A Multiphase Software Reliability Model: From Testing to Operational Phase

Teebu Philip, Peter N. Marinos, Kishor S. Trivedi *
Center for Advanced Computing and Communication
Department of Electrical and Computer Engineering
Duke University
Durham, NC 27708-0291
{teebu, pnm, kst}@ee.duke.edu

Jay Lala
The Charles Stark Draper Lab
555 Technology Square
Cambridge, MA 02139
lala@draper.com

Abstract

Accurate prediction of release times, reliability and availability of a software product requires the formulation of a reliability model which captures in a quantitative manner critical elements of the software testing process such as: (i) test coverage and (ii) the effect of operational profiles. This paper presents a Non-Homogeneous Poisson Process (NHPP) based model which accounts explicitly for test coverage and can make predictions about the operational phase. The model provides, also, for defective fault detection and test coverage during testing and operational phases. The model yields reliability and availability functions which utilize test and operational data for making performance predictions during the operational phase. Stopping rules are developed for determining optimal software release times subject to various constraints such as cost, operational reliability and operational availability requirements.

Index Terms: Software reliability modeling; Test coverage; Software availability

1 Introduction

Software is an integral part of modern systems, and it is the dominant element of complexity and cost. Additionally, high quality software is required by many critical applications such as commercial avionics, banking, nuclear power generation, and medical instrumentation. A methodology for certifying software integrity in an objective and fairly accurate manner is absolutely essential. All these factors converge to a fundamental point, namely, the need for a far more accurate and cost-effective software reliability model than those presently in use.

*This work was supported in part by the US AIR FORCE Rome Laboratory as a core project in the Center for Advanced Computing and Communication and by a contract from the Charles Stark Draper Laboratory.
A software reliability model must account quantitatively for: (i) test coverage and (ii) operational profiles for the software product. The literature is replete with references to these important elements[13, 15, 20], but very little has been accomplished in terms of their direct and quantitative incorporation into existing software reliability models. Since software reliability estimates impact significantly the release time of a software product, and thus its development and maintenance costs, the model accuracy becomes crucial and it should account for program testability, test coverage, test quality and test effectiveness. In addition to test coverage, one must formulate a model which allows for predictions into the operational phase of the product. The model should be flexible enough to handle the various operational profiles and be able to recognize the distinctions between operational and testing environments.

2 Background and Motivation

Program testability and test coverage are related concepts. The first refers to the ease with which one may test a program while the second provides a measure of how thoroughly a test has “covered” all potential fault-sites in a program. A potential fault-site is defined very broadly to mean any structurally or functionally described program element whose integrity may require verification and validation via an appropriately designed test. The factors which impact program testability and test coverage include: (i) program complexity, (ii) software development philosophy or approach, (iii) software tools employed, (iv) test quality, and (v) test effectiveness. It is necessary to provide background for addressing these issues, and create the framework for an enhanced software reliability model capable of estimating accurately the reliability of software products utilizing appropriately obtained test data.

2.1 Existing Notions of Coverage

The importance of test coverage has been recognized by several researchers [7, 9], and empirical evidence strongly suggests that testing which is carried out without some form of test coverage measurement may fail to sensitize as much as 45% of the code. Software designers and testers must develop effective performance evaluation tools capable of measuring test coverage and pointing out design deficiencies contributing to poor software testability. Such tools should also provide data which lead to improvements in the quality and effectiveness of a software test as well as information about its adequacy when deciding product-release times.

There are two types of coverage definitions in literature: control-flow and data-flow coverage[2, 7, 8, 19]. Each coverage criterion proposed in the literature captures some important aspect of a program’s structure. In general, test coverage is a measure of how well a test covers all the potential fault-sites in a software product under test. It should be obvious that how one defines a potential fault-site and how well such fault-sites are sensitized influence greatly the significance of this important metric. Potential fault-sites are introduced here to mean program entities representing either structural or functional program elements whose sensitization is deemed essential towards establishing the operational integrity of the software product. Specific instances of potential fault-sites range from source-language and machine-code statements to high-level specification language statements and program blocks with I/O specification and functional description, only.

The first important step is taken here towards offering a unifying definition for test coverage which accommodates all the proposed specializations of the concept (i.e., statement coverage, block coverage, decision coverage, condition/decision coverage, etc.) without the burden of the specificity they impose on the modeling process used to estimate or predict the reliability of software products.
Definition 2.1: (Test Coverage): Given a software product and its companion test set, one defines test coverage to be the ratio of the number of potential fault-sites sensitized by the test divided by the total number of potential fault-sites under consideration.

As mentioned earlier, there are several definitions of test coverage but the one offered here is most general and easily adaptable to situations which may benefit from a specialized application of the concept. The definition allows also for the possibility of defective coverage; that is, the possibility of having a subset of non-sensitizable potential fault-sites.

In Section 3, a framework is developed which will allow us to relate test coverage to program testability and test length. This is a necessary step towards the ultimate objective, namely, that of incorporating test coverage into reliability modeling.

2.2 Existing Software Reliability Models

In this paper, the position is taken that all reliability models[12] which are widely used for software quality assessment share the Markov property[18]. Software faults are known to display the behavior of a Non-homogeneous Poisson Process (NHPP) in which the parameter of the stochastic process, \( \lambda(t) \), is time-dependent. The function \( \lambda(t) \) denotes the instantaneous failure intensity. The Markov chain for the NHPP model is given in Figure 1, where \( N(t) \) is the cumulative number of software faults detected by time \( t \).

Given \( \lambda(t) \), the mean value function \( m(t) = E[N(t)] \) satisfies the relation,

\[
m(t) = \int_0^t \lambda(s)ds
\]

and,

\[
\frac{dm(t)}{dt} = \lambda(t).
\]

(2)

\( N(t) \) as defined follows a Poisson distribution with parameter \( m(t) \), that is, the probability that \( N(t) \) is a given positive integer \( n \) is expressed as

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

(3)

All the time domain models which assume the failure process to be a NHPP differ in the approach they use for determining \( \lambda(t) \) or \( m(t) \).

2.2.1 The GO Model - A brief description

Although NHPP models have been studied for quite some time, the GO[5] model has had a strong influence on software reliability modeling. It is a very simple model whose parameters
have a physical interpretation, and it can easily accept failure data in the form of times between failures or as number of failures in given time intervals. Also, applicability of the model over a broad class of projects is another factor in its favor.

The basic assumptions made by the GO model are:

- The cumulative number of faults detected by time $t$ is Poisson distributed.
- All faults are independent and have the same chance of being detected.
- All detected faults are removed immediately and no new faults are introduced.

Specifically, the GO model states that the failure process is modeled by NHPP with a mean value function $m(t)$ given by

$$m(t) = a(1 - e^{-bt}), \quad a \geq 0, b \geq 0$$

(4)

where $a$ is the expected number of faults that would be observed by the testing process given infinite testing time, and $b$ is interpreted as failure occurrence rate per fault.

The failure intensity function $\lambda(t)$ is given as

$$\lambda(t) = a e^{-bt}.$$  

(5)

If the last failure occurred at time $s$, the reliability function is given by

$$R(t|s) = e^{-\int_s^{t+s} \lambda(\tau) d\tau} = e^{-a(e^{-bs} - e^{-bt})}$$

(6)

where $t$ is the time measured from the occurrence of the last failure event. It is important to note that the GO model does not make any predictions concerning the operational phase of the product, and this deficiency has been one of the main criticisms of the model.

3 An Enhanced NHPP Model

Prior to embarking upon the development of the enhanced NHPP model, we restate our goal: the objective is to define a metric which captures software testability and incorporate it into software reliability modeling. We show:

1. how testability can be reflected in test coverage, and

2. how time-dependent test coverage can be incorporated in the software reliability model under consideration.

3.1 Testability and Test Coverage

It is well-known among hardware reliability engineers that coverage (that is, fault-coverage) is an extremely important parameter in estimating system metrics such as reliability, availability and safety. It is also intuitively obvious that the quality of the tests generated for the purpose of evaluating a software product depends greatly on the testability characteristics of the software under test. Therefore, software testability and test coverage must be quantified and analytically incorporated into software reliability models for more accurate estimation of software product reliability and release-times. The previously given definition of potential fault-site allows for
the introduction into the area of software reliability a number of well-developed analytical
techniques and notions which have been successfully applied to reliability studies in hardware
systems.

A program has a finite number of potential fault-sites. Any potential fault-site has a prob-
ability of sensitization of \(x\). The fraction of potential fault-sites [3] have a probability of sensi-
tization equal to \(x\) is denoted by \(f(x)\) in the continuous case and by \(p(x)\) in the discrete case.
The continuous function \(f(x)\) satisfies the relation

\[
\int_{0}^{1} f(x) \, dx = 1
\]  

and similarly, the discrete function, \(p(x)\), satisfies the relation

\[
\sum_{x} p(x) = \sum_{k=1}^{m} p_k = 1
\]  

where \(p_k\) denotes the fraction of all potential fault-sites sensitizable with a probability between
\(x_k\) and \(x_{k+1}\).

Figure 2 shows a histogram for all the potential fault-sites. Figures 3 and 4 denote the
respective discrete and continuous functions for the fraction of potential fault-sites sensitized
as deduced from Figure 2.

If one assumes that each test vector is an independent Bernoulli trial, then one can assign
the probability that any potential fault-site will be sensitized on a test to be \(x\). The probability
that a potential fault-site will be sensitized on the \(n^{th}\) test is

\[
(1 - x)^{n-1} x.
\]  

Using expression (9), one can derive the equations (10) and (11) which give the expected
fraction of all potential fault-sites sensitized after \(n\) tests for the continuous and discrete cases,
respectively.

\[
E[X] = c_n = \int_{0}^{1} \sum_{j=1}^{n} f(x)(1 - x)^{j-1} x \, dx
\]

\[
E[X] = c_n = \sum_{k=1}^{m} \sum_{j=1}^{n} p_k(1 - x_k)^{j-1} x_k
\]

These equations will be the working definitions for coverage achieved after \(n\) tests, and they are
reducible to the forms given by Equations (12) and (13), respectively.

\[
c_n = 1 - \int_{0}^{1} (1 - x)^{n} f(x) \, dx
\]

\[
c_n = 1 - \sum_{k=1}^{m} (1 - x_k)^{n} p_k
\]

Unfortunately, it may not be always possible to sensitise all potential fault-sites, and the
effect on coverage will be an impulse of magnitude, \(p_0\), at origin of \(f(x)\); or a mass, \(p_0\), of
$p(x = 0)$ as shown in Equations (15) and (16). One can consider $p_0$ to be the defect in $c_n$ at infinity, so that

$$\lim_{n \to \infty} c_n = 1 - p_0.$$  

(14)

In view of expression (14), Equations (12) and (13) are rewritten as follows:

$$c_n = (1 - p_0) - \sum_{k=1}^{m} p_k (1 - x_k)^n$$  

(15)

$$c_n = (1 - p_0) - \int_0^1 (1 - x)^n f(x) dx$$  

(16)

“Defective or imperfect coverage” is defined as the inability to sensitize all potential fault-sites through an infinite number of tests. Figures 3 and 4 show discrete and continuous functions with defective coverage. The mass at origin, $p_0$, in $p(x)$ or $f(x)$ denotes the fraction of potential faults which cannot be sensitized. This mass at origin is shown in a defective $c(t)$ function as given in Figure 5.

These fundamental relationships which have been successfully applied to hardware systems can now be extended to software testing. Equation (10) relates test coverage, $c_n$, to testability, $f(x)$ or $p(x)$, and the length of the test, $n$. The function $f(x)$ or $p(x)$ can be obtained through the use of testability evaluation tools such as ATAC. Similarly, experimental data, such as those presented in Table 1, can be easily collected and utilized to obtain test coverage, $c(t)$, by associating coverage information given in column-2 with the cumulative execution time given in column-5.
Figure 3: Discrete function, \( p(x) \), with a mass at origin

Figure 4: Continuous function, \( f(x) \), with a mass at origin
Figure 5: A defective distribution function: $F(t) = (1 - p)F^C(t)$

Table 1: Experimental Data

<table>
<thead>
<tr>
<th>Test Number</th>
<th>Mean Coverage</th>
<th>Clock time</th>
<th>Execution time</th>
<th>Cumulative execution time</th>
<th>Failure occurrence (y/n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$e_1$</td>
<td>$T_1$</td>
<td>$t_1$</td>
<td>$t_1$</td>
<td>n (no)</td>
</tr>
<tr>
<td>2</td>
<td>$e_2$</td>
<td>$T_2$</td>
<td>$t_2$</td>
<td>$t_1 + t_2$</td>
<td>y (yes)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>i</td>
<td>$e_i$</td>
<td>$T_i$</td>
<td>$t_i$</td>
<td>$\sum_{j=1}^{i} t_j$</td>
<td>n</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>n</td>
</tr>
<tr>
<td>n</td>
<td>$e_n$</td>
<td>$T_n$</td>
<td>$t_n$</td>
<td>$\sum_{j=1}^{n} t_j$</td>
<td>y</td>
</tr>
</tbody>
</table>
3.2 Relating Test Coverage to Software Reliability

The reliability model proposed here states that the rate at which faults are removed is proportional to the rate at which potential fault-sites are covered. This assumption is in contrast to the GO model which assumes that the fault removal rate is proportional only to the number of remaining faults in the software. The following basic assumptions are made for the Enhanced NHPP model:

- faults are uniformly distributed,
- when a potential fault-site is sensitized, any fault present at that site is detected with probability \( c_d(t) \), and
- repairs are effected instantly and without introduction of new faults.

Analytically, the model is based on the expression

\[
\frac{dm(t)}{dt} = ac_d(t)\frac{dc(t)}{dt}.
\]

or

\[
m(t) = a \int_0^t c_d(\tau) * d(\tau)d\tau
\]

where \( a \) is defined as the total number of faults which are expected to be observed given infinite testing time, perfect fault detection coverage \( (c_d(t) = 1) \) and perfect test coverage \( (c(\infty) = 1) \).

The expected number of faults detected by time \( t \) is \( m(t) \). If one assumes \( c_d(\tau) \) to be a constant value \( K \), then

\[
m(t) = aK c(t).
\]

Equation (19) is intuitively simple: the expected number of faults one should find by time \( t \) is equal to the total number of faults in the product times the probability of detecting a fault times the percent coverage gained by time \( t \).

This results in a failure intensity function \( \lambda(t) \) and a reliability expression \( R(t|s) \) as follows:

\[
\lambda(t) = \frac{dm(t)}{dt} = aK c'(t)
\]

\[
R(t|s) = e^{-\int_s^{t+1} \lambda(\tau)d\tau} = e^{-aK[c(s+t) - c(s)]}
\]

where \( s \) is the time of the last failure and \( t \) is the time measured from the last failure. Although equations (19) through (21) were developed for use during the development test phase of the software product, these equations can easily be extended to the operational phase with minor modifications.

4 Predictions in the Operational Phase

It has been widely stated that the operational profile of a software product is radically different from the profile used during product testing[10, 11]. In addition to establishing the link between software reliability and test coverage, the new enhanced NHPP model can make accurate predictions of the operational behavior of a software product.

As one transitions from the development test phase into the operational phase, it becomes essential to provide estimates of key parameters needed in making accurate predictions during
the operational phase. To facilitate this process, one must adjust the notions of test coverage, \(c(t)\), and fault detection, \(c_d(t)\), to reflect the user’s operational environment and specialized needs.

Towards this end, one must first compute the number of remaining faults at the end of the development test phase. Assuming either defective coverage function, \(c(t)\), or defective fault detection, \(c_d(t)\), or both, the faults remaining at the end of the development test phase, \(a_H\), are expressed as follows:

\[
a_H = a - m_T(\infty) = a - a \times K_T \times (1 - p_{0\tau}) = a(1 - (K_T \times (1 - p_{0\tau}))
\]

(22)

where \(m_T(\infty)\) is the number of faults found given infinite test time, \(K_T\) is a constant fault detection probability and \(p_{0\tau}\) is the defect in test coverage all taken during the development test phase. The quantity \(a_H\) is known as Heisen bugs[6], or faults that will never be found even after infinite testing time. Using Equations (19) and (22), one obtains the mean number of faults detected by time \(t\) during the operational phase as

\[
m_H(t) = a \times (1 - K_T \times (1 - p_{0\tau})) \times K_L \times c_L(t)
\]

(23)

where \(K_L\) is the fault detection probability and \(c_L(t)\) the coverage during the operational phase. \(K_L\) and \(c_L(t)\) capture the effects of the operational profile[10, 11, 13] of the software product.

From Equation (23), one has

\[
\frac{d n_H(t)}{dt} = \lambda_H(t) = a \times (1 - K_T \times (1 - p_{0\tau})) \times K_L \times c'_L(t)
\]

(24)

Release-time values must be incorporated into equations (23) and (24) in order to obtain their proper forms for use in the operational phase. Noting that \(m_T(t_r)\) represents the number of faults removed during the development test phase at release time \(t_r\), one has

\[
m_L(t) = (a - m_T(t_r)) \times K_L \times c_L(t) = a \times (1 - (K_T \times c_T(t_r))) \times K_L \times c_L(t)
\]

(25)

\[
\lambda_L(t) = (a - m_T(t_r)) \times K_L \times c'_L(t) = a \times (1 - (K_T \times c_T(t_r))) \times K_L \times c'_L(t).
\]

(26)

It is obvious that during development test phase, one would like \(K_T\) and \(c_T(t_r)\) to be as high as possible.

The quantity \(a - m_T(t_r) - m_L(\infty)\) denotes the number of faults which remain in the software product after infinite time of operational use. These faults may be the result of defective test coverage or of imperfect fault detection. Conditioning on the faults that manifest during system operation, one may derive the corresponding reliability function[16] for the operational phase to be:

\[
R_L^c(t) = \frac{e^{-m_L(t)} - e^{-m_L(\infty)}}{1 - e^{-m_L(\infty)}}.
\]

(27)

Predictions about the reliability, failure intensity and availability all taken during the operational phase will now consider the operational profile and the effect of the testing process of the software product. Equation (27) will be used to compute Software Availability as described in the sequel. In a similar manner, one can extend the above ideas to any two sequential test phases by making minor modifications to Equations (25) and (26).
5 Software Availability

In many real-time applications, such as telecommunications software, software availability is a more critical system metric than reliability\[4\]. The expression for software availability, \( A \), during the operational phase, is as follows:

\[ A = \frac{MTTF}{MTTF + MTTR} \]  

where \( MTTF \) is the mean time to failure and \( MTTR \) is the mean time to repair. In this case, faults occurring during operation are not being fixed either because they are too expensive to do so or because they are not easily reproducible to initiate an effective repair. Faults continue to reside in the software, and there is no opportunity for availability growth. \( MTTR \) is given as the mean time to reboot the system.

\( MTTF \) is given by:

\[ MTTF = \int_0^\infty R_L^c(t)dt = \int_0^\infty \frac{e^{-m_L(t)} - e^{-m_L(\infty)}}{1 - e^{-m_L(\infty)}}dt. \]  

Assuming \( c_L(t) = 1 - e^{-\beta_L t} \), one can derive the expression for \( MTTF \) as follows:

\[ MTTF = \left( \frac{e^{-a_T K_L}}{gL(1 - e^{-a_T K_L})} \right) \sum_{i=1}^{\infty} \frac{(a_L K_L)^i}{i!} \]  

6 Impact on Test Adequacy

The basic question while testing software is to decide when to stop testing. The software release problem is one of the most important issues addressed by software reliability models.

6.1 Software Release Times

The determination of software release times is typically an optimization problem. This problem is usually solved either in terms of availability, or reliability or cost of testing. There are at least six test stopping criteria which can influence product-release time:

- **Stopping rule using the number of remaining faults.** This rule normally is used in the development test phase. If testing is to stop when a fraction \( \rho \) of all detectable faults are removed, Equation (19) yields

\[ c_T(t) = \frac{\rho}{K_T}. \]  

where \( \rho = \frac{m_T(t)}{a_T} \), and \( K_T \) is the constant fault detection probability during the development test phase.

- **Stopping rule using the failure intensity requirements.** This rule applies with appropriate modification in both development test and operational phases. If testing is to stop when the failure intensity reaches a specified value \( \lambda_f \), then the release time \( t_f \) can be determined from Equation (20) which yields,

\[ c_T(t) = \frac{\lambda_f t_f}{a K_T} + C_0 \]  

where \( C_0 = c_T(0) \).
An alternative approach to obtaining $t_f$ is through the use of operational phase parameters. Using Equations (26), one obtains,

$$c_T(t_f) = \frac{1}{K_T} - \frac{\lambda_L(0)}{aK_TK_Lc'_L(0)}$$

where $\lambda_L(0)$ denotes the failure intensity at the start of the operational phase.

- **Stopping rule using reliability requirements.** This rule is used in product development utilizing operational phase parameters. If the required conditional reliability is $R_r$ at time $t_0$ after product release, then the release time $t_r$ can be determined, using Equation (27) as

$$c_T(t_r) = \frac{\ln(1 - (1 - e^{-m_L(\infty)})(1 - R_r)) + aK_Lc_L(t_0)}{aK_TK_Lc_L(t_0)}$$

- **Stopping rule using cost requirement.** Following the cost model given in GO[5], one can derive the optimal release time $t_r$ from parameters $d_1$, $d_2$, and $d_3$, where $d_1$ is the expected cost of removing a fault during testing; $d_2$ is the expected cost of removing a fault during operation($d_2 > d_1$); and $d_3$ is the expected cost of software testing per unit time. The total cost equation for testing is:

$$d_1 * m_T(t_r) + d_2 * m_L(\infty) + d_3 * t_r = TC$$

which can be expressed as:

$$d_1(a * K_T * c_T(t_r)) + d_2(a * (1 - K_T * c_T(t_r)) * K_L * (1 - p_0)) + d_3 * t_r = TC.$$  

(36)

From Equation (36), one obtains the time $t_r$ which minimizes the cost of testing, $TC$. The result is given as Equation (37)

$$c_T(t_r) = \frac{d_3 * t_r}{a * K_T * (d_2 * K_L * (1 - p_0) - d_1)} + C_0$$

where $C_0 = c_T(0)$.

- **Stopping rule based on availability.** Assuming that MTTR is the time needed to reboot and there is no availability growth, one can derive a stopping rule based on availability solving numerically equation (38) for $a_L$ and substituting into Equation (39).

$$\left(\frac{e^{-a_LK_L}}{g_L(1 - e^{-a_LK_L})}\right) \sum_{i=1}^{\infty} \frac{(a_LK_L)^i}{i * i!} = \frac{A_r * MTTR * g_L}{1 - A_r}$$

(38)

$$c_T(t_a) = \frac{a - a_L}{aK_T}$$

(39)

### 7 Parameter Estimation

Software failure and coverage data can be collected and organized similar to that of Table 1. There are two options in determining the coverage function $c(t)$. 


Table 2: Failure and coverage data

<table>
<thead>
<tr>
<th>Start of Time Interval</th>
<th>End of Time Interval</th>
<th>Number of Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_0 )</td>
<td>( s_1 )</td>
<td>( n_1 )</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>( s_2 )</td>
<td>( n_2 )</td>
</tr>
<tr>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td>( s_{k-1} )</td>
<td>( s_k )</td>
<td>( n_k )</td>
</tr>
</tbody>
</table>

7.1 Option 1

One can hypothesize that \( c(t) \) has a specific form of an S-shaped curve

\[
   c(t) = [1 - (1 + gt)e^{-nt}].
\]

(40)

One can express \( m(t) \) as follows:

\[
   m(t) = aK[1 - (1 + gt)e^{-nt}].
\]

(41)

This equation has two unknown parameters, \( a \) and \( g \). Maximum likelihood estimation and data from Table 2, which can be abstracted from Table 1, is used to determine values for the unknown parameters.

The likelihood function for a NHPP model with mean value function \( m(t) \) is given by:

\[
   L(n_1, n_2, \ldots, n_k) = \prod_{i=1}^{k} \frac{[m(s_i) - m(s_{i-1})]^{n_i} e^{m(s_{i-1}) - m(s_i)}}{n_i!}
\]

(42)

where \( n_i \) is the number of faults detected in the time interval \( [s_{i-1}, s_i] \) and \( 0 = s_0 < s_1 < s_2 < \ldots < s_k \); and \( s_i \) represents running times since software testing has begun in a single test phase.

Upon substituting in Equation (42), one can obtain the following expression:

\[
   \ln(L) = n \ln(aK) + \left( \sum_{i=1}^{k} n_i \ln \left( (1 + g s_{i-1}) e^{-g s_{i-1}} - (1 + g s_i) e^{-g s_i} \right) \right)
   + aK \left( (1 + g s_k) e^{-g s_k} - 1 \right) - \left( \sum_{i=1}^{k} \ln(n_i!) \right)
\]

(43)

After obtaining expressions for the partial derivative of \( \ln L \) with respect to \( a \) and \( g \) and setting them equal to zero, one obtains Equations (44) and (45) which may be solved numerically for \( a \) and \( g 

\[
   a = \frac{n}{K \ast (1 - (1 + g s_k)e^{-g s_k})}
\]

(44)
0 = \left( \sum_{i=1}^{k} \left( s_{i-1} e^{(-g s_{i-1})} - (1 + g s_{i-1}) s_{i-1} e^{(-g s_{i-1})} - s_i e^{(-g s_i)} + (1 + g s_{i}) s_i e^{(-g s_i)} \right) / \left((1 + g s_{i-1}) e^{(-g s_{i-1})} - (1 + g s_i) e^{(-g s_i)} \right) \right) + K a \left( s_k e^{(-g s_k)} - (1 + g s_k) s_k e^{(-g s_k)} \right) \tag{45}

7.2 Option 2

This option makes use of the failure and coverage data presented in Figure 1 and Table 1 to determine the coverage function \( c(t) \) and the function \( f(x) \). Knowledge of these two functions allows determination of the function \( m(t) \) as in Option 1 except that \( m(t) \) involves only one unknown parameter to be estimated, \( a \).

7.3 Example Using the Model

In this example, the shape of coverage function \( c(t) \) has been hypothesized to be an S-shaped curve for computational convenience. This choice is based on the intuitive notion that initially the testing team is not familiar with the software, hence fault removal is slow, but after a certain time, the team gains sufficient experience and knowledge about the behavior of the product under test which leads to higher rates of fault removal until a time is reached when a large number of faults have been detected and removed thus becoming increasingly more difficult to remove new ones[21]. Using the data in Table 1 and the maximum likelihood technique, as explained in Option 1, one can estimate the parameters \( a \) and \( g \). Assuming the outcome of such a computation to be: \( a = 150 \) and \( g = 0.04 \), one can next evaluate the mean value.
function $m(t)$. Figures 6 through 8 give the mean number of faults, $m(t)$, the failure intensity, $\lambda(t)$, and the conditional reliability, $R(t|s)$, for the enhanced NHPP model respectively. These predictions are made for a single test phase.

8 Conclusions

In this paper, the dependence of software reliability on test coverage and software testability has been established. The enhanced NHPP model is based on an experimentally derived time-dependent test coverage function, $c(t)$. The model will handle defects in the test coverage and in fault detection. In addition to the link to test coverage, this model will allow explicitly for predictions into the operational phase of a software product. Equations for reliability and availability were generated using test and operational profiles. Stopping rules for optimal software release times are determined subject to constraints such as operational availability, cost and operational reliability.

Research currently underway will extend the model to cover the cases of:

- multiple fault types,
- repair with the possibility of introducing multiple new faults, and
- nonuniform distribution of faults over all potential fault sites.

The models developed will be validated using experimental data obtained from real software projects.

References

Figure 8:


