Incorporating Fault Debugging Activities Into Software ReliabilityModels: A Simulation Approach

Swapna S. Gokhale, *Member, IEEE*, Michael R. Lyu, *Fellow, IEEE*, and Kishor S. Trivedi, *Fellow, IEEE*

***Abstract*—A large number of software reliability growth models have been proposed to analyse the reliability of a software application based on the failure data collected during the testing phase of the application. To ensure analytical tractability, most of these models are based on simplifying assumptions of instantaneous & perfect debugging. As a result, the estimates of the residual number of faults, failure rate, reliability, and optimal software release time obtained from these models tend to be optimistic. To obtain realistic estimates, it is desirable that the assumptions of instantaneous & perfect debugging be amended. In this paper we discuss the various policies according to which debugging may be conducted. We then describe a rate-based simulation framework to incorporate explicit debugging activities, which may be conducted according to the different debugging policies, into software reliability growth models. The simulation framework can also consider the possibility of imperfect debugging in conjunction with any of the debugging policies. Further, we also present a technique to compute the failure rate, and the reliability of the software, taking into consideration explicit debugging. An economic cost model to determine the optimal software release time in the presence of debugging activities is also described. We illustrate the potential of the simulation framework using two case studies.**

***Index Terms*—Debugging, imperfect debugging, software reliability growth models.**

Notation

$\lambda(n,t)$	Failure rate of the software at time t, when the number of faults detected is n
$\mu(j,t)$	Debugging rate of the software at time t, when the number of faults pending to be debugged is j
$m(t)$	Expected number of faults detected & debugged assuming instantaneous debugging by time t
$m_D(t)$	Expected number of faults detected by time t with explicit debugging
$m_R(t)$	Expected number of faults removed by time t with explicit debugging
μ, k, α, β	Parameters of the debugging rate
ϕ	Number of faults which should be detected before debugging can commence in case of deferred debugging
p	Probability of reducing the fault content by 1
q	Probability of no change in the fault content
r	Probability of increasing the fault content by 1
$\lambda'(n,t)$	Failure rate of the software after accounting for debugging activities
C_1	Cost of activities involved in testing the software
C_2	Cost of resolving a failure and fixing the fault in the testing phase
C_3	Cost of fixing a failure in the operational phase
C_4	Cost to customer operations in the field caused by a failure
C_{21}	Cost of resolving a failure and detecting a fault in the testing phase
C_{22}	Cost of debugging a fault in the testing phase
η	Expected execution time of the software release per field site
l	Number of field sites

Manuscript received January 24, 2000; revised May 2005 and June 2005. This work was supported in part by the Research Grants Council of the Hong Kong Special Administrative Region, China, under Project CUHK4205/04E. Associate Editor: M. A. Vouk.

S. S. Gokhale is with the Department of CSE, University of Connecticut, Storrs, CT 06269 USA (e-mail: ssg@engr.uconn.edu).

M. R. Lyu is with the Department of CSE, Chinese University of Hong Kong, Shatin, NT, Hong Kong (e-mail: lyu@cse.cuhk.edu.hk).

K. S. Trivedi is with the Department of ECE, Duke University, Durham, NC 27708 USA (e-mail: kst@ee.duke.edu).

Digital Object Identifier 10.1109/TR.2006.874911

Acronyms[1]

SRGM	Software Reliability Growth Model
NHCTMC	Non Homogeneous Continuous Time Markov Chain

I. Introduction

SOFTWARE reliability is defined as the probability of failure-free software[2] operation for a specified period of time in a specified environment [21]. A large number of software reliability growth models (SRGM) have been proposed to analyse the reliability of a software application based on the failure data collected during the testing phase of the application. A detailed overview of these models can be obtained from elsewhere [7]. To ensure analytical tractability, most of these models assume that a software fault is fixed immediately upon detection, and that no new faults are introduced during the debugging process. In practice, however, the time taken to debug a fault is finite, and this debugging time has a direct impact on the residual number of faults, and hence the reliability of the software application [4], [23], [33]. For example, Fig. 1 shows the cumulative number of open & closed modification requests

[1]The singular and plural of an acronym are always spelled the same.

[2]The terms software, software application, application, system, and software system are used interchangeably in this paper.

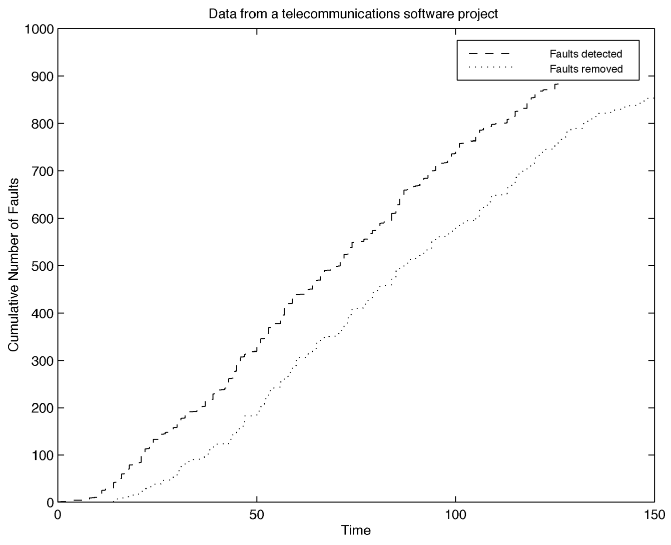


Fig. 1. Data from a telecommunications software.

(MR) as a function of time from a large telecommunications software project during its development of a particular release. Open MR represent the number of faults detected, and closed MR represent the number of faults fixed. As can be seen from the figure, at any given time, the number of faults fixed is less than the number of faults detected. In addition, as suggested by [2], [16], most of the faults encountered by customers are ones reintroduced during debugging of the faults detected during testing. Thus, imperfect debugging also affects the residual number of faults, and in fact at times can be a major cause of field failures & customer dissatisfaction. Thus, to obtain a realistic estimate of the residual number of faults, and the reliability, it is necessary to amend the assumption of instantaneous & perfect debugging.

A number of researchers have recognized this shortcoming, and have attempted to incorporate explicit debugging into some of the software reliability models. Smidts [29], [30] incorporate debugging time into a software reliability model that considers human errors. Levendel [16], and Kremer [15] develop a birth-death model which takes into consideration debugging time. Dalal [3] assumes that the software debugging follows a constant debugging rate, and incorporates debugging into an exponential order statistics software reliability model. Schneidewind [24], [26], [27] incorporates a constant debugging rate into the Schneidewind software reliability model [25]. Gokhale *et al.* [11] incorporates explicit repair into SRGM using a numerical solution. Jones *et al.* [14] consider imperfect debugging in the context of infinite failures models using simulation. However, the objective of Jones *et al.* is to examine the appropriateness of the infinite failures models. Imperfect debugging has also been considered by other researchers [8], [12], [15], [16]. The above research efforts, which seek to incorporate debugging into software reliability models, consider only one or two of the SRGM. Also, they assume that debugging is conducted according to a constant rate. In practice, debugging may be conducted according to different policies, which may be reflective of the scheduling and budget constraints of a project. Also, a number of these efforts do not consider imperfect debugging, and even when it is considered, either instantaneous debugging or a constant debugging rate is assumed. Furthermore, these efforts only provide means to estimate the residual number of faults in the software in the presence of explicit & imperfect debugging. They do not provide methods to estimate the failure rate, reliability, and optimal software release time considering explicit & imperfect debugging.

In this paper, we describe various debugging policies according to which debugging may be conducted. We then incorporate these policies along with the possibility of imperfect debugging into a generic rate-based simulation framework. The framework can be used to consider explicit & imperfect debugging in conjunction with any one of the several prominent software reliability growth models. In addition, we describe a technique to compute the failure rate, and the reliability of the software in the presence of debugging. We also present an economic cost model to determine the optimal release time of the software taking into account debugging activities. We illustrate the potential of the simulation framework using two case studies. In the first case study, we use the data shown in Fig. 1 for the purpose of illustration. Optimal software release time with, and without debugging are also computed & compared for these data. Using the second case study, we illustrate how the simulation framework can be used to analyse the impact of various debugging policies as well as of imperfect debugging on the residual number of faults. Using the results of the two case studies, we discuss how the simulation framework can be used to guide decision making regarding the allocation of resources toward testing & debugging so that a maximum number of faults are detected & debugged in a cost-effective manner.

The layout of the paper is as follows: Section II provides a unification framework for some of the most popular software reliability growth models by casting them as special cases of a generic non homogeneous continuous time Markov chain (NHCTMC) process. It also presents a rate-based simulation procedure for a generic NHCTMC process. Section III describes the various debugging policies, and enhances the simulation procedure for NHCTMC processes to incorporate these debugging policies. A simulation procedure which considers imperfect debugging is also presented in this section. Section IV presents a technique to compute the failure rate, and the reliability in the presence of debugging. Section V presents an economic model to determine the optimal software release time with explicit debugging. Section VI illustrates the simulation framework using two case studies. Section VII offers concluding remarks, and directions for future research.

II. NHCTMC Processes

The objective of this paper is to offer a generic framework to incorporate debugging activities into several software reliability growth models. Toward this end, it is first necessary to unify these models along a common theme. In this section we describe a unification framework for some of the most popular software reliability growth models by casting them as special cases of a NHCTMC process. The discussion in the subsequent subsections is described in the context of a generic NHCTMC

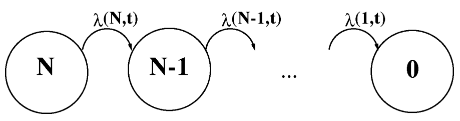


Fig. 2. Pure death NHCTMC.

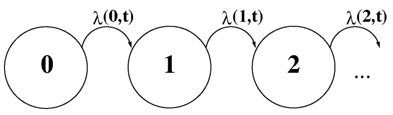


Fig. 3. Pure birth NHCTMC.

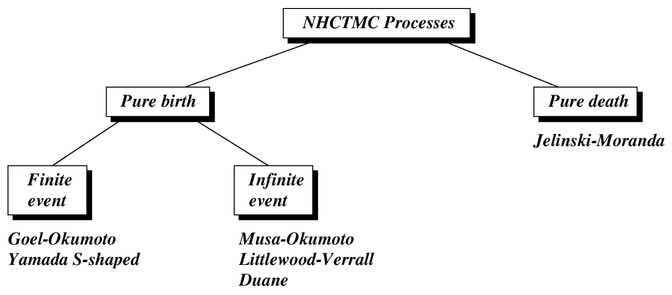


Fig. 4. Classification of SRGM as NHCTMC processes.

process, and is applicable for those software reliability growth models which are special cases of the NHCTMC process.

A. Overview

We consider a class of NHCTMC processes, where the behavior of the stochastic process $\{X(t)\}$ of interest depends only on a rate function $\lambda(n,t)$. The rate function $\lambda(n,t)$ depends on the state n of the system at time t. Let $X(t)$ be the number of "events" occurring in an interval $(0,t)$. "Events" here refers to the number of times the phenomenon of interest occurs (number of failures, for example), and this number n denotes the state of the system. $\{X(t)\}$ can be viewed as a pure death process if we assume that the maximum number of events that can occur in the time interval of interest is fixed, and the remaining number of events forms the state space of the NHCTMC. Thus, the system is said to be in state i at time t if we assume that the maximum number of events that can occur is N, and $N-i$ events have occurred by time t. It can also be viewed as a pure birth process if the number of occurrences of the event forms the state space of the system. In this case, the system is said to be in state i at time t if the event has occurred i number of times up to time t. Let $N(t)$ denote the cumulative number of events in the interval $(0,t)$, and $m(t)$ denote its expectation, i.e., $m(t) = E[N(t)]$. Pure birth processes can be further classified as "finite events," and "infinite events" processes[3], based on the value that $m(t)$ can assume in the limit. In the case of a finite event pure birth process, the expected number of events occurring in an infinite interval is finite (i.e., $\lim_{t\to\infty} m(t) = a$, where a denotes the expected number of events that can occur in an infinite interval), whereas in the case of an infinite event process, the expected number of events occurring in an infinite interval is infinite (i.e., $\lim_{t\to\infty} m(t) = \infty$). Figs. 2 and 3 show the pure death, and pure birth NHCTMC, respectively.

Some of the popular software reliability models are non-homogeneous continuous time Markov chain (NHCTMC)-based. For example, Goel-Okumoto model [8], Yamada S-shaped [35], Musa-Okumoto model [19], Duane model [5], and Littlewood-Verrall model [17] can be cast as pure birth NHCTMC processes, whereas the Jelinski-Moranda model [13] can be cast as a pure death NHCTMC model. The pure birth NHCTMC models can be further classified into finite failures pure birth, and infinite failures pure birth models. The classification of the prominent software reliability models as different types of NHCTMC processes is depicted in Fig. 4.

[3]If the events of interest are failures, then these events become finite failures, and infinite failures

B. Rate-based Simulation

In Section II-A, we described how the stochastic failure process of a software application embodied in many of the software reliability growth models can be described by a NHCTMC. Introducing debugging into this stochastic failure process gives rise to a birth-death process with complex failure & debugging rates, and makes it impossible to obtain closed form expressions. Simulation, on the other hand, can take into account the fault detection as well as the debugging process in an integrated manner. As a result, in this section we provide an overview of rate-based simulation for NHCTMC processes.

We consider a pure birth process, although the technique is equally applicable for a pure death process with suitable modifications. For a pure birth process, the conditional probability that an event occurs in an infinitesimal interval $(t, t+dt)$, provided that it has not occurred prior to time t, is given by $\lambda(0,t)dt$, where $\lambda(0,t)$ is called the event occurrence rate. The probability that the event will not occur in the time interval $(0,t)$, denoted by $P_0(t)$, is given by

$$P_0(t) = e^{-m(t)} \tag{1}$$

where

$$m(t) = \int_0^t \lambda(0,\tau)d\tau \tag{2}$$

When the events of interest are failures, $\lambda(0,t)$ is often referred to as the hazard rate, and q is the cumulative hazard. The cumulative distribution function $F_1(t)$, and the probability density function $f_1(t)$ of the time to occurrence of the first event are then given by [32]

$$F_1(t) = 1 - P_0(t) = 1 - e^{-m(t)}, \tag{3}$$

and

$$f_1(t) = \frac{d}{dt}F_1(t) = \lambda(0,t)e^{-m(t)} \tag{4}$$

```
double single_event(double t, double dt, double (* lambda)(int,double))
{
        int event = 0;
        while (event == 0)
        {
                if (occurs(lambda(0,t)*dt))
                        event++
                t += dt;
        }
        return t;
}
```

Fig. 5. Procedure A: Single event simulation procedure.

```
void recurrent_event(double ta, double tmax, double (* lambda) (int, double),
                     double dt, int *events, int max_events)
{
        double t = ta;
        while ((t <= tmax) && (*events < max_events))
        {
                if (occurs(lambda(*events, t)*dt)
                        ++(*events);
                t += dt;
        }
}
```

Fig. 6. Procedure B: Recurrent event simulation procedure.

Expressions for occurrence times of further events are rarely analytically tractable [32]. These processes are also known as conditional event-rate processes [31].

Rate-based simulation techniques can be used to obtain a possible realization of the arrival process of a NHCTMC. The occurrence time of the first event of a NHCTMC process can be generated using Procedure A expressed in a C-like form shown in Fig. 5. The function $single\_event()$ returns the occurrence time of the event. In the code segment in Procedure A, $occurs(x)$ compares a random number with x, and returns 1 if $random() < x$, or 0 otherwise. The $recurrent\_event()$ procedure presented in Procedure B in Fig. 6 is a simple extension of the $single\_event()$ procedure, and counts the number of occurrences of the event over the interval (t_a, t_{max}). The $events$ parameter must be initialized by the calling program to the number of occurrences prior to time t_a, and it will contain an updated count of the number of occurrences after the function returns.

The use of rate-based simulation to obtain a realization of the failure process modeled by some software reliability growth models has been described by Tausworthe *et al.* [31]. To the best of our knowledge, however, this is the first attempt to provide a unifying framework for the different types of software reliability models along the common theme of NHCTMC processes, and further providing a simulation procedure for the generic NHCTMC process. In recent years, rate-based simulation has also been used for architecture-based software reliability analysis [10].

III. Simulation Framework

In this section, we present a framework based on the rate-based simulation technique to incorporate explicit debugging activities into the black-box software reliability models. Toward this end, we first describe the various policies according to which debugging may be conducted.

A. Debugging Policies

We assume that the testing process is unaffected by debugging activities; that is, testing continues even during debugging. The detected faults are queued to be debugged. The fault detection rate is $\lambda(n,t)$, and depends on the number of faults detected, or time, or both. The debugging rate, or the rate at which the faults are removed, $\mu(j,t)$, also depends on time, or the number of faults queued to be debugged, or both.[4] Thus, at time t, if the number of faults detected is n, and the number of faults queued to be debugged is j, then $n-j$ faults have been debugged.

Debugging may be conducted according to different debugging policies, which are reflective of the budget & scheduling constraints of the project. These debugging policies manifest as different types of debugging rates. The debugging rate $\mu(j,t)$ can be of the following types:

- Constant: This is the simplest possible situation where the debugging rate is independent of the number of faults pending as well as time. The debugging process discussed by Kremer [15], Levendel [16], and Dalal [3] is of this type. The debugging rate $\mu(j,t)$ in this case is given by
$$\mu(j,t) = \mu \tag{5}$$
- Fault-dependent: The debugging rate could depend on the number of faults queued. As the number of faults pending increases, it is likely that additional resources are allocated for debugging, and hence the faults are removed faster, which reflects as a faster debugging rate. If j is the number of faults pending, the debugging rate $\mu(j,t)$ is given by
$$\mu(j,t) = j * k \tag{6}$$
where the constant k can depend on the portion of resources allocated for debugging.
- Increasing time-dependent: The debugging rate could also be time-dependent. Intuitively, the debugging rate is lower at the beginning of the testing phase, and increases as testing progresses or as the project deadline approaches. The debugging rate reaches a constant value beyond which it cannot increase, and this may reflect budget constraints or exhaustion of resources, etc. The time-dependent debugging rate is hypothesized to be of the form
$$\mu(j,t) = \alpha(1 - e^{-\beta t}) \tag{7}$$
for some constants α & β, which reflect the characteristics of a particular project; and for t, the length of the test

[4]In this case, the state of the system 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 debugged. 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 debugged, and hence is independent of j. Similarly, we assume that the debugging 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 debugging rate.

```
void recurrent_event_repair(double tmax, double ta, double dt, int max_events,
                            double (*lambda)(int, double), double (*mu) (int, double),
                            int *events, int *pending, int *removed, int fd_delay,
                            double time_delay, double time_detected[])
{
    double t = ta;
    while ((t < tmax) && (*events < max_events))
    {
        if ((occurs(lambda(*events, t)*dt))
        {
            ++(*events);
            ++(*pending);
            time_detected[*events] = t;
        }
        if (((*pending+*removed) > fd_delay) && (t > time_del))
        {
            if ((*pending > 0)  && (t > time_detected[*removed+1] + time_del))
            {
                if (occurs(mu(*pending, t)*dt))
                {
                    --(*pending);
                    ++(*removed);
                }
            }
        }
        t += dt;
    }
}
```

Fig. 7. Procedure C: Simulation procedure with explicit debugging.

interval. We refer to this as the time-dependent debugging rate #1.

- Decreasing time-dependent: Debugging rate could also be time-dependent in the case of latent faults, which are inherently harder to remove, and can be hypothesized to be

$$\mu(j,t) = \alpha e^{-\beta t} \tag{8}$$

We refer to this as the time-dependent debugging rate #2.

- General time-dependent: Time-dependent debugging rate could also have any other functional form as dictated by the software process for a particular project.

Debugging activities can also be deferred to a later point in time in the case of some software development scenarios, based on the following two constraints:

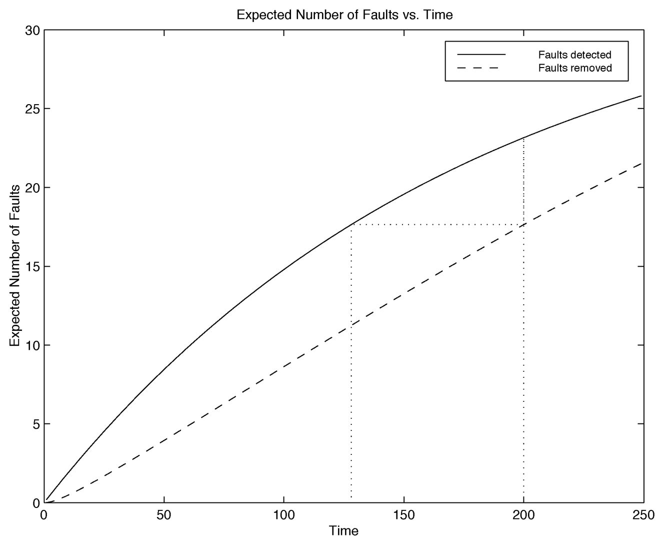
- Debugging can be deferred until a certain number ϕ of faults are detected, and are pending to be debugged.
- Debugging may have to be suspended for a certain average amount of time τ after the detection of a fault, or in other words there is a time lag of τ units between the detection of a fault, and the initiation of its debugging. Debugging may also be delayed for a certain period of time after testing begins.

Once initiated, debugging could proceed according to any one of the policies described above.

The $recurrent\_event()$ simulation procedure shown in Procedure B in Fig. 6 is modified as in Procedure C shown in Fig. 7 to count the number of faults detected as well as debugged for the various debugging policies in the interval (t_a, t_{max}). The calling program must initialize *events* to the number of detected faults, *pending* to the number of faults remaining to be debugged, and *removed* to the number of faults already debugged prior to time t_a. Procedure C is general, and can represent any of the specialized debugging policies or a combination of them by initializing the appropriate parameters to the desired values. For example, fault-dependent delay can be incorporated by setting the parameter fd_delay to the number of faults after which the debugging activity begins. Also, $time\_lag$ can be initialized to reflect the expected time lag between the detection of a fault, and the initiation of its debugging. The procedure at each

```
void imp_fault_removal(double ta, double tmax, double dt, double q, double r,
                       double (*mu)(int, double), int *pending, int *removed)
{
    double t = ta;
    while ((t < tmax) && (*pending) > 0))
    {
        if (occurs(mu(*pending, t)*dt))
        {
            temp = random();
            if (temp < r)
                ++(*pending);
            else
            {
                temp = random();
                if (temp > q+r)
                    --(*pending);
                    ++(*removed);
            }
        }
    }
}
```

Fig. 8. Procedure D: Simulation procedure with imperfect debugging.



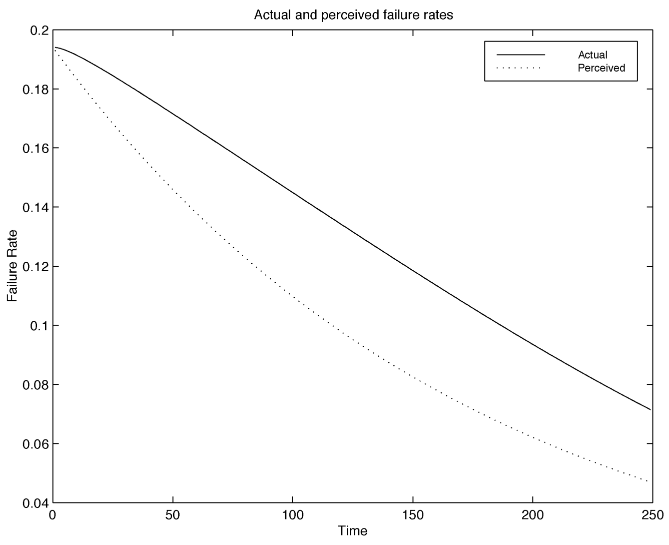


Fig. 9. An example for failure rate adjustment.

time step checks for pending faults if any, and invokes the debugging process. Upon return, the parameters *events*, *pending*, and *removed* contain the updated counts of the number of faults detected, pending to be debugged, and debugged, respectively.

B. Imperfect Debugging

The framework discussed in the previous section is extended in this section to account for fault reintroduction/imperfect debugging based on the following considerations. Whenever a fault is detected, there are three mutually exclusive possibilities to the corresponding debugging 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. Thus $p + q + r = 1$ [15]. It is important to note that simulation does not impose any restrictions on the nature of the debugging process, and fault reintroduction could be used in conjunction with any of the debugging policies described in the previous section. Also, although we have assumed that other cases (additional fault removal or fault reintroduction) are rare & negligible, the simulation framework is sufficiently powerful & generic to consider the reintroduction of multiple faults.

The simulation procedure with imperfections in the debugging activity is presented in Procedure D in Fig. 8. For now, we ignore the testing process; and we assume that a certain number of faults are pending to be debugged, and that we are attempting to debug these pending faults. The calling program must initialize the parameters $pending$ & $removed$ to the number of faults pending to be debugged, and the number of faults debugged, prior to time t_a. These parameters contain the updated counts of these quantities when the function returns.

IV. Computation of Failure Rate and Reliability

In this section, we describe a technique to compute the failure rate & the reliability of the software in the presence of debugging. The estimates of failure rate & the reliability obtained using this approach are valid only if testing is conducted according to the operational profile of the application [20]. Under the ideal assumption of instantaneous & perfect debugging, the expected number of faults debugged is the same as the expected number of faults detected. However, if we take into consideration the time required for debugging, the expected number of faults debugged at any given time is less than the expected number of faults detected as seen in Fig. 9. Thus at any time t, $\lambda(n,t)$, which is the failure rate of the software based on the assumption of instantaneous & perfect debugging, needs to be adjusted to reflect the expected number of faults that have been detected, but not yet debugged. We calculate this adjustment as follows: let $m_R(t)$ denote the expected number of faults debugged by time t, and $m_D(t)$ denote the expected number of faults detected by time t. The approach consists of computing time $t_R \leq t$, such that $m_D(t_R) = m_R(t)$; that is, the time t_R at which the expected number of faults detected as well as debugged under the assumption of instantaneous debugging is equal to the expected number of faults debugged with explicit fault removal. Whereas the perceived failure rate at time t, under the assumption of instantaneous & perfect debugging, is $\lambda(n,t)$, we postulate that the actual failure rate (failure rate after adjustment), denoted by $\lambda'(n,t)$, can be approximately given by $\lambda(n,t_R)$, where $t_R \leq t$. The condition $t = t_R$ represents the situation of instantaneous & perfect debugging. This can be considered as a "roll-back" in time, and is like saying that accounting for fault detection & debugging separately up to time t is equivalent to instantaneous & perfect debugging up to a prior time t_R. Expressions to compute t_R for finite failure NHPP models have been derived elsewhere [9].

We illustrate this approach with the help of an example. Referring to the plot on the left-hand side of Fig. 9, the expected number of faults detected, $m_D(t)$, by time $t = 200$ is 23.16, while the expected number of faults debugged by time t, $m_R(t)$, is 17.64. The failure rate for this particular example is assumed to be that of the Goel-Okumoto model [8], and is given by $\lambda(n,t) = 34.05 * 0.0057 * e^{-0.0057t}$. t_R, computed using these values, is given by 128.1. Thus, the perceived failure rate is 0.062, whereas the actual failure rate after adjustment is 0.093. The approach described here was repeated for every time step

dt, and the plot on the right hand side in Fig. 9 shows the perceived as well as the actual failure rates for the entire time interval. In other words, if the software were released at time $t = 200$, its failure rate taking into account the debugging activities will be 0.093, as opposed to 0.062, which would be its failure rate assuming instantaneous debugging.

V. Optimal Software Release Time

Software testing is an expensive process, and typically consumes about one-third to one-half of the cost of a typical software development project [3]. Overzealous testing can increase the cost of testing, and delay the introduction of a product into the market, in which an early product release may mean the difference between success, and failure. On the other hand, if testing stops too soon, there is a risk of releasing the software with latent bugs, and fixing a fault in a released system is an order of magnitude more expensive than fixing the fault during the test phase. In addition, there is a cost of customer dissatisfaction & loss of goodwill, and of system down time & restoration. Thus there is a tradeoff, and the issue is to find an optimal point at which costs justify the stop decision.

The stopping rule problem has been addressed by several researchers in the literature [3], [6], [22], [28], [35]. As discussed by Ehlrich *et al.* [6], the economic consequences E, involved in stopping testing at time t_s units, or releasing the software at t_s units after test execution, should take into consideration the following costs:

- The cost of testing activities, like running test cases & analysing data, the amount of man-power, and the CPU time spent by the time t_s, or equivalent "testing-effort" [34], [36] is denoted by C_1. The cost associated with test planning & test case development is normally completed before testing, so it is not included in this value.
- The cost of resolving a failure, which consists of activities like opening a modification request, diagnosing the underlying fault, removing the fault, and verifying that the failure no longer occurs, is denoted by C_2.
- The cost of fixing a failure in the operational phase, is denoted by C_3.
- The cost to customer operations in the field is denoted by C_4, which is a function of the failure rate, $\lambda(n, t_s)$ of the software at the release time, the expected execution time η of the software release per field site, and the number of field sites, l.

The economic model is thus given by

$$E = C_1(t_s) + C_2 m(t_s) + C_3 \left(a - m(t_s)\right) + C_4 \left(\lambda(n, t_s)\eta l\right) \tag{9}$$

where a denotes the total expected number of faults in the software, and $m(t_s)$ denotes the expected number of faults detected & hence debugged if the debugging process is assumed to be instantaneous.

These costs are based on software reliability models which assume that the fault is debugged as soon as it is detected, and the debugging process is perfect. The time required to debug a fault, however, cannot be neglected; and hence at any given time, the number of faults debugged will be less than the number of faults detected. Thus, the cost of resolving a failure actually consists of two parts: the cost of opening a modification request & diagnosing the fault that caused a failure, and the cost of removing a fault & verifying that the failure no longer occurs. The former depends on the fault detection process, and the latter depends on the debugging process. Let C_{21} denote the cost associated with the former, and C_{22} with the latter. For a release time t_s, the economic model presented in (9) can be modified to be

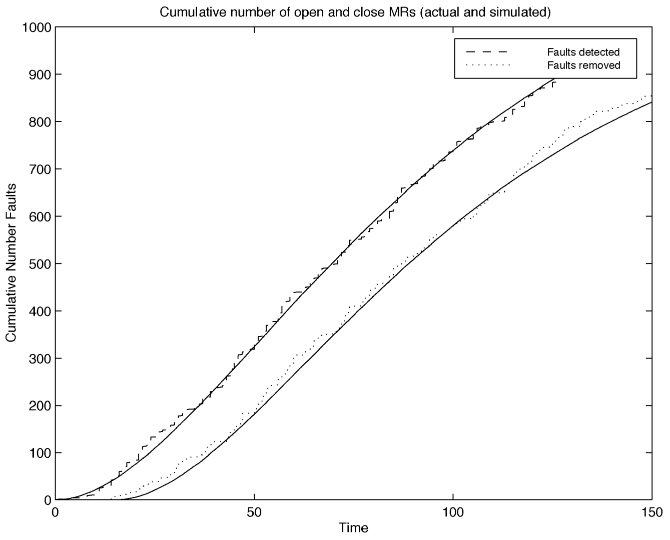


Fig. 10. Actual and simulated MR.

$$\begin{aligned} E = {} & C_1(t_s) + C_{21} m_D(t_s) + C_{22} m_R(t_s) \\ & + C_3 \left(a - m_R(t_s)\right) + C_4 \left(\lambda'(n, t_s)\eta l\right) \end{aligned} \tag{10}$$

where $m_D(t_s)$, and $m_R(t_s)$ denote the expected number of faults detected, and removed respectively, by time t_s. Note that C_4, which is the cost to customer operations in the field, is now a function of the adjusted failure rate $\lambda'(n, t_s)$ of the software.

VI. Illustrative Examples

In this section, we demonstrate the potential of the simulation framework using two case studies. In the first case study, we demonstrate the capability of the framework to simulate the data shown in Fig. 1. We also illustrate the economic cost model using these data. In the second case study, we demonstrate how the rate-based simulation framework can be used to assess the impact of the parameters of the different debugging policies, and of imperfect debugging on the residual number of faults. Through this case study, we also discuss how the results produced by simulation can be used to guide the allocation of resources to achieve the desired reliability in a cost-effective manner.

A. Case Study I

We use the rate-based simulation framework to simulate the open & closed MR data shown in Fig. 1. This figure clearly indicates that the open & closed MR profiles follow two distinct

TABLE I
SIMULATION PARAMETERS FOR CASE STUDY I

Simulation parameter	Notation	Value
Fault detection rate	$\lambda(n,t)$	$1257.5 * (0.0198)^2 * t * e^{-0.0198*t}$
Fault debugging rate	$\mu(j,t)$	$1352.5 * (0.0143)^2 * t * e^{-0.0143*t}$
Initial time	t_a	0.0
Maximum time	t_{max}	500.0
Initial detected faults	$events$	0
Initial pending faults	$pending$	0
Maximum number of events	max_events	1000
Initial removed faults	$removed$	0
Time delay	$time\_delay$	0.0
Fault delay	fd_delay	0
Time step	dt	1

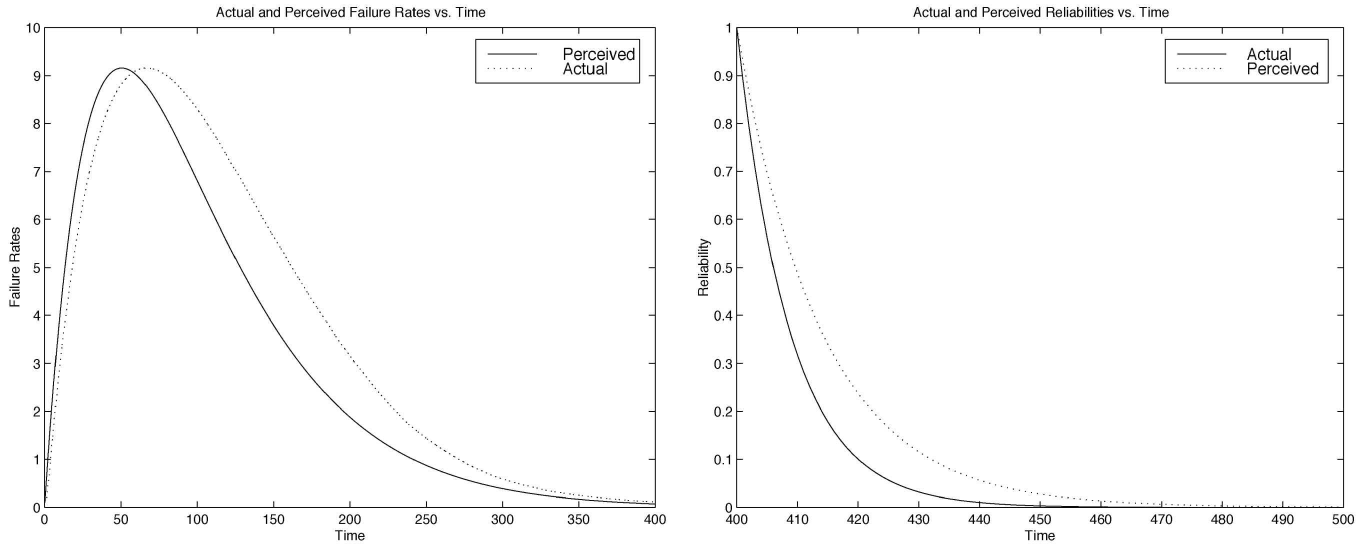


Fig. 11. Actual, and perceived failure rates, and reliabilities for telecommunications data.

processes; and in the conventional software reliability realm, these curves would have to be modeled separately using two different analytical models, which makes the underlying reliability process difficult to understand. Data like the one in Fig. 1 can be easily simulated using a software reliability model for the open MR curve, and a suitable debugging process for the closed MR curve. We simulated the two profiles using the simulation procedure shown in Fig. 7. The open MR profile is simulated using the rate function of the S-shaped model [35], while the closed MR profile is simulated using a time-dependent debugging rate. These two processes were chosen because they provided the best possible fit to the observed curves. The fault detection rate $\lambda(n,t)$ is given by $1257.5 * (0.0198)^2 * t * e^{-0.0198*t}$, and the fault debugging rate $\mu(j,t)$ is given by $1352.5 * (0.0143)^2 * t * e^{-0.0143*t}$. The parameters used in the simulation, along with the fault detection & debugging rates, are summarized in Table I. 1000 simulation runs were conducted, and an average of these runs was obtained. The simulation results are shown in Fig. 10. For the graphs reported in Fig. 10, the standard deviation is within approximately 1% of the mean, and is not shown here to avoid visual clutter.

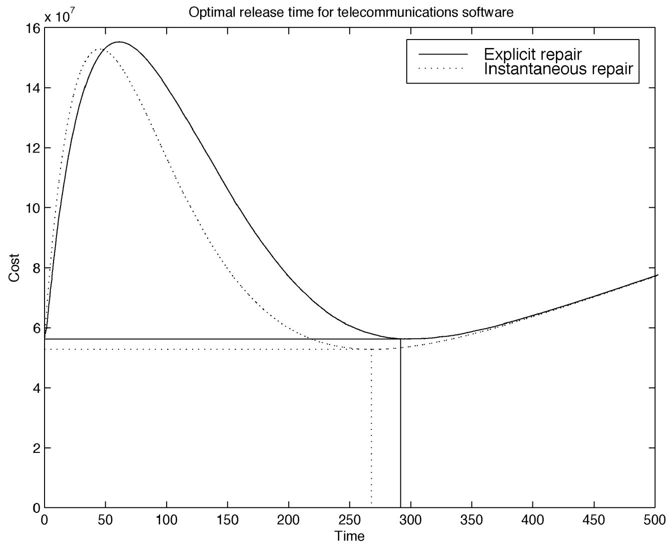


Fig. 12. Optimal software release times with, and without debugging.

The perceived, and the actual failure rate of the software are then computed for the time period from $t = 0.0$ to $t = 400.0$ based on the technique proposed in Section IV, and are shown on the left in Fig. 11. Assuming that both the testing as well

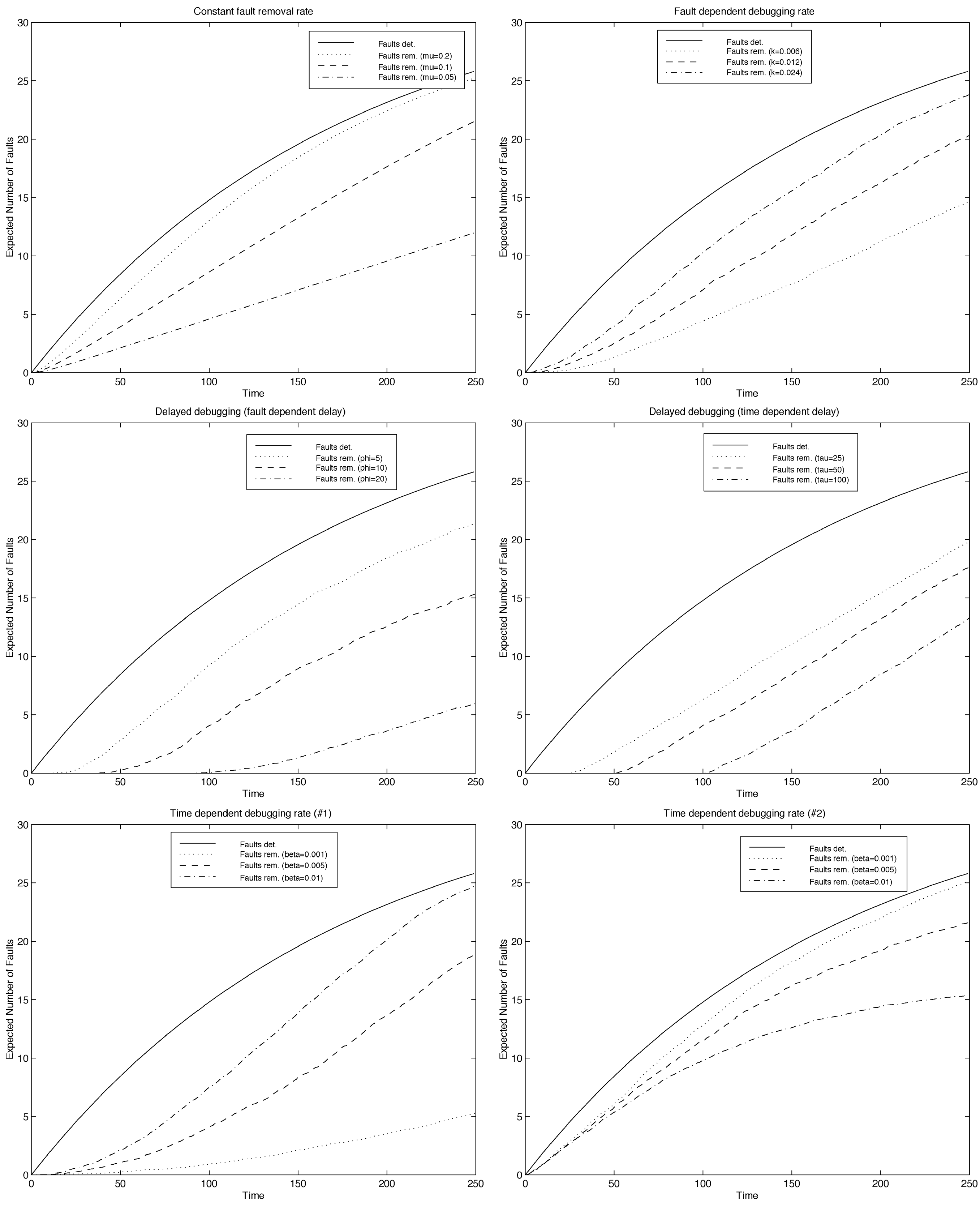


Fig. 13. Expected number of detected & debugged faults for debugging policies.

as debugging activities are stopped at time $t = 400.0$, the reliability of the software under the assumption of instantaneous debugging, and taking into account explicit debugging, is shown on the right in Fig. 11.

TABLE II
SIMULATION PARAMETERS FOR CASE STUDY II

Simulation parameter	Notation	Value
Fault detection rate	$\lambda(n,t)$	$34.05 * 0.0057 * e^{-0.0057*t}$
Initial time	t_a	0.0
Maximum time	t_{max}	300
Initial detected faults	$events$	0
Initial pending faults	$pending$	0
Maximum number of events	max_events	40
Initial removed faults	$removed$	0
Time step	dt	1

We now assess the impact of explicit debugging on the optimal software release time using the telecommunications data. To enable this, we adapt the parameters of the economic cost model described in (9) from Ehlrich *et al.* [6]. The values of the parameters are specified in terms of staff units rather than actual units to preserve the proprietary nature of resource use & cost data. The cost of resolving failures during system test, denoted by C_1, is assumed to be 60 staff units per fault. This cost includes the cost of failure identification, fault diagnosis, and fault removal. C_2, the effort per CPU test-execution unit, is assumed to be 1900 staff units. The effort to resolve failures after system release, C_3, is assumed to be 600 staff units per failure. This cost is based on the observations of Boehm [1], and Dalal *et al.* [3], that the cost of fixing a software fault after system release is an order of magnitude greater than the cost of fixing while testing. C_1, C_2, and C_3 were multiplied by a value of 75 to arrive at the value of the staff, assuming a loaded salary of 75 monetary units per staff-unit. To determine the consequences of field failures, we assume that the system would typically execute 371 CPU units at a single field site before a new version was installed, and that there were six field sites. The economic effect of the system failure was assumed to be 5000 monetary units. For the modified economic cost model which takes into account debugging activities, the cost of resolving a failure during system test, denoted by C_2, is split into two costs, namely, C_{21} which is the cost of failure identification & fault diagnosis, and C_{22} which is the cost of fault removal. We assume both C_{21} & C_{22} to be 30 staff units. Fig. 12 shows the optimal software release times for the data shown in Fig. 1. As can be seen from the figure, the release time as well as the cost at release is higher if debugging activities are explicitly accounted for, instead of assuming instantaneous & perfect debugging.

B. Case Study II

In this case study, for the purpose of illustration, we use the failure rate of the Goel-Okumoto model [8]. The parameters of the rate function of the Goel-Okumoto model for NTDS data [8] were estimated using CASRE [18]. The failure rate used is given by $\lambda(n,t) = 34.05 * 0.0057 * e^{-0.0057*t}$.

1) Parameters of Debugging Policies: In the experiments reported in this section, the objective was to analyse the impact of the debugging policies on the expected number of faults detected & debugged. The parameters used in these experiments are reported in Table II. This table summarizes only those parameters that remain invariant across all the experiments considered in this section. In particular, because the impact of the parameters of the debugging policies is evaluated in these experiments, the parameters pertaining to the debugging rate are different in each scenario, and are not reported in the table. Using the parameters reported in Table II, 1000 simulation runs were conducted using the simulation procedure shown in Fig. 7 in each experiment, and the average of these runs was computed. In each one of these experiments, the standard deviation is within approximately 1% of the mean, and is not shown here to avoid visual clutter.

The expected number of faults detected & debugged for the various debugging policies is shown in Fig. 13. Initially, we simulated the expected number of faults detected & debugged for various values of constant debugging rate, μ. The values of the debugging rate μ were set to be approximately 100%, 50%, 25%, and 12.5% of the maximum fault detection rate. The expected number of faults debugged decreases as μ decreases, as expected. The cumulative fault removal curve has a form similar to the cumulative fault detection curve, and as the debugging rate increases, the fault removal curve almost follows the fault detection curve. We then simulated the expected number of faults debugged as a function of time, when the debugging rate depends on the number of pending faults (6). The expected number of faults debugged is directly related to the proportionality constant k in (6). The debugging rate in this case does not have a closed form expression, but can be computed with a small extension to the simulation procedure. As k increases, the expected number of faults removed increases. The expected number of faults detected & removed as a function of time for time-dependent debugging rate #1 as given by (7) was simulated next. The value of α is held at 0.19, 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 & debugged faults depends on the value of β. As β increases, the debugging rate increases, and the expected number of faults debugged increases. The expected number of faults debugged for time-dependent debugging rate #2 as given by (8) was then simulated. In this case, as the value of β increases, the debugging rate decreases, and the expected number of faults debugged decreases. The expected number of faults detected & debugged as a function of time for delayed fault removal, where the expected delay between the detection of a fault & the initiation of its debugging is τ units, was simulated next. The expected number of faults debugged decreases with increasing τ. The expected number of faults detected & debugged for delayed fault removal, where debugging begins only after a certain number of faults ϕ are accumulated, was simulated next, for different values of ϕ. As ϕ increases, the expected number of faults debugged decreases.

Fig. 13 depicts that, for higher values of debugging rates, the fault removal profile follows the fault detection profile very closely, and the estimates of the residual number of faults based on the ideal assumption of instantaneous debugging are close to the ones with explicit debugging. However, as the debugging rate increases, the debugging resources could be under-utilized. This relation is illustrated in Fig. 14, which shows the utilization of the debugging mechanism for various values of the constant debugging rate μ. As seen from the figure, for lower values of μ,

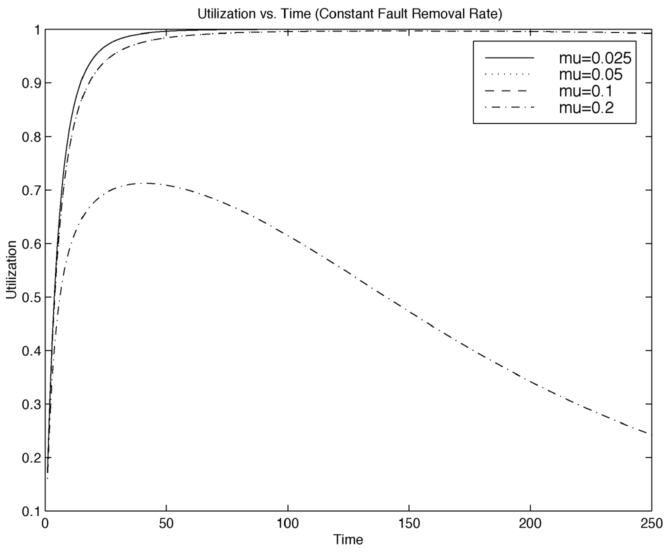


Fig. 14. Utilization for constant debugging rate.

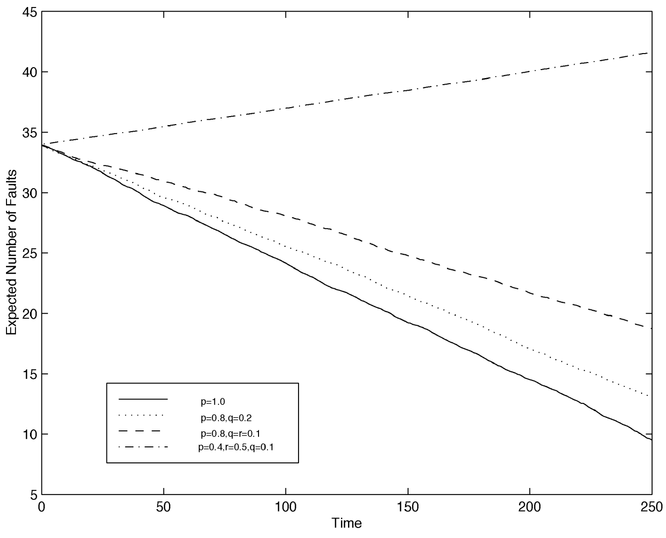


Fig. 15. Expected number of remaining faults.

the debugging mechanism is fully utilized, whereas for higher values of μ, the utilization decreases. Under-utilization of the resources allocated for debugging may not be very cost-effective, especially with increasing budget & deadline constraints facing modern software development organizations. Simulation with explicit debugging can be used to guide resource allocation decisions so that a maximum number of faults are detected & debugged prior to product shipment. Simulation with explicit debugging can thus be used to guide decision making about the allocation of resources to the crucial, perhaps the most important activity of debugging, so that a maximum number of faults are detected & debugged before the shipment of the software product, in a cost-effective manner.

2) Parameters of Imperfect Debugging: For the sake of illustration, the scenario of imperfect fault removal is simulated assuming a constant debugging rate of 0.1, and the number of faults pending for removal is 34. The expected number of faults remaining in the system for different values of p, q, and r [15] simulated using the simulation procedure in Fig. 8 is shown in Fig. 15. The expected number of remaining faults depends on the probability of perfect debugging, p, the probability of introducing one fault, r, and the probability of no change in the fault content, q. As p decreases & r increases, the expected number of faults remaining increases, and beyond a certain threshold of p & r, the fault content of the software may actually increase.

VII. Conclusion and Future Research

In this paper, we incorporated explicit debugging activities along with the possibility of imperfect debugging into the NHCTMC-based black-box software reliability models, using rate-based simulation. We discussed various debugging policies according to which debugging may be conducted. The approach presented here may reflect the testing phase in a software development cycle more closely than the conventional black-box software reliability models. We also presented a technique to compute the failure rate & the reliability of the software, taking into account explicit debugging activities. Further, we described an economic cost model to determine the optimal software release time in the presence of debugging activities. We illustrated the potential of the simulation framework using two case studies.

References

[1] B. W. Boehm and P. N. Papaccio, “Understanding and controlling software costs,” *IEEE Trans. on Software Engineering*, vol. 14, no. 10, pp. 1462–1477, October 1988.
[2] P. J. Boland and N. Chuiv, “Cost implications of imperfect repair in software reliability,” *International Journal of Reliability and Applications.*, vol. 2, no. 3, pp. 147–160, 2001.
[3] S. R. Dalal and C. L. Mallows, “Some graphical aids for deciding when to stop testing software,” *IEEE Trans. on Software Engineering*, vol. 8, no. 2, pp. 169–175, February 1990.
[4] M. Defamie, P. Jacobs, and J. Thollembeck, “Software reliability: assumptions, realities and data,” in *Proc. of International Conference on Software Maintenance*, September 1999.
[5] J. T. Duane, “Learning curve approach to reliability monitoring,” *IEEE Trans. on Aerospace*, vol. AS-2, pp. 563–566, 1964.
[6] W. Ehrlich, B. Prasanna, J. Stampfel, and J. Wu, “Determining the cost of a stop-test decision,” *IEEE Software*, vol. 10, no. 2, pp. 33–42, March 1993.
[7] “Software Reliability Modeling Survey,” in *Handbook of Software Reliability Engineering.* M. R. Lyu, Ed. New York: McGraw-Hill, 1996, pp. 71–117.
[8] A. L. Goel and K. Okumoto, “Time-dependent error-detection rate models for software reliability and other performance measures,” *IEEE Trans. on Reliability*, vol. R-28, no. 3, pp. 206–211, August 1979.
[9] S. Gokhale, “Software failure rate and reliability incorporating repair policies,” in *Proc. of METRICS 04*, September 2004.
[10] S. Gokhale, M. R. Lyu, and K. S. Trivedi, “Reliability simulation of component-based software systems,” in *Proc. of Ninth Intl. Symposium on Software Reliability Engineering (ISSRE 98)*, Paderborn, Germany, November 1998, pp. 192–201.
[11] ——, “Analysis of software fault removal policies using a non homogeneous continuous time Markov chain,” *Software Quality Journal*, pp. 211–230, September 2004.
[12] S. Gokhale, P. N. Marinos, K. S. Trivedi, and M. R. Lyu, “Effect of repair policies on software reliability,” in *Proc. of Computer Assurance (COMPASS 97)*, Gatheirsburg, Maryland, June 1997, pp. 105–116.
[13] Z. Jelinski and P. B. Moranda, “Statistical Computer Performance Evaluation,” in *Software Reliability Research*, W. Freiberger, Ed. New York: Academic Press, 1972, pp. 465–484, chapter.
[14] W. Jones and D. Gregory, “Infinite-failures models for a finite world: a simulation study of fault discovery,” *IEEE Trans. on Reliability*, vol. 43, no. 2, pp. 520–526, 1994.
[15] W. Kremer, “Birth and death bug counting,” *IEEE Trans. on Reliability*, vol. R-32, no. 1, pp. 37–47, April 1983.

[16] L. Levendel, "Reliability analysis of large software systems: Defect data modeling," *IEEE Trans. on Software Engineering*, vol. 16, no. 2, pp. 141–152, February 1990.

[17] B. Littlewood, "A Bayesian reliability growth model for computer software," *Journal of Royal Statistical Society*, vol. 22, no. 3, pp. 332–346, 1973.

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

[19] J. D. Musa, "A theory of software reliability and its application," *IEEE Trans. on Software Engineering*, vol. SE-1, no. 1, pp. 312–327, September 1975.

[20] ——, "Operational profiles in software-reliability engineering," *IEEE Software*, vol. 10, no. 2, pp. 14–32, March 1993.

[21] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability—Measurement, Prediction, Application*. New York: McGraw Hill, 1987.

[22] H. Pham and X. Zhang, "A software cost model with warranty and risk costs," *IEEE Trans. on Computers*, vol. 48, no. 1, pp. 71–75, January 1999.

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

[24] N. Schneidewind, "Assessing reliability risk using fault correction profiles," in *Proc. of Eighth Intl. Symposium on High Assurance Systems Engineering (HASE 04)*, 2004, pp. 139–148.

[25] N. F. Schneidewind, "Software reliability model with optimal selection of failure data," *IEEE Trans. on Software Engineering*, vol. 19, no. 11, pp. 1095–1014, November 1993.

[26] ——, "Modeling the fault correction process," in *Proc. of Intl. Symposium on Software Reliability Engineering*, Hong Kong, November 2001, pp. 185–191.

[27] ——, "An integrated failure detection and fault correction model," in *Proc. of Intl. Conference on Software Maintenance*, December 2002, pp. 238–241.

[28] N. D. Singpurwalla, "Determining an optimal time interval for testing and debugging software," *IEEE Trans. on Software Engineering*, vol. 17, no. 4, pp. 313–319, April 1991.

[29] C. Smidts, "A stochastic model of human errors in software development: impact of repair times," in *Proc. of 10th Intl. Symposium on Software Reliability Engineering*, Boca Raton, FL, November 1999, pp. 94–103.

[30] M. A. Stutzke and C. S. Smidts, "A stochastic model of fault introduction and removal during software development," *IEEE Trans. on Reliability*, vol. 50, no. 2, pp. 184–193, June 2001.

[31] "Software Reliability Simulation," in *Handbook of Software Reliability Engineering*. M. R. Lyu, Ed. New York: McGraw-Hill, 1996, pp. 661–698.

[32] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. Englewood Cliffs, New Jersey: Prentice-Hall, 1982.

[33] A. Wood, "Software reliability growth models: assumptions vs. reality," in *Proc. of Eighth Intl. Symposium on Software Reliability Engineering*, Albuquerque, NM, November 1997, pp. 136–141.

[34] S. Yamada, J. Hishitani, and S. Osaki, "Software-reliability growth with a Weibull test effort: a model & application," *IEEE Trans. on Reliability*, vol. 42, no. 1, pp. 100–105, March 1993.

[35] S. Yamada, M. Ohba, and S. Osaki, "S-shaped reliability growth modeling for software error detection," *IEEE Trans. on Reliability*, vol. R-32, no. 5, pp. 475–485, December 1983.

[36] S. Yamada, H. Ohtera, and H. Narihisa, "Software reliability growth models with testing-effort," *IEEE Trans. on Reliability*, vol. R-35, no. 1, pp. 19–23, April 1986.

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

Michael R. Lyu received the B.S. degree in electrical engineering from National Taiwan University, Hsinchu, Taiwan, and the M.S. degree in computer engineering and Ph.D. degree in computer science from University of California, Los Angeles, in 1981, 1985, and 1988, respectively. He is a Professor in the Computer Science and Engineering Department of the Chinese University of Hong Kong, Shatin, NT, Hong Kong. He worked at the Jet Propulsion Laboratory, Bellcore, and Bell Labs, and taught at the University of Iowa. He is the Editor-in-Chief for two book volumes: Software Fault Tolerance (Wiley, 1995) and the Handbook of Software Reliability Engineering (IEEE and McGraw-Hill, 1996). He has participated in more than 30 industrial projects and helped to develop many commercial systems and software tools. He was frequently invited as a keynote or tutorial speaker to conferences and workshops in U.S., Europe, and Asia. His research interests are software reliability engineering, software fault tolerance, distributed systems, image and video processing, multimedia technologies, and mobile networks. He has published over 200 papers in these areas. Dr. Lyu initiated the International Symposium on Software Reliability Engineering (ISSRE) and was the Program Chair for ISSRE'1996, Program Co-Chair for WWW10 and SRDS'2005, and General Chair for ISSRE'2001 and PRDC'2005. He was also the recipient of the Best Paper Awards in ISSRE'98 and in ISSRE'2003. He was an Associate Editor of the IEEE TRANSACTIONS ON RELIABILITY, the IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, and the *Journal of Information Science and Engineering*.

Kishor S. Trivedi holds the Hudson Chair in the Department of Electrical and Computer Engineering at Duke University, Durham, NC. He has been on the Duke faculty since 1975. He is the author of a well-known text entitled Probability and Statistics with Reliability, Queuing and Computer Science Applications whose revised second edition is under publication. He has also published two other books entitled Performance and Reliability Analysis of Computer Systems (Kluwer Academic Publishers) and Queueing Networks and Markov Chains (John Wiley). He has published over 300 articles, lectured extensively on the area of reliability and performance assessment, and has supervised 39 Ph.D. dissertations. He has made seminal contributions in software rejuvenation, solution techniques for Markov chains, fault trees, stochastic Petri nets, and performability models. He has actively contributed to the quantification of security and survivability. He is a co-designer of the HARP, SAVE, SHARPE, and SPNP software packages that have been well circulated. His research interest is reliability and performance assessment of computer and communication systems. Mr. Trivedi is a Golden Core Member of the IEEE Computer Society.