Software Reliability Growth Models: Overview and Applications

1 Razeef Mohd., 2Mohsin Nazir
1, 2 Department of Information Technology, Central University of Kashmir, India
1 m.razeef@gmail.com, 2 mohsin.kawoosa@yahoo.com

ABSTRACT

A software reliability growth model is one of the fundamental techniques used to assess software reliability quantitatively. The software reliability growth model is required to have a good performance in terms of goodness-of-fit, predictability, and so forth. A number of analytical models have been proposed during the past three decades for assessing the reliability of the software system. In this paper we will summarize some existing Software Reliability Growth Models (SRGMs), provide a critical analysis of the underlying assumptions, and assess the applicability of these models during the software development cycle using an example.

Keywords: Software Reliability, Software Reliability Growth Models (SRGMs), Fault, Failure, Non Homogenous Poisson Process (NHPP), fault count models, fault seeding, input domain models, times between failures.

1. INTRODUCTION

Software reliability can be defined as the probability of failure-free software operation for a specified period of time in a specified environment [25], [27], [28], [23]. In highly complex modern software systems, reliability is the most important factor, since it quantifies software failures during the process of software development and software quality control. Software reliability can also be defined as the probability of execution without failures for some specified interval of natural units or time [28]. A failure is a departure of system behavior in execution from user requirements and it is the result of a fault. A fault is a defect that causes or can potentially cause the failure when executed [28]. The models applicable to the assessment of software reliability are called SRGMs. SRGM are useful for estimating how software reliability improves as faults are detected and repaired. It can be used to predict when a particular level of reliability is likely to attained and also helps in determining when to stop testing to attain a given reliability level. SRGMs help in decision making in many software development activities such as number of initial faults, failure intensity, reliability within a specified interval of time period, number of remaining faults, cost analysis and release time etc. A software reliability model describes failures as random process as failures are result of two processes: The introduction of faults and then activation through selection of input states, both of these processes are random in nature. So SRGM is generally described as the probability distribution of the value of the random process at each point in time. SRGMs are developed in general by probability distribution of failure times or the number of failures experienced and by the nature of the random process with time. To achieve highly reliable software systems, many software fault detection/ removal techniques can be used by programmers or testing teams. In applying these techniques, the SRGM are important, because they can provide quite useful information for developers and testers during the testing/debugging phase. Many researchers have also tried to compare various SRGMs [38] by using actual failure data.

Numerous SRGMS have been developed during the last three decades and they can provide very useful information about how to improve reliability [25], [28], [43], [37]. The effort index or the execution time is better time domain for software reliability modeling than the calendar-time because the shape of observed reliability growth curve depends strongly on the time distribution of testing-effort [27], [34].

2. SOFTWARE RELIABILITY GROWTH MODELS

A Software Reliability Growth Model is one of the fundamental techniques to assess software reliability quantitatively [26], [38]. The Software Reliability Growth Model required having a good performance in terms of goodness-of-fit, predictability, and so forth. In order to estimate as well as to predict the reliability of software systems, failure data need to be properly measured by various means during software development and operational phases. Any software required to operate reliably must still undergo extensive testing and debugging. This can be a costly and time consuming process, and managers require accurate information about how software reliability grows as a result of this process in order to effectively manage their budgets and projects. The effects of this process, by which it is hoped software is made more reliable, can be modeled through the use of Software Reliability Growth Models, hereafter referred to as SRGMs. Research efforts in software reliability engineering have been conducted over the past three decades and many software reliability growth models (SRGMs) have been proposed. SRGMs can estimate the number of initial faults, the software reliability, the failure intensity, the mean time-interval between failures, etc.
Ideally, these models provide a means of characterizing the development process and enable software reliability practitioners to make predictions about the expected future reliability of software under development. Such techniques allow managers to accurately allocate time, money, and human resources to a project, and assess when a piece of software has reached a point where it can be released with some level of confidence in its reliability. Unfortunately, these models are often inaccurate. A comparative study of Software Reliability Growth Models [38] allows to determine with models is suited well and under what conditions.

A number of analytical models have been proposed to address the problem of software reliability measurement. [26], [28], [38]. These approaches are based mainly on the failure history of software and can be classified according to the nature of the failure process studied as indicated below.

a. Times between Failures Models

In this class of models, the process under study is the time between failures. The most common approach is to assume that the time between, say, the \((i-1)\)st and the \(i\)th failures, follows a distribution whose parameters depend on the number of faults remaining in the program during this interval. Estimates of the parameters are obtained from the observed values of times between failures and estimates of software reliability, mean time to next failure, etc., are then obtained from the fitted model. Another approach is to treat the failure times as realizations of a stochastic process and use an appropriate time-series model to describe the underlying failure process.

d. Input Domain Based Models

The basic approach taken here is to generate a set of test cases from an input distribution which is assumed to be representative of the operational usage of the program. Because of the difficulty in obtaining this distribution, the input domain is partitioned into a set of equivalence classes, each of which is usually associated with a program path. An estimate of program reliability is obtained from the failures observed during physical or symbolic execution of the test cases sampled from the input domain.

Model Assumptions

Table 1 list various assumptions underlying the models defined above. Not all of the assumptions listed here are relevant to any given model but, as a totality, they provide an insight into the kind of limitations imposed by them on the use of the software reliability models. It should be pointed out that the arguments presented here are not likely to be universally applicable because the software development process is very environment dependent. What holds true in one environment may not be true in another. Because of this, even assumptions that seem reasonable, e.g., during the testing of one function or system, may not hold true in subsequent testing of the same function or system. The ultimate decision about the appropriateness of the underlying assumptions and the applicability of the models will have to be made by the user of a model. What is presented here should be helpful in determining whether the assumptions associated with a given model are representative of the user's development environment and in deciding which model, if any, to use [44].

<table>
<thead>
<tr>
<th>Table 1: List of Key Assumptions by Model Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Times Between Failures (TBF) Models</td>
</tr>
<tr>
<td>• Independent times between failures.</td>
</tr>
<tr>
<td>• Equal probability of the exposure of each fault.</td>
</tr>
<tr>
<td>• Embedded faults are independent of each other.</td>
</tr>
<tr>
<td>• Faults are removed after each occurrence.</td>
</tr>
<tr>
<td>• New faults introduced during correction, i.e. perfect fault removal.</td>
</tr>
<tr>
<td>Fault Count (FC) Models</td>
</tr>
<tr>
<td>• Testing intervals are independent of each other.</td>
</tr>
<tr>
<td>• Testing during intervals is reasonably homogenous.</td>
</tr>
<tr>
<td>• Numbers of faults detected during non-overlapping intervals are independent of each other.</td>
</tr>
</tbody>
</table>
3. TIMES BETWEEN FAILURES MODELS

This is one of the earliest classes of models proposed for software reliability assessment. When interest is in modeling times between failures, it is expected that the successive failure times will get longer as faults are removed from the software system. For a given set of observed values, this may not be exactly so due to the fact that failure times are random variables and observed values are subject to statistical fluctuations.

A number of models have been proposed to describe such failures. Let a random variable $T_i$ denote the time between the $(i-1)$th and the $i$th failures. Basically, the models assume that $T_i$ follows a known distribution whose parameters depend on the number of faults remaining in the system after the $(i-1)$th failure. The assumed distribution is supposed to reflect the improvement in software quality as faults are detected and removed from the system. The key models in this class are described below.

a. Jelinski and Moranda (JM) Degradation Model

This is one of the earliest and probably the most commonly used model for assessing software reliability [38]. It assumes that there are $N$ software faults at the start of testing, each is independent of others and is equally likely to cause a failure during testing. A detected fault is removed with certainty in a negligible time and no new faults are introduced during the debugging process. The software failure rate, or the hazard function, at any time is assumed to be proportional to the current fault content of the program. In other words, the hazard function during, the time between the $(i-1)th$ and $i$th failures, is given by

$$Z(t_i) = \Theta [N - (i - 1)]$$

where $\Theta$ is a proportionality constant. Note that this hazard function is constant between failures but decreases in steps of size $k$ following the removal of each fault. A typical plot of the hazard function for $N = 100$ and $k = 0.01$ is shown in Fig. 1.

![Fig 1: A typical plot of $Z(t_i)$ for the JW model (N=100, $\Theta=0.01$).](image)

A variation of the above model was proposed by Moranda [43] to describe testing situations where faults are not removed until the occurrence of a fatal one at which time the accumulated group of faults is removed. In such situation, the hazard function after a restart can be assumed to be a fraction of the rate which attained when the system crashed. For this model, called the geometric degradative model, the hazard function during the $i$th testing interval is given by

$$Z(t_i) = Dk^{i-1}$$

Where $D$ is the fault detection rate during the first interval and $k$ is a constant ($0 < k < 1$).

![Fig 2: A typical plot of the hazard function for the SW model (N=150, $\Theta=0.02$).](image)

b. Schick and Wolverton (SW) Model

This model is based on the same assumptions as the JM model except that the hazard function is assumed to be proportional to the current fault content of the program as well as to the time elapsed since the last failure [14] is given by

$$Z(t_i) = \Theta [N - (i - 1)]$$

where $\Theta$ is a proportionality constant. Note that this hazard function is constant between failures but decreases in steps of size $k$ following the removal of each fault. A typical plot of the hazard function for $N = 100$ and $k = 0.01$ is shown in Fig. 1.
where the various quantities are as defined above. Note that in some papers $t_i$ has been taken to be the cumulative time from the beginning of testing. We note that the above hazard rate is linear with time within each failure interval, returns to zero at the occurrence of a failure and increases linearly again but at a reduced slope, the decrease in slope being proportional to 4. A typical behavior of $Z(t_i)$ for $N = 150$ and $\alpha = 0.02$ is shown in Fig. 2.

A modification of the above model was proposed in [15] whereby the hazard function is assumed to be parabolic in test time and is given by

$$Z(t_i) = \alpha \left[ N - (i - 1) \right] \lambda$$

Where $\alpha$, $b$, $c$ are constants and the other quantities are as defined above. This function consists of two components. The first is basically the hazard function of the JM model and the superimposition of the second term indicates that the likelihood of a failure occurring increases rapidly as the test time accumulates within a testing interval. At failure times ($t_i = 0$), the hazard function is proportional to that of the JM model.

c. Goel and Okumoto Imperfect Debugging Model

The above models assume that the faults are removed with certainty when detected. However, in practice [40] that is not always the case. To overcome this limitation, Goel and Okumoto [7], [4] proposed an imperfect debugging model which is basically an extension of the JM model. In this model, the number of faults in the system at time $t$, $X(t)$, is treated as a Markov process whose transition probabilities are governed by the probability of imperfect debugging. Times between the transitions of $X(t)$ are taken to be exponentially distributed with rates dependent on the current fault content of the system. The hazard function during the interval between the $(i - 1)^{th}$ and the $i^{th}$ failures is given by

$$Z(t_i) = \alpha \left[ N - p(i - 1) \right] \lambda$$

Where $N$ is the initial fault content of the system, $p$ is the probability of imperfect debugging, and $\lambda$ is the failure rate per fault.

d. Littlewood - Verrall Bayesian Model

Littlewood and Verall [9], [10] took a different approach to the development of a model for times between failures. They argued that software reliability should not be specified in terms of the number of errors in the program. Specifically, in their model, the times between failures are assumed to follow an exponential distribution but the parameter of this distribution is treated as a random variable with a gamma distribution, viz.

$$f(t_i | \lambda_i) = \lambda_i e^{-\lambda_i t_i}$$

and

$$f(\lambda_i | \psi(i)) = \frac{\lambda_i^{\alpha-1} e^{-\psi(i)\lambda_i}}{\Gamma \alpha}$$

In the above, $\psi(i)$ describes the quality of the programmer and the difficulty of the programming task. It is claimed that the failure phenomena in different environments can be explained by this model by taking different forms for the parameter $\psi(i)$.

4. FAULT COUNT MODELS

This class of models is concerned with modeling the number of failures seen or faults detected in given testing intervals. As faults are removed, from the system, it is expected that the observed number of failures per unit time will decrease. If this is so, then the cumulative number of failures versus time curve will eventually level off. Note that time here can be calendar time, CPU time, number of test cases run or some other relevant metric. In this setup, the time intervals may be fixed a priori and the observed number of failures in each interval is treated as a random variable.

Several models have been suggested to describe such failure phenomena. The basic idea behind most of these models is that of a Poisson distribution whose parameter takes different forms for different models. It should be noted that Poisson distribution has been found to be an excellent model in many fields of application where interest is in the number of occurrences.

One of the earliest models in this category was proposed by Shooman [31]. Taking a somewhat similar approach, Musa [17] later proposed another fault count model based on execution time. Schneidewind [33] took a different approach and studied the fault counts over a series of time intervals. Goel and Okumoto [7] introduced a time dependent failure rate of the underlying Poisson process and developed the necessary analytical details of the models. A generalization of this model was proposed by Goel [2]. These and some other models in this class are described below.

a. Goel-Okumoto Nonhomogeneous Poisson Process Model

In this model Goel and Okumoto [6] assumed that a software system is subject to failures at random times caused by faults present in the system. Letting $N(t)$ be the
cumulative number of failures observed by time \( t \), they proposed that \( N(t) \) can be modeled as a non-homogeneous Poisson process, i.e., as a Poisson process with a time-dependent failure rate. Based on their study of actual failure data from many systems, they proposed the following form of the model

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

where

\[
m(t) = a(1 - e^{-bt}), \quad \lambda(t) = m'(t) = abc e^{-bt}.
\]

Here \( m(t) \) is the expected number of failures observed by time \( t \) and \( \lambda(t) \) is the failure rate. Typical plots of the \( m(t) \) and \( \lambda(t) \) functions are shown in Fig. 3.

\[\text{Fig 3: A typical plot of the } m(t) \text{ and } \lambda(t) \text{ functions for the Goel-Okumoto NHPP model (} a = 175, b = 0.05)\]

In this model, \( a \) is the expected number of failures to be observed eventually and \( b \) is the fault detection rate per fault. It should be noted that here the number of faults to be detected is treated as a random variable whose observed value depends on the test and other environmental factors. This is a fundamental departure from the other models which treat the number of faults to be a fixed unknown constant.

In some environments a different form of the \( m(t) \) function might be more suitable than the one given above, see, e.g., Ohba [29] and Yamada et al. [148].

Using a somewhat different approach than described above, Schneidewind [33] had earlier studied the number of faults detected during a time interval and failure counts over a series of time intervals. He assumed that the failure process is a non-homogeneous Poisson process with an exponentially decaying intensity function given by

\[
d(i) = \alpha e^{-\beta i}, \quad \alpha, \beta > 0, \quad i = 1, 2, \ldots
\]

Where \( \alpha \) and \( \beta \) are the parameters of the model.

b. Goel Generalized Nonhomogeneous Poisson Process Model

Most of the times between failures and failure count models assume that a software system exhibits a decreasing failure rate pattern during testing. In other words, they assume that software quality continues to improve as testing progresses. In practice, it has been observed that in many testing situations, the failure rate first increases and then decreases. In order to model this increasing/decreasing failure rate process, Goel [2], [3] proposed the following generalization of the Goel-Okumoto NHPP model.

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

\[
m(t) = a(1 - e^{-bt}),
\]

Where \( a \) is expected number of faults to be eventually detected, and \( b \) and \( c \) are constants that reflect the quality of testing. The failure rate for the model is given by

\[
\lambda(t) = m'(t) = abc e^{-bt} t^{-1}
\]

Typical plots of the \( m(t) \) and \( \lambda(t) \) functions are shown in Fig. 4.

\[\text{Fig 4: A typical plot of the } m(t) \text{ and } \lambda(t) \text{ functions for the Goel generalized NHPP model (} a = 500, b = 0.015, c = 1.5)\]

c. Musa Execution Time Model

In this model, Musa [17] makes assumptions that are similar to those of the JM model except that the process modeled is the number of failures in specified execution
time intervals. The hazard function for this model is given by
\[ z(\tau) = \Theta f(N - n_c(\tau)) \]

where \( \tau \) is the execution time utilized in executing the program up to the present, \( f \) is the linear execution frequency (average instruction execution rate divided by the number of instructions in the program), \( \Theta \) is a proportionality constant, which is a fault exposure ratio that relates fault exposure frequency to the linear execution frequency, and \( n_c \) is the number of faults corrected during \((0, \tau)\). One of the main features of this model is that it explicitly emphasizes the dependence of the hazard function on execution time. Musa also provides a systematic approach for converting the model so that it can be applicable for calendar time as well.

d. Shooman Exponential Model

This model is essentially similar to the JM model. For this model the hazard function is of the following form
\[ z(t) = k \left( \frac{N}{I} - \frac{n_c(t)}{I} \right) \]

Where \( t \) is the operating time of the system measured from its initial activation, \( I \) is the total number of instructions in the program, \( \tau \) is the debugging time since the start of system integration, \( n_c(t) \) is the total number of faults corrected during \( \tau \), normalized with respect to \( I \), and \( k \) is proportionality constant.

e. Generalized Poisson Model

This is a variation of the NHPP model of Goel and Okumoto and assumes a mean value function of the following form.
\[ m(t_i) = \Theta (N - M_{i-1}) \tau^a_i \]

Where \( M_{i-1} \) is the total number of faults removed up to the end of the \((i-1)\) st debugging interval, \( \Theta \) is a constant of proportionality, and \( a \) is a constant used to rescale time.

f. IBM Binomial and Poisson Models

In these models Brooks and Motley consider the fault detection process during software testing to be a discrete process, following a binomial or a Poisson distribution. The software system is assumed to be developed and tested incrementally. They claim that both models can be applied at the module or the system level.

g. Musa-Komodo Logarithmic Poisson Execution Time Model

In this model the observed number of failures by some time \( T \) is assumed to be a NHPP, similar to the Goel-Okumoto model, but with a mean value function which is a function of \( \tau \), viz.
\[ \mu(\tau) = \frac{1}{\Theta} \ln(\lambda_0 \theta + 1), \]

Where \( \lambda_0 \) and \( \Theta \) represent the initial failure intensity and the rate of reduction in the normalized failure intensity per failure, respectively. This model is also closely related to Moranda's geometric de-eutrophication model [45] and can be viewed as a continuous version of this model.

5. FAULT SEEDING AND INPUT DOMAIN BASED MODELS

In this section we give a brief description of a few time independent models that have been proposed for assessing software reliability. As mentioned earlier, the two approaches proposed for this class of models are fault seeding and input domain analysis. In fault seeding models, a known number of faults are seeded (planted) in the program. After testing, the numbers of exposed seeded and indigenous faults are counted. Using combinatorial and maximum likelihood estimation, the number of indigenous faults in the program and the reliability of the software can be estimated. The basic approach in the input domain based models is to generate a set of test cases from an input (operational) distribution. Because of the difficulty in estimating the input distribution, the various models in this group partition the input domain into a set of equivalence classes. An equivalence class is usually associated with a program path. The reliability measure is calculated from the number of failures observed during symbolic or physical execution of the sampled test cases.

a. Mills Seeding Model

The most popular and most basic fault seeding model is Mills' Hyper geometric model [16]. This model requires that a number of known faults be randomly seeded in the program to be tested. The program is then tested for some amount of time. The number of original indigenous faults can be estimated from the numbers of indigenous and seeded faults uncovered during the test by using the hyper geometric distribution. The procedure adopted in this model is similar to the one used for estimating population of fish in a pond or for estimating wildlife. These models are also referred to as tagging models since a given faults tagged as seeded or indigenous.

Lipow [30] modified this problem by taking into consideration the probability of finding a fault, of either
kind, in any test of the software. Then, for statistically independent tests, the probability of finding given numbers of indigenous and seeded faults can be calculated. In another modification, Basin [39] suggested a two stage procedure with the use of two programmers which can be used to estimate the number of indigenous faults in the program.

b. Nelson Model

In this input domain based model [12], the reliability of the software is measured by running the software for a sample of n inputs. Then inputs are randomly chosen from the input domain set \( E = (E_i; i = 1, \ldots, N) \) where each \( E_i \) is the set of data values needed to make a run. The random sampling of n inputs is done according to a probability distribution \( P_i \); the set \( (P_i; i = 1, N) \) is the operational profile or simply the user input distribution. If \( n_e \) is the number of inputs that resulted in execution failures, then an unbiased estimate of software reliability \( \hat{R}_1 \) is \( \{1 - (n_e/n)\} \). However, it may be the case that the test set used during the verification phase may not be representative of the expected operational usage. Brown and Lipow [21] suggested an alternative formula for \( \hat{R} \) which is

\[
\hat{R}_2 = 1 - \left( \frac{f_i}{n_j} \right) p(E_j) \left( \frac{f_i}{n_j} \right)
\]

where \( i \) is the number of runs sampled from input sub domain \( E_j \) and \( f \) is the number of failures observed out of \( n_j \) runs.

The main difference between Nelson's RI and Brown and Lipow's \( \hat{R}_2 \) is that the former explicitly incorporates the usage distribution or the test case distribution while the latter implicitly assumes that the accomplished testing is representative of the expected usage distribution. Both models assume prior knowledge of the operational usage distribution.

c. Ramamurthy and Bastani Model

In this input domain based model, the authors are concerned with the reliability of critical, real-time, process control programs. In such systems no failures should be detected during the reliability estimation phase, so that the reliability estimate is one. Hence, the important metric of concern is the confidence in the reliability estimate. This model provides an estimate of the conditional probability that the program is correct for all possible inputs given that it is correct for a specified set of inputs. The basic assumption is that the outcome of each test case provides at least some stochastic information about the behavior of the program for other points which are close to the test point. The specific model is discussed in [17], [11].

A main result of this model is

\[
P \{ \text{program is correct for all points in } [a, a + V] \ | \text{it is correct for test cases having successive distances } x_j, j = 1, \ldots, n - 1 \} = e^{-\lambda y} \prod_{j=1}^{n-1} \left[ \frac{2}{1 + e^{-\lambda y}} \right]
\]

Where \( \lambda \) is a parameter which is deduced from some measure of the complexity of the source code.

Unlike other sampling models, this approach allows any test case selection strategy to be used. Hence, the testing effort can be minimized by choosing test cases which exercise error-prone constructs. However, the model concerning the parameter \( X \) needs to be validated experimentally.

A related model based on fuzzy set theory is discussed in [13].

6. APPLICABILITY OF SOFTWARE RELIABILITY MODELS

In this section we consider the four classes of Software Reliability Models and assess their applicability during the design, unit testing, integration testing, and operational phases of the software development process.

a. Design Phase

During the design phase, faults may be detected visually or by other formal or informal procedures. Existing software reliability models are not applicable during this phase because the test cases needed to expose faults as required by fault seeding and input domain based models do not exist, and the failure history required by time dependent models is not available.

b. Unit Testing

The typical environment during module coding and unit testing phase is such that the test cases generated from the module input domain do not form a representative sample of the operational usage distribution. Further, times between exposures of module faults are not random since the test strategy employed may not be random testing. In fact, test cases are usually executed in a deterministic fashion. Given these conditions, it seems that the fault seeding models are applicable provided it can be assumed that the indigenous and seeded faults have equal probabilities of being detected. However, a difficulty could arise if the programmer is also the tester in this phase. The input domain based models seem to be applicable, except that matching the test profile to operational usage distribution could be
difficult. Due to these difficulties, such models, although applicable, may not be usable.

The time dependent models, especially the time between failures models, do not seem to be applicable in this environment since the independent times between failures assumption is seriously violated.

c. Integration Testing

A typical environment during integration testing is that the modules are integrated into partial or whole systems and test cases are generated to verify the correctness of the integrated system. Test cases for this purpose may be generated randomly from an input distribution or may be generated deterministically using a reliable test strategy, the latter being probably more effective. The exposed faults are corrected and there is a strong possibility that the removal of exposed faults may introduce new faults. Under such testing conditions, fault seeding models are theoretically applicable since we still have the luxury of seeding faults into the system. Input domain based models based on an explicit test profile distribution are also applicable. The difficulty in applying them at this point is the very large number of logic paths generated by the whole system.

If deterministic testing (e.g., boundary value analysis, path testing) is used, times between failures models may not be appropriate because of the violation of the independence of inter failure times assumption. Fault count models may be applicable if sets of test cases are independent of each other, even if the tests within a set are chosen deterministically. This is so because in such models the system failure rate is assumed to decrease as a result of executing a set of test cases and not at every failure.

If random testing is performed according to an assumed input profile distribution, then most of the existing software reliability models are applicable. Input domain based models, if used, should utilize a test profile distribution which is statistically equivalent to the operational profile distribution. Fault seeding models are applicable likewise, since faults can be seeded and the equal probability of fault detection assumption may not be seriously violated. Thesis due to the random nature of the test generation process.

Times between failures and failure count models are most applicable with random testing. The next question could be about choosing a specific model from a given class. Some people prefer to try a couple of these model sons the same failure history and then choose one. However, because of different underlying assumptions of these models, there are subtle distinctions among them. Therefore, as far as possible, the choice of a specific model should be based on the development environment considerations.

d. Acceptance Testing

During acceptance testing, inputs based on operational usage are generated to verify software acceptability. In this phase, seeding of faults is not practical and the exposed faults are not usually immediately corrected. The fault seeding and times between failures models are thus not applicable. Many other considerations here are similar to those of integration testing so that the fault count and input domain based models are generally applicable.

e. Operational Phase

When the reliability of the software as perceived by the developer or the "friendly users" is already acceptable, the software is released for operational use. During the operational phase, the user inputs may not be random. This is because the user may use the same software function or path on a routine basis. Inputs may also be correlated (e.g., in real-time systems), thus losing their randomness. Furthermore, faults are not always immediately corrected. In this environment, fault-count models are likely to be most applicable and could be used for monitoring software failure rate or for determining the optimum time for installing a new release.

f. An Example Illustrating Software Reliability Modeling

We now employ the above procedure to illustrate the development of a software reliability model based on failure data from a real-time, command and control system. The delivered number of object instructions for this system was 21 700 and it was developed by Bell Laboratories. The data were reported by Musa [18] and represent the failures observed during system testing for 25 hours occupy time.

For purposes of this illustration, we employ the NHPP model of Goel and Komodo [6]. We do so because of its simplicity and applicability over a wide range of testing situations as also noted by Misra [36], who successfully used this model to predict the number of remaining faults in a space shuttle software subsystem.

Step 1: The original data were reported as times between failures. To overcome the possible lack of independence among these values, we summarized the data into numbers of failures per hour of execution time. The summarized data are given in Table II. A plot of the hourly data is shown in Fig. 5 and a plot of N(t), the cumulative number of failures by t, is shown in Fig. 6. Some other plots shown in Fig. 6 will be discussed later.

Step 2: A study of the data in Table II and of the plotting Fig. 6 indicates that the failure rate (number of
failures per hour) seems to be decreasing with test time. This means that an NHPP with a mean value function

\[ m(t) = a(1 - e^{-bt}) \]

should be a reasonable model to describe the failure process.

**Step 3:** For the above model, two parameters, \( a \) and \( b \), are to be estimated from the failure data. We chose to use the method of maximum likelihood for this purpose [5],[2]. The estimated values for the two parameters are \( a = 142.32 \) and \( b = 0.1246 \). Recall that \( a \) is an estimate of the expected total number of faults likely to be detected and \( b \) represents the number of faults detected per fault per hour.

**Step 4:** The fitted model based on the data of Table II and the parameters estimated in Step 3 is

\[ m(t) = 142.32 \left( 1 - e^{-0.1246t} \right) \]

and

\[ \lambda(t) = 17.73 \cdot e^{-0.1246t} \]

**Step 5:** In this case, we used the Kolmogorov-Smirnov goodness-of-fit test for checking the adequacy of the model. For details of this test, see Goal [3]. Basically, the test provides a statistical comparison between the actual data and the model chosen in Step 2. The fitted modeling Step 4 passed this test so that it could be considered a good descriptor of the data in Table II. The plots in Fig. 6 also provide a visual check of the goodness-of-fit of the model.

**Table 2:** Failures in 1 hour (execution time) intervals and cumulative failures

<table>
<thead>
<tr>
<th>Hour</th>
<th>Number of failures</th>
<th>Cumulative failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>43</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>54</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>75</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>82</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>84</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>89</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>92</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>93</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>97</td>
</tr>
</tbody>
</table>
Fig 7: Estimated remaining number of faults and confidence bounds.

Plots of the confidence bounds for the expected cumulative number of failures, and the expected number of remaining faults are also shown in Figs. 6 and 7, respectively. A study of these plots indicates that the chosen NHPP model provides an excellent fit to the data and can be used for purposes of describing the failure behavior as well as for prediction of the future failure process. The information available from this can be used for planning, scheduling, and other management decisions as indicated below.

Step 7: The model developed above can be used for answering a variety of questions about the failure process and for determining the additional test effort required until the system is ready for release. This type of information can be sought at various points of time and one does not have to wait until the end of testing. For illustrative purposes, suppose that failure data through only 16 hours of testing are available, and a total of 122 failures (see Table II) have been observed. Based on these data, the fitted model is

\[ m(t) = 138.37 \left(1 - e^{-0.13t}\right). \]

An estimate of the remaining number of faults is 16.37 with a 90 percent confidence interval of (4.64-28.11). Also, the estimated one hour ahead reliability is 0.165 and the corresponding 90 percent confidence interval is (0.019-0.310).

Now, suppose that a decision to release software for operational use is to be based on the number of remaining faults. Specifically, suppose that we would release the system if the expected number of remaining faults is less than or equal to 10. In the above analysis we saw that the best estimate of this quantity at present is 16.37, which means that we should continue testing in the hope that additional faults can be detected and removed. If we were to carry on a similar analysis after each additional hour of testing, the expected number of remaining faults after 20 hours would be 9.85 so that the above release criterion would be met.

The above simple example was meant to illustrate the kind of information that can be obtained from a software reliability model. In practice, determination of release time, additional testing effort, etc. are based on much more elaborate considerations than remaining faults. The results from models such as the ones developed here can be used as inputs into the decision-making process.

7. CONCLUSION

In this research effort, we have provided an overview of some existing software reliability growth models. Four classes of analytical models, along with their underlying assumptions, were described. The use and applicability of such models during software development and operational phase was also discussed. A methodology for developing a model from failure data was illustrated via an example. The objective was to provide a user an insight into the usefulness of such models that will be helpful in determining which model to use in a given software development environment. Distinctive feature of this research is that we do not add any new models to the already large collection of SRGMs.

REFERENCES


