed and that the number of failures in a given time period follows a Poisson distribution. If the failure intensity is λ and the period of execution of the program is τ, then the number of failures during this period is

distributed Poisson with parameter $\lambda\tau$.
A principal factor that causes reliability to vary with time is the correction of faults that have caused failures. In general, the time of correction does not coincide with the time of original failure. This could lead to substantial complication in characterizing the failure process. However, it can be handled by assuming instantaneous repair and not counting the reoccurrence of the same failure. It would be recounted, however, if the recurrence were due to inability to locate and repair the fault. Although the result is not precisely equivalent, it is a very good approximation. All the leading models take this approach.

Particularization

The model specifies the general form of the dependence of the failure process on the variables mentioned. The specific form can be determined from the general form, at least in theory, through determination of parameters. For the execution time component, this can occur in one of two ways [12]:

(1) Prediction--properties of the software product and the development process are used to particularize the model by determination of its parameters (this can be done prior to any execution of the program).
(2) Estimation--inference (for example, parameter estimation) procedures are applied to failure data.

A model and an inference procedure are commonly associated. Together, they provide projection through time. Without a model, we could not make inferences about reliability outside the time period for which failure data have been taken. In fact, we could not make inferences at all, because the size of the failure sample would be one. The model provides the structure that relates behavior at different points in time. It Thus, in effect, provides for a sample of reasonable size to be taken.

Note that the inference procedure that has historically been associated with a model is not necessarily the "best". We may wish to consider alternatives. Inference also generally includes the determination of the range of uncertainty. We either establish confidence intervals for the parameters or determine posterior probability distributions for significant quantities in the case of Bayesian inference. The determination of ranges of uncertainty is generally extended to quantities that are derived from the models as well.

Failure data are most commonly available in the form of times of failures or number of failures in a given time interval. Time can be specified in either calendar time, the actual chronological period that has passed, or execution time, the processor (CPU) time accumulated.

Model limitations

In fitting any model to a given data set, we are cautioned about some limitations for this type of analysis. First, we must be aware of a given model's assumptions. For example, if a selected model makes the assumption that the time intervals over which the software is observed or tested are all of the same magnitude, don't attempt to use this model if this is not the case for the data. There are other assumptions that may not hold, but the model may be fairly robust with respect to violations. One such assump-

tion is the distributional one about the number of failures per unit time or the time between failures. One can still do a credible job in fitting the data for a selected model even if its distributional assumption is violated. The only way to tell is to ask just how well the model is doing in tracking and predicting the data.

A second model limitation concerns future predictions. If the environment in which the software is being tested or observed changes considerably from the one in which the data have been collected, we can't expect to do well in predicting future behavior. If the software is being operated in a different manner (i.e., new capabilities are being Exercised that were not used before, or a different testing methodology is used), the failure history of the past will not reflect these changes, and poor predictions may be result. Too many times model users tend to extrapolate either too far into the future or make reliability predictions for an environment in which little if any data have been gathered. For both violations of assumptions and considerations for predictions, one option that may be available to the practitioner is to use the most recent data if sufficient current data are available. Recent data may be more representative of the environment in which the software is employed than data collected in the distant past. This same reasoning applies to violations of assumptions. Current data may be more stable and reflective of the assumptions than past data.

3. Software reliability simulation techniques

3.1. Introduction

The software reliability models attempt to assess expected reliability or future operability by using observed failure data and statistical inference techniques. Most of these treat only the exposure and handling of failures during testing or operations. They are restricted in their life-cycle scope and adaptability to general use for a number of reasons, including their foundation on oversimplified assumptions and their main focus on testing and operations phases. Some modelers may have relaxed an assumption here or there in attempts to provide more generality, but as models become more and more realistic, the likelihood of obtaining simple analytic solutions plunges to impossibility. And reliability modeling ultimately requires good data. But software projects do not always collect data sets that are comprehensive, complete, or consistent enough for effective model application or research. Additionally, industrial organizations are reluctant to release their reliability data for use by outside parties.

Simulation presents a particularly attractive computational alternative for investigating software reliability because it averts the need for overly restrictive assumptions and because it can model a wider range of reliability phenomena than mathematical analyses can cope with. Also, it can provide an "virtual" environment to predict or study software reliability for some software projects. Here, simulation is refers to the technique of imitating the character of an object or process in a way that permit us to make quantified inferences about the real object or process. In the area of software reliability, simulation can mimic key characteristics of the processes that create, validate, and revise documents and code. It can mimic faulty observation of a failure when one has, in fact, occurred, and additionally, can mimic system outages due to failures. Furthermore, simulation can distinguish faults that have been removed from those that have not, and thus can readily reproduce multiple failures due to the same asyet unrepaired fault. Some reliability subprocesses may be sensitive to the passage of execute

time (e.g., operational failures), while others may depend on wall-clock, or calendar, time (e.g., project phases); still others may depend on the amount of human effort expended (e.g., fault repair) or on number of test cases applied. A simulator can relate model-pertinent resource dependencies to a common base via resource schedules, such as workforce loading and computer utilization profiles.

There are two main types of software reliability simulation ways, one is rate-based simulation, the other is artifact-based simulation. For the artifact-based simulation: we consider many aspects of program construction and testing to investigate the effect of static features on dynamic behavior, the inputs may include those which characterize code structure, coding errors, test input data, test conduct, failure characteristics, debugging effectiveness, and computing environment. In this paper, we used rate-based simulation way to get some results for a switching system software.

3.2. Rate-based simulation

It is a rate-controlled event process simulation way, the fundamental basis of this simulation method is the representation of a stochastic phenomenon of interest by a time series x(t) whose behavior depends only on a rate function, call it $\beta(t)$, where $\beta(t)dt$ acts as the conditional probability that a specified event occurs in the infinitesimal interval (t,t+dt). We can treat the event as a fault/failure in a software.

3.2-1. Event process statistics

If S_0 and S_1 denote the states of an event ε, S_0 in effect before the event and S_1 after its Occurrence, then a particular member of the stochastic time series defined by $\{\beta_0(t), S_0, S_1\}$beginning at time t=0 is a sample function, or realization, of the general rate-based discrete-event stochastic process. The zero subscript on $\beta_0(t)$ signifies the S_0, or zero occurrences, starting state.

The statistical behavior of this process is well known: the probability that event ε will not have occurred prior to a given time t is given by the expression

$$P_0(t) = e^{-\lambda_0(t,0)}$$

where

$$\lambda_0(t,t_0) = \int_{t_0}^{t} \beta_0(\tau) d\tau$$

The form of $\beta_0(t)$ is unrestricted, but generally must satisfy

$\beta_0(t) \geq 0$ and $\lambda_0(\infty,0)=\infty$ Eq. (3.3)

The first of these prevents the event from occurring at a negative rate, and the second stipulates that the event must eventually occur. If the second condition is violated, there will be a finite probability that the event will never occur.

Where the events of interest are failures, $\beta_0(t)$ is often referred to as the process hazard function and $\lambda_0(t,0)$ is the total hazard. The cumulative distribution function and probability density function for the time of an occurrence are then

$$F_1(t)=1-P_0(t)$$

$$f_1(t) = \beta_0(t) e^{-\lambda_0(t,0)}$$

The mean time of occurrence is

$$E(t) = \int_0^{\infty} t\beta_0(t) e^{-\lambda_0(t,0)} dt$$

If $\lambda_0(t,0)$ is known in closed form, we may sometimes be able to write down and analyze the event probability and mean time of occurrence functions directly. In all but the simplest cases, however, we will require the assistance of a computer. When we can not express the integrals in closed form, we can still evaluate them using str-aightforward numerical analysis.

3.2-2. Rate function of simulation models

We use the following rate functions in our implemented simulator. Except these may differ significantly in their assumptions about underlying failure mechanism, they differ mathematically only in the forms of their rate functions.

(1) The Goel-Okumoto (GO) model treats an overall reliability growth process with $\beta(t)=n_0\phi e^{-\phi t}$, where n_0 and ϕ are input parameters, $n_0\phi$ is the initial failure rate, and ϕ is the failure rate decay factor. Strictly speaking, this rate function violates the conditions on $\lambda(t,0)$ imposed in Eq.(3.3), because $\lambda_0(\infty,0)=n_0$ and $P_0=e^{-n_0}$. In practicality, n_0 is usually fairly large, so the consequences may be negligible.
(2) The Jelinski-Moranda (JM) model describes statistics of failure time intervals under the presumption that $\beta_n(t)=\beta_0(1-n/n_0)$, where n_0 is the estimated (unknown) number of initial software faults and β_0 is initial failure rate.
(3) The Duane model deals with another overall reliability growth model with failure rate function as, $\beta(t)=kbt^{b-1}$, where k and b are input parameters. Eq.(3.3) requires that $0<\beta<1$.
(4) The Littlewood-Verrall inverse linear model is an overall reliability growth model with $\beta(t)=\beta_0/(1+\theta t)^{1/2}$ where β_0 is the initial failure rate and θ is a rate decay factor.
(5) The Musa-Okumoto model [13], in which $\beta(t)= \beta_0/(1+\theta t)$, where β_0 is the initial failure rate and θ is a rate decay factor. Both β_0 and θ are input parameters.
(6) The Yamada S-shaped model, its failure rate function is $\beta(t)=ab^2te^{-bt}$, where a is the number of failures to be expect occur and b corresponding to a failure detect rate.
(7) The Musa's basic execution time model, its failure rate function is $\beta(t)= \beta_0 \beta_1 e^{-\beta_1 t}$. Where β_0 is the total number of faults that would be detected, β_1 is the factor of fault reduction.

In [14] and Chapter 3 of [2], there are detailed description about software reliability models.

3.3 Simulator implementation

We have implemented a simulator which has black-box and white-box simulation functions for software reliability. It is a failure rate-based simulator, the above seven failure rate functions are used as simulation model respectively. The simulator can be used for data fitting, model validation and failure evaluating for software reliability engineering.

3.3-1 General simulation assumptions

For the simulator we have the following assumptions, they can be seen as the most common assumptions for software reliability models.

(1) The faults of program are introduced randomly, the occurrence of failure is a random process.
(2) The software is tested remains essentially unchanged throughout testing, except for the removal of faults as they are found.
(3) Removing a fault does not affect the chance that a different fault will be found.
(4) "Time" is measured in such a way that testing effort is constant.
(5) At any time the future evolution of the testing process depends only on the present state (the current time, the number of faults found and remaining, and the overall parameters of the model), and not on details of the past history of the testing process.
(6) All faults are of equal importance (contribute equally to the failure rate).
(7) At the start of testing, there is some finite total number of faults, which may be fixed (known or unknown) or random; if random, their distribution may be known or of known form with unknown parameters. Alternatively, the "number of faults" is not assumed finite, so that if testing continues indefinitely, an ever-increasing number of faults will be found.
(8) Between failures, the failure rate follows a known functional form.

3.3-2 Simulation approaches

1). Black-box simulation

We have two methods for implementing the simulator, one is black-box simulation, another is white-box simulation. For the black-box simulation, we treat software as a whole only its interactions with the outside world are modeled, the internal structure and component combinations are not concerned. It is relatively a simple simulation approach, the basic algorithm for black-box simulation is as follows

Initialization(e.g. set maxtime_step, dt, simulation run times…)

⇩

While(maxtime_step>1){

⇩

Produce a random number: 0<occurs<1

⇩

If (vr*dt<occurs)

⇩

Failure_num=Failure_number+1 and t=t+1 }

Where, vr is the value of the failure rate function at that time, Failure_number is the number of cumulative failures at that time.

The input of black-box simulation is a failure behavior file, this file includes the parameters of failure rate functions. The parameters can be obtained by using CASRE

(Computer Aided Software Reliability Estimation) which is a tool for software reliability measurement. There are detailed description about CASRE in [2] Appendix A. The output or results of black-box simulation are the number of cumulative failures and the failure intensity of the software. In the later of this paper we give some black-box simulation results discussion.

2). White-box simulation

In the black-based simulation approach for software reliability we treat the software system as a whole, without looking into its internal structure, and using one model to simulate the whole software. However, in practical, a lot of software packages are consist of components (component-based). With the advancement and widespread use of object oriented systems design and internet-based development, the use of component-based development is on the rise. The software components can be commercially available off the shelf (COTS), developed in house, or developed contractually. Thus, the whole software is developed in a heterogeneous (multiple teams in different environments) fashion, and hence it may be inappropriate to simulate the overall failure process of such a software using the black-box approach. Thus, taking into account some information about the internal structure of the software for analyzing its reliability is absolutely essential [15].

In the white-box simulation approach we assume that the software comprising of m components begins execution with component 1, and terminates upon the execution of component m. The architecture of the software is specified by the intercomponent transition probabilities, denoted by w_{ij}. w_{ij} represents the probability that component j is executed upon the completion of component i. We can apply different software reliability model for different component, of course we also can use the identical model for the whole software. The basic algorithm of white-box simulation as follows

```
Initialization(set max-time, dt; curr_comp=1…)
        ⇩
While (time<max-time){
        ⇩
Check_point=0; produce a random number 0<a<1
        ⇩
if (vr*dt<a)
        ⇩
Failure_num_comp+1 and Failure_num_T+1; time+dt
        ⇩
for (i=1; i<=m;i++){
  Check_point=check_poin+transition_p[curr_comp][i]
  if (a<=check_point)
   break; }
 curr_comp=i; }
```

In the above, vr is equal to the value of failure rate function at that time; curr_comp represents the current executing component number; the Failure_num_comp is the cumulative failure number of current component, and the Failure_num_T is the cumulative failure number of the whole software. dt is time interval for simulation. The transition_p is an array which contains the transition probabilities between components of the software. The "for loop" is actually used for determining which component will execute at next time.

The input of white-box simulation are failure behavior of all components and transition probabilities file. A five-component software failure behavior file is as follows:

```
go 130.6 0.0048
go 108.7 0.0053
jm 63.78 0.3288
ys  88.5 0.00988
go 78.66 0.0056
```

In which, each row corresponds one component (first row is for component 1, second row is for component 2,etc.). First column indicate which model will be used for the component (e.g. "go" represent the Goul-Okumoto model, "ys" indicate the Yamada S-shaped model is being used for component 4). The other real numbers of each row are the parameters of the used software reliability model. These parameters can be estimated by using CASRE.

The following is a five-component software transition probability file

```
5
0.00 0.80 0.20 0.00 0.00
0.30 0.00 0.70 0.00 0.00
0.00 0.00 0.00 0.70 0.30
0.00 0.00 0.20 0.00 0.80
0.60 0.20 0.00 0.20 0.00
```

In which, the first row has an integer number, it indicates the number of components belong to the software. Each real number in other rows is in [0.00, 1.00], it represents the transition probability from component i to component j.

The output or results of white-box simulation are number of cumulative failures for each component and the whole software. We can also get the failure intensity of each component and the whole software. Figure 3.1 give the simulation results of a five-component software. The failure behavior and transition probability of this software are the same as above.

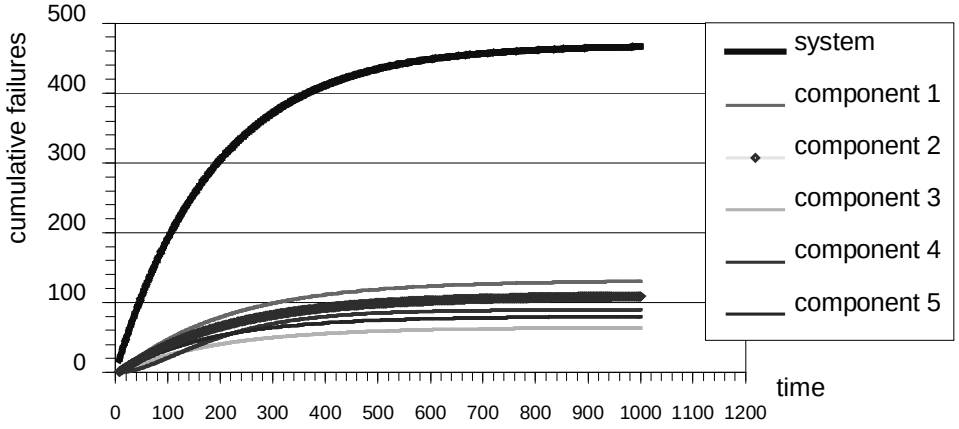


Figure 3.1 simulate results for a five-component software

From the Figure 3.1 we can know that the cumulative of whole software system has identical convergence trend with all components, and the total number of failures in the system is approximately equal to the sum of all components. In most situations, this is the case.

4. Project applications

We have applied the simulation approaches into a project software for analyzing its reliability features. This section introduce the application results and some comparisons.

4.1 General description of the software (About the project see [16])

This is the system software of three successive generations of the Brazilian switching system, TROPICO-R. It is developed jointly by the R&D center for Brazilian Telecommunications and some Brazilian manufacturers. To dates, three successive products have been developed, and referred to as PRA,PRB and PRC. The software can be decomposed into two main parts; the applicative software and the executive software. Two categories of components can be distinguished in the TROPICO-R software: i) Elementary Implementation Blocks (EIB), which fulfil elementary functions and ii) groups of elementary implementation blocks according to the main four functions of the system. These groups are:

- Telephony (TEL): local call processing, charge-metering, etc.
- Defense (DEF): on-line testing, traffic measurement, error detection, etc.
- Interface (INT): communication with local devices (memories, terminals),…
- Management (MAN): communication with external devices (trunk),…

The software were coded in Assemble language. In order to analyze the evolution of TROPICO-R software and to compare the successive products, we have defined the following types of EIBs:

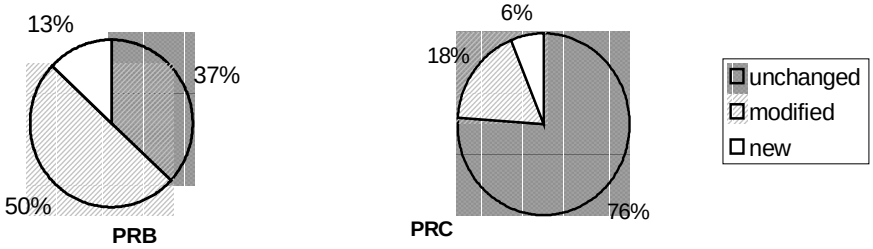
- new: developed specifically for a given product;
- modified: developed for a given product and then modified to meet the requirements of the new product;
- unchanged: EIBs of a previous product included in a new product without functional modification.

Table 4.1 lists the number of EIBs and the size of the software for the three products. It can be seen that the software size progressively increased. A 10 percent increase of the PRB size can be noticed relative to PRA and 20 percent in PRC code compared to PRB. Only one EIB from PRA was not included in PRB, while all others were reused with or without modifications for PRB. Additionally, four new EIBs were developed. With respect to PRC, only six EIBs from PRB were functionally modified, the remain PRB EIBs were unchanged. Also, two new EIBs were developed specifically for PRC

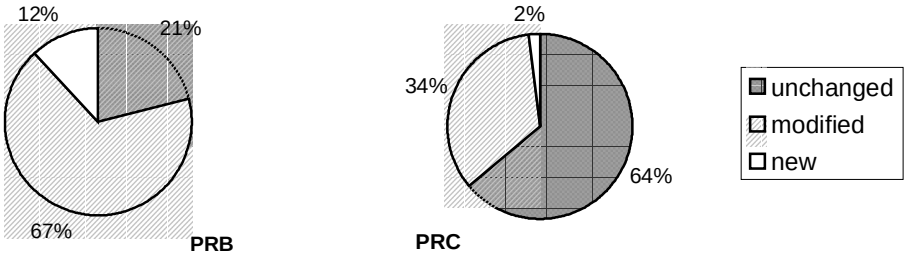
	#EIB	Size (kbytes)
PRA	29	320
PRB	32	351
PRC	34	421

Table 4.1 Number of EIBs and size of PRA, PRB and PRC

Figure 4.1 shows the amount of modification introduced on PRB with respect to PRA and on PRC with respect to PRB, according to the number of EIBs and to the software size. 67% of PRB code results from the modification of the PRA code. About 75% of the modified EIBs belong to the applicative software and 84% of unchanged EIBs to the executive. Thus, the increase of the TROPICO-R capacity mainly led to major modifications of the applicative software with only minor modifications to the executive. With respect to PRC, since the processing capacity of the system was the same as that of PRB, only 34% of PRB code was modified. Most modifications were introduced on the applicative software. When considering the four software functions, it appears that, for both PRB and PRC , most modifications concerned telephony and defense functions.



a) according to the number of EIBs



b) according to the size of EIBs

Figure 4.1 Distribution of unchanged, modified and new EIBs in PRB and PRC

4.2 Test Environment and Collected Data

4.2-1 Test Program

The software test program drawn up for TROPICO-R include four series of tests: unit test, integrated test, validation test, and field test. The first three correspond to the test phases usually defined for a software life cycle. Field test consists of testing a prototype in a real environment, similar to the operational environment. It uses a system configuration (hardware and software) that has reached an acceptable level of quality after completing the laboratory tests.

The test program completed during validation and field testing is made up of four types of test (functional, quality, performance and overload). The whole quality con-

trol program established for TROPICO-R is described in [17]. PRA and PRB validation was carried out according to this program. Also, testers followed a similar test program for the test and validation of PRC. In addition to software testing based on software execution, they used code inspections for static analysis. These inspections were performed during the development of PRC and continued during the operational phase.

4.2-2 Data collection

The failures and troubles impacting the software were reported in appropriate failure or trouble report sheets. A failure report, denoted FR, is filled in whenever a discrepancy is found between the expected and the observed system behavior during software execution. A trouble report, denoted TR, records each fault uncovered during static analysis.
The failure or trouble reports contain the following:

- date of failure or date of detection of faults by static analysis;
- description of system configuration in which the failure was observed and of the conditions of failure occurrence for FRs;
- type of FR or TR: hardware, software, documentation with an indication of EIBs concerned;
- analysis: identification and classification of the fault(s) which led to an abnormal software behavior (coding, specification, interface,…);
- solutions: the proposed solutions and those retained;
- modification control: control of the corrected EIBs;
- regression testing: results of the tests applied to the corrected EIBs.

Only one FR (resp. TR) is kept per observed failure (resp. per detected trouble): rediscoveries are not recorded. In other words, if several FRs (resp.TRs) cover the same failure (resp. the same trouble), only one (the first) is enter into the database. In fact, an FR (TR) is both a failure report (trouble report) and a correction report since it also contains information about the fault(s) that resulted in an abnormal behavior of the software.
For each product and each phase, table 4.2 gives the data collection period. No field test were performed for PRB. This is because many PRA components were reused for the development of PRB, which was then installed in operational sites while PRA had already been operating for several months. For PRC , data collection started at the beginning of the of the operational phase of the system. The data provided by ELEBRA* only refer to this phase and the failures or troubles encountered during validation and field test were not reported.

	validation	Field test	operation
PRA	10 months	4 months	13 months
PRB	8 months	0	24 months
PRC	0	0	47 months

Table 4.2. Validation, field test and operation length for the period of data collection

* Brazilian Telecommunications

4.2-3 Statistics on failures and corrected faults

Table 4.3 gives the number of failures (#FR) and troubles (#TR) reported, as well as the number of corrected faults (#CF) for each product. Note that for PRC, the number of failures and the number of troubles are indicated in order to distinguish between the failures observed during software execution and the troubles identified by static analysis. Clearly, the number of reported trouble is important. This result shows that code inspections are effective and allow a high proportion of software faults to be detected. Experimental studies reported for example in [18] and [19] have shown that thorough static analyses can lead to the detection of 75% to 95% of faults before software execution. The results obtained for PRC show that static analyses could also be helpful in operation.

Table 4.3 shows that less failures occurred in PRB and PRC even though:i) the period of data collection is longer for these products than that of PRA (see Table 4.2) and ii) more PRB and PRC systems have been in use during the operation phase.

Because some failures led to the modification of more than one EIB, the number of corrected faults indicated in Table 4.3 exceeds the number of failures. Table 4.4 give the statistics concerning the number of EIBs that have been corrected because of a software failure[*]. Clearly, the results are similar for the three products. More than 70% of the failures led to the correction of only one EIB. This shows that there is a slight failure interdependence among EIBs.

The analyses of the data corresponding to failures which is involving more than one component allowed us to identify two pairs of EIBs that are strongly dependent in terms of failure occurrence. For these two pairs, it was found that the probability of simultaneous modification of both EIBs exceeds 0.5 whenever a failure was due to a fault located in one of them. This result was obtained for the three products. More generally, this type of analysis can have great help for software maintenance. It allows software debuggers to identify the stochastically dependent components with regards to failure occurrence and to take them into account when looking for the origin of failures.

	#FR / #TR	#CF
PRA	465/-	637
PRB	210/-	282
PRC	212 / 105	394

Table 4.3 Number of failures and corrected faults in PRA, PRB, PRC

# corrected EIBs	# FR in PRA	# FR in PRB	(#FR+#TR) in PRC
1	362 (77.8%)	165 (78.6%)	228 (71.9%)
2	72 (15.5%)	33 (15.7%)	69 (21.8%)
≥3	31 (6.7%)	12 (5.7%)	20 (6.3%)

Table 4.4 Statistics on the number of EIBs affected by one failure

* For the sake of simplicity, in the following, we will define a failure as a discrepancy between the expected and the observed software behavior irrespective of whether it is observed during software execution or detected by static analysis. The distinction between failures and troubles will only be made if required.

4.2-4 Collected failure data of each function for TROPICO-R

Table 4.5 gives the number of failures (#FR) or troubles (#TR), and the number of corrected faults (#CF) attributed to the four functions: TEL, DEF, INT and MAN (as defined in section 4.1). The sum of failure reports attributed to the functions is higher than the total number of failures reports indicated in table 4.4, this is because when a failure impacts different functions, an FR is attributed to each one. The real failure database include "cumulative failure number vs. time" and "failure intensity" are in Appendix B.

	PRA			PRB			PRC		
	size	#FR	#CF	size	#FR	#CF	size	#FR/#TR	#CF
TEL	72	146	190	75	74	102	111	65 / 52	155
DEF	93	138	164	117	67	71	130	63 / 21	87
INT	113	170	191	115	61	68	129	72 / 27	112
MAN	42	78	92	44	31	41	51	25 / 10	40
Sum	320	532	637	351	233	282	421	225 / 110	394

Table 4.5 Size (in Kbytes) and number of failures and corrected faults per function

4.3 Simulate results for TROPICO-R and comparisons

We have applied our simulator to simulate the software reliability of three successive generations products (TROPICO-R, PRA, PRB and PRC). First , we simulated each function of each product, then we made simulations for each product. There are three models are used in these simulation processes, they are: GO (Goel-Okumoto) model, JM (Jelinski-Moranda) model and Yamada S-shaped model. The parameters in these failure rate functions were got by using a software reliability estimation tool CASRE (see [2] Appendix A). The results of simulations are cumulative number of failures and failure intensity of each function component and whole product. Here, we just give the simulation results of "TEL" component and system for each product. The all simulation data are in appendix B.

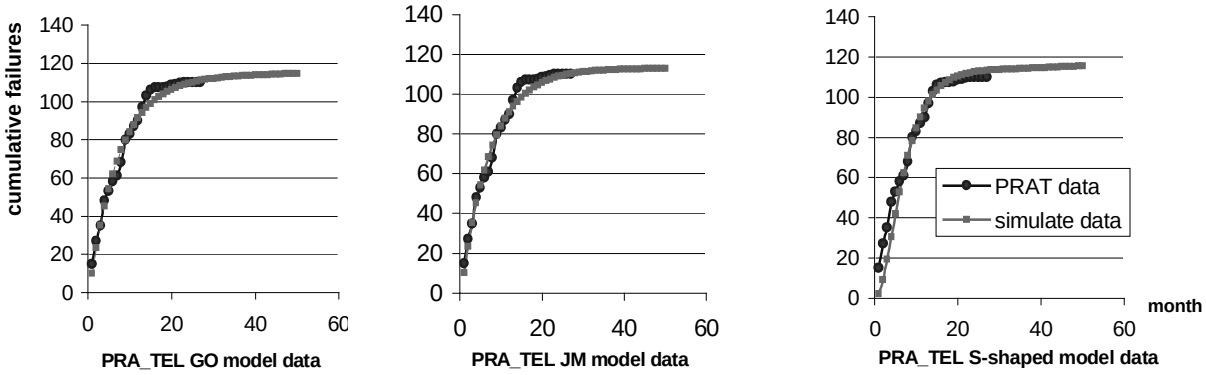


Figure 4.2 PRA_TEL simulate results (with real data)

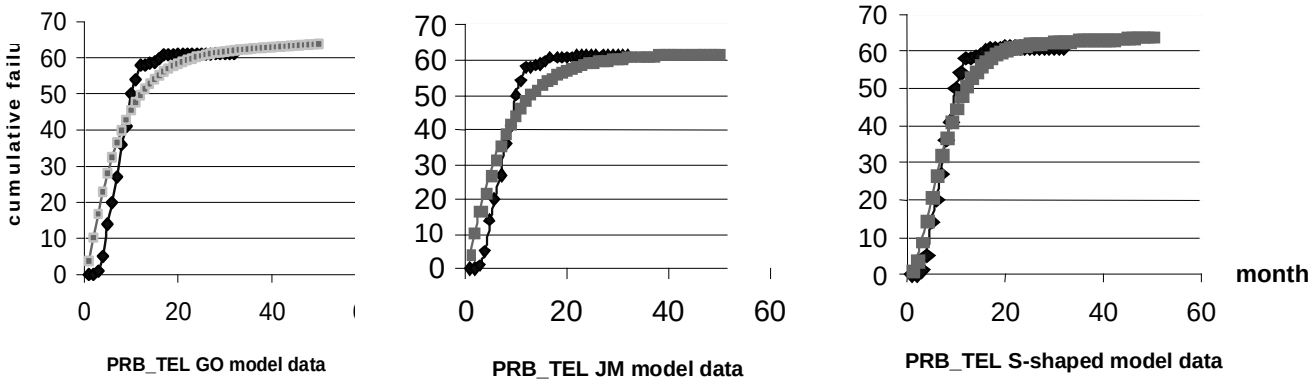


Figure 4.3 PRB_TEL simulate results (with real data)

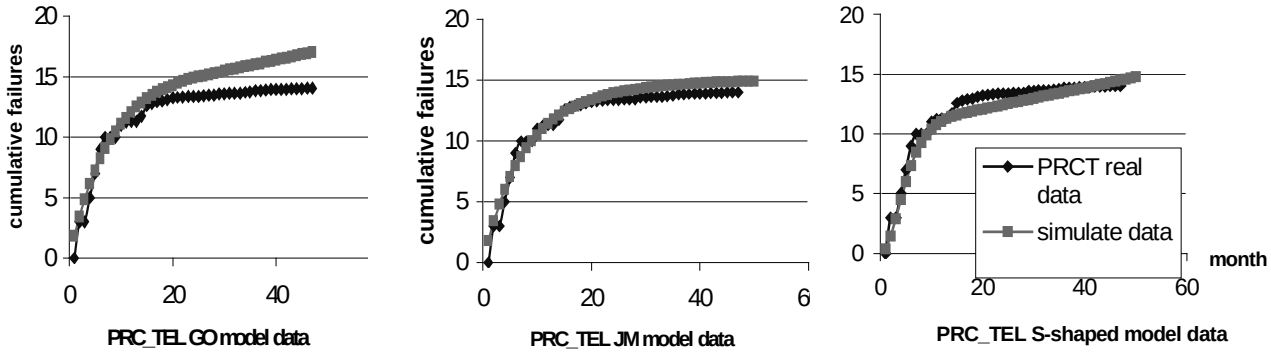


Figure 4.4 PRC_TEL simulate results (with real data)

Figure 4.2 to 4.4 show the comparisons between simulate data with real data for TEL function of each product. The time unit is month, there are 27 months observed failure data for PRA. There are 32 months,47 months observed failure data for PAB and PRC respectively. In order to get prediction, we made 50 months simulation results. From these figures we can see that in case of large number of failures the three models have better fitting and prediction. When there are small total failures number (TEL function of PRC), JM model has better fitting and prediction (see Figure 4.4), GO model has worst prediction. This result indicates that GO model has more dependency with the Eq. 3.3 . From the simulation results, we can also know that in some cases theYamada S-shaped model is closer to the real data at early phase and it has similar prediction with GO model. The simulation results for other functions (DEF, INT, MAN) of the three products (see Appendix B) have similar features with above.

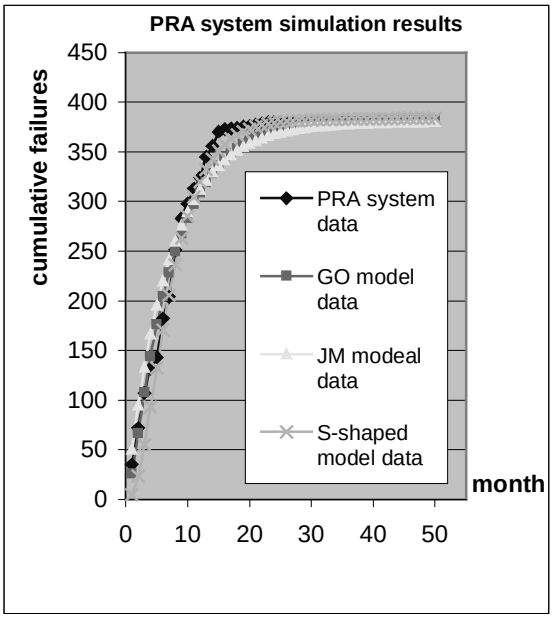


Figure 4.5 PRA system simulation results

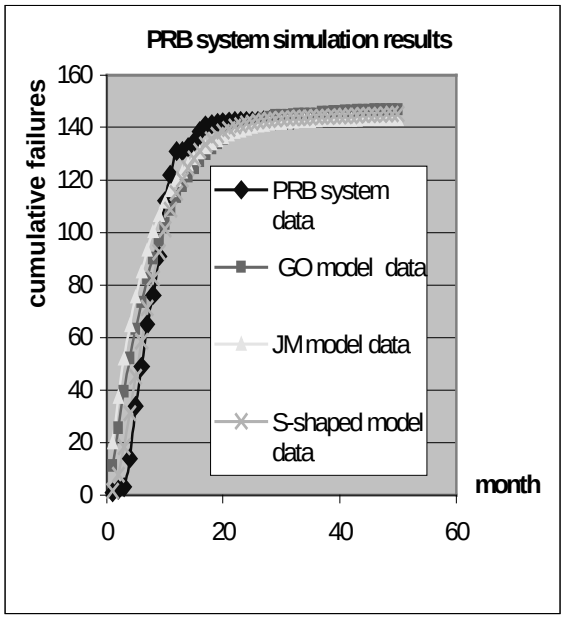


Figure 4.6 PRB system simulation results

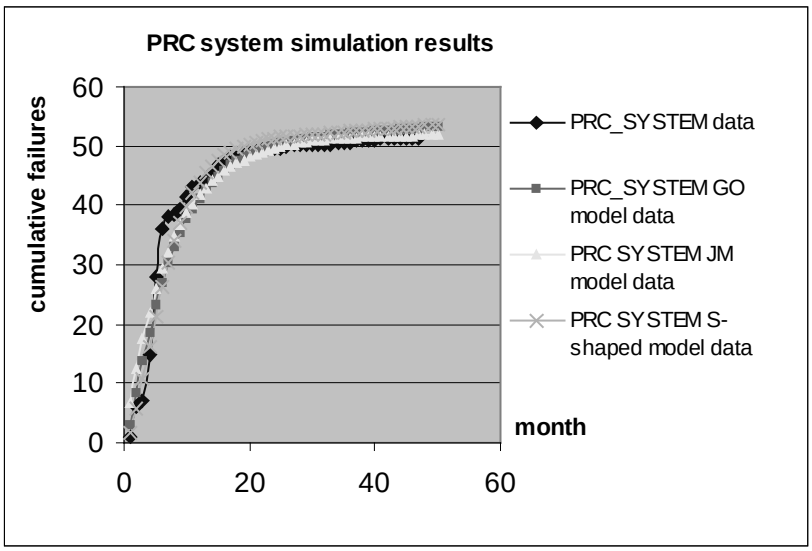


Figure 4.7 PRC system simulation results

Figure 4.5-4.7 give the simulation results of PRA, PRB and PRC. PRA has the largest number of failures . The simulation results of PRA have good fitting and prediction. And for PRA, GO model and JM model have better fitting than S-shaped model during early phase. This may be explained as: PRA is the first generation product, there was no inherited experience for software developer and tester. Thus, the faults have more homogeneous exposure rate during testing phase. For PRB and PRC the S-shaped model has better fitting during early phase, it can be thought that in successive generations software, latent faults are more difficult (take more time) to be found.

4.4 The simulation results deviation

In order to evaluate the accuracy of simulation we define simulation deviation as: the observed failure data value minus the simulation value at that time.
Here, we give the deviations of system simulation results for PRA, PRB and PRC.

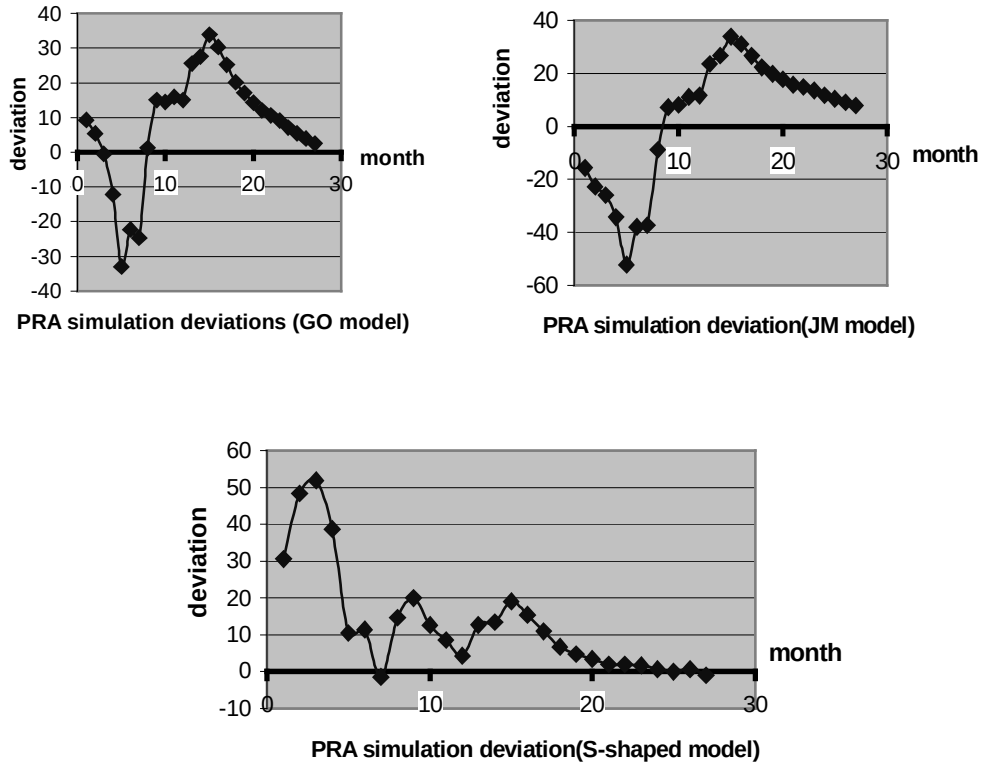


Figure 4.8 PRA system simulation deviations

From figure 4.8 we can see that GO model and JM model simulation deviation are similar. At some time points there are exist large simulation deviations in the three models. In general, it seems that the JM model results has smaller deviations. With comparing Figure 4.5, we can find that the simulation results are very close to the practical data curve, however, with taking account into the time, it is difficult to have accurate failure evaluation or prediction with exact time point. In other words, for a random failure process, simulation can give a trend or general prediction, and it can not give the accurate number with exact occurrence time.
The deviations of PAB and PRC system simulation results are showed in Figure 4.9 and Figure 4.10. As the total failure number is smaller in PRC, the deviation range in Figure 4.10 is smaller.
We noted that near the time of 10 months in the three figures the deviation value have a transition trend, comparing with Figure 4.7, this indicate that at the time point of 10 month the failure occurrence rate begin change.

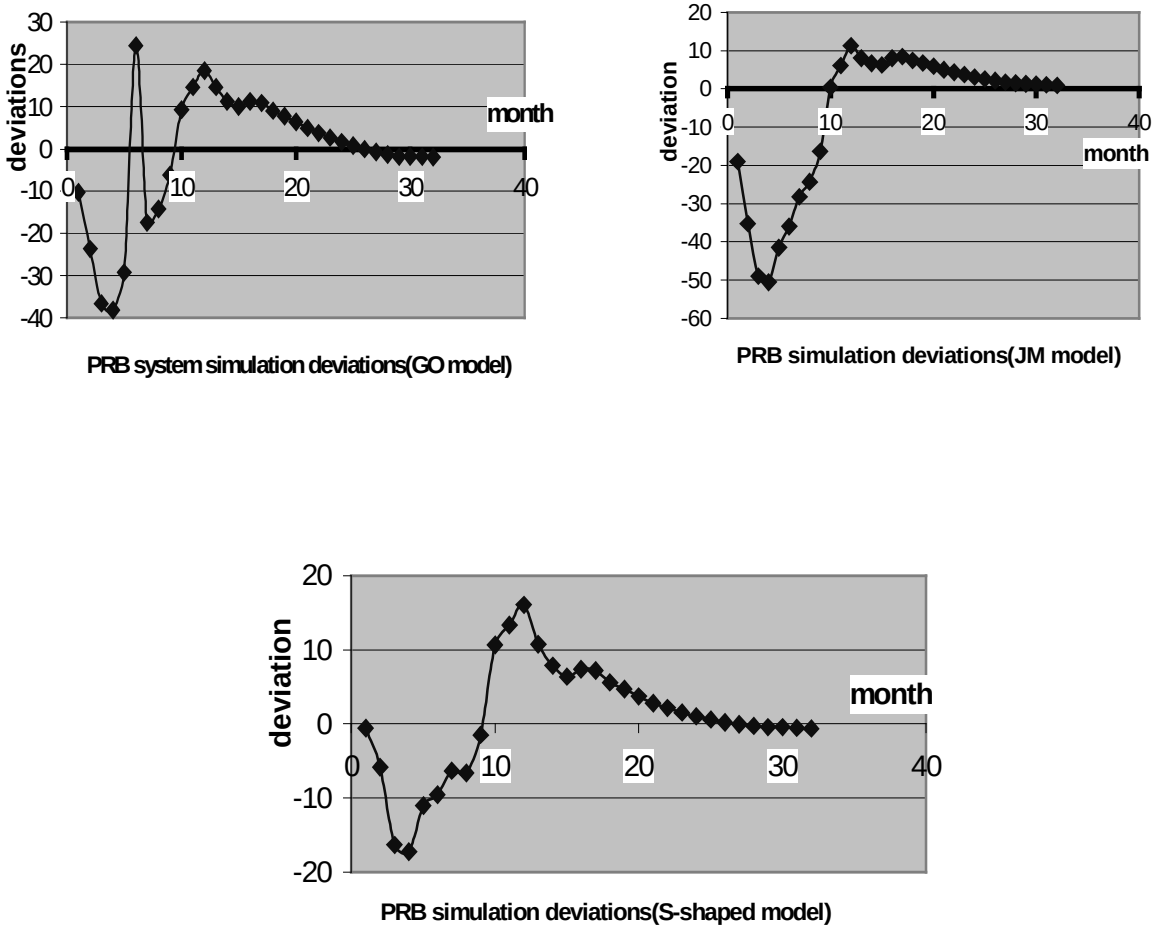
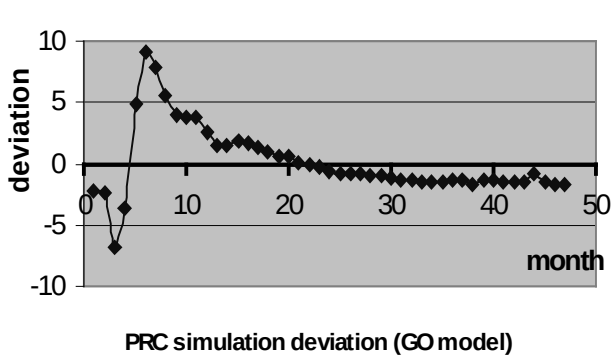
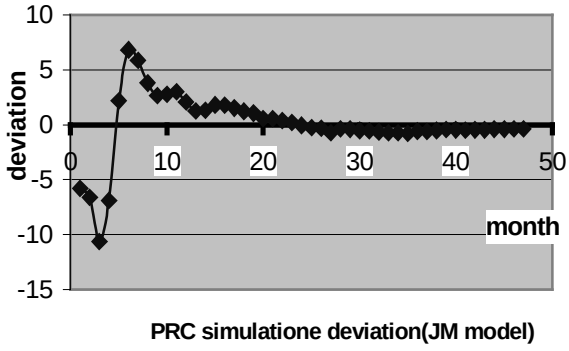


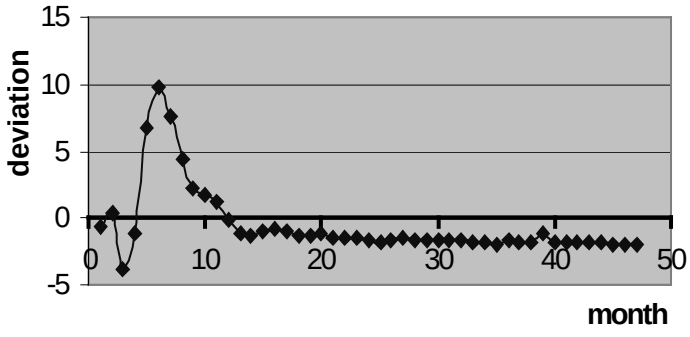
Figure 4.9 PRB system simulation results deviation



PRC simulation deviation (GO model)



PRC simulatione deviation(JM model)



PRC simulation deviation(S-shaped model)

Figure 4.10 PRC system simulation results deviation

5 Conclusion and future work

Many papers deal with software reliability growth modeling and evaluation. However, papers following a global method are seldom found.
In our work, we combined analytical models into simulation approaches to give a effective and practical simulate method for software reliability measures. The main contributions of our work is: implemented a rate-based software reliability measure simulator. Its advantages are: no computation intensive, enable models combination application, taking account into internal structure or dependency of software. The project application demonstrate it can be used for analysis, prediction and evaluation in software reliability literature.

References

[1] Institute of Electrical and Electronics Engineering , *ANSI/IEEE Standard Glossary of Software Engineering Terminology,* IEEE Std. (1991) pp. 729-1991

[2] Michael R.Lyu, *Handbook of Software Reliability Engineering*, McGraw-Hill, New York, (1996)

[3] Goel, A.L. and Okumoto, K., "Time-dependent Error-detection Rate Model for Software Reliability and Other Performance Measures," *IEEE Trans. Reliability*, R-28, (1979) , pp. 206-211

[4] Ohba, M.et al., "S-shaped Software Reliability Growth Curve: How Good Is It?," *COMPSAC'82*, (1982), pp. 38-44

[5] Yamada, S.et al., "S-shaped Software Reliability Growth Models and Their Applications," *IEEE Trans. Reliability*, R-33, (1984), pp. 289-292

[6] Schagen, I.P., "A New Model for Software Failure," *Reliability Engineering*, (1987), pp. 205-221

[7] Langberg, N. and Singpurwalla, N.D., "A Unification of Some Software Reliability Models," *SIAM J. Scientific and Statistical Computation*, 6. (1985), pp. 781-790

[8] Miller, D.R., "Exponential Order Statistical Models of Software Reliability Growth," *IEEE Trans. Software Engineering*, SE-12, (1986), pp. 12-24

[9] Mellor, P., "Experiments in Software Reliability Estimation," *Reliability Engineering*, 18, (1987), pp. 117-129

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

[11] Musa, J.D., "Validity of Execution Time Theory of Software Reliability," *IEEE Trans. Reliability*, R-28(3), (1979), pp. 181-191

[12] Hecht, H., "Measurement, Estimation, and Prediction of Software Reliability," *Software Engineering Technology-Volume* 2, Infotech International, Maidenhead, Berkshire, England, (1979), pp. 209-224

[13] Musa, J.D., and Okumoto, K., "A Logarithmic Poisson Execution Time Model for Software Reliability Measurement," *Proceedings seventh International Conference on Software Engineering*, Orlando, Florida, (1984), pp. 230-238

[14] M. Xie, *Software Reliability Modelling* World Scientific Publishing Co. Pte. Ltd. , (1991)

[15] Swapna S. Gokhale, Michael R. Lyu, Kishor S. Trivedi, "Reliability Simulation of Component-Based Software Systems," *IEEE Proceedings*, (1998)

[16] Karama Kanoun, Marta Rettelbusch de Martini, and Jorge Moreira de Souza "A Method for Software Reliability Analysis and Prediction Application to the TROPICO-R Switching System," *IEEE Trans. Software Engineering*, vol.17,

[17] Vianna, B.,"R&D at TELEBRAS-CPqD:The TROPICO System," in *Proceedings International Conference Communications (ICC88)*, philadelphia, PA, USA, June 1988

[18] Fagan, M.E., "Advances in Software Inspection," *IEEE Trans. Software Eng.*, 12 (7), (1986), pp. 744-751

[19] Shen, V.Y., et al., "Indentifying Error-Prone Software--An Empirical Study," *IEEE Trans. Software Eng.*, SE-11(4), (1985), pp. 317-324