Software Quality Journal, 12, 211–230, 2004
© 2004 Kluwer Academic Publishers. Manufactured in The Netherlands.

Analysis of Software Fault Removal Policies Using a Non-Homogeneous Continuous Time Markov Chain

SWAPNA S. GOKHALE ssg@engr.uconn.edu
Department of Computer Science and Engineering, University of Connecticut, Storrs, CT 06269, USA

MICHAEL R. LYU lyu@cse.cuhk.edu.hk
Computer Sciences and Engineering Department, The Chinese University of Hong Kong, Shatin, NT, Hong Kong

KISHOR S. TRIVEDI kst@ee.duke.edu
Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708, USA

Abstract. Software reliability is an important metric that quantifies the quality of a software product and is inversely related to the residual number of faults in the system. Fault removal is a critical process in achieving desired level of quality before software deployment in the field. Conventional software reliability models assume that the time to remove a fault is negligible and that the fault removal process is perfect. In this paper we examine various kinds of fault removal policies, and analyze their effect on the residual number of faults at the end of the testing process, using a non-homogeneous continuous time Markov chain. The fault removal rate is initially assumed to be constant, and it is subsequently extended to cover time and state dependencies. We then extend the non-homogeneous continuous time Markov chain (NHCTMC) framework to include imperfections in the fault removal process. A method to compute the failure intensity of the software in the presence of explicit fault removal is also proposed. The fault removal scenarios can be easily incorporated using the state-space view of the non-homogeneous Poisson process.

Keywords: software reliability, fault removal, non-homogeneous continuous time Markov chain

1. Introduction

The residual faults in a software system directly contribute to its failure intensity, causing software unreliability. Therefore, the problem of quantifying software reliability can be approached by obtaining the estimates of the residual number of faults in the software. The number of faults that remain in the code is also an important measure for the software developer from the point of view of planning maintenance activities. This is especially true for the developer of a commercial off-the-shelf software application that is expected to run on thousands of individual systems. In the case of such applications, users routinely report the occurrence of a specific failure to the software development organization, with the presumption of getting the underlying fault fixed, so that the failure does not recur. Thus commercial software organizations focus on the residual number of faults, in addition to reliability as a measure of software quality (Kenney, 1993).

Most of the black-box software reliability models (Farr, 1996) reported in the literature assume that a software fault is fixed immediately upon detection, and no new faults are introduced during the fault removal process. This assumption of instantaneous and perfect fault removal is impractical (Defamie et al., 1999; Wood, 1997), and should be amended in order to present more realistic testing scenarios. The time lag between the detection and removal of a fault is not explicitly accounted for in the traditional software reliability models, as it complicates the stochastic process significantly, making it impossible to obtain closed-form expressions for various metrics of interest. However, the estimates of the residual number of faults in the software is influenced not only by the detection process, but also by the time required to remove the detected faults. Fault removal process thus affects the number of faults remaining in the software and consequently its reliability, and makes a direct impact on the quality of a software product. The other stringent assumption is that of perfect fault removal. Studies have shown that most of the faults encountered by customers are the ones that are reintroduced during the removal of the faults detected during testing. Thus imperfect fault removal also affects the residual number of faults in the software, and can at times be a major cause of its unreliability and hence customer dissatisfaction (Levendel, 1990).

Conventional software reliability growth models (SRGMs) seek to obtain a closed form analytical solution for various metrics of interest because of which they cannot incorporate explicit and imperfect fault removal. However, the non-homogeneous Poisson process (NHPP) that underlies these SRGMs can also be regarded as a non-homogeneous continuous time Markov chain (NHCTMC), where the evolution of the process is represented in the form of state transitions. As a result, NHCTMC is often referred to as the "state-space view" of NHPP. By resorting to a numerical solution of the NHCTMC, we incorporate explicit and imperfect fault removal into SRGMs. We also analyze the effect of various fault removal strategies on the residual number of faults using the NHCTMC framework. We also describe how the framework could be used to incorporate imperfections in the fault removal process. We propose a method to compute the failure intensity of the software in the presence of fault removal.

The layout of this paper is as follows: Section 2 provides a brief overview of finite failure NHPP models. Section 3 presents the state-space view of non-homogeneous Poisson process, and describes a numerical technique to obtain the solution of a NHPP. Section 4 describes various fault removal strategies, presents a framework to incorporate fault removal activities into the finite failure software reliability growth models, extends the framework to include imperfect fault removal and proposes a method to compute the failure intensity of the software in the presence of fault removal. Section 5 presents some numerical results. Section 6 concludes the paper.

2. Finite failure NHPP models

This section provides an overview of the finite failure non-homogeneous Poisson process software reliability models. This class of models is concerned with the number of faults detected in a given time and hence are also referred to as "fault count models" (Goel and Okumoto, 1979).

Software failures are assumed to display the behavior of a non-homogeneous Poisson process (NHPP), the stochastic process $\{N(t), t \geq 0\}$, with a rate parameter $\lambda(t)$ that is time-dependent. The function $\lambda(t)$ denotes the instantaneous failure intensity.

Given $\lambda(t)$, the mean value function $m(t) = E[N(t)]$, where $m(t)$ denotes the expected number of faults detected by time t, satisfies the relation:

$$m(t) = \int_0^t \lambda(s)\,\mathrm{d}s \tag{1}$$

and

$$\lambda(t) = \frac{\mathrm{d}m(t)}{\mathrm{d}t}. \tag{2}$$

The random variable $N(t)$ follows a Poisson distribution with parameter $m(t)$, that is, the probability that $N(t)$ takes a given non-negative integer value n is determined by:

$$P\{N(t) = n\} = \frac{[m(t)]^n \cdot \mathrm{e}^{-m(t)}}{n!}, \quad n = 0, 1, \ldots, \infty. \tag{3}$$

The time domain models which assume the failure process to be a NHPP differ in the approach they use for determining $\lambda(t)$ or $m(t)$. The NHPP models can be further classified into finite failures and infinite failures models. We will be concerned only with finite failure NHPP models in this paper.

Finite failures NHPP models assume that the expected number of faults detected during an infinite amount of testing time will be finite and this number is denoted by a (Farr, 1996). The mean value function $m(t)$ in case of finite failure NHPP models can be written as (Gokhale et al., 1996):

$$m(t) = aF(t), \tag{4}$$

where $F(t)$ is a distribution function.

The failure intensity $\lambda(t)$ for finite failure NHPP models can be written as:

$$\lambda(t) = aF'(t). \tag{5}$$

The failure intensity $\lambda(t)$ can also be written as:

$$\lambda(t) = \big[a - m(t)\big]\frac{F(t)}{1 - F'(t)}, \tag{6}$$

where

$$h(t) = \frac{F(t)}{1 - F'(t)} \tag{7}$$

is the failure occurrence rate per fault or the hazard rate. Referring to Equation (6), $[a - m(t)]$ represents the expected number of faults remaining, and hence is a non-increasing function of testing time. Thus, the nature of the failure intensity is governed by the nature of the failure occurrence rate per fault (Gokhale and Trivedi, 1999). Popular finite failure NHPP models can be classified according to the nature of their failure occurrence rate per fault. The Goel–Okumoto software reliability growth model has a constant failure occurrence rate per fault (Goel and Okumoto, 1979), the generalized Goel–Okumoto model can exhibit an increasing or a decreasing failure occurrence rate per fault (Goel, 1985), the S-shaped model exhibits an increasing failure occurrence rate per fault (Yamada et al., 1983), and the log-logistic model exhibits an increasing/decreasing failure occurrence rate per fault (Gokhale and Trivedi, 1998).

3. State-space view of NHPP

The NHPP models described above provide a closed-form analytical expression for the expected number of faults detected given by the mean value function, $m(t)$. However, the mean value function $m(t)$ can also be obtained using numerical techniques. In this section we present a brief overview of homogeneous and non-homogeneous Poisson process, and describe a method to obtain a solution of the NHPP processes using numerical techniques.

A discrete state, continuous-time stochastic process $\{N(t), t \geq 0\}$ is called a Markov chain if for any $t_0 < t_1 < \cdots < t_n < t$, the conditional probability mass function of $N(t)$ for given values of $N(t_0), N(t_1), \ldots, N(t_n)$, depends only on $N(t_n)$. Mathematically the above statement can be expressed as (Trivedi, 2001):

$$\begin{aligned} &P\big[N(t) = x \mid N(t_n) = x_n, \ldots, N(t_0) = x(t_0)\big] \\ &\quad = P\big[N(t) = x \mid N(t_n) = x_n\big]. \end{aligned} \tag{8}$$

Equation (8) is known as the Markov property.

A Markov chain $\{N(t), t \geqslant 0\}$ is said to be time-homogeneous, if the following property holds:

$$P\big[N(t) = x \mid N(t_n) = x_n\big] = P\big[N(t - t_n) = x \mid N(0) = x_n\big]. \tag{9}$$

Equation (9) also known as the property of stationary increments implies that the next state of the system depends only on the present state and not on the time when the system entered that state. Thus in case of a homogeneous continuous time Markov chain (CTMC), the holding time in each state is exponentially distributed.

Markov chains which do not satisfy the property of stationary increments given by Equation (9) are called as non-homogeneous Markov chains (Kulkarni, 1995). For a non-homogeneous CTMC, the holding time distribution in state 0 is $1-\mathrm{e}^{-m(t)}$, whereas the distribution of the holding times in the remaining states is fairly complicated.

A homogeneous Poisson process is a homogeneous continuous time Markov chain, whereas a non-homogeneous Poisson process is a non-homogeneous continuous time Markov chain. Figure 1 shows the representation of a non-homogeneous Poisson process by a non-homogeneous continuous time Markov chain (NHCTMC). In the figure, the state of the process is given by the number of events observed. When the NHCTMC is used to model the fault detection process, each event represents fault detection and hence the state of the process is given by the number of faults detected. The process starts in state 0, since initially at time $t = 0$ which marks the beginning of the testing process no faults are detected. Upon the detection of the first fault, the process transitions to state 1, and the rate governing this transition is the failure intensity function $\lambda(t)$. Similarly, upon the detection of the second fault the process transitions to state 2 and so on. Note that the time origin of a transition rate emanating from any state is the beginning of system operation (this is called global time) and not from the time of entry into that state (the so-called local time). Globally time dependent transition rates imply a non-homogeneous Markov chain as in the present case whilst locally time dependent transition rates would imply a semi-Markov process.

The expected number of faults detected given by the mean value function $m(t)$ can also be computed numerically by solving the Markov chain shown in Figure 1 using SHARPE (Sahner et al., 1996). SHARPE stands for Symbolic Hierarchical Automated Reliability and Performance Evaluator, and is a powerful tool that can be used to solve stochastic models of performance, reliability and performability. SHARPE was first developed in 1986 for three groups of users, namely, practicing engineers, researchers in performance and reliability modeling, and students in science and engineering courses. Since then several revisions have been made to SHARPE to fix bugs and to adopt new requirements. Details of SHARPE can be obtained from (Sahner et al., 1996). While using SHARPE to solve the chain, two issues need to be resolved.

The first issue is that the chain in Figure 1 has an infinite number of states. In order to overcome this issue, the chain can be truncated to θ states. Assuming that we are unlikely find more faults than six times the expected number of faults present in the beginning, we use:

$$\theta = 6\lceil a \rceil, \tag{10}$$

where a is the expected number of faults that can be detected given infinite testing time as in case of Equation (4). The second issue is that SHARPE is designed to solve homogeneous CTMCs. We get around this problem using the time-stepping techniques by dividing the time axis uniformly into small sub-intervals, where within each time sub-interval, the failure intensity, $\lambda(t)$, can be assumed to be constant. Thus, within each time sub-interval, the non-homogeneous continuous time Markov chain reduces to a homogeneous continuous time Markov chain which can then be solved using SHARPE. This value of $\lambda(t)$ is used to obtain the state probability vector of the Markov chain at the end of that time sub-interval. These state probabilities then form the initial probability vector for the next time sub-interval. Let $p_i(t)$ denote the probability of being in state i at time t. The mean value function, $m(t)$, can then be computed as:

$$m(t) = \sum_{i=0}^{\theta} i \cdot p_i(t). \tag{11}$$

The accuracy of the solution produced by solving the NHCTMC using SHARPE will be influenced by the length of the time sub-interval used to approximate the value of the failure intensity function $\lambda(t)$. In general, the smaller the sub-interval, the better will be the accuracy. However, as the length of the sub-interval decreases, the computational cost involved in obtaining a solution increases. Thus, an appropriate choice of the length of the time sub-interval is a matter of tradeoff between the computational cost and the desired accuracy of the solution.

We demonstrate the computational equivalence of the mean value function obtained using the expressions presented in Section 2 (referred henceforth as the closed-form analytical method) and the numerical solution method, using the NTDS data (Goel and Okumoto, 1979; Jelinski and Moranda, 1972). The NTDS data is from the U.S.

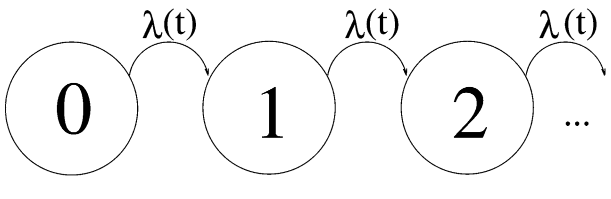

Figure 1. Non-homogeneous Markov chain for NHPP models.

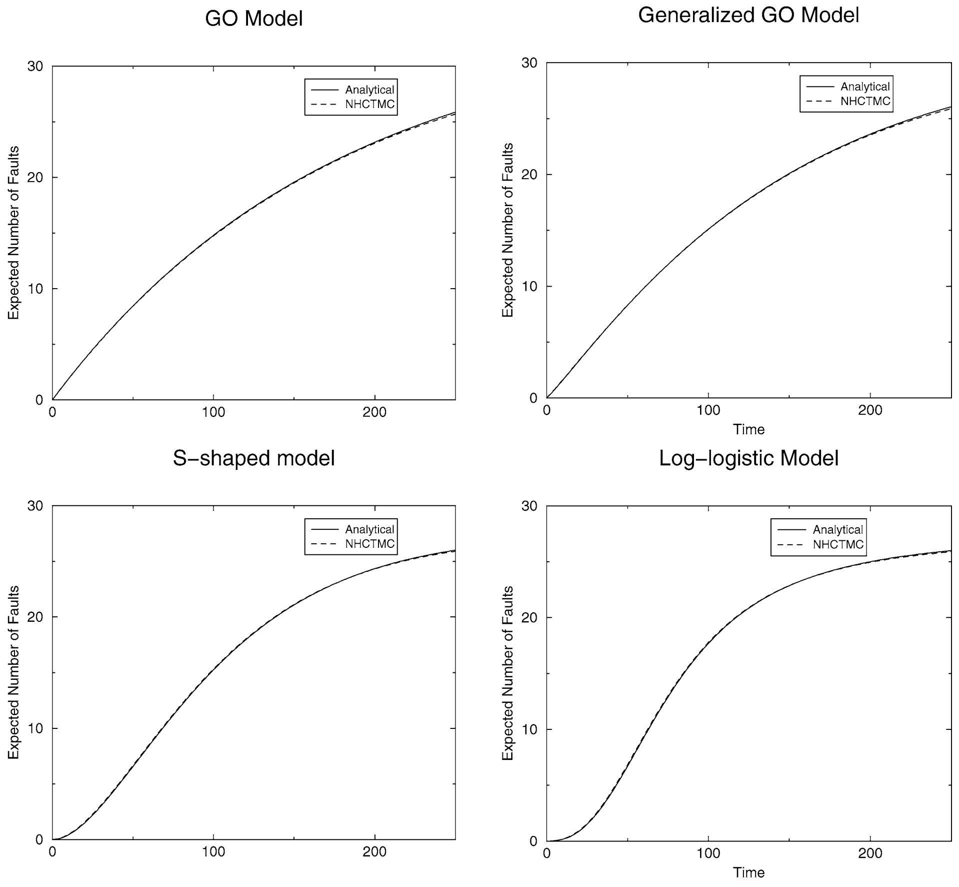

Figure 2. Comparison of analytical and numerical mean value functions.

Navy Fleet Computer Programming Center consisting of errors in the development of software for a real-time, multicomputer complex which forms the core of the Naval Tactical Data System (NTDS). The NTDS software consisted of 38 different modules. Each module was supposed to follow three stages: the production (development) phase, the test phase, and the user phase. The parameters of the four NHPP models described above were estimated from the NTDS data using SREPT (Ramani et al., 1998), and then the mean value function is computed for each of these models by solving the Markov chain in Figure 1. Figure 2 shows the closed-form mean value function and the one obtained using a numerical solution of the non-homogeneous continuous time Markov chain (NHCTMC) for Goel–Okumoto, generalized Goel–Okumoto, S-shaped, and log-logistic models, respectively.

As observed from Figure 2, the numerical solution of the mean value function obtained using the state-space view of the non-homogeneous Poisson process gives us a very good approximation to the analytical solution. In the next section, we describe how we exploit the flexibility offered by the state-space view to incorporate more realistic features into the NHPP software reliability models, which were initially based on oversimplifying assumptions in order to ensure mathematical tractability.

4. Fault removal into finite failure NHPP models

In this section we discuss the various fault removal strategies and the incorporation of explicit fault removal into the finite failure NHPP models. We then extend the NHCTMC framework with explicit fault removal to include imperfect fault removal. We also propose a method to compute the failure intensity of the software in the presence of fault removal.

4.1. Fault removal policies

We assume that the testing process is unaffected by fault removal activity, i.e., testing continues even during fault removal. The detected faults are queued to be removed. The fault detection rate $\lambda(n, t)$ depends on the number of faults detected, or time, or both. The fault removal rate $\mu(j, t)$ also depends on time, or the number of faults queued to be removed, or both. Thus at time t, if the number of faults detected is n, and the number of faults pending to be removed is j, then $n - j$ faults have been removed.[1]

The fault removal rate, $\mu(j, t)$ is assumed to be of the following types:

- Constant: This is the simplest possible situation where the fault removal rate is independent of the number of faults pending as well as time. The fault removal process discussed by Kremer (1983), Levendel (1990), Dalal and Mallows (1990), and Schneidewind (2002; 2003) is of this type. The fault removal rate $\mu(j, t)$ in this case is given by:

$$\mu(j, t) = \mu. \tag{12}$$

- Fault dependent: The fault removal rate could depend on the number of faults pending to be removed. As the number of faults pending increases, it is likely that more resources are allocated for fault removal and hence the faults are removed faster, which reflects as a faster fault removal rate. If j is the number of faults pending, the fault removal rate $\mu(j, t)$ can be given by:

$$\mu(j, t) = j \cdot k, \tag{13}$$

  where the constant k can reflect the portion of resources allocated for fault removal.

- The fault removal rate could also be time-dependent. Intuitively, the fault removal rate is lower at the beginning of the testing phase and increases as testing progresses or as the project deadline approaches. The fault removal rate reaches a constant value beyond which it cannot increase, and this may be an indicator of budget constraints or exhaustion of resources, etc. The time-dependent fault removal rate is hypothesized to be of the form:

$$\mu(j,t) = \alpha\left(1 - \mathrm{e}^{-\beta t}\right) \tag{14}$$

  for some constants α and β which may reflect resource allocation and scheduling constraints of a particular project. We refer to this as the time-dependent fault removal rate # 1.
- Fault removal rate could also be time-dependent in the case of latent faults, which are inherently harder to remove, and can be hypothesized to be:

$$\mu(j,t) = \alpha \mathrm{e}^{-\beta t}. \tag{15}$$

  We refer to this as the time-dependent fault removal rate # 2.
- Time-dependent fault removal rate could also have any other functional form as dictated by the process of a particular software project.

Fault removal activities can also be delayed to a later point in time in case of some software development scenarios. Fault removal can be delayed based on the following two constraints:

- Fault removal can be delayed till a certain number ϕ of faults are detected and are pending to be removed.
- Fault removal may be suspended for a certain average amount of time $1/\mu_1$ after the detection of the fault, or in other words there is an average time lag of $1/\mu_1$ units between the detection of the fault and the initiation of its removal. Fault removal can also be delayed for a certain period of time after testing begins, and once initiated it can proceed as per any of the fault removal policies described above.

The fault removal rate could have any of the forms described above in case of deferred fault removal.

4.2. *Incorporating explicit fault removal*

The finite failure NHPP models represented as a non-homogeneous continuous time Markov chain (NHCTMC) in Section 3 are extended in this section to incorporate the various fault removal strategies described above. The state-space of the NHCTMC in this case is a tuple (i, j), where i is the number of faults removed and j is the number of faults detected, and pending to be removed. The faults are removed one at a time, and thus the detected faults form a queue up to a maximum of $\theta - i$, where i is the number of faults removed and θ is as given in Equation (10). Figure 3 shows the NHCTMC with a constant fault removal rate and Figure 4 shows the NHCTMC with a fault dependent removal rate. Expressions for the expected number of faults removed, $m_{\mathrm{R}}(t)$ and the expected number of faults detected, $m_{\mathrm{D}}(t)$, by time t in case of Figures 3 and 4 are given by Equations (16) and (17), respectively.

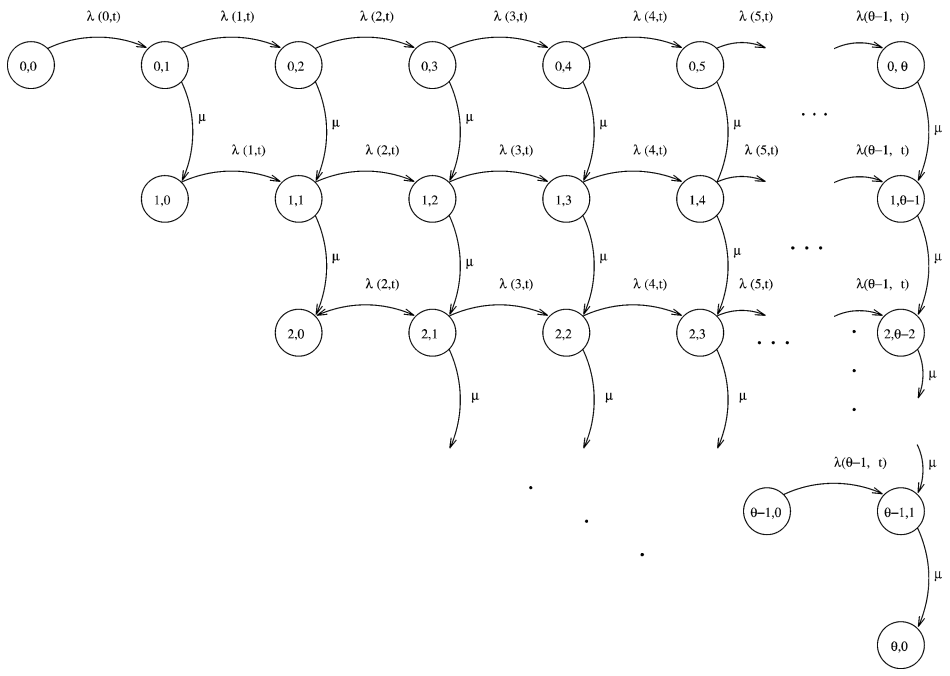

Figure 3. NHCTMC—constant fault removal rate.

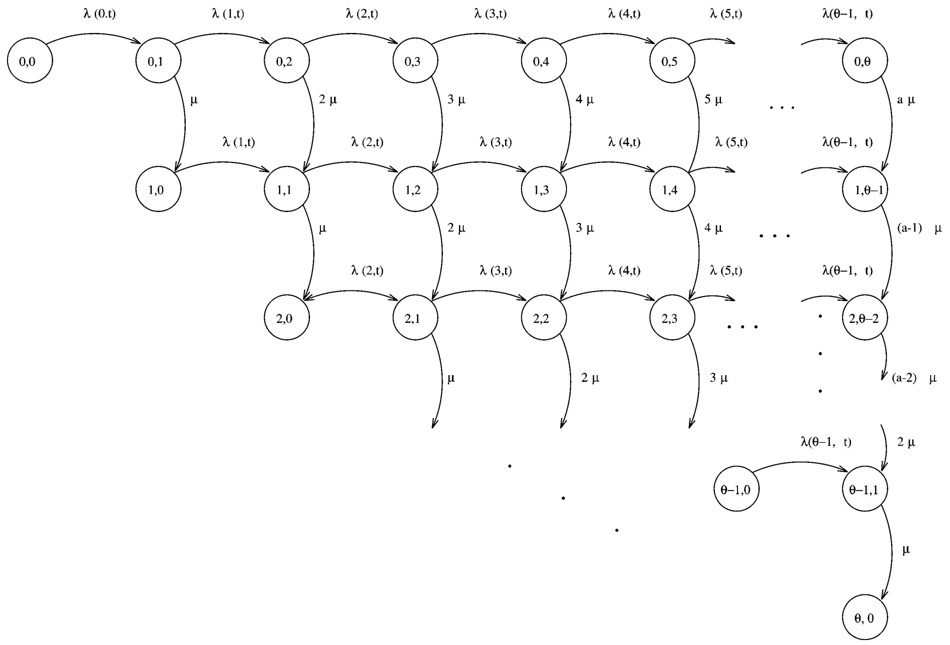

Figure 4. NHCTMC—fault dependent removal rate.

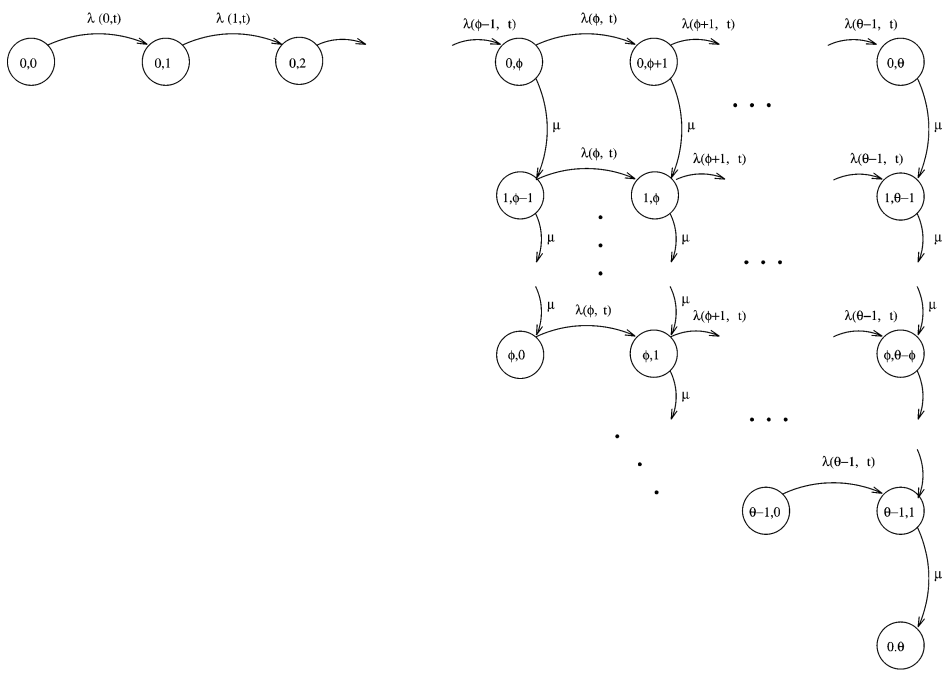

Figure 5. NHCTMC—delayed (fault dependent delay) fault removal rate.

$$m_{\mathrm{R}}(t) = \sum_{i=0}^{\theta} \sum_{j=0}^{\theta-i} i \cdot p_{i,j}(t), \tag{16}$$

$$m_{\mathrm{D}}(t) = \sum_{i=0}^{\theta} \sum_{j=0}^{\theta-i} (i+j) \cdot p_{i,j}(t). \tag{17}$$

NHCTMC model with delayed fault removal, where the fault removal activities are delayed until a certain number ϕ of the faults is removed (fault dependent delay) is shown in Figure 5. The expected number of faults removed, $m_{\mathrm{R}}(t)$, and the expected number of faults detected, $m_{\mathrm{D}}(t)$, in this case are given by Equations (16) and (17), respectively.

Delayed fault removal, where the fault removal activities are suspended till a certain time is elapsed after the detection of a fault can be incorporated in the NHCTMC model using a phase type distribution (Kulkarni, 1995). Figure 6 shows the NHCTMC where the removal is carried out in two phases. The mean time spent in phase 1 or the idle phase is given by $1/\mu_1$, and the mean time spent in phase 2 or the phase where the actual removal activities are carried out is given by $1/\mu_2$. Thus, the mean fault removal time $1/\mu$ is given by:

$$\frac{1}{\mu} = \frac{1}{\mu_1} + \frac{1}{\mu_2}. \tag{18}$$

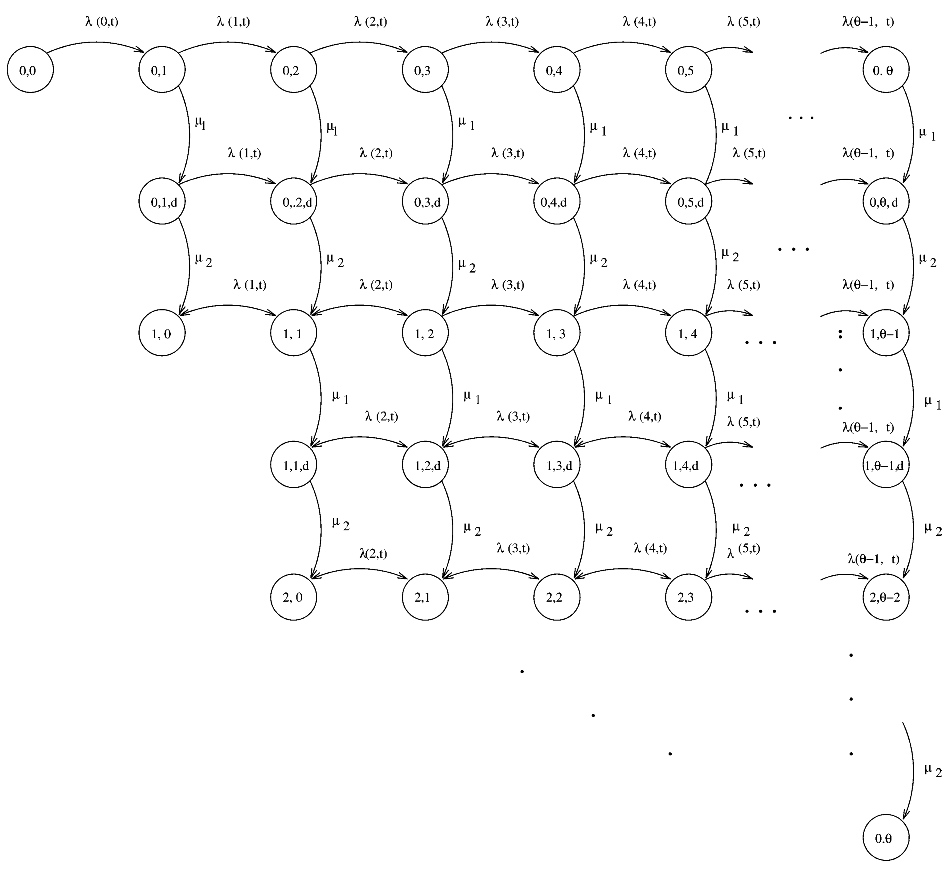

Figure 6. NHCTMC—(time delayed) fault removal rate.

In Figure 6, state $(i,\ j,\ \mathrm{d})$ implies that i faults have been removed, j faults have been detected and are queued for removal, and "d" implies intermediate phase of repair. The expected number of faults removed, $m_{\mathrm{R}}(t)$ and the expected number of faults detected, $m_{\mathrm{D}}(t)$ is given by Equations (19) and (20), respectively.

$$m_{\mathrm{R}}(t) = \sum_{i=0}^{\theta} \sum_{j=0}^{\theta-i} i \cdot \bigl(p_{i,j}(t) + p_{i,j,\mathrm{d}}(t)\bigr), \tag{19}$$

$$m_{\mathrm{D}}(t) = \sum_{i=0}^{\theta} \sum_{j=0}^{\theta-i} (i+j) \cdot \bigl(p_{i,j}(t) + p_{i,j,\mathrm{d}}(t)\bigr). \tag{20}$$

As mentioned earlier in Figures 3–5, i is the number of faults removed and j is the number of faults detected, and pending to be removed. The NHCTMC transitions either in the horizontal direction or in the vertical direction depending on whether a fault is detected or removed. In state $(i,\ j)$ when a fault is detected the NHCTMC transitions to state $j+1$, since the number of faults pending to be removed increases by 1. On the other hand, in state (i, j) if a fault is removed the NHCTMC transitions to state $(i + 1, j - 1)$. The former transition is governed by the failure intensity function $\lambda(t)$, and the latter transition is governed by the fault removal rate. Similar arguments hold for the transitions shown in Figure 6, except that the state representation is given by a 3-tuple.

4.3. *Incorporating imperfect fault removal*

The importance of fault reintroduction has been recognized by several researchers and a few models have been proposed/extended (Goel and Okumoto, 1979; Gokhale et al., 1997; Kapure et al., 1992; Kremer, 1983; Levendel, 1990; Ohba and Chou, 1989; Sumita and Shantikumar, 1986) to incorporate imperfect fault removal. However, most of these are restricted to either instantaneous fault removal or constant fault removal rate. We extend the NHCTMC framework described in the previous section to account for fault reintroduction based on the following assumptions. Whenever a fault is detected, there are three mutually exclusive possibilities to the corresponding fault removal effort: reduction in the fault content by 1 with probability p, no change in the fault content with probability q, and an increase in the fault content by 1 with probability r. We assume other cases (additional fault removal or fault reintroduction) are rare and negligible. Thus $p + q + r = 1$ (Kremer, 1983).

The NHCTMC with imperfect fault removal is shown in Figure 7. In Figure 7, in state (i, j), when a fault is removed the process transitions to state $(i + 1, j - 1)$ with rate $\mu \cdot p$ representing reduction in fault content by 1, remains in the same state with rate $\mu \cdot q$ representing no change in fault content, and transitions to state $(i, j + 1)$ with rate $\mu \cdot r$ representing an increase in fault content by 1. The fault removal rate is assumed to be a constant in case of Figure 7. However, imperfect fault removal can be taken into account in accordance with any of the fault removal strategies described earlier. Expressions for the expected number of faults removed and detected are given by Equations (16) and (17), respectively.

4.4. *Computation of failure intensity*

In this section we describe a method to compute the failure intensity of the software in the presence of fault removal. Under the idealized assumption of instantaneous and perfect fault removal, the expected number of faults removed is the same as the expected number of faults detected. However, if we take into consideration the time required for removal, the expected number of faults removed at any given time is less than the expected number of faults detected as seen in Figure 8. Thus at any time t, $\lambda(n, t)$, which is the failure intensity of the software based on the assumption of instantaneous and perfect fault removal, needs to be adjusted in order to reflect the expected number of faults that have been detected but not yet removed. We calculate this adjustment as follows: let $m_{\mathrm{R}}(t)$ denote the expected number of faults removed at time t, and $m_{\mathrm{D}}(t)$ denote the expected number of faults detected at time t. The approach consists of computing time $t_{\mathrm{R}} \leqslant t$, such that $m_{\mathrm{D}}(t_{\mathrm{R}}) = m_{\mathrm{R}}(t)$, i.e., time t_{R} at which the expected number of faults detected as well as removed under the assumption

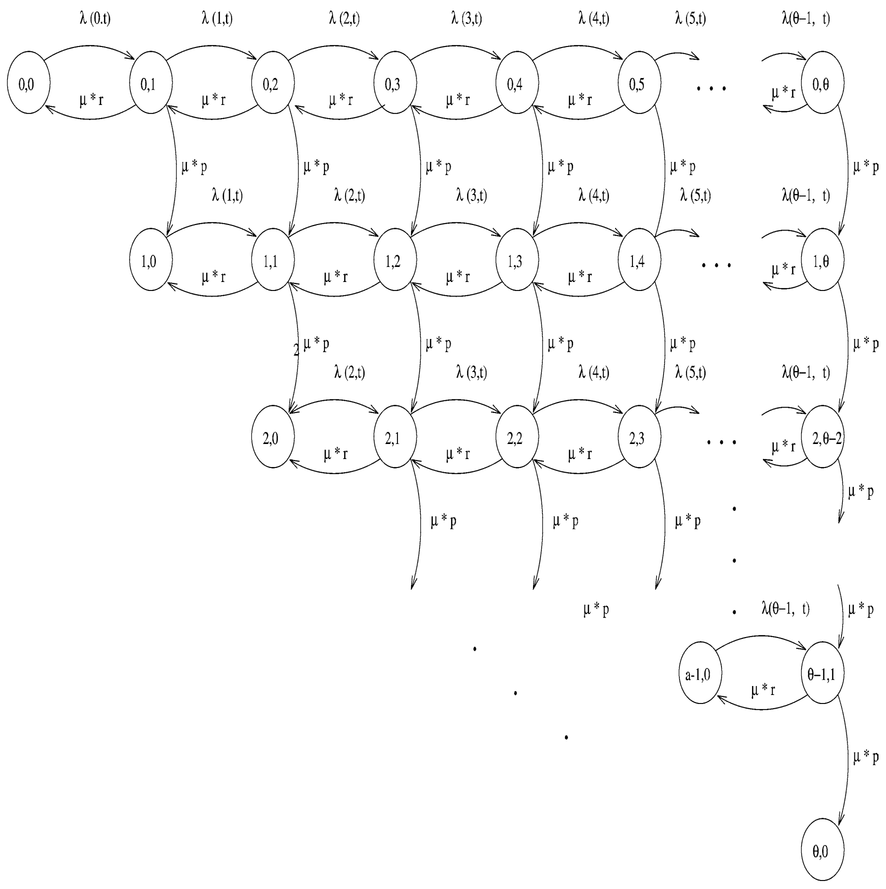

Figure 7. NHCTMC with imperfect repair.

of instantaneous fault removal is equal to the expected number of faults removed with explicit fault removal. Whereas the perceived failure intensity at time t, under the assumption of instantaneous and perfect fault removal is $\lambda(n, t)$, we postulate that the actual failure intensity (failure intensity after adjustment), denoted by $\lambda'(n, t)$, can be approximately given by $\lambda(n, t_R)$, where $t_R \leqslant t$. The condition $t = t_R$ represents the situation of instantaneous and perfect fault removal. This can be considered as a "roll-back" in time, and is like saying that accounting for fault detection and fault removal separately up to time t is equivalent to instantaneous and perfect fault removal up to time t_R.

We illustrate this approach with the help of an example. Referring to the left plot in Figure 8, the expected number of faults detected, $m_D(t)$, at time $t = 200$ is 23.16, while the expected number of faults removed at time t, $m_R(t)$ is 17.64. The failure intensity for this particular example is assumed to be of the Goel–Okumoto model and is given by $\lambda(n, t) = 34.05 \cdot 0.0057 \cdot e^{-0.0057 \cdot t}$. The value of t_R computed using these values is 128.1. Thus if the software is released at $t = 200$, its perceived failure intensity would be 0.0622, whereas the actual failure intensity after adjustment would

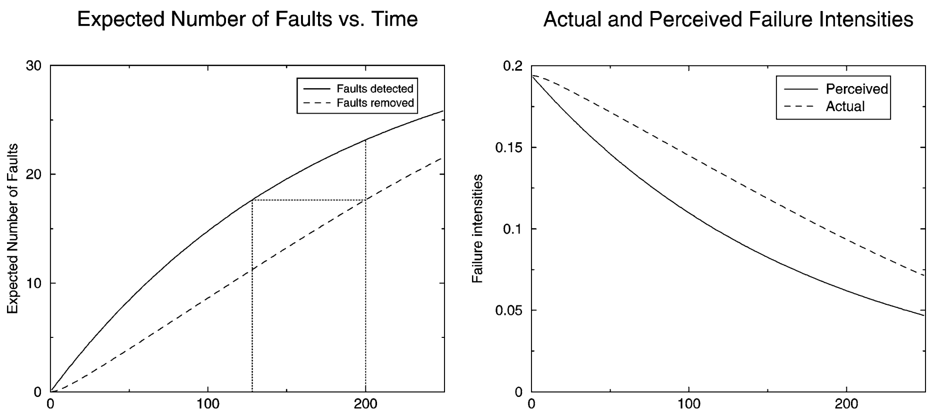

Figure 8. An example for failure intensity adjustment.

be 0.093. The method discussed here was repeated for every time step dt, and the plot on the right in Figure 8 shows the perceived as well as the actual failure intensities for the entire time interval.

5. Numerical results

In this section we demonstrate the utility of the state-space technique to generate fault detection and fault removal profiles, with the help of some case studies. Without loss of generality, we use the failure intensity of the Goel–Okumoto model for the case studies in this section. We estimate the parameters of the failure intensity function of the Goel–Okumoto model using data set obtained from Charles Stark Draper Laboratories using CASRE (Lyu and Nikora, 1992). The failure intensity used is given by $\lambda(n, t) = 39.0082 \cdot 0.0037 \cdot e^{-0.0037 \cdot t}$.

The expected number of faults detected and removed for the various fault removal policies is shown in Figure 9. Initially, we obtain the expected number of faults detected and removed for various values of constant fault removal rate, μ. The values of the fault removal rate μ were set to be approximately 100%, 70%, 35% and 17.5% of the maximum fault detection rate. The expected number of faults removed decreases as μ decreases, and expectedly so. The cumulative fault removal curve has a form similar to the cumulative fault detection curve, and as the fault removal rate increases, the fault removal curve almost follows the fault detection curve. We then obtain the expected number of faults removed as a function of time, when the fault removal rate depends on the number of pending faults (Equation (13)). The expected number of faults removed is directly related to the proportionality constant k in Equation (13). The fault removal rate in this case does not have a closed form expression but can be computed by assigning appropriate reward rates to the states of the NHCTMC. As k increases, the expected number of faults removed increases. The expected number of faults detected and removed as a function of time for time-dependent fault removal rate #1 as given by Equation (14) was obtained next. The value of α is held at 0.15, which is approximately the maximum value of the fault detection rate. The cumulative fault removal curve in this case is also similar to the cumulative fault detection curve, and the difference between the expected number of detected and removed faults depends on the value of β. As β increases, the fault removal rate increases and the expected number of faults removed increases. The expected number of faults removed for time-dependent fault removal rate #2 as given in Equation (15) was then obtained. In this case, as the value of β increases, the fault removal rate decreases and the expected number of faults removed decreases. The expected number of faults detected and removed as a function of time for delayed fault removal, where the expected delay between the detection of the fault and the initiation of its removal is μ_1 units, was obtained next. The expected number of faults removed decreases with increasing μ_1. The expected number of faults detected and removed for delayed fault removal where fault removal begins only after a certain number of faults ϕ are accumulated was obtained next, for different values of ϕ, setting the fault removal rate μ to be 0.1. As ϕ increases, the expected number of faults removed decreases.

Figure 9 depicts that for higher values of fault removal rates, the fault removal profile follows the fault detection profile very closely, and the estimates of the residual number of faults based on the idealized assumption of instantaneous fault removal are close to the ones with explicit fault removal. However, as the fault removal rate increases the fault removal resources could be under utilized. For the sake of illustration, we measure the utilization of the fault removal resources in case of the constant fault removal rate. This can be achieved easily by assigning proper rewards to the states of the NHCTMC. The utilization is shown in Figure 10. As seen from the figure, for higher values of μ, the utilization is low, and the debugging resources are not used up to their complete capacity for an extended period of time. This may not be very cost effective, especially with increasing budget and deadline constraints facing modern software development organizations. Incorporating explicit fault removal into the software reliability growth models can thus be used to guide decision making about the allocation of resources to the crucial, but perhaps the most important activity of fault removal, so that a maximum number of faults are detected and removed before the shipment of the software product in a cost effective manner.

Figure 11 shows the expected number of faults removed for different values of p, q and r, in case of imperfect fault removal. The figure shows that the expected number of faults removed decreases as the probability of perfect fault removal p decreases.

Figure 12 shows the actual and the perceived failure intensities computed for the time period from $t = 0$ to $t = 275$ using the method explained in Section 4.4. As can be seen from the figure, the actual failure intensity which accounts for fault removal is higher than the perceived failure intensity which relies on the assumption of instantaneous repair. Thus if the software were to be released at $t = 275$ time units its actual failure intensity would be 0.0633 while its perceived failure intensity would be 0.0522.

The key points observed from the numerical results can be summarized as follows:

- The fault removal profile closely follows the fault detection profile for high values of fault removal rates. However, for very high fault removal rates the fault removal resources are severely under utilized which could lead to an inefficient use of the overall resources.

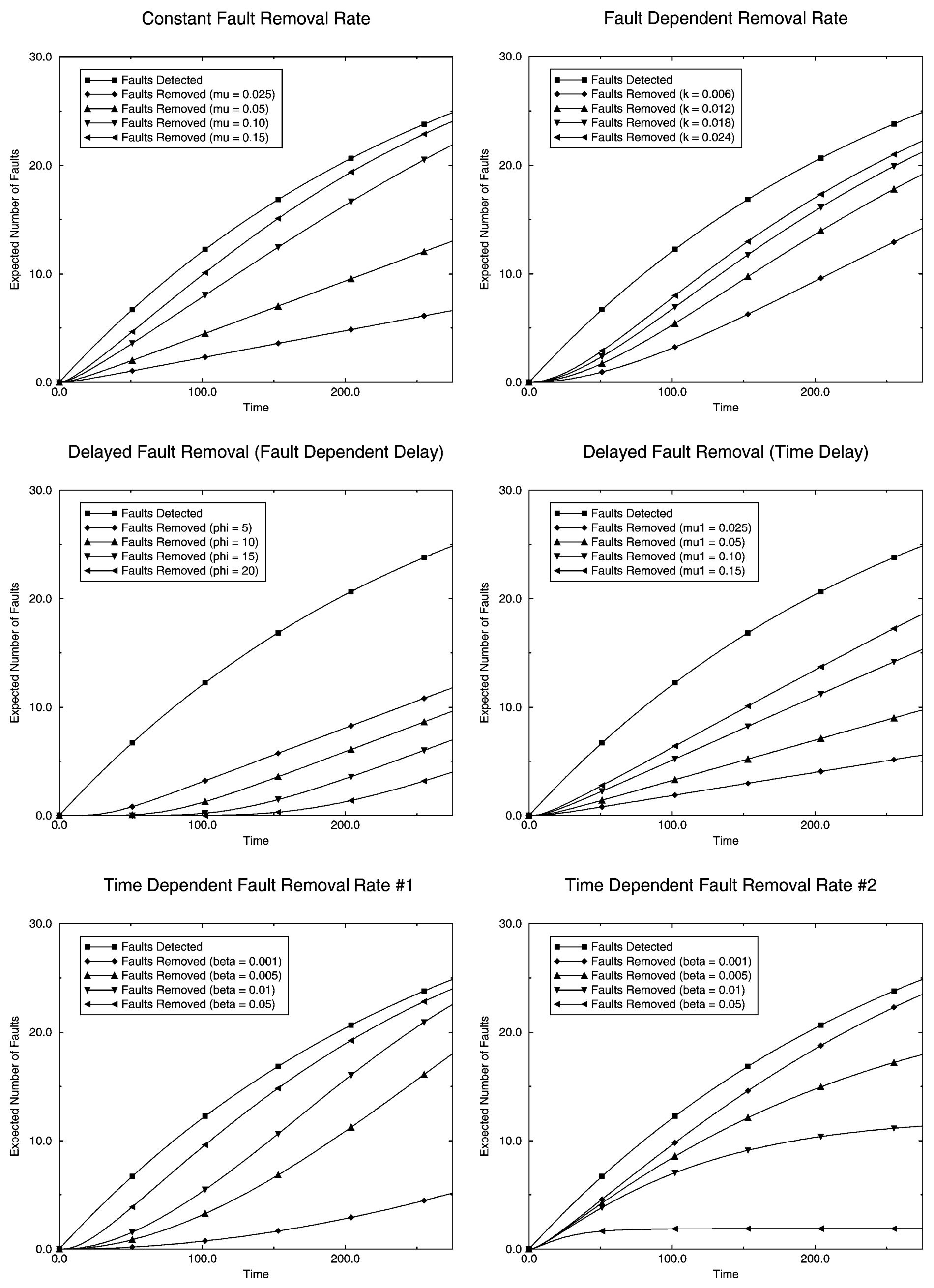

Figure 9. Expected number of detected and removed faults for fault removal policies.

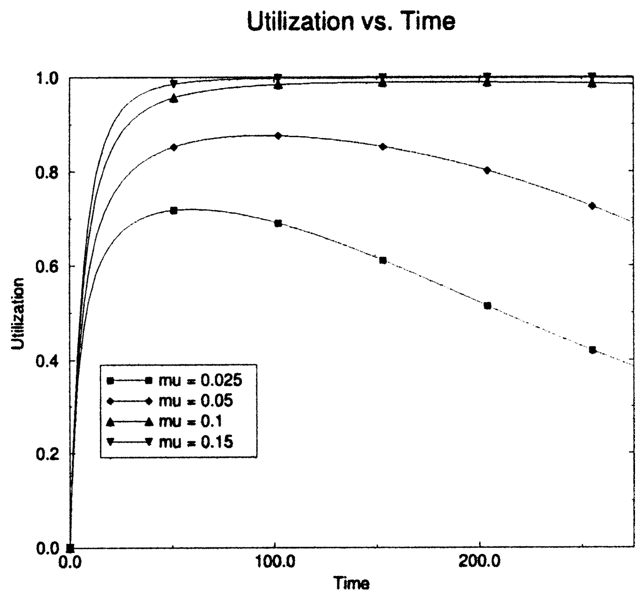

Figure 10. Utilization for constant fault removal rate.

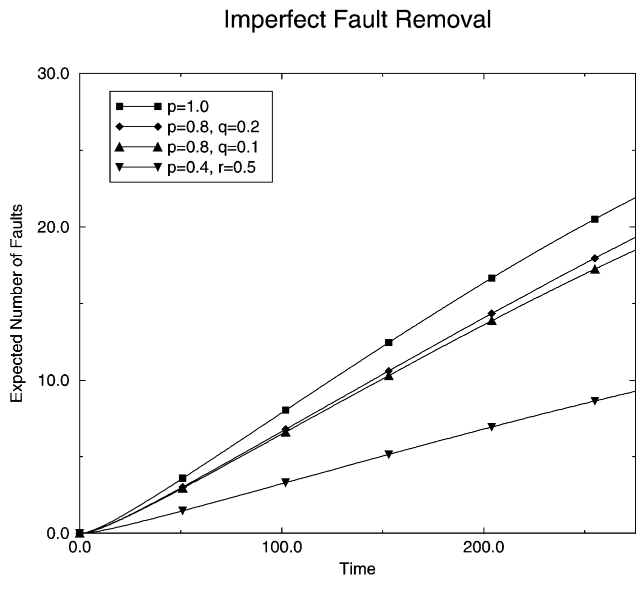

Figure 11. Expected number of faults removed for imperfect fault removal.

- The number of faults removed decreases as the probability of perfect fault removal decreases. Thus, the residual fault density in the software depends not only on the fault detection and fault removal rates, but is also affected by the probability of perfect fault removal.

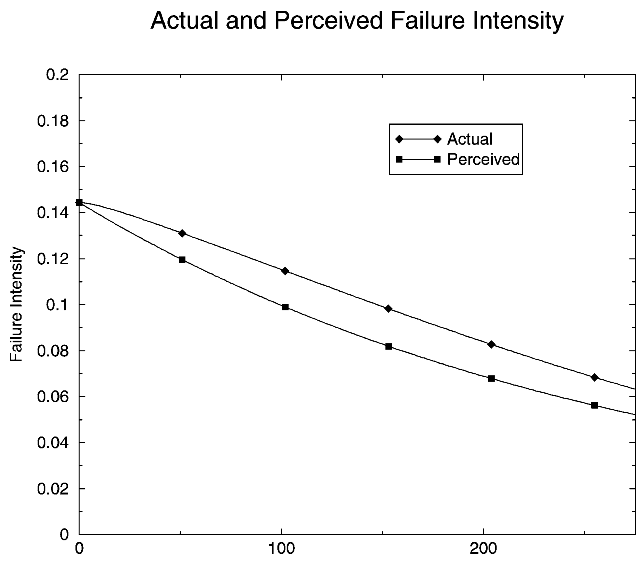

Figure 12. Actual and perceived failure intensities.

- The actual failure intensity of the software (taking into account explicit fault removal) is higher than the perceived failure intensity of the software, which does not consider the effects of fault removal process explicitly. As a result, budget and software release time decisions based on the perceived failure intensity rather than the actual failure intensity may tend to be optimistic.

The NHCTMC framework incorporating explicit and imperfect fault removal could be used in a variety of ways. At the end of the testing phase it could be used to obtain an estimate of the residual number of faults in the software. More importantly, it could be used to obtain a realistic estimate of the optimal testing time, when costs justify the stop test decision (Gokhale, 2003). The framework could also be used for predictive or forecasting purposes to analyze the impact of various factors such as the impact of additional testing, and the impact of delayed or deferred fault removal. For such analysis, the solution of the model can be obtained for the desired parameter values. These parameters include the length of the time interval of testing, the fault removal rate, and the duration of the interval for which fault removal is to be deferred.

6. Conclusions

In this paper we have proposed a framework to account for explicit fault removal along with fault reintroduction into the finite failure software reliability growth models. This framework is based on the state-space view of non-homogeneous Poisson processes, and relies on numerical techniques to obtain the mean value function of the NHPP. An analysis of different fault removal strategies is also presented. We describe a method to compute the failure intensity of the software taking into consideration explicit fault removal, which accounts for the faults which have been detected but not removed.

Models with explicit fault removal can be used to obtain realistic estimates of the residual number of faults in the software, as well as the failure intensity of the software and hence guide decision making about the allocation of resources in order to achieve timely release of the software.

Note

1. We note that in this case the state of the software is given by a 2-tuple (n, j) where n represents the number of faults detected and j represents the number of faults pending to be removed. Consequently, the failure rate should be given by $\lambda(n, j, t)$ and the debugging rate should be given by $\mu(n, j, t)$. However, we assume that the failure rate is unaffected by the number of faults pending to be removed and hence is independent of j. Similarly, we assume that the fault removal rate is independent of the number of faults detected, represented by n. As a result, we use the simplified notation of $\lambda(n, t)$ to represent the failure rate and $\mu(j, t)$ to represent the fault removal rate.

References

Dalal, S.R. and Mallows, C.L. 1990. Some graphical aids for deciding when to stop testing software, *IEEE Trans. on Software Engineering* 8(2): 169–175.

Defamie, M., Jacobs, P., and Thollembeck, J. 1999. Software reliability: Assumptions, realities and data, *Proc. of International Conference on Software Maintenance*, September.

Farr, W. 1996. Software reliability modeling survey: *Handbook of Software Reliability Engineering*, ed. M.R. Lyu, pp. 71–117, New York, McGraw-Hill.

Goel, A.L. and Okumoto, K. 1979. Time-dependent error-detection rate models for software reliability and other performance measures, *IEEE Trans. on Reliability* R-28(3): 206–211.

Goel, A.L. 1985. Software reliability models: Assumptions, limitations and applicability, *IEEE Trans. on Software Engineering* SE-11(12): 1411–1423.

Gokhale, S. and Trivedi, K.S. 1998. Log-logistic software reliability growth model, *Proc. of High Assurance Systems Engineering (HASE 98)*, Washington, DC, November, pp. 34–41.

Gokhale, S. and Trivedi, K.S. 1999. A time/structure based software reliability model, *Annals of Software Engineering* 8: 85–121.

Gokhale, S., Philip, T., Marinos, P.N., and Trivedi, K.S. 1996. Unification of finite failure non-homogeneous Poisson process models through test coverage, *Proc. of Intl. Symposium on Software Reliability Engineering (ISSRE 96)*, White Plains, NY, October, pp. 289–299.

Gokhale, S., Marinos, P.N., Trivedi, K.S. and Lyu, M.R. 1997. Effect of repair policies on software reliability, *Proc. of Computer Assurance (COMPASS 97)*, Gatheirsburg, MD, June, pp. 105–116.

Gokhale, S. 2003. Optimal software release time incorporating fault correction, *Proc. of 28th IEEE/NASA Workshop on Software Engineering*, Annapolis, MD, December.

Jelinski, Z. and Moranda, P.B. 1972. Software reliability research: *Statistical Computer Performance Evaluation*, ed. W. Freiberger, pp. 465–484, New York, Academic Press.

Kapur, P.K., Sharma, K.D., and Garg, R.B. 1992. Transient solutions of software reliability model with imperfect debugging and error generation, *Microelectronics and Reliability* 32(1): 475–478.

Kenney, G.Q. 1993. Estimating defects in commercial software during operational use, *IEEE Trans. on Reliability* 42(1): 107–115.

Kremer, W. 1983. Birth and death bug counting, *IEEE Trans. on Reliability* R-32(1): 37–47.

Kulkarni, V.G. 1995. *Modeling and Analysis of Stochastic Systems*. New York, Chapman-Hall.

Levendel, Y. 1990. Reliability analysis of large software systems: Defect data modeling, *IEEE Trans. on Software Engineering* 16(2): 141–152.

Lyu, M.R. and Nikora, A.P. 1992. CASRE—a computer-aided software reliability estimation tool, *CASE '92 Proceedings*, Montreal, Canada, July, pp. 264–275.

Ohba, M. and Chou, X. 1989. Does imperfect debugging affect software reliability growth?, *Proc. of Intl. Conference on Software Engineering*, April, pp. 237–244.

Ramani, S., Gokhale, S., and Trivedi, K.S. 1998. SREPT: Software reliability estimation and prediction tool, *10th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation—Performance Tools 98*, Palma de Mallorca, Spain, September, pp. 27–36.

Sahner, R.A., Trivedi, K.S., and Puliafito, A. 1996. *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*. Boston, Kluwer Academic.

Scheidewind, N.F. 2003. Fault correction profiles, *Proc. of Intl. Symposium on Software Reliability Engineering*, Denver, CO, November, pp. 257–267.

Schneidewind, N.F. 2002. An integrated failure detection and fault correction model, *Proc. of Intl. Conference on Software Maintenance*, December, pp. 238–241.

Sumita, U. and Shantikumar, G. 1986. A software reliability model with multiple-error introduction and removal, *IEEE Trans. on Reliability* R-35(4): 459–462.

Trivedi, K.S. 2001. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. Wiley.

Wood, A. 1997. Software reliability growth models: Assumptions vs. reality, *Proc. of 8th Intl. Symposium on Software Reliability Engineering*, Albuquerque, NM, November, pp. 136–141.

Yamada, S., Ohba, M., and Osaki, S. 1983. S-shaped reliability growth modeling for software error detection, *IEEE Trans. on Reliability* R-32(5): 475–485.

Swapna S. Gokhale received B.E. (Hons.) in electrical and electronics engineering and computer science from the Birla Institute of Technology and Science, Pilani, India in June 1994, and M.S. and Ph.D. in electrical and computer engineering from Duke University in September 1996 and September 1998, respectively. Currently, she is an Assistant Professor in the Department of Computer Science and Engineering at the University of Connecticut. Prior to joining UConn, she was a Research Scientist at Telcordia Technologies in Morristown, NJ. Her research interests include software reliability and performance, software testing, software maintenance, program comprehension and understanding, and wireless and multimedia networking.

Dr. Michael R. Lyu is currently an Associate Professor at the Computer Science and Engineering department of the Chinese University of Hong Kong. He worked at the Jet Propulsion Laboratory as a technical staff member from 1988 to 1990. From 1990 to 1992 he was with the Electrical and Computer Engineering Department at the University of Iowa as an Assistant Professor. From 1992 to 1995, he was a member of the technical staff in the applied research area of the Bell Communications Research (Bellcore). From 1995 to 1997 he was a research member of the technical staff at Bell Labs, which was first part of AT&T and later became part of Lucent Technologies. Dr. Lyu's research interests include software reliability engineering, distributed systems, fault-tolerant computing, wireless communication networks, Web technologies, digital library, and electronic commerce. He has published over 90 refereed journal and conference papers in these areas.

Kishor S. Trivedi received the B.Tech. degree from the Indian Institute of Technology (Bombay), and M.S. and Ph.D. degrees in computer science from the University of Illinois, Urbana-Champaign. He holds the Hudson Chair in the Department of Electrical and Computer Engineering at Duke University, Durham, NC. He also holds a joint appointment in the Department of Computer Science at Duke. His research interests are in reliability and performance assessment of computer and communication systems. He has published over 300 articles and lectured extensively on these topics. He has supervised 37 Ph.D. dissertations. He is the author of a well known text entitled *Probability and Statistics with Reliability, Queuing and Computer Science Applications*, originally published by Prentice-Hall, with a thoroughly revised second edition being published by Wiley. He is a Fellow of the Institute of Electrical and Electronics Engineers. He is a Golden Core Member of IEEE Computer Society.