Software Reliability Analysis Incorporating Fault Detection and Debugging Activities

Swapna S. Gokhale[1*] Michael R. Lyu[2†] Kishor S. Trivedi[3‡]

[1] Bourns College of Engg.
University of California
Riverside, CA 92521
swapna@cs.ucr.edu

[2] Dept. of Computer Science & Engg.
Chinese University of Hong Kong
Shatin NT, Hong Kong
lyu@cse.cukh.edu.hk

[3] Dept. of Electrical Engg.
CACC, Duke University
Durham, NC 27708
kst@ee.duke.edu

Abstract

Software reliability measurement problem can be approached by obtaining the estimates of the residual number of faults in the software. Traditional black-box based approaches to software reliability modeling assume that the debugging process is instantaneous and perfect. The estimates of the remaining number of faults, and hence reliability, are based on these oversimplified assumptions and they tend to be optimistic. In this paper, we propose a framework relying on rate-based simulation technique for incorporating explicit debugging activities along with the possibility of imperfect debugging into the black-box software reliability models. We present various debugging policies and analyze the effect of these policies on the residual number of faults in the software. In addition, we propose a methodology to compute the reliability of the software, taking into account explicit debugging activities. An economic cost model to determine the optimal software release criteria in the presence of debugging activities is described. Finally, we present the high-level architecture of a tool, called SRSIM, for the purpose of automating the simulation techniques described in this paper.

1 Introduction

Software reliability is accepted as a key attribute in software quality, and is defined as the probability of failure-free software operation for a specified period of time in a specified environment [11]. The residual faults in the software system directly contribute to the failure rate, causing software unreliability. Therefore, the problem of measuring 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 specially true for the developer of a commercial off-the-shelf software package that will run on thousands of individual systems. The reliability of a commercial software is important to its users, however, the users never report their reliability experience. They 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, rather than reliability as a measure of software quality [7].

A plethora of black-box software reliability models [4] have appeared in the literature, and most of them, assume that a software fault is fixed immediately upon detection, and no new faults are introduced during the debugging process. This assumption of instantaneous and perfect debugging is impractical [15], and should be amended in order to present more realistic testing scenarios. The time lag between the detection and debugging of a fault is not explicitly accounted for in the traditional software reliability models, as it complicates the failure 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 debug the detected faults. Debugging 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

*This work was done when the author was a graduate student at Duke University

†Supported by the Direct Grant from the Chinese University of Hong Kong

‡Supported by a contract from Charles Stark Draper Laboratory and in part by Bellcore as a core project in the Center for Advanced Computing and Communication

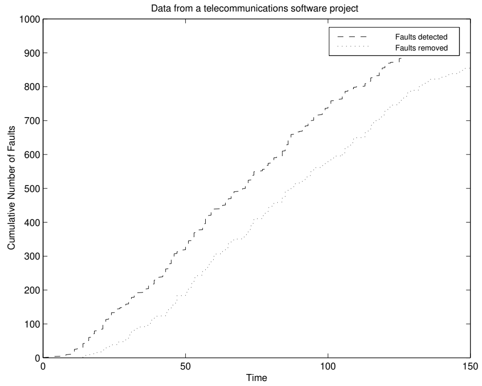

Figure 1. Telecommunications software data

that of perfect debugging. Studies have shown that most of the faults encountered by customers are the ones that are reintroduced during debugging of the faults detected during testing. Thus imperfect debugging 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 [9]. Figure 1 shows cumulative number of open and closed modification requests (MRs) as a function of time from a large telecommunications software project during its development of a particular release. Open MRs represent the number of faults detected, and closed MRs 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. Whereas conventional software reliability models cannot account for this difference between the detected and debugged faults, simulation offers a powerful, yet simple alternative to take this difference into consideration. Data as shown in the Figure 1 can be easily simulated as we will demonstrate in the sequel.

In this paper, we develop procedures to incorporate explicit debugging activity into the black-box models, and analyze the effect of various debugging policies on the residual number of faults. We also present a procedure to incorporate imperfections in the debugging process. We propose a methodology to compute the failure rate of the software in the presence of debugging. An economic cost model to determine the optimal release time of the software taking into account explicit debugging is also presented. The simulation results could be used to guide decision making regarding the allocation of resources towards testing and debugging, so that a maximum number of faults are detected and debugged in a cost-effective manner. Simulation can enable us to assess quickly and safely the implications of an intended resource allocation policy before it is implemented. Controlled experiments to develop new insights into the project and sensitivity analysis to the variations in some of the resource factors can also be easily conducted using simulation. In addition, we describe a software reliability simulation tool (SRSIM) to automate the simulation process. The objective of the tool is to encapsulate the simulation techniques in a systematic, user-friendly environment to allow practical applications.

The layout of this paper is as follows: Section 2 presents an overview of non-homogeneous continuous time Markov chains (NHCTMC) processes, and rate-based simulation for these processes, Section 3 presents the simulation scheme with debugging activities, which includes a discussion of the debugging policies, imperfect debugging, a methodology to compute the failure rate in the presence of debugging, and an economic model to determine the optimal software release criteria, Section 4 presents some numerical results, simulates the open and closed MRs data shown in Figure 1, computes the failure rate for the data shown in Figure 1, and studies the influence of explicit debugging activities on the optimal software release times, Section 5 presents the simulation tool, and Section 6 presents concluding remarks.

2 NHCTMC processes

Some of the popular software reliability models are non-homogeneous continuous time Markov chain (NHCTMC) based, namely, Goel-Okumoto model, Musa-Okumoto model, Yamada S-shaped model, Duane model, and Littlewood-Verrall [13]. Thus the stochastic failure process can be described by a NHCTMC. Introducing debugging into this stochastic failure process gives rise to a birth-death process with complex failure and debugging rates, and makes it impossbile to obtain closed-form expressions. Simulation, on the other hand can take into account the fault detection as well as the debugging process under a unified framework as will be discussed in the sequel. Towards this end, in this section, we present a brief overview of the NHCTMC processes, and rate-based simulation procedure for these processes.

2.1 Overview

We consider a class of non-homogeneous continuous time Markov chain (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 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 $N_0(0,t)$ denote the cumulative number of events in the interval $(0,t)$, and $m_0(0,t)$ denote its expectation, thus $m_0(0,t) = E[N_0(0,t)]$. The notation $m_0(0,t)$ indicates that the process starts at time $t=0$, and the subscript 0 indicates no events have occurred prior to that time. Pure birth processes can be further classified as "finite events" and "infinite events" processes (if the events of interest are failures, then finite failures and infinite failures), based on the value that $m_0(0,t)$ can assume in the limit. In case of a finite event pure birth process, the expected number of events occurring in an infinite interval is finite (i.e., $\lim_{t\to\infty} m_0(0,t) = a$, where a denotes the expected number of events that can occur in an infinite interval), whereas in case of an infinite event process, the expected number of events occurring in an infinite interval is infinite (i.e., $\lim_{t\to\infty} m_0(0,t) = \infty$). Although these definitions are presented for specific initial conditions (the state of the process is 0 at $t=0$), they hold in the case of more general scenarios. In the sequel we assume that the process starts from time $t=0$, and no events have occurred prior to that time, and denote the number of events occurring in the interval $(0,t)$ by $m(t)$.

2.2 Rate-based simulation

Rate-based simulation technique can be used to obtain a possible realization of the arrival process of a NHCTMC. The occurrence time of the first event of a pure-birth NHCTMC process can be generated using Procedure A expressed in a C-like form [13], in Appendix A. 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 $random() < x$, and 0 otherwise. The $recurrent\_event()$ procedure presented in Procedure B (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. Though the procedure is described for a pure-birth process, it is equally applicable to a pure-death process with suitable modifications.

Thus the rate-based simulation technique offers a very attractive mechanism for the study and enhancement of conventional software reliability models.

3 Simulation scheme for debugging activities

In this section we present a framework based on the rate-based simulation technique to incorporate explicit debugging activity into the black-box software reliability models. Towards this end, we first describe the assumptions and the various debugging policies.

3.1 Assumptions and debugging policies

We assume that the testing process is unaffected by debugging activity, i.e., 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. 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.

The debugging rate, $\mu(j,t)$ is assumed to 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 [8], Levendel [9] and Dalal [2] has been of this type. The debugging rate $\mu(j,t)$ in this case is given by:

$$\mu(j,t) = \mu \tag{1}$$

- Fault dependent: The debugging rate could depend on the number of faults queued. As the number of faults pending increases, it is likely that more 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:

$$\mu(j,t) = j * k \tag{2}$$

where the constant k can depend on the portion of resources allocated for debugging.

- 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:

$$\mu(j,t) = \alpha(1 - e^{-\beta t}) \tag{3}$$

for some constants α and β which reflect the characteristics of a particular project, and t is the length of the test interval. We refer to this as the time-dependent debugging rate # 1.

- 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:

$$\mu(j,t) = \alpha e^{-\beta t} \qquad (4)$$

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

- 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 case of some software development scenarios, based on the following two constraints:

- Debugging can be deferred till 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 the fault, or in other words there is a time lag of τ units between the detection of the fault and the initiation of debugging. Debugging can also be delayed for a certain period of time after testing begins, and once initiated it can proceed as per any of the debugging policies described above.

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

The $recurrent\_event()$ simulation procedure shown in Procedure B (Appendix A) is modified as in Procedure C (Appendix A) to count the number of faults detected as well as debugged for the various debugging policies in an 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, $time\_lag$ can be initialized to reflect the expected time lag between the detection of the fault and its debugging, etc. The algorithm at each 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.

3.2 Imperfect debugging

The importance of fault reintroduction has been recognized by several researchers and a few models have been proposed/extended [5, 6, 8, 9] to incorporate imperfect debugging, but most of them are restricted to either instantaneous debugging or constant debugging rate. The framework discussed in the previous section is extended here to account for fault reintroduction based on the following assumptions. 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. We assume other cases (additional fault removal or fault reintroduction) are rare and negligible. Thus $p + q + r = 1$ [8]. It is important to note that simulation does not impose any restrictions on the nature of the debugging process, and reintroduction could be used in conjunction with any of the debugging policies described in the previous section.

The simulation procedure with imperfections in the debugging activity is presented in Procedure D (Appendix A). For the time being we ignore the testing process, and assume that a certain number of faults are pending to be debugged, and we are attempting to debug these pending faults. The calling program must initialize the parameters $pending$ and $removed$ 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.

3.3 Computation of failure rate

In this section we propose a methodology to compute the failure rate of the software in the presence of debugging. Under the idealized assumption of instantaneous and 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 by any given time is less than the expected number of faults detected as seen in Figure 2. Thus at any time t, $\lambda(n,t)$, which is the failure rate of the software based on the assumption of instantaneous and perfect debugging, needs to be adjusted in order 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, such that $m_D(t_R) = m_R(t)$, i.e., time t_R at which the expected number of faults detected as well as debugged under the assumption of instantaneous

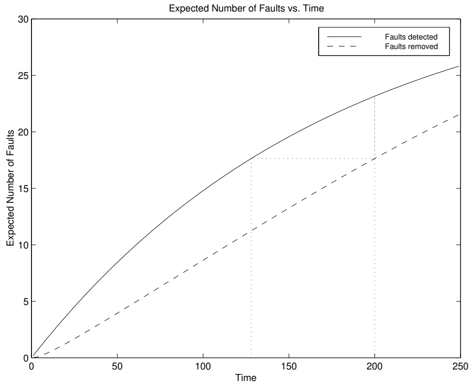

Figure 2. An example of failure rate adjustment

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 and 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 and perfect debugging. This can be considered as a "roll-back" in time, and is like saying that accounting for fault detection and debugging separately up to time t is equivalent to instantaneous and perfect debugging up to time t_R.

We illustrate this approach with the help of an example. Referring to Figure 2, 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 of the Goel-Okumoto model 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 0.062072, whereas the actual failure rate after adjustment is 0.093043. In other words if the software were released at time $t = 200$, its failure rate will be 0.093043. The methodology discussed here can be applied for every time step dt, which will give the actual failure rate during testing.

3.4 Optimal software release criteria

Software testing is an expensive process, and typically consumes about one-third to one-half of the cost of a typical software development project. Overzealous testing can increase the cost of testing, and delay the introduction of the product into a 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 order of magnitude more expensive than fixing the fault during the test phase. In addition, there is a cost of customer dissatisfaction and loss of goodwill, and of system downtime and restoration. Thus there is a tradeoff, and the issue is to find a optimal point at which costs justify the stop decision.

The stopping rule problem has been addressed by several researchers in the literature [2, 3, 12, 17]. As discussed by Ehlrich *et al* [3], the economic consequences E, involved in stopping testing at time t_r units or releasing the software at t_r units after test execution, should take into consideration the following costs:

- The cost of testing activities, like running test cases and analyzing data, the amount of man-power, and the CPU time spent by the time t_r, or equivalently "testing-effort" [16, 18] denoted by C_1. The cost associated with test planning and 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, denoted by C_2.
- The cost of fixing a failure in the operational phase, denoted by C_3
- The cost to customer operations in the field, denoted by C_4, which is a function of the failure rate, $\lambda(n, t_r)$ of the software at the release time, the expected execution time ϕ of the software release per field site, and the number of field sites, l.

The economic model is thus given by:

$$E = C_1(t_r) + C_2 m(t_r) + C_3(a - m(t_r)) \\ + C_4(\lambda(n, t_r)\phi l) \quad (5)$$

where a denotes the total expected number of faults in the software, and $m(t_r)$ denotes the expected number of faults detected and 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 first part being the cost of opening a modification request and diagnosing the fault that caused a failure, and the second part being the cost of removing a fault and verifying that the failure no longer occurs, during the testing phase. The

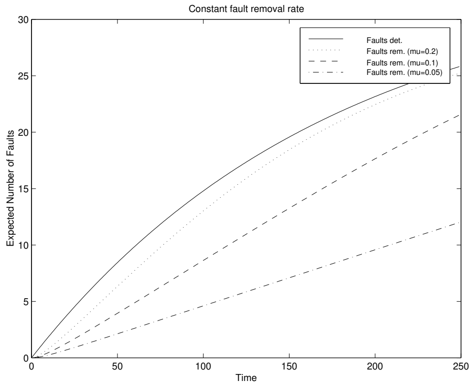

Figure 3. Profile for constant debugging rate

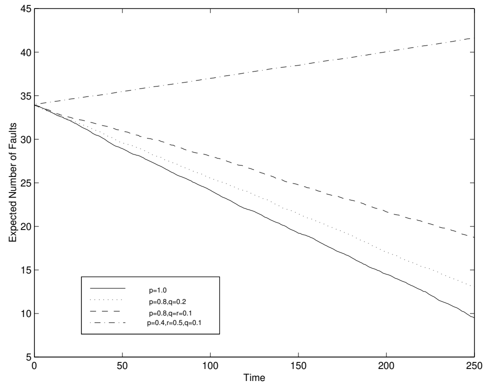

Figure 4. Profile for imperfect debugging

former depends on the fault detection process and the later depends on the debugging process. Let C_{21} denote the cost associated with the former and C_{22} with the later. For a release time t_r, the economic model presented in Equation (5) can be modified to reflect this as follows:

$$\begin{aligned} E = C_1(t_r) + C_{21}m_D(t_r) + C_{22}m_R(t_r) \\ +C_3(a - m_R(t_r)) + C_4(\lambda^{'}(n, t_r)\phi l) \end{aligned} \tag{6}$$

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

4 Numerical results

In this section we demonstrate the utility of the simulation technique to generate the fault detection and debugging profiles, with the help of some case studies. Without loss of generality, we use the failure rate of the Goel-Okumoto model for the case studies in this section. The parameters of the rate function of the Goel-Okumoto model for NTDS data [5] were estimated using CASRE [10]. The failure rate used is given by $\lambda(n, t) = 34.05 * 0.0057 * e^{-0.0057*t}$.

The expected number of faults detected and debugged for the various debugging policies described in Section 3.1 were simulated, however, we present the results only for constant debugging rate due to space constraints. We simulated the expected number of faults detected and 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, and expectedly so. 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.

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 [8] is shown in Figure 4. 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 and r increases, the expected number of faults remaining increases, and beyond a certain threshold of p and r, the fault content of the software may actually increase.

We then simulate the open and closed MRs data shown in Figure 1. Figure 1 clearly indicates that the open and closed MRs profiles follow two distinct 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 Figure 1 can be easily simulated using a software reliability model for the open MRs curve, and a suitable debugging process for the closed MRs curve. We simulated these two profiles, and the results are shown in Figure 5. The open MRs profile is simulated using the rate function of the S-shaped model, while the closed MRs 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 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 methodology proposed in Section 3.3, and are shown on the in Figure 6.

We now assess the impact of explicit debugging on the optimal software release criteria with the help of an example. To enable this, we adapt the parameters of the economic cost model described in Equation (5) from Ehlrich *et*

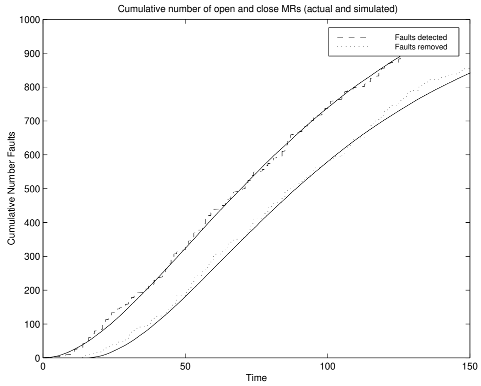

Figure 5. Actual and simulated MRs

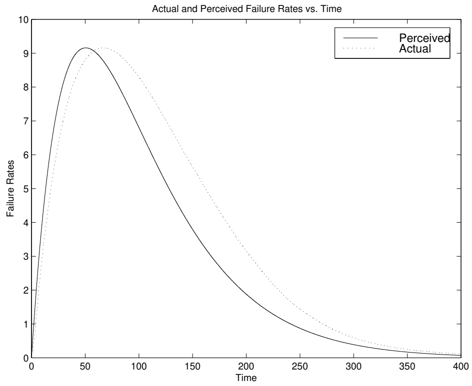

Figure 6. Actual and perceived failure rates

al. [3]. The values of the parameters are specified in terms of staff units rather than actual units due to the proprietary nature of resource use and 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, 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 *at al.* [2], 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, and fault diagnosis, and C_{22} which is the cost of fault removal. We assume both C_{21} and C_{22} to be 30 staff units. Figure 7 shows the optimal software release criteria for the data shown in Figure 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 and perfect debugging.

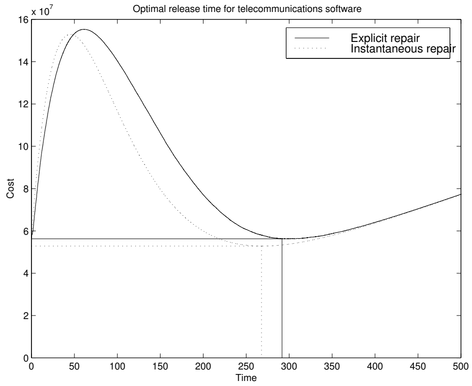

Figure 7. Optimal software release criteria with and without debugging

5 Tool

The simulation framework described in this paper is available in a prototype tool called SRSIM. In its present form, SRSIM is capable of simulating the fault detection profile of all six software reliability models enumerated in Section 2. It can simulate the fault removal profile corresponding to all the debugging policies except delayed debugging. The graphical user interface of the tool is adapted from CASRE [10], and is developed using Tcl/Tk for the X-windows environment. A pull-down menu-driven user interface guides the user through the necessary steps. SRSIM comprises of seven major functional areas: File operations, Parameters to control the simulation, Selection of the models, Parameters of the selected models, Selection of the debugging process, Parameters of the debugging process, and Graphics display window. Figure 8 shows the high level architecture of SRSIM.

SRSIM allows the user to set the following simulation parameters: maximum simulation time, time step used for simulation, units of time, maximum number of faults that can be detected, and number of simulation runs. The parameters maximum number of faults and maximum simulation time control the duration of the simulation runs. It is imperative that these parameters be set by the user before proceeding to select the models and the debugging process to be simulated since no default values of the parameters are supplied. The debugging option is set to be instantaneous by default. The user can then choose to select a debugging process or proceed directly to select the models whose fail-

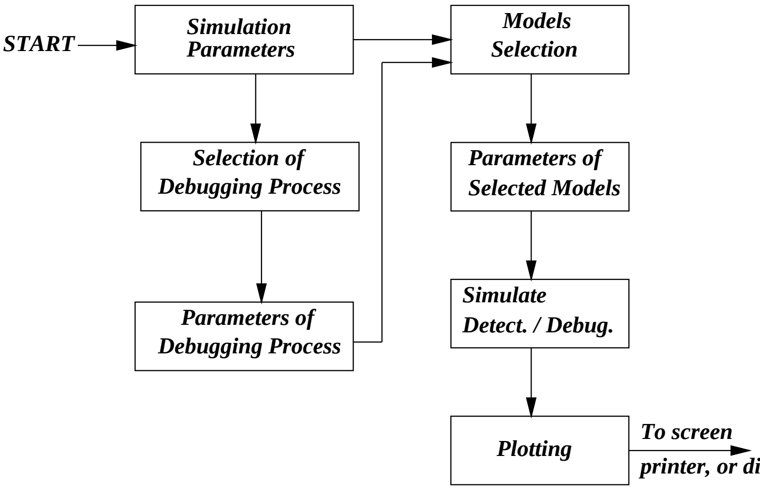

Figure 8. High level architecture of the SRSIM tool

ure occurrence profile is to be simulated. After selecting the debugging process, the user is prompted to provide the parameters of the selected debugging process. The user can then select from one of the following six software reliability models, viz., Littlewood-Verrall, Goel-Okumoto, Musa-Okumoto, Yamada S-shaped, Jelinski-Moranda, and Duane. After the selection of the models, the user is prompted to supply the values of the parameters of the selected models. Once the specification of the parameters corresponding to the fault detection and debugging is complete, simulation results are plotted on a canvas, and the simulated fault detection and removal data is displayed on the workspace on a separated window.

Figure 9 shows a typical window dump from SRSIM. The fault detection is simulated using the rate function of the Goel-Okumoto model, and the debugging rate is set to constant. The left hand side shows the workspace in which the simulated fault detection and debugging data is displayed, while the right hand side shows the plots of these profiles. The menu items on the right hand side canvass include **Plot** which allows the user to save the plot in a file or to draw a plot from a saved file, **Sim Results** which allows the user to display the simulated data in the workspace on the right, **Display** which allows the user to display other metrics of interest like interfailure and interdebugging times, reliability, failure rate, etc., **Settings** provides options for controlling the attributes of the display like color, grey-scale display, etc., and **Help** would provide online assistance. The **File** option on the left hand side allows for file operations, **Edit** allows the editing of the simulated data, **Sim** has sub-menus for setting the simulation parameters, selecting and setting the parameters of the debugging process, model selection and setting the parameters of the selected models, of which model selection menu is shown in the figure, **Setup** allows the user to include external applications (e.g., editors) into the **Edit** menu and **Help** would provide on-line help.

6 Conclusions

In this paper, we have incorporated explicit debugging activities into the NHCTMC based black-box software reliability models, using rate-based simulation. We have discussed various debugging policies and analyzed their effect on the residual number of faults in the software. The approach presented here reflects the testing phase in a software development cycle more closely than the conventional black-box software reliability models. Various metrics of interest like reliability, failure rate, etc., which were computed based on the idealized assumptions of instantaneous and perfect debugging, can now be re-computed taking into account explicit and imperfect debugging, to give more accurate and realistic values. Optimal release times of the software based on the black-box models can now take into account time and resources expended in debugging activities. We have also presented the SRSIM tool which is currently under development, to automate the simulation process. The purpose of SRSIM is to automate the simulation task, and to aid in managerial decision making about the allocation of resources to testing, debugging, and allied activities.

References

[1] B. W. Boehm and P. N. Papaccio. "Understanding and Controlling Software Costs". *IEEE Trans. on Software Engineering*, 14(10):1462–1477, October 1988.

[2] S. R. Dalal and C. L. Mallows. "Some Graphical Aids for Deciding When to Stop Testing Software". *IEEE Trans. on Software Engineering*, 8(2):169–175, February 1990.

[3] W. Ehrlich, B. Prasanna, J. Stampfel, and J. Wu. "Determining the Cost of a Stop-Test Decision". *IEEE Software*, 10(2):33–42, March 1993.

[4] W. Farr. *Handbook of Software Reliability Engineering, M. R. Lyu, Editor*, chapter Software Reliability Modeling Survey, pages 71–117. McGraw-Hill, New York, NY, 1996.

[5] A. L. Goel and K. Okumoto. "Time-Dependent Error-Detection Rate Models for Software Reliability and Other Performance Measures". *IEEE Trans. on Reliability*, R-28(3):206–211, August 1979.

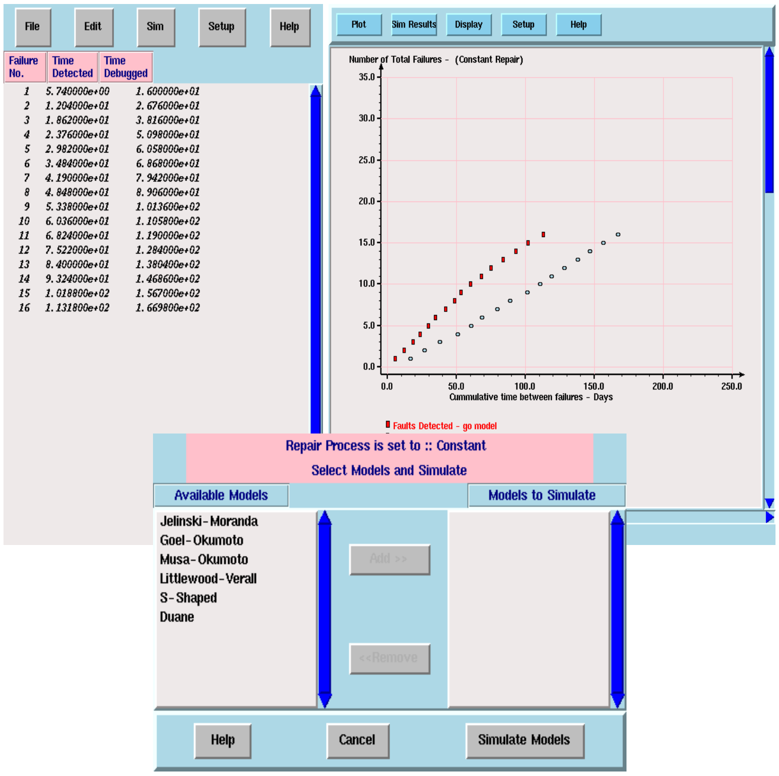

Figure 9. A Typical Window Dump of SRSIM

[6] 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)*, pages 105–116, Gatheirsburg, Maryland, June 1997.

[7] G. Q. Kenney. "Estimating Defects in Commercial Software During Operational Use". *IEEE Trans. on Reliability*, 42(1):107–115, January 1993.

[8] W. Kremer. "Birth and Death Bug Counting". *IEEE Trans. on Reliaability*, R-32(1):37–47, April 1983.

[9] Y. Levendel. "Reliability Analysis of Large Software Systems: Defect Data Modeling". *IEEE Trans. on Software Engineering*, 16(2):141–152, February 1990.

[10] M. R. Lyu and A. P. Nikora. "CASRE-A Computer-Aided Software Reliability Estimation Tool". In *CASE '92 Proceedings*, pages 264–275, Montreal, Canada, July 1992.

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

[12] N. D. Singpurwalla. "Determining an Optimal Time Interval Interval for Testing and Debugging Software". *IEEE Trans. on Software Engineering*, 17(4):313–319, April 1991.

[13] R. C. Tausworthe and M. R. Lyu. *Handbook of Software Reliability Engineering, M. R. Lyu, Editor*, chapter Software Reliability Simulation, pages 661–698. McGraw-Hill, New York, NY, 1996.

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

[15] A. Wood. "Software Reliability Growth Models: Assumptions vs. Reality". In *Proc. of Eigth Intl. Symposium on Software Reliability Engineering*, pages 136–141, Albuquerque, NM, November 1997.

[16] S. Yamada, J. Hishitani, and S. Osaki. "Software-Reliability Growth with a Weibull Test Effort: A Model & Application". *IEEE Trans. on Reliability*, 42(1):100–105, March 1993.

[17] S. Yamada, M. Ohba, and S. Osaki. "S-Shaped Reliability Growth Modeling for Software Error Detection". *IEEE Trans. on Reliability*, R-32(5):475–485, December 1983.

[18] S. Yamada, H. Ohtera, and H. Narihisa. "Software Reliability Growth Models with Testing-Effort". *IEEE Trans. on Reliability*, R-35(1):19–23, April 1986.

Appendix A: Simulation Procedures

Procedure A: Single Event Simulation Procedure

```
/* Input parameters and functions are
assumed to be defined at this point  */
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;
}
```

Procedure B: Recurrent Event Simulation Procedure

```
/* Input parameters and functions are
assumed to be defined at this point  */
int recurrent_event (double ta, double
   tmax, double dt, double (* lambda)
           (int,double), int *events)
{
    double t = ta;
    while ((t < tmax) &&
             (*events < max_events)) {
    if (occurs(lambda (*events, T) * dt))
            ++(*events);
        t += dt;
    }
}
```

Procedure C: Simulation Procedure with Explicit Debugging

```
/* Input parameters and functions are
assumed to be defined at this point  */
void recurrent_event_repair (double tmax,
    double ta, double dt, double
    max_events, double (*m) (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 (m (*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;
        }
    }
}
```

Procedure D: Simulation Procedure with Fault Reintroduction

```
/* Input parameters and functions are
assumed to be defined at this point  */
void imp_fault_removal (double ta,
     double tmax, double dt, double q,
     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);
                }
            }
        }
    }
}
```