Software Reliability Models and Their Applications: A Survey

Shigeru Yamada†

† Department of Social System Engineering
Faculty of Engineering
Tottori University
Tottori-shi
680-8552, Japan
E-mail: yamada@sse.tottori-u.ac.jp
Preface

Software reliability is one of the most important characteristics of software quality. Its measurement and management technologies during the software life-cycle are essential to produce and maintain quality/reliable software systems. In this paper, we discuss software reliability modeling and its applications. As to software reliability modeling, hazard rate and NHPP models are investigated particularly for quantitative software reliability assessment. Further, imperfect debugging and software availability models are also discussed for incorporating practical factors of dynamic software behavior. And three software management problems are discussed as an application technology of software reliability models: Optimal software release problem, statistical testing-progress control, and optimal testing-effort allocation problem.

Key words
software reliability measurement and assessment, reliability growth model, imperfect debugging model, availability model, optimal release problem, testing-progress control, optimal testing-effort allocation problem
## Contents

1 Introduction 58

2 Definitions and Software Reliability Model 58

3 Software Reliability Growth Modeling 60

4 Imperfect Debugging Modeling 64
   4.1 Imperfect debugging model with perfect correction rate 64
   4.2 Imperfect debugging model for introduced faults 65

5 Software Availability Modeling 66
   5.1 Model description 66
   5.2 Software availability measures 67

6 Application of Software Reliability Assessment 68
   6.1 Optimal software release problem 68
      6.1.1 Maintenance cost model 69
      6.1.2 Maintenance cost model with reliability requirement 70
   6.2 Statistical software testing-progress control 71
   6.3 Optimal testing-effort allocation problem 72
1 Introduction

In recent years, many computer system failures have been caused by software faults which were introduced during the software development process. This is an inevitable problem since a software system installed in the computer system is an intellectual product consisting of documents and source programs developed by human activities. Then, total quality management (TQM) is considered to be one of the key technologies to produce more highly quality software products (see [1] and [2]). In case of TQM for software development, all phases of the development process, i.e. requirement specification, design, coding, and testing, have to be controlled systematically to prevent software bug- or fault-introduction as much as possible and to detect the introduced faults in the software system as early as possible. Basically, the concept of TQM means to assure the quality of the products in each phase to the next phase. Particularly, quality control carried out at the testing phase which is the last stage of the software development process is very important. During the testing phase, the product quality and the software performance during the operation phase are evaluated and assured. Concretely, a lot of software faults introduced in the software system through the first three phases of the development process by human activities are detected, corrected, and removed. Figure 1 shows a general software development process called a water-fall paradigm.

Therefore, TQM for software development, i.e. software TQM, has been emphasized. Software TQM aims to manage the software life-cycle comprehensively, considering productivity, quality, cost and delivery simultaneously. In particular, the management technologies for improving software reliability are very important. The quality characteristic of software reliability is that computer systems can continue to operate regularly without the occurrence of failures on software systems.

In this paper, we discuss a quantitative technique for software quality/reliability measurement and assessment as one of the key software reliability technologies, which is a so-called software reliability model, and its applications.

2 Definitions and Software Reliability Model

Generally, a software failure caused by software faults latent in the system cannot occur except for a special occasion when a set of special data is put into the system under a special condition, i.e. the program path including software faults is executed. Therefore, the software reliability is dependent on the input data and the internal condition of the program. We summarize the definitions of the technical terms related to the software reliability in the following.

A software system is a product which consists of the programs and documents produced through the software development process discussed in previous chapter (see Figure 1). Specification derived by analyzing user requirements for the software system is a document which describes the expected performance of the system. When the software performance deviates from the specification and output variable has an improper

Fig.1 : A general software development process (water-fall paradigm).
value or the normal processing is interrupted, it is said that a software failure occurs. That is software failure is defined as an unacceptable departure of program operation from the program requirements. The cause of software failure is called a software fault. Then, software fault is defined as a defect in the program which causes a software failure. The software fault is usually called a software bug. Software error is defined as human action that results in the software system containing a software fault (see [3] and [4]). Thus, the software fault is considered to be a manifestation of software errors.

Based on the basic definitions above, we can describe a software behavior as Input(I)-Program(P)-Output(O) model [5], [6] as shown in Figure 2.

\[ T : \text{Input subspace of } I, \text{ verified and validated by testing (Testing-Domain)} \]
\[ E : \text{Input subspace mapped into the output subspace of } O' \text{ by software faults} \]
\[ O' : \text{Output subspace of } O, \text{ constituting events of software failures caused by software faults latent in program } P \]

Fig.2 : An Input-Program-Output model for software behavior.

In this model a program is considered as a mapping from the input space constituting input data available on use to the output space constituting output data or interruptions of normal processing. Testing space T is an input subspace of I, of which the performance can be verified and validated by software testing. Software faults detected and removed during the testing phase map the elements of input subspaces E into an output subspace O' constituting the events of a software failure. That is, the faults detected during the testing phase belong to the intersection of subspace E and T. Software faults remaining in the operation phase belong to the subspace E but not to the testing space T.

Under the definitions for technical terms above, software reliability is defined as the attribute that a software system will perform without causing software failures over a given time period under specified conditions, and is measured by its probability (see [3] and [4]). A software reliability model is a mathematical analysis model for the purpose of measuring and assessing software quality/reliability quantitatively. Many software reliability models have been proposed and applied to practical use because software reliability is considered to be a “must-be quality” characteristic of a software product. The software reliability models can be divided into two classes [6], [7]. One treats the upper software development process, i.e. design and coding phases, and analyze the reliability factors of the software products and processes. The other deals with testing and operation phases by describing a software failure-occurrence phenomenon or software fault-detection phenomenon, by applying the stochastic/statistics theories, and can estimate and predict the software reliability.

In the former class, a software complexity model or a static software reliability model is well-known and can measure the reliability by assessing the complexity based the structural characteristics of products and the process features to produce the products. In the latter class, a software reliability growth model is especially...
well-known. Further, this model is divided into three categories [6], [7]:

1. **software failure-occurrence time model**
   The model which is based on the software failure-occurrence time or the software fault-detection time.

2. **software fault-detection count model**
   The model which is based on the number of software failure-occurrences or the number of detected faults.

3. **software availability model**
   The model which describes the time-dependent behavior of software system alternating up (operation) and down (fault correction) states.

The software reliability growth models are utilized for assessing the degree of achievement of software quality, deciding the time to software release for operational use, and evaluating the maintenance cost for faults undetected during the testing phase. We discuss the software reliability growth models and their applications in the following.

3 Software Reliability Growth Modeling

Generally, a mathematical model based on stochastic and statistics theories is useful to describe the software fault-detection phenomena or the software failure-occurrence phenomena and estimate the software reliability quantitatively. During testing phase in the software development process, software faults are detected and removed with a lot of testing-effort expenditures. Then, the number of faults remaining in the software system is decreasing as the testing goes on. This means that the probability of software failure-occurrence is decreasing so that the software reliability is increasing and the time-interval between software failures becomes longer with the testing time (see Figure 3).

![Software Reliability Growth Model](image)

**Fig.3 : Software reliability growth.**

A mathematical tool which describe software reliability aspect is a *software reliability growth model* [6], [8], [9].

Based on the definitions discussed in previous chapter, we can develop a software reliability growth model based on the assumptions for actual environments during the testing phase or the operation phase. Then, we
can define the following random variables on the number of detected faults and the software failure-occurrence time (see Figure 4):

\[ N(t) = \text{the cumulative number of software faults (or the cumulative number of observed software failures) detected up to time } t, \]
\[ S_i = \text{the } i\text{-th software failure-occurrence time (} i = 1, 2, \ldots; S_0 = 0), \]
\[ X_i = \text{the time-interval between } (i - 1)\text{-st and } i\text{-th software failures (} i = 1, 2, \ldots; X_0 = 0). \]

Figure 4 shows the occurrence of event \( \{N(t) = i\} \) since \( i \) faults have been detected up to time \( t \). From these definitions, we have

\[ S_i = \sum_{k=1}^{i} X_k, \quad X_i = S_i - S_{i-1}. \] 

(1)

Assuming that the hazard rate, i.e. the software failure rate, for \( X_i(i = 1, 2, \ldots) \), \( z_i(x) \), is proportional to the current number of residual faults remaining in the system, we have

\[ z_i(x) = (N - i + 1)\lambda(x) \quad (i = 1, 2, \ldots, N; x \geq 0, \lambda(x) > 0), \] 

(2)

where \( N \) is the initial fault content and \( \lambda(x) \) the software failure rate per fault remaining in the system at time \( x \). If we consider two special cases in eq. (2) as

\[ \lambda(x) = \phi \quad (\phi > 0), \] 
\[ \lambda(x) = \phi x^{m-1} (\phi > 0, m > 0), \] 

(3)
(4)

then two typical software hazard rate models respectively called Jelinski-Moranda model [10] and Wagoner model [11] can be derived where \( \phi \) and \( m \) are constant parameters. Usually, it is difficult to assume that a software system is completely fault-free or failure-free. Then, we have a software hazard rate model called Moranda model [12] for the case of the infinite number of software failure-occurrences as

\[ z_i(x) = Dk^{i-1} \quad (i = 1, 2, \ldots; D > 0, 0 < k < 1), \] 

(5)

where \( D \) is the initial software hazard rate and \( k \) the decreasing ratio. Eq. (5) describes a software failure-occurrence phenomenon where a software system has high frequency of software failure-occurrence during the early stage of the testing or the operation phase and it gradually decreases after then. Based on the software hazard rate models above, we can derive the software reliability function for \( X_i(i = 1, 2, \ldots) \) as

\[ R_i(x) = \exp \left[ -\int_0^x z_i(x) \, dx \right] \quad (i = 1, 2, \ldots). \] 

(6)
In the following, we discuss NHPP models [8], [13]–[15], which are modeled for random variable \( N(t) \) as typical software reliability growth models. In the NHPP models, a nonhomogeneous Poisson process (NHPP) is assumed for the random variable \( N(t) \) of which the distribution function is given by

\[
\Pr\{N(t) = n\} = \frac{\{H(t)\}^n}{n!} \exp[-H(t)] \quad (n = 1, 2, \ldots),
\]

where \( \Pr\{A\} \) means the probability of event \( A \). \( H(t) \) in eq. (7) is called a mean value function which indicates the expectation of \( N(t) \), i.e. the expected cumulative number of faults detected (or the expected cumulative number of software failures occurred) in the time-interval \((0, t]\), and \( h(t) \) in eq. (7) called an intensity function which indicates the instantaneous fault-detection rate at time \( t \).

From eq. (7), various software reliability assessment measures can be derived. For examples, the expected number of faults remaining in the system at time \( t \) is given by

\[
n(t) = a - H(t),
\]

where \( a = H(\infty) \), i.e. parameter \( a \) denotes the expected initial fault content in the software system. Given that the testing or the operation has been going on up to time \( t \), the probability that a software failure does not occur in the time-interval \((t, t + x) (x \geq 0)\) is given by conditional probability \( \Pr\{X_i > x|S_{i-1} = t\} \) as

\[
R(x|t) = \exp[H(t) - H(x + t)] \quad (t \geq 0, x \geq 0).
\]

(9)

\( R(x|t) \) in eq. (9) is a so-called software reliability. Measures of MTBF (mean time between software failures or fault-detections) can be obtained follows:

\[
\text{MTBF}_I(t) = \frac{1}{h(t)}, \\
\text{MTBF}_C(t) = \frac{t}{H(t)}.
\]

(10)

(11)

MTBF’s in eqs. (10) and (11) are called instantaneous and cumulative MTBF’s, respectively.

It is obvious that the lower the value of \( n(t) \) in eq. (8), the higher the value \( R(x|t) \) for specified \( x \) in eq. (9), or the longer the value MTBF’s in eqs. (10) and (11), the higher the achieved software reliability is. Then, analyzing actual test data with accepted NHPP models, these measures can be utilized to assess software reliability during the testing or operation phase, where statistical inferences, i.e. parameter estimation and goodness-of-fit test, are usually performed by a method of maximum-likelihood.

To assess the software reliability actually, it is necessary to specify the mean value function \( H(t) \) in eq. (7). Many NHPP models considering the various testing or operation environment for software reliability assessment have been proposed in the last decade. Typical NHPP models are summarized in Table 1. As discussed above, a software reliability growth is described as the relationship between the elapsed testing or operation time and the cumulative number of detected faults, and can be shown as the reliability growth curve mathematically (see Figure 5). Among the NHPP models in Table 1, exponential and modified exponential software reliability growth models are appropriate when the observed reliability growth curve shows an exponential curve ((A) in Figure 5). Similarly, delayed S-shaped and inflection S-shaped software reliability growth models are appropriate when it shows an S-shaped curve ((B) in Figure 5).

In addition, as for computer makers or software houses in Japan, logistic curve and Gompertz curve models have been often used as software quality assessment models by assuming that software fault-detection phenomena can be shown by S-shaped reliability growth curves [30], [31]. In these deterministic models, the cumulative number of faults detected up to testing \( t \) is formulated by the following growth equations:

\[
L(t) = \frac{k}{1 + me^{-\alpha t}} \quad (m > 0, \alpha > 0, k > 0),
\]

(12)

\[
G(t) = ka^{(t)} \quad (0 < a < 1, 0 < b < 1, k > 0).
\]

(13)

In eqs. (12) and (13), assuming that \( a \) convergence value of each curve \((L(\infty) \) or \( G(\infty))\), i.e. parameter \( k \), represents the initial fault content in the software system, it can be estimated by a regression analysis.
### Table 1: A summary of NHPP models.

<table>
<thead>
<tr>
<th>NHPP model</th>
<th>Mean Value Function $H(t)$</th>
<th>Intensity Function $h(t)$</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponential software reliability growth model [16],[17]</td>
<td>$m(t) = a(1 - e^{-bt})$  \hspace{0.5cm} ($a &gt; 0, b &gt; 0$)</td>
<td>$h_m(t) = ab\epsilon^{-bt}$</td>
<td>A software failure-occurrence phenomenon with a constant fault-detection rate at an arbitrary time is described.</td>
</tr>
<tr>
<td>Modified exponential software reliability growth model [19],[18]</td>
<td>$m_\alpha(t) = a \sum_{i=1}^{2} p_i(1 - e^{-b_1^i t})$  \hspace{0.5cm} ($a &gt; 0, 0 &lt; b_2 &lt; b_1 &lt; 1$, $\sum_{i=1}^{2} p_i = 1, 0 &lt; p_i &lt; 1$)</td>
<td>$h_\alpha(t) = a \sum_{i=1}^{2} p_i b_1 e^{-b_1^i t}$</td>
<td>A difficulty of software fault-detection during the testing is considered ($b_1$ is the fault-detection rate for easily detectable faults: $b_2$ is the fault-detection rate for hardly detectable faults).</td>
</tr>
<tr>
<td>Delayed S-shaped software reliability growth model [20],[21]</td>
<td>$M(t) = a[1 - (1 + bt)e^{-bt}]$ \hspace{0.5cm} ($a &gt; 0, b &gt; 0$)</td>
<td>$h_M(t) = ab^2 t e^{-bt}$</td>
<td>A software fault-detection process is described by successive two phenomena, i.e. failure-detection process and fault-isolation process.</td>
</tr>
<tr>
<td>Infection S-shaped software reliability growth model [22],[23]</td>
<td>$I(t) = \frac{a(1 - e^{-bt})}{1 + ce^{-bt}}$ \hspace{0.5cm} ($a &gt; 0, b &gt; 0, c &gt; 0$)</td>
<td>$h_I(t) = \frac{ab(1 + c)e^{-bt}}{(1 + ce^{-bt})^2}$</td>
<td>A software failure-occurrence phenomenon with mutual dependency of detected faults is described.</td>
</tr>
<tr>
<td>Testing-effort dependent software reliability growth model [24],[25]</td>
<td>$T(t) = a[1 - e^{-W(t)}]$ \hspace{0.5cm} ($0 &lt; r &lt; 1$, $a &gt; 0, \alpha &gt; 0, \beta &gt; 0, m &gt; 0$)</td>
<td>$h_r(t) = a\alpha\beta \times m t^{m-1} e^{-r W(t)}$</td>
<td>The time-dependent behavior of the amount of testing-effort and the cumulative number of detected faults is considered.</td>
</tr>
<tr>
<td>Testing-domain dependent software reliability growth model [26],[27]</td>
<td>$D(t) = a[1 - \frac{1}{\nu - b}(e^{-bt} - be^{-W(t)})]$ \hspace{0.5cm} ($\nu \neq b$)</td>
<td>$h_D(t) = \frac{ab\nu}{\nu - b}(e^{-bt} - e^{-\nu t})$</td>
<td>The testing-domain which is the set of software functions influenced by executed test-cases is considered.</td>
</tr>
<tr>
<td>Logarithmic Poisson execution time model [28],[29]</td>
<td>$\mu(t) = \frac{1}{\theta} \ln(\lambda_0 \theta t + 1)$ \hspace{0.5cm} ($\lambda_0 &gt; 0, \theta &gt; 0$)</td>
<td>$\lambda(t) = \frac{\lambda_0}{\lambda_0 \theta t + 1}$</td>
<td>When the testing or operation time is measured on the basis of the number of CPU hours, a exponentially decreasing software failure rate is considered with respect to the cumulative number of software failures.</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
    a &= \text{the expected number of initial fault content in the software system} \\
    b, b_i, \alpha &= \text{the parameter representing the fault-detection rate} \\
    p_i &= \text{the fault content ratio of Type } i \text{ fault} \\
    c &= \text{the parameter representing the inflection factor of test personnel} \\
    \alpha, \beta, m &= \text{the parameters which determine the testing-effort function } W(t) \\
    \nu &= \text{the testing-domain growth rate} \\
    \lambda_0 &= \text{the initial software failure rate} \\
    \theta &= \text{the reduction rate software failure rate}
\end{align*}
\]
4 Imperfect Debugging Modeling

Most software reliability growth models proposed so far are based on the assumption of perfect debugging, i.e. that all faults detected during the testing and operation phases are corrected and removed perfectly. However, debugging actions in real testing and operation environments are not always performed perfectly. For example, type misses invalidate the fault correction activity or fault removal is not carried out precisely due to incorrect analysis of test results (see [32]). It is interesting, therefore, to develop a software reliability growth model which assumes an imperfect debugging environment (cf. [33], [34]). Such an imperfect debugging model is expected to estimate reliability assessment measures more accurately.

4.1 Imperfect debugging model with perfect correction rate

To model an imperfect debugging environment, the followings are assumed:

1. Each fault which causes a software failure is corrected perfectly with probability $p (0 \leq p \leq 1)$. It is not corrected with probability $q (= 1 - p)$. We call $p$ the perfect debugging rate or the perfect correction rate.

2. The hazard rate is given by eq.(5), and geometrically decreases whenever each detected fault is corrected.

3. The probability that two or more software failures occur simultaneously is negligible.

4. No new faults are introduced during the debugging. At most one fault is removed when it is corrected and the correction time is not considered.

Let $X(t)$ be a random variable representing the cumulative number of faults corrected up to the testing time $t$. Then, $X(t)$ forms a Markov process [35]. That is, from assumption (1), when $i$ faults have been corrected by arbitrary testing time $t$,

$$X(t) = \begin{cases} i & \text{with probability } q \\ i + 1 & \text{with probability } p \end{cases}$$

(see Figure 6). Then, the one-step transition probability for the Markov process that after making a transition into state $i$, the process $\{X(t), t \geq 0\}$ makes a transition into state $j$ by time $t$ is given by

$$Q_{ij}(t) = p_{ij}(1 - \exp[-DK^i t]),$$

(15)

where $p_{ij}$ are the transition probabilities from state $i$ to state $j$ are given by

$$p_{ij} = \begin{cases} q & (i = j) \\ p & (j = i + 1) \\ 0 & \text{(elsewhere)} \end{cases}$$

(16)

\[65\]
Eq. (15) represents the probability that if $i$ faults have been corrected at time zero, $j$ faults are corrected by time $t$ after the next software failure occurs. Therefore, based on Markov analysis by using the assumptions and stochastic quantities above, we have the software reliability function and the mean time between software failures for $X_i(i = 1, 2, \ldots)$ as

$$R_i(x) = \sum_{s=0}^{i-1} \binom{i-1}{s} p_s q^{i-1-s} \exp[-Dk^sx], \quad (17)$$

$$E[X_i] = \int_0^{\infty} R_i(x) dx = (p/k + q)^{i-1}/D. \quad (18)$$

And if the initial fault content in the system, $N$, is specified, the expected cumulative number of faults debugged imperfectly up to time $t$ is given by

$$M(t) = \frac{q}{p} \sum_{n=1}^{N} \sum_{i=0}^{n-1} A_{i,n} (1 - \exp[-pDk^i t]), \quad (19)$$

where $A_{i,n}$ is

$$A_{0,1} = 1, \quad A_{i,n} = \frac{k^{(1/2)n(n-1)-i}}{\prod_{j=0}^{n-1} (k^j - k^i)} \quad (n = 2, 3, \ldots; i = 0, 1, 2, \ldots, n - 1). \quad (20)$$

### 4.2 Imperfect debugging model for introduced faults

Besides the imperfect debugging factor above in fault-correction activities, we consider the possibility of introducing new faults in the debugging process. It is assumed that there exist the following two kinds of software failures in the dynamic environment [36], [37], i.e. the testing or user operation phase:

- (F1) software failures caused by faults originally latent in the software system prior to the testing (which are called inherent faults),
- (F2) software failures caused by faults introduced during the software operation due to imperfect debugging.

In addition, it is assumed that one software failure is caused by one fault and that it is impossible to discriminate whether the fault that caused the occurred software failure is F1 or F2. As to the software failure-occurrence rate due to F1, the inherent faults are detected with the progress of the operation time. In order to consider two kinds of time dependencies on the decreases of F1, let $a_i(t)(i = 1, 2)$ denote the software failure-occurrence rate for F1. On the other hand, the software failure-occurrence rate due to F2 is denoted as constant $\lambda(\lambda > 0)$ since we assume that F2 occurs randomly throughout the operation. When we consider the software failure-occurrence phenomena due to F1 and F2 simultaneously, the software failure-occurrence rate at operation time $t$ is given by

$$h_i(t) = \lambda + a_i(t) \quad (i = 1, 2). \quad (21)$$

From eq. (21), the expected cumulative number of software failures in the time-interval $(0, t]$ (or the expected cumulative number of detected faults) is given by

$$H_i(t) = \lambda t + A_i(t)$$

$$A_i(t) = \int_0^t a_i(x) dx \quad (i = 1, 2). \quad (22)$$

Then, we have two imperfect debugging models based on an NHPP discussed in section 3, where $h_i(t)$ in eq. (21) and $H_i(t)$ in eq. (22) are used as the intensity functions and the mean value functions $(i = 1, 2)$ for
an NHPP, respectively. Especially, exponential and delayed S-shaped software reliability growth models are assumed for describing software failure-occurrence phenomena due to the inherent faults as (see Table 1)

\[ a_1(t) = ab e^{-bt} \quad (a > 0, b > 0), \]
\[ a_2(t) = ab^2 e^{-bt} \quad (a > 0, b > 0), \]

where \( a \) is the expected number of initially latent inherent faults and \( b \) the software failure-occurrence rate per inherent fault. Therefore, the mean value functions of NHPP models for the imperfect debugging factor are given by

\[ H_1(t) = \lambda t + a(1 - e^{-bt}), \]
\[ H_2(t) = \lambda t + a[1 - (1 + bt)e^{-bt}]. \]

From these imperfect debugging models we can derive several software reliability measures for the next software failure-occurrence time-interval \( X \) since current time \( t \) such as the software reliability function \( R_i(x|t) \), the software hazard rate \( z_i(x|t) \), and the mean time between software failures \( E_i[X|t] \) \((i = 1, 2):\)

\[ R_i(x|t) = \exp[H_i(t) - H_i(t + x)] \quad (t \geq 0, x \geq 0), \]
\[ z_i(x|t) = -\frac{d}{dx} \frac{R_i(x|t)}{R_i(x|t)} = h_i(t + x), \]
\[ E_i[X|t] = \int_0^\infty R_i(x|t)dx. \]

5 Software Availability Modeling

Recently, software performance measures such as the possible utilization factors begin to be taken an interest as well as the hardware products. That is, it is very important to measure and assess software availability, which is defined as the probability that the software system is performing successfully, according to the specification, at a specified time point [38]-[40]. Several stochastic models have been proposed so far for software availability measurement and assessment. [41] has proposed a software availability model considering a reliability growth process, taking notice of the cumulative number of corrected faults. [42]-[44] have constructed the software availability models describing the uncertainty of fault removal. [45] and [46] have incorporated the increase of the difficulty of fault removal.

The actual operational environments need to be more reflected in software availability modeling since software availability is a customer-oriented metrics. [46] and [47] have developed a plausible model, assuming that there exist two types of software failures occurring during the operation phase. Furthermore, [48] has built an operational software availability model from the viewpoint of restoration scenarios.

The above models have employed Markov processes for describing the stochastic time-dependent behaviors of the systems which alternate between the operating regularly and the restoration state (down state) that a system is inoperable [49]. Several stochastic metrics for software availability measurement in dynamic environment are derived from the respective models.

We discuss a fundamental software availability model [44] in the following.

5.1 Model description

The following assumptions are made for software availability modeling:

(1) The software system is unavailable and starts to be restored as soon as a software failure occurs, and the system can not operate until the restoration action is complete.

(2) The restoration action implies the debugging activity, which is performed perfectly with probability \( a(0 < a \leq 1) \) and imperfectly with probability \( b(= 1 - a) \). We call \( a \) the perfect debugging rate. One fault is corrected and removed from the software system when the debugging activity is perfect.

(3) When \( n \) faults have been corrected, the time to the next software failure-occurrence and the restoration time follow exponential distributions with mean \( 1/\lambda_n \) and \( 1/\mu_n \), respectively.
(4) The probability that two or more software failures occur simultaneously is negligible.

Consider a stochastic process \( \{X(t), t \geq 0\} \) with the state space \((W, R)\) where up state vector \(W = \{W_n; n = 0, 1, 2, \ldots\}\) and down state vector \(R = \{R_n; n = 0, 1, 2, \ldots\}\). Then, the events \(\{X(t) = W_n\}\) and \(\{X(t) = R_n\}\) mean that the system is operating and inoperable, respectively, due to the restoration action at time \(t\), when \(n\) faults have already been corrected.

From the assumption (2), when the restoration action has been complete in \(\{X(t) = R_n\}\),

\[
X(t) = \begin{cases} 
W_n & \text{with probability } b \\
W_{n+1} & \text{with probability } a 
\end{cases}
\]  (30)

We use the Moranda model discussed in chapter 3 to describe the software failure-occurrence phenomenon, i.e. when \(n\) faults have been corrected, the software hazard rate \(\lambda_n\) is given by

\[
\lambda_n = Dk^n \quad (n = 0, 1, 2, \ldots; D > 0, 0 < k < 1).
\]  (31)

The expression of eq. (31) comes from the point of view that software reliability depends on the debugging efforts, not the residual fault content. We do not note how many faults remain in the software system.

Next, we describe the time-dependent behavior of the restoration action. The restoration action for software systems includes not only the data recovery and the program reload but also the debugging activities for manifested faults. From the viewpoint of the complexity, there are cases where the faults detected during the early stage of the testing or operation phase have low complexity and are easy to correct/remove, and as the testing is in progress, detected faults have higher complexity and are more difficult to correct/remove [8]. In the above case, it is appropriate that the mean restoration-time becomes longer with the increasing number of corrected faults. Accordingly, we express \(\mu_n\) as follows:

\[
\mu_n = Er^n \quad (n = 0, 1, 2, \ldots; E > 0, 0 < r \leq 1),
\]  (32)

where \(E\) and \(r\) are the initial restoration rate and the decreasing ratio of the restoration rate, respectively. In eq. (32) the case of \(r = 1\), i.e. \(\mu_n = E\), means that the complexity of each fault is random.

Let \(T_n\) and \(U_n(n = 0, 1, 2, \ldots)\) be the random variables representing the next software failure-occurrence and the next restoration time-intervals when \(n\) faults have been corrected, in other words the sojourn times in states \(W_n\) and \(R_n\), respectively. Furthermore, let \(Y(t)\) be the random variable representing the cumulative number of faults corrected up to time \(t\). The sample behavior of \(Y(t)\) is illustrated in Figure 7. It is noted that the cumulative number of corrected faults is not always coincident with that of software failures or restoration actions. The sample state transition diagram of \(X(t)\) is illustrated in Figure 8.

### 5.2 Software availability measures

We can obtain the state occupancy probabilities that the system is in states \(W_n\) and \(R_n\) at time point \(t\) as

\[
P_{W_n}(t) \equiv \Pr\{X(t) = W_n\} = \frac{g_{n+1}(t)}{a\lambda_n} + \frac{g_{n+1}(t)}{a\lambda_n \mu_n} \quad (n = 0, 1, 2, \ldots),
\]  (33)

\[
P_{R_n}(t) \equiv \Pr\{X(t) = R_n\} = \frac{g_{n+1}(t)}{a\mu_n} \quad (n = 0, 1, 2, \ldots),
\]  (34)

respectively, where \(g_n(t)\) is the probability density function of random variable \(S_n\), which denotes the first passage time to state \(W_n\), and \(g'_n(t) \equiv dg_n(t)/dt\). \(g_n(t)\) and \(g'_n(t)\) can be given analytically.

The following equation holds for arbitrary time \(t\):

\[
\sum_{n=0}^{\infty} [P_{W_n}(t) + P_{R_n}(t)] = 1.
\]  (35)
The instantaneous availability is defined as

\[ A(t) \equiv \sum_{n=0}^{\infty} P_{W_n}(t), \tag{36} \]

which represents the probability that the software system is operating at specified time point \( t \). Furthermore, the average software availability over \((0, t]\) is defined as

\[ A_{av}(t) \equiv \frac{1}{t} \int_{0}^{t} A(x)dx, \tag{37} \]

which represents the ratio of system’s operating time to the time-interval \((0, t]\). Using eqs. (33) and (34), we can express eqs. (36) and (37) as

\[ A(t) = \sum_{n=0}^{\infty} \left[ \frac{g_{n+1}(t)}{a \lambda_n} + \frac{g_{n+1}'(t)}{a \lambda_n \mu_n} \right], \tag{38} \]

\[ A_{av}(t) = \frac{1}{t} \sum_{n=0}^{\infty} \left[ \frac{G_{n+1}(t)}{a \lambda_n} + \frac{g_{n+1}(t)}{a \lambda_n \mu_n} \right], \tag{39} \]

respectively, where \( G_n(t) \) is the distribution function of \( S_n \).

6 Application of Software Reliability Assessment

It is very important to apply the results of software reliability assessment to management problems on software projects for attaining higher productivity and quality. We discuss three software management problems as application technologies of software reliability models.

6.1 Optimal software release problem

Recently, it is becoming increasingly difficult for the developers to produce highly-reliable software systems efficiently. Thus, it has been necessary to control a software development process in terms of quality, cost, and release time. In the last phase of software development process, the testing is carried out to detect and fix software faults introduced by human work, prior to its release for the operational use. The software faults that cannot be detected and fixed remain in the released software system after the testing phase. Thus, if a software failure occurs during the operational phase, then a computer system stops working and it may cause serious damage to our daily life.

If the length of software testing is long, we can remove many software faults in the system and its reliability increases. However, it leads to increase the testing cost and to delay software delivery. In contrast, if the length of software testing is short, a software system with low reliability is delivered and it includes many software faults which have not been removed in the testing phase. Thus, the maintenance cost during the operation phase increases.

So, it is very important in terms of software management that we solve for the optimal length of software testing, what is called an optimal release time. Such a decision problem is called an optimal software release problem \([50]-[57]\). These decision problems have been studied in the last decade by many researchers. We discuss optimal software release problems which consider both a present value and a warranty period (in the operational phase) during which the developer has to pay the cost for fixing any faults detected. Then it is very important with respect to software development management that we solve an optimal software testing time by integrating the total expected testing cost and the reliability requirement.
6.1.1 Maintenance cost model

The following notations are defined:

\begin{align*}
    c_0 &= \text{the cost for the minimum amount of testing which must be done}, \\
    c_t &= \text{the testing cost per unit time}, \\
    c_w &= \text{the maintenance cost per one fault during the warranty period}, \\
    T &= \text{the software release time, i.e. additional total testing time}, \\
    T^* &= \text{the optimum software release time}.
\end{align*}

We discuss a maintenance cost model for formulation of optimal release problem. The maintenance cost during the warranty period is considered. The concept of a present value is also introduced into the cost factors. Then, the total expected software maintenance cost \( WC(T) \) can be formulated as:

\[ WC(T) \equiv c_0 + c_t \int_0^T e^{-\alpha t} dt + C_w(T), \]  

(40)

where \( C_w(T) \) is the maintenance cost during the warranty period. The parameter \( \alpha \) in eq. (40) is a discount rate of the cost. When we apply an exponential software reliability growth model based on an NHPP with mean value function \( m(t) \) and intensity function \( h_m(t) \) discussed in section 3 (see Table 1), we discuss the following three cases in terms of the behavior of \( C_w(T) \) (see Figure 9):

(Case 1)

When the length of the warranty period is constant and the software reliability growth is not assumed to occur after the testing phase, \( C_w(T) \) is represented as:

\[ C_w(T) = c_w \int_T^{T+T_w} h_m(t) e^{-\alpha t} dt. \]  

(41)

(Case 2)

When the length of the warranty period is constant and the software reliability growth is assumed to occur even after the testing, \( C_w(T) \) is given by:

\[ C_w(T) = c_w \int_T^{T+T_w} h_m(t) e^{-\alpha t} dt. \]  

(42)

(Case 3)

When the length of the warranty period obeys a distribution function \( W(t) \) and the software reliability growth is assumed to occur even after the testing phase, \( C_w(T) \) is represented as:

\[ C_w(T) = c_w \int_0^\infty \int_T^{T+T_w} h_m(t) e^{-\alpha t} dt dW(T_w), \]  

(43)

where we assume that the distribution of the warranty period is a truncated normal distribution:

\[ \frac{dW(t)}{dt} = \frac{1}{A \sqrt{2\pi\sigma}} \exp\left[-(t - \mu)^2/(2\sigma^2)\right] \quad (t \geq 0, \mu > 0, \sigma > 0), \]  

(44)

\[ A = \frac{1}{\sqrt{2\pi\sigma}} \int_0^\infty \exp[-(t - \mu)^2/(2\sigma^2)] dt. \]  

(45)
Let us consider the optimal release policies for minimizing \( WC(T) \) in eq. (40) with respect to \( T \) of Case 1 which is a typical case for optimal software release problems. Substituting eq. (41) into eq. (40), we rewrite it as:

\[
WC(T) = c_0 + c_1 \int_0^T e^{-\alpha t} dt + c_w h_m(T) \int_T^{T+T_w} e^{-\alpha t} dt. \tag{46}
\]

Differentiating eq. (46) in terms of \( T \) and equating it to zero yields:

\[
h_m(t) = \frac{c_t}{c_w T_w(b + \alpha)}. \tag{47}
\]

Note that \( WC(T) \) is a convex function with respect to \( T \) because \( d^2WC(T)/dT^2 > 0 \). Thus, the equation \( dWC(T)/dT = 0 \) has only one finite solution when the condition \( h_m(0) > c_t/[c_w T_w(b + \alpha)] \) holds. The solution \( T_1 \) of eq. (47) and the optimum release time can be shown as follows:

\[
T^* = T_1 = \frac{1}{b} \ln \left[ \frac{abc_w T_w(b + \alpha)}{c_t} \right] \quad (0 < T_1 < \infty). \tag{48}
\]

When the condition \( h_m(0) < c_t/[c_w T_w(b + \alpha)] \) holds, \( WC(T) \) in eq. (46) is a monotonically increasing function in terms of the testing time \( T \). Then, the optimum release time \( T^* = 0 \). Therefore, we can obtain the optimal release policies as follows:

[Optimal Release Policy 1]

1. If \( h_m(0) > c_t/[c_w T_w(b + \alpha)] \), then the optimum release time is \( T^* = T_1 \).

2. If \( h_m(0) < c_t/[c_w T_w(b + \alpha)] \), then the optimum release time is \( T^* = 0 \).

Similarly we can obtain the optimal release policies for Case 2 and Case 3 (see [55] and [58]).

### 6.1.2 Maintenance cost model with reliability requirement

Next, we discuss the optimal release problem with the requirement of software reliability goal. In the actual software development, the manager must spend and control the testing resources as minimizing the total software cost and satisfying reliability requirement rather than only minimizing the cost. From the exponential software reliability growth model, the software reliability function can be defined as the probability that a software failure does not occur during the time-interval \((T, T + x)\) after the total testing time \( T \), i.e. the release time. The software reliability function is given as follows:

\[
R(x|T) = \exp[m(T) - m(T + x)]. \tag{49}
\]

From eq. (49), we derive the software reliability function as follows:

\[
R(x|T) = \exp[-e^{-bt} \cdot m(x)]. \tag{50}
\]

Let the software reliability objective be \( R_0(0 < R_0 \leq 1) \). We can evaluate optimum release time \( T = T^* \) which minimizes eq. (40) with satisfying the software reliability objective \( R_0 \). Thus, the optimal software release problem is formulated as follows:

\[
\text{minimize } WC(T) \quad \text{subject to } \quad R(x|T) \geq R_0. \tag{51}
\]

For the optimal release problem formulated by eq. (51), let \( T_R \) be the optimum release time with respect to \( T \) satisfying the relation \( R(x|T) = R_0 \) for specified \( x \). By applying the relation \( R(x|T) = R_0 \) into eq. (50), we can obtain the solution \( T_R \) as follows:

\[
T_R = \frac{1}{b} \left\{ \ln m(x) - \ln \left( \frac{1}{R_0} \right) \right\}. \tag{52}
\]

Then, we can derive the optimal release policies to minimize the total expected software maintenance cost and to satisfy the software reliability objective \( R_0 \).

For Case 1, the optimal release policies are given as follows:

[Optimal Release Policy 2]
(2.1) If \( h_m(0) > c_t/[c_w T_w(b + \alpha)] \) and \( R(x|0) < R_0 \), then the optimum release time is \( T^* = \max\{T_1, T_R\} \).

(2.2) If \( h_m(0) > c_t/[c_w T_w(b + \alpha)] \) and \( R(x|0) \geq R_0 \), then the optimum release time is \( T^* = T_1 \).

(2.3) If \( h_m(0) \leq c_t/[c_w T_w(b + \alpha)] \) and \( R(x|0) < R_0 \), then the optimum release time is \( T^* = T_R \).

(2.4) If \( h_m(0) \leq c_t/[c_w T_w(b + \alpha)] \) and \( R(x|0) \geq R_0 \), then the optimum release time is \( T^* = 0 \).

Similarly, we can obtain the optimal release policies for Case 2 and Case 3 (see [58]).

### 6.2 Statistical software testing-progress control

As well as quality/reliability assessment, software-testing managers should assess the degree of testing-progress. We can construct a statistical method for software testing-progress control based on a control chart method as follows [6], [59]. This method is based on several instantaneous fault-detection rates derived from software reliability growth models based on an NHPP. For example, the intensity function based on the delayed S-shaped software reliability growth model in section 3 (see Table 1) is given by

\[
h_M(t) = \frac{dM(t)}{dt} = ab^2 te^{-bt} \quad (a > 0, b > 0).
\]

From (53), we can derive

\[
\ln Z_M(t) = \ln a + 2 \cdot \ln b - bt,
\]

\[
Z_M(t) = \frac{h_M(t)}{t}.
\]

The mean value of instantaneous fault-detection rate which is represented by eq. (55) is defined as \textit{average-instantaneous fault-detection rate}. Eq. (54) means that the relation between the logarithm value of \( Z_M(t) \) and the testing time has linear property. If the testing phase progresses smoothly and the reliability growth is stable in the testing, the logarithm of average-instantaneous faults-detection rate decreases linearly with the testing time. From eq. (54), we can also estimate the unknown parameters \( a \) and \( b \) by the method of least-squares, and assess the testing-progress by applying a regression analysis to the observed data. It is assumed that the form of the data is \((t_k, Z_k)(k = 1, 2, \ldots, n)\) where \( t_k \) is the \( k \)th testing time and \( Z_k \) is the realization of average-instantaneous fault-detection rate \( Z_M(t) \). Letting the estimated unknown parameters be \( \hat{a} \) and \( \hat{b} \), we obtain the estimator of \( Y (= \ln Z_M(t)) \) as follows:

\[
\hat{Y} = \ln \hat{Z}_M(t) = \ln \hat{a} + 2 \cdot \ln \hat{b} - \hat{b}t
\]

\[
= \bar{Y} - \hat{b}(t - \bar{t}),
\]

where

\[
\bar{Y} = \frac{1}{n} \sum_{k=1}^{n} Y_k, \quad Y_k = \ln Z_k, \quad \bar{t} = \frac{1}{n} \sum_{k=1}^{n} t_k \quad (k = 1, 2, \ldots, n).
\]

The variation which is explained as the regression to the dependent variable \( Y \) is

\[
S_b = \sum_{k=1}^{n} (Y_k - \bar{Y})^2 = \hat{b}^2 \sum_{k=1}^{n} (t_k - \bar{t})^2.
\]

On the other hand, the error-variation unexplained as the regression is represented as the summation of residual squares. That is,

\[
S_e = \sum_{k=1}^{n} (Y_k - \hat{Y}_k)^2.
\]
The unbiased variance from eqs. (57) and (58) are:

\[ V_b = S_b, \quad V_e = \frac{S_e}{n - 2}. \]  

(59)

About eq. (56), we discuss the logarithm of average-instantaneous fault-detection rate \( Y_0 = \ln Z_M(t_0) \) at \( t = t_0 \) (\( t_0 \geq t_n \)) by using the results of the analysis of variance. The 100(1 - \( \alpha \)) percent confidence-interval to \( \hat{Y}_0 \) is given by

\[ \hat{Y}_0 \pm t \left( n - 2, 1 - \frac{\alpha}{2} \right) \sqrt{\text{Var}[\hat{Y}_0]}, \]

\[ \text{Var}[\hat{Y}_0] = \left\{ 1 + \frac{1}{n} + \frac{(t_0 - t)^2}{\sum_{k=1}^{m}(t_k - t)^2} \right\} V_e. \]  

(60)

Var[\( \hat{Y}_0 \)] in eq.(60) is the variance of \( \hat{Y}_0 \). \( t(h, p) \) in eq. (60) is 100\( p \) percent point of \( t \)-distribution at degree-of-freedom \( h \). We now make the control chart which consists of the center-line by the logarithm of average-instantaneous fault-detection rate, and the upper and lower control limits which are given by eq. (60). We can assess the testing-progress by applying a regression analysis to the observed data.

The testing-progress assessment indices for the other NHPP models are given by the following intensity function:

- \( h_m(t) = abe^{-bt} \) with relation \( \ln h_m(t) = (\ln a + \ln b) - bt \) (for the exponential software reliability growth model),
- \( h_\mu(t) = \frac{\lambda_0}{(\lambda_0 \theta t + 1)} \) with relation \( \ln h_\mu(t) = \ln \lambda_0 - \theta \mu(t) \) (for the logarithmic Poisson execution time model),
- \( h_\lambda(t) = \lambda \beta t^{\beta-1} \) with relation \( \ln h_\lambda(t) = (\ln \lambda + \ln \beta) + (\beta - 1) \ln t \) (for the Weibull process model [6], [8]).

The procedure of testing-progress control is shown as follows:

Step 1: An appropriate model is selected to apply and the model parameters are estimated by the method of least-squares.

Step 2: To certify goodness-of-fit of the estimated regression equation for the observed data, we use the \( F \)-test.

Step 3: Based on the result of the \( F \)-test, the center-line and upper and lower control limits of the control chart are calculated. The control chart is drawn.

Step 4: The observed data is plotted on the control chart and the stability of the testing-progress is judged.

6.3 Optimal testing-effort allocation problem

We discuss a management problem to achieve a reliable software system efficiently during module testing in the software development process by applying a testing-effort dependent software reliability growth model based on an NHPP (see Table 1). We take notice of the relationship between the testing-effort spent during the module testing and the detected software faults where the testing-effort is defined as resource expenditures spent on software testing, e.g. man-power, CPU hours, and executed test-cases. The software development manager has to decide how to use the specified testing-effort effectively in order to maximize the software quality and reliability [60]. That is, to develop a quality and reliable software system, it is very important for the manager to allocate the specified amount of testing-effort expenditures for each software module under some constraints. We can observe the software reliability growth in the module testing in terms of a time-dependent behavior of the cumulative number of faults detected during the testing phase.

Based on the testing-effort dependent software reliability growth model, we consider the following testing-effort allocation problem [61], [62]:

73
(1) The software system is composed of \( M \) independent modules. The number of software faults remaining in each module can be estimated by the model.

(2) The total amount of testing-effort expenditures for the module testing is specified.

(3) The manager has to allocate the specified total testing-effort expenditures to each software module so that the number of software faults remaining in the system may be minimized.

The following are defined:

\( a \) = the expected initial fault content,
\( r \) = the fault-detection rate per unit testing-effort expenditures \((0 < r < 1)\),
\( i \) = the subscript for each software module number \( i = 1,2,\ldots, M \),
\( w_i \) = the weight for each module \((w_i > 0)\),
\( n_i \) = the expected number of faults remaining in each module,
\( q_i, Q \) = the amount of testing-effort expenditures for each module to be allocated and the total testing-effort expenditures before module testing \((q_i \geq 0, Q > 0)\).

From eq. (8) and Table 1, i.e. \( n(t) = a \cdot \exp[-rW(t)] \), the estimated number of remaining faults for module \( i \) is formulated by

\[
    n_i = a_i \cdot \exp[-r_i q_i] \quad (i = 1, 2, \ldots, M).
\]

Thus, the optimal testing-effort allocation problem is formulated as:

\[
    \text{minimize} \quad \sum_{i=1}^{M} w_i n_i = \sum_{i=1}^{M} w_i a_i \cdot \exp[-r_i q_i],
\]

so that \[
    \sum_{i=1}^{M} q_i \leq Q, \quad q_i \geq 0 \quad (i = 1, 2, \ldots, M),
\]

where it is supposed that the parameter \( a_i \) and \( r_i \) have been already estimated by the model.

To solve the problem above, we consider the following Lagrangian:

\[
    L = \sum_{i=1}^{M} w_i a_i \cdot \exp[-r_i q_i] + \lambda \left( \sum_{i=1}^{M} q_i - Q \right),
\]

and the necessary and sufficient conditions \([63]\) for the minimum are

\[
    \begin{align*}
    \frac{\partial L}{\partial q_i} &= -w_i a_i r_i \cdot \exp[-r_i q_i] + \lambda \geq 0 \\
    q_i \cdot \frac{\partial L}{\partial q_i} &= 0 \quad (i = 1, 2, \ldots, M) \\
    \sum_{i=1}^{M} q_i &= Q \\
    q_i &\geq 0 \quad (i = 1, 2, \ldots, M)
    \end{align*}
\]

where \( \lambda \) is a Lagrange multiplier.

Without loss of generality, setting \( A_i = w_i a_i r_i (i = 1, 2, \ldots, M) \), we can assume that the following condition is satisfied for the tested modules:

\[
    A_1 \geq A_2 \geq \cdots \geq A_{k-1} \geq A_k \geq A_{k+1} \geq \cdots \geq A_M.
\]

74
This means that it is arranged in order of fault detectability for the tested modules. Now, if $A_k > \lambda \geq A_{k+1}$, from eq. (65) we have

$$q_i = \max \left\{ 0, \frac{1}{r_i} (\ln A_i - \ln \lambda) \right\},$$

i.e.

$$q_i = \begin{cases} \frac{1}{r_i} (\ln A_i - \ln \lambda) & (i = 1, 2, \ldots, k) \\ 0 & (i = k + 1, \ldots, M) \end{cases}.$$  \hspace{1cm} (67)

From eqs. (65) and (67), $\ln \lambda$ is given by

$$\ln \lambda = \frac{\sum_{i=1}^{k} \frac{1}{r_i} \ln A_i - Q}{\sum_{i=1}^{k} \frac{1}{r_i}} \quad (k = 1, 2, \ldots, M).$$  \hspace{1cm} (68)

Let $\lambda_k$ denote the value of the right-hand side of eq. (68). Then, the optimal Lagrange multiplier $\lambda^*$ exists in the set $\{\lambda_1, \lambda_2, \ldots, \lambda_M\}$. Hence, we can obtain $\lambda^*$ by the following procedures:

(1) Set $k = 1$.

(2) Compute $\lambda_k$ by eq. (68).

(3) If $A_k > \lambda_k \geq A_{k+1}$, then $\lambda^* = \lambda_k$(stop). Otherwise, set $k = k + 1$ and go back to (2).

The optimal solutions $q^*_i$ ($i = 1, 2, \ldots, M$) are given by

$$q^*_i = \begin{cases} \frac{1}{r_i} (\ln A_i - \ln \lambda^*) & (i = 1, 2, \ldots, k) \\ 0 & (i = k + 1, \ldots, M) \end{cases}.$$  \hspace{1cm} (69)

which means that the amount of testing-effort expenditures is needed more for the tested modules containing more faults.
References


Fig. 6: A diagrammatic representation of transitions between states of $X(t)$.

Fig. 7: A sample realization of $Y(t)$.

Fig. 8: State transition diagram for software availability modeling.
Fig. 9: Software reliability growth aspects during the warranty period.