# Reliability Simulation of Component-Based Software Systems

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

*Prevalent Markovian and semi-Markovian methods to predict the reliability and performance of component-based heterogeneous systems suffer from several limitations: they are subject to an intractably large state-space for more involved scenarios, and they cannot take into account the influence of various parameters such as reliability growth of the individual components, dependencies among the components, etc., in a single model. Discrete-event simulation on the other hand offers an attractive alternative to analytical models as it can capture a detailed system structure, and can be used to study the influence of different factors separately as well as in a combined fashion on dependability measures. In this paper we demonstrate the flexibility offered by discrete-event simulation to analyze such complex systems through two case studies, one of a terminating application, and the other of a real-time application with feedback control. We simulate the failure behavior of the terminating application with instantaneous as well as explicit repair. We also study the effect of having fault-tolerant configurations for some of the components on the failure behavior of the application. In the second case of the real-time application, we initially simulate the failure behavior of a single version taking into account its reliability growth. Later we study the failure behavior of three fault tolerant systems, viz., DRB, NVP and NSCP, which are built from the individual versions of the real-time application. Results demonstrate the flexibility offered by simulation to study the influence of various factors on the failure behavior of the applications for single as well as fault-tolerant configurations.*

*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

## 1 Introduction

Prevalent approaches to software reliability modeling are black-box based, i.e., the software system is treated as a whole and only its interactions with the outside world are modeled. The black-box based approaches have been shown to be inadequate to capture the behavior of modern systems built using a combination of components off-the-shelf, components developed in-house, and components developed contractually [9]. Component-based systems are thus developed in a heterogeneous fashion, and modeling the failure behavior of such systems using only one of the software reliability growth models [3] can be misrepresentative. With the advancement of object-oriented systems design and web-based development, component-based software systems have become more of a norm than an exception, and thus developing techniques to predict the reliability of such systems taking into account their architecture and failure behavior of the individual components is absolutely essential.

Most of the existing analytical methods to predict the reliability and performance of component-based systems are based on the Markovian assumption [1, 12], and rely on exponential failure- and repair-time distributions. Semi-Markov [14] and Markov regenerative models attempt to relax this assumption in a restrictive manner. Both the Markov and the semi-Markov models are also subject to an intractably large state space. Methods have been proposed to model the reliability growth of the components which cannot be accounted for by the conventional analytical methods [7, 8, 13], but they are also subject to the state-space explosion problem, and/or are computationally very intensive. Some methods have also been proposed to study the effect of correlated versions on various fault-tolerant configurations [10, 15]. However, a single analytical model which takes into account all such features is intractable. Discrete-event simulation, on the other hand, offers an attractive alternative to analytical models as it can capture a detailed system structure, and facilitate the study of the influence of various factors such as reliability growth, various repair policies, correlations among the various versions etc., on dependability measures. The main contribution of this paper lies in demonstrating the flexibility offered by discrete-event simulation to analyze complex and heterogeneous systems through two case studies. One of the case studies models a terminating application, whereas the other one deals with a real-time application with feedback control.

The rest of the paper is organized as follows: Section 2 describes the stochastic failure process, Section 3 presents generalized simulation procedures for a terminating and a real-time control application, Section 4 describes the terminating and the real-time application studied in this paper, and simulates the failure profile of these two applications under different scenarios, and Section 5 presents conclusions and directions for future research.

## 2 The stochastic failure process

In this section we provide a brief overview of the stochastic failure processes, and rate-based simulation for the process.

### 2.1 Overview of NHCTMC processes

We assume that the failure behavior of each component is described by a class of non-homogeneous continuous time Markov chain (NHCTMC), where the behavior of the stochastic process $\{X(t)\}$ of interest depends only on a rate function $\lambda(n,t)$, where $n$ is the state of the component. Some of the popular software reliability models are NHCTMC based, viz., Goel-Okumoto model, Musa-Okumoto model, Yamada S-shaped model, Duane model, Littlewood-Verrall, and Jelinski-Moranda model [18]. The state of the component depends on the number of failures observed from the component. $\{X(t)\}$ can be viewed as a pure death process if we assume that the maximum number of failures that can occur in the time interval of interest is fixed, and the remaining number of failures forms the state-space of the NHCTMC. Thus, the component is said to be in state $i$ at time $t$, if we assume that the maximum number of failures that can occur is $N$, and $N-i$ failures have occurred by time $t$. It can also be viewed as a pure birth process, if the number of failure occurrences forms the state space of the NHCTMC. In this case, the component is said to be in state $i$ at time $t$, if $i$ failures have occurred up to time $t$. Let $N_0(0,t)$ denote the number of failures 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 failures have occurred prior to that time. Pure birth processes can be further classified as "finite failures" and "infinite failures" processes, based on the value that $m_0(0,t)$ can assume in the limit. In case of a finite failures pure birth process, the expected number of failures 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 failures that can occur in an infinite interval), whereas in case of an infinite failures process, the expected number of failures 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.

Analytical closed form expressions can be derived for the failure process of a single component described either by a pure birth and pure death NHCTMC. Now, consider a system composed of $k$ components. In addition to the state of each component, we need to keep track of the amount of execution time experienced by each component separately. The failure rate of the system is denoted by $\Lambda(\mathbf{n},\tau,t)$, where the vector $\mathbf{n} = (n_1, n_2, \ldots, n_k)$, with $n_i$ denoting the number of failures observed from component $i$ up to time $t$, the vector $\tau = (\tau_1, \tau_2, \ldots, \tau_n)$, with $\tau_i$ denoting the amount of time spent in component $i$ up to time $t$, and $t$ is the total time spent executing the system such that $\sum_{i=1}^{k} \tau_i = t$. The failure behavior of such a system can no longer be described by a NHCTMC, and is analytically intractable. Rate-based simulation described in the sequel offers an attractive alternative to study the failure behavior of such systems.

### 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 [18], 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. Procedure A can be easily extended to generate the failure process of a multicomponent system, by keeping track of the number of failures observed from every component, and the execution time experienced by every component.

## 3 Simulation procedures and assumptions

In this section we describe generalized simulation procedures to simulate the failure profile of a terminating application and a real-time application with feedback control.

### 3.1 Terminating application

We assume that the terminating application comprising of $m$ components begins execution with component 1, and terminates upon the execution of component $m$. The architecture of the application is specified by the intercomponent transition probabilities, denoted by $w_{i,j}$. $w_{i,j}$ represents the probability that component $j$ is executed upon the completion of component $i$. The procedure is presented for the case when the failure behavior of each of the components described by one of the software reliability growth models listed in Section 2, but it can be easily modified to take into account the scenarios when the failure behavior is described either by a constant failure rate or a fixed probability of failure. We assume that the application spends $\phi$ units of time in a component per visit. $\phi$ is assumed to be approximately 100 times that of $dt$, which is the time step used in Procedure A in Appendix A. The failure profile of a terminating application, during a single execution, assuming independent failures of the components, can be simulated using Procedure B in Appendix A. The function $generate\_failure(curr\_comp, dt)$, accepts the current component as input, and checks if the component fails in $\phi$ time units, based on the number of failures observed from that component, and the time spent in that component, i.e. based on $\Lambda(\mathbf{n}, \tau, t)$. $generate\_failure(curr\_comp, dt)$ calls Procedure A every $dt$ time steps to check for the occurrence of a failure. The procedure in its current form returns the number of time units necessary for a single execution of the application, and can be easily modified to return the counts of the total number of faults detected, number of faults detected from every component, number of time units spent in every component, etc. The procedure can also be suitably modified to simulate the failure behavior of the application during testing, where test cases are executed in succession for a certain period of time. The time period can be pre-specified if there is a constraint in terms of a deadline, or it can also be determined as the testing progresses based on optimizing some measure such as cost, reliability, failure rate etc., of the individual components or the application as a whole. Various enhancements to this basic model can also be made to simulate other scenarios such as finite debugging time, fault tolerant configurations for some selected or all of the components, dependent failures etc., as will be demonstrated using a case study.

### 3.2 Application with feedback control

In this section we develop a simulation procedure to capture the failure behavior of a real-time application with feedback control. We assume that there are $k$ modes of operation, and the system spends a fixed amount of time in each of these $k$ modes. Upon completion of the $kth$ mode, the application terminates. Computations for certain attributes are performed during each time frame, and the values computed in the $(i-1)st$ time frame are used for the computations in the $ith$ frame. We assume that the application terminates after $l$ time frames, and it spends $l_i$ frames in mode $i$, such that $\sum_{i=1}^{k} l_i = l$. The algorithm is presented for the case when the failure behavior of the components in the application is specified by one of the software reliability growth models listed in Section 2, and can be generalized when the failure behavior is specified either by a constant failure rate or a fixed probability of failure. One of the ways to test such a application is to execute the application for a number of test cases, where each test case consists of $l$ time frames. Procedure C presented in Appendix A simulates the failure behavior of the application during the testing phase, for one test case. For every time frame the procedure determines the mode of operation, and checks if the application fails in that mode of operation in that time frame. Error detection is performed at the end of every time frame. We assume that every module takes $\phi$ time units to complete execution. $\phi$ is assumed to be approximately 100 times that of $dt$, which is the time step used in Procedure A in Appendix A.

These kind of applications are typically used in critical systems and operate in fault tolerant configurations such as DRB, NVP, and NSCP [11]. The simulation procedures for these three fault tolerant systems are reported in [6], and we apply these procedures to a real world case study in this paper.

## 4 Project applications

In this section, we present two project applications, one of a terminating application and the other of a real-time application with feedback control.

### 4.1 Terminating application

We use the terminating application reported in [1] as a running example in this section. The application consists of 10 components, and begins execution with component 1, and terminates upon the execution of component 10. The control-flow graph of the application is shown in Figure 1, and the intercomponent transition probabilities are shown in Table 1. Initially we assume that the failure behavior of the components is specified by their reliabilities. We simulate the expected time to completion of the application, and the reliability of the application. The expected number of time units necessary to complete the execution of the application is 6.450346 (assuming $\phi = 1$), and the reliability of the

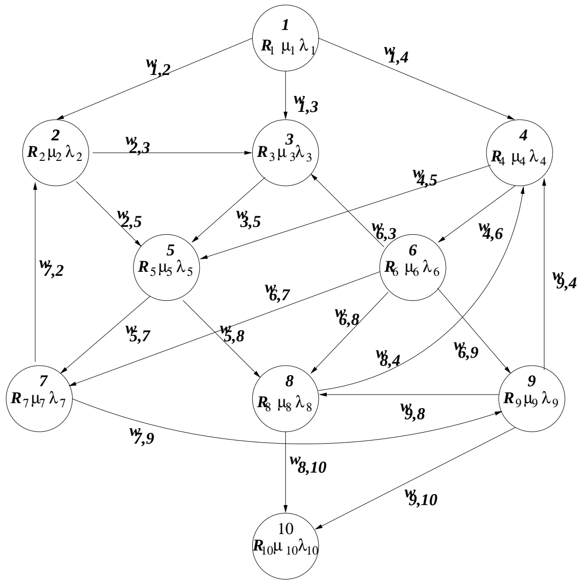


**Figure 1. Control-flow Graph of a Terminating Application**

**Table 1. Transition probabilities in Figure 1**

| | | | |
|---|---|---|---|
| $w_{1,2} = 0.60$ | $w_{1,3} = 0.20$ | $w_{1,4} = 0.20$ | |
| $w_{2,3} = 0.70$ | $w_{2,5} = 0.30$ | | |
| $w_{3,5} = 1.00$ | | | |
| $w_{4,5} = 0.40$ | $w_{4,6} = 0.60$ | | |
| $w_{5,7} = 0.40$ | $w_{5,8} = 0.60$ | | |
| $w_{6,3} = 0.30$ | $w_{6,7} = 0.30$ | $w_{6,8} = 0.10$ | $w_{6,9} = 0.30$ |
| $w_{7,2} = 0.50$ | $w_{7,9} = 0.50$ | | |
| $w_{8,4} = 0.25$ | $w_{8,10} = 0.75$ | | |
| $w_{9,8} = 0.10$ | $w_{9,10} = 0.90$ | | |

application is $0.8811$. These values are very close to the analytical results reported in [1, 5].

We then assign time-dependent failure rates to each of the 10 components. Without loss of generality, we assume that the failure rate of each of the components is given by the failure rate of the Goel-Okumoto model, i.e. $\lambda(n, t) = ae^{-bt}$, where $a$ is the expected number of faults that would be detected from the component if it were to be tested for infinite time, and $b$ is the failure occurrence rate per fault [3]. The parameters $a$ and $b$ are estimated from the field data reported in [4], and the failure rate is given by $34.05 * 0.0057 * e^{-0.0057*t}$. The objective now is to simulate the failure profile of the application given the failure rates of its components, and the transition probabilities between them for a typical testing scenario, where the application is tested by executing several test cases one after the other, for a certain specified duration of $t$ units. The failure profile of the application is given by the expected number of failures observed as a function of time.

Initially, we assume that the failure of any of the components is regarded as failure of the application. The faults are debugged instantaneously upon detection, and the execution of the application proceeds uninterrupted after the instantaneous debugging. Thus the expected total number of faults detected is equal to the expected number of failures of the application. However, in real-life testing scenarios, there is a definite non-zero time interval that is necessary to debug the detected fault [20]. We then introduce finite debugging time into the simulation model, in order to obtain a realistic estimate of the expected total number of faults fixed (repaired or debugged), and the expected number of faults debugged from every component at a given time. Initially we consider sequence dependent failures, where there is no independent debugging for every component. A single debugging facility with a constant debugging rate of $\mu = 0.1$ is shared among all the components, and the faults are debugged in the order of their detection. Next, we assume that each of the 10 components has an independent debugging facility with a debugging rate which is one-tenth of the debugging rate in the case of sequence dependent repair. Thus the faults detected from component $i$ are being debugged at the rate of $\mu_i = 0.01$, for $i = 1, \ldots, 10$. The expected number of faults debugged as a function of time, for $t = 1000$ time units, in case of instantaneous, shared and independent repair is depicted in Figure 2. As seen from Figure 2, the expected total number of faults debugged in case of independent debugging is lower than the expected total number of faults debugged in case of shared debugging. This is due to the fact that all the components are not tested to the same extent, resulting in an unequal expected number of faults being detected from all the components. As a result, the debugging mechanism belonging to certain components would be over utilized, whereas for certain components it will be under utilized. Results like these could aid in making decisions regarding allocation of resources towards testing and debugging of each component so as to maximize some dependability measure of interest. This could also help towards tracking and controlling the schedule and the budget of a project.

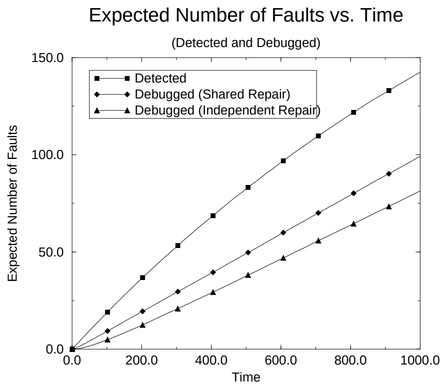


**Figure 2. Shared and independent repair**

We now simulate the failure behavior of the application assuming fault-tolerant configurations for some of the components. Suppose if the correct functioning of component 2 is critical for the successful completion of the application, and we wish to conduct a "what-if" analysis to determine which of the two configurations, namely, $\mathrm{NVP}/1/1$ [11] with three parallel versions of component 2 running on three hardware hosts, or a $\mathrm{RB}/1/1$ [11], i.e., Distributed Recovery Block (DRB) configuration, with two hardware hosts running two recovery blocks [17] is more effective. For both the configurations we assume that the hardware hosts running the software versions/alternates do not fail, and the voter in the NVP system is perfect. The failure rate of the acceptance test (AT) in case of the DRB was set to two orders of magnitude lower than the failure rate of the alternates. We simulate the failure behavior for $1000$ units of time beyond $t = 1000$, with NVP and RB configurations for component 2, for two values of correlation among the two alternates in case of DRB configuration, and two versions in case of NVP configuration, viz., $0.0$ and $0.2$. The expected number of failures observed from the entire application, with different levels of correlation among the alternates/versions of component 2 are shown in Figure 3. As the probability of a related fault among the versions/alternates increases, the expected number of failures of the application increases, due to an increase in the number of failures of component 2. The expected number of failures with the NVP configuration are slightly higher than the expected number of failures with the DRB configuration for component 2. A second way of guarding the application against the failures of component 2, is to cover the failures of component 2 with a certain probability known as coverage [2]. Figure 3 shows the expected number of failures of the application for different values of coverage for component 2. Normally, coverage is the ability of the system to tolerate faults, and is provided by the fault-tolerant capability built into the system. The effectiveness of the fault-tolerant capability to guard against failures will depend on the correlation among the versions which constitute the fault-tolerant configuration. Thus, intuitively, a low correlation among the versions should result in high coverage. As seen from Figure 3, the expected number of faults observed assuming complete independence among the versions constituting the NVP configuration, (correlation of $0.0$ among the versions), and when the coverage is $0.8$, are approximately the same. Thus, we can roughly say that a correlation of $0.0$ for the NVP configuration, corresponds to a coverage of $0.8$. Although, no mathematical relationship exists between correlation and coverage, simulation can be thus used to determine an approximate empirical relationship between two related variables, for a particular system. Studies like these, which help assess the sensitivity of the parameters to the behavior of the system can be of great importance in carrying out "what-if" analysis to enable a choice among a set of alternatives.

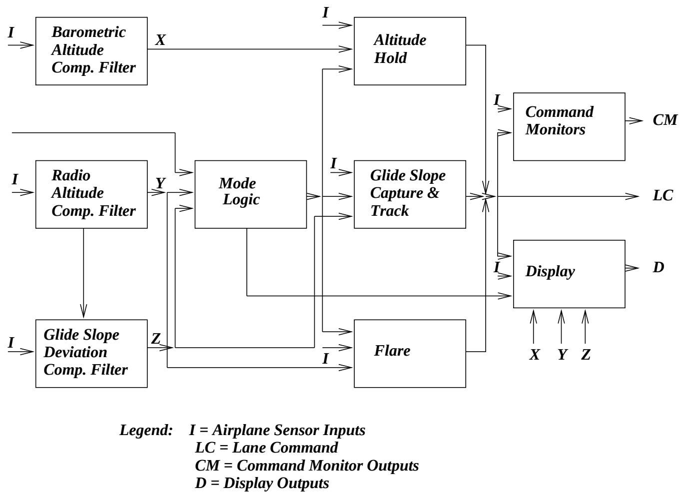


**Figure 4. Autoland system functions and data flow diagram**

### 4.2 Application with feedback control

The real-time application with feedback control under consideration is the *autopilot* project which was developed by multiple independent teams at the University of Iowa and the Rockwell / Collins Avionics Division [16]. The application program is an automatic flight control function (also known as the *autopilot*) for the landing of commercial airliners that has been implemented by the avionics industry. The major system functions of the *autopilot* and its data flow are shown in Figure 4. In this application, the autopilot is engaged in the flight control beginning with the initialization of the system in the altitude hold mode, at a point approximately ten miles ($52800$ feet) from the airport. The initial altitude is about $1500$ feet, and the initial speed is about $120$ knots ($200$ feet per second).

15 versions of the autoland application were developed independently by 15 programming teams at U/Iowa and the Rockwell Avionics Division. The details of the experiment are reported in [16]. The failure data collected from the testing of the 12 accepted versions of the *autopilot*, served as a guideline to assign failure rates to the modules in the application for the purpose of simulation. The number of faults detected from each of the modules during testing was used as an estimate of the parameter $a$ of the Goel-Okumoto model [4]. The parameter $b$ reflects the sensitivity of faults to failures. Arbitrary values for parameter $b$ were chosen in a decreasing order, for the modules Mode Logic, Interface routines, Inner Loop, Glide Slope Control Outer Loop, Altitude Hold Outer Loop, Radio Filter, Barometric Filter, GS Filter, Command Monitor, and Display. There are sixteen different modes in which the data can be displayed, and the complexity of the display increases from mode 1 to mode 16. The display modules were not tested in the original experiment, and hence no failure data were available. As a result, we assign arbitrary

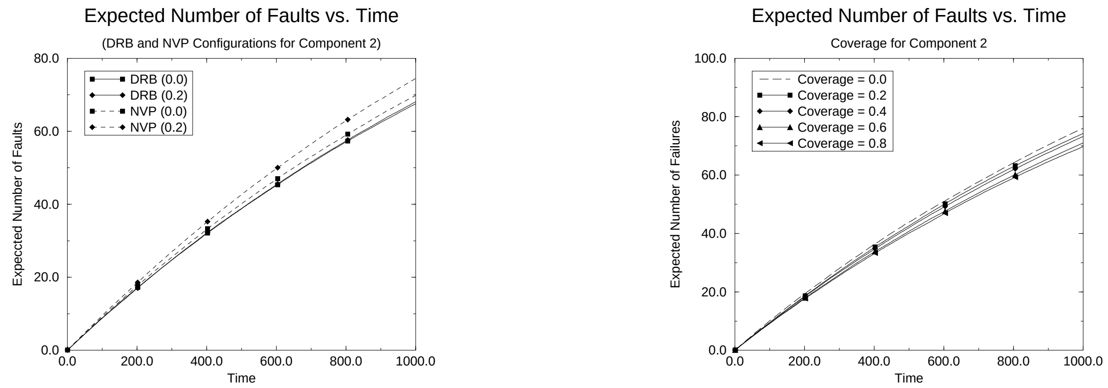


**Figure 3. Operational scenarios of fault tolerance and coverage**

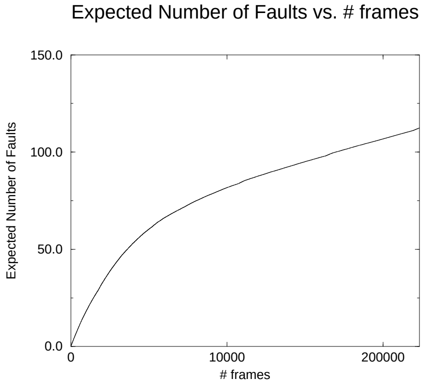


**Figure 5. Failure profile for 4 simulations**

failure rates to the sixteen modes of display module, with a higher failure rate reflecting higher complexity of the display. The failure rate of the display in mode $i$ is assumed to be $3.0*0.00001*(10+(i-1)*5)*e^{(-0.00001*(10+(i-1)*5))}$. The number of faults detected from each of the modules for the 12 versions are summarized in Table 2.

A single run of the $autopilot$ consists of $5582$ frames, with each frame of 50 ms duration for a total landing time of approximately 264 seconds. Thus a single test case in this context consists of $5582$ frames. We assume that software error detection is performed at the end of each frame. For every frame, the display mode is generated randomly, assuming that all 16 modes are equally likely. The expected number of failures of the $autopilot$ for 4 flight simulations from all the 12 versions, which consist of 22328 frames is shown in Figure 5.

Having analyzed the reliability growth of the individual versions in the testing phase, we then proceed to evaluate the failure behavior of three fault-tolerant configurations of the *autopilot* in operation. Thus the failure behavior is analyzed at two levels: the first level being the analysis of the failure behavior during testing, and the second one is to analyze the failure behavior in three FT configurations, namely, RB$/1/1$, NVP$/1/1$, and NSCP$/1/1$ [11] during operation.

We do not address performability issues in this study, and the interested reader is referred to [19] for performability analysis. Each of the $autopilot$ versions comprising the fault-tolerant configuration is now treated as a black-box, and error detection is performed at the end of every test case. A fixed probability of failure per test case is assigned to every $autopilot$ version. We assume that the hardware hosts running the $autopilot$ versions are perfect. In addition, we also assume that the probability of activation of a related fault between any two autopilot versions is fixed for every test case. Initially, the probability of a failure of a single version for every test case is assumed to be $0.0195$, and the probability of a related fault among two software versions is assumed to be $0.0167$. These two values have been guided by a systematic parameter estimation case study for the $autopilot$ reported in [2]. The results obtained using simulation procedures using these parameter values are in close accordance with the ones reported in [2] for all the three systems. We then study the effect of correlation among the alternates$/$versions on the failure behavior of the DRB, NVP and NSCP systems. Starting from a value of $0.0167$ for the correlation, we simulate the expected number of failures that would be observed for $1000$ runs for correlations of $0.3$, $0.4$, $0.6$ and $0.8$. The probability of a related fault among all the versions is assumed to be the same as the probability of a related fault among two versions. The expected number of failures increase with increasing correlation, and in case of DRB, they are less than the number of failures that would be expected from a single version with the same failure probability. However, in case of NVP and NSCP, beyond a certain threshold value of correlation, the expected number of failures that would be observed from a single version with the same failure probability as that of one of the versions constituting the fault-tolerant configuration are in fact lesser than the number of failures that would be observed from the NVP and NSCP configurations. The effect of the failure behavior of the acceptance test (AT) in case of the DRB system and voter in case of NVP and NSCP systems is studied next. The expected number of failures for AT$/$voter failure probabilities of $0.001$, $0.01$ and $0.1$ are simulated. The increase in the number of failures is significant as the failure probability of the AT$/$voter increases from $0.01$ to $0.1$, than the increase observed when the failure probability of the AT$/$voter increases from $0.001$ to $0.01$. Thus for higher values of the

**Table 2. Summary of faults detected from 12 versions of Autopilot**

| Module | b | Faults Detected | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $\beta$ | $\gamma$ | $\epsilon$ | $\zeta$ | $\eta$ | $\theta$ | $\kappa$ | $\lambda$ | $\mu$ | $\nu$ | $\xi$ | $\omega$ |
| Mode Logic | 0.0009 | 2 | 1 | 6 | 5 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Interface Routines | 0.0008 | 1 | 0 | 2 | 0 | 1 | 0 | 2 | 2 | 0 | 0 | 2 | 1 |
| Radio Filter | 0.0003 | 1 | 0 | 2 | 2 | 1 | 0 | 1 | 2 | 0 | 0 | 0 | 0 |
| Barometric Filter | 0.0002 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| GS Filter | 0.0001 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 0 | 4 | 0 | 0 | 0 |
| AHD Outer Loop | 0.0004 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 2 | 0 |
| GSCD Outer Loop | 0.0006 | 2 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 2 | 2 | 1 |
| Flare Outer Loop | 0.0005 | 2 | 2 | 0 | 1 | 0 | 0 | 1 | 2 | 1 | 0 | 0 | 0 |
| Inner Loop | 0.0007 | 2 | 1 | 0 | 1 | 3 | 3 | 2 | 2 | 2 | 2 | 4 | 0 |
| Command Monitor | 0.00009 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

failure probabilities of AT/voter, failures of the AT/voter dominate the system analysis. The expected number of failures for DRB, NVP and NSCP systems, for different correlations, and the failure probabilities of AT/voter are shown in Figure 6.

Failures experienced by a fault tolerant system can be classified into two categories, viz., detected and undetected. Undetected failures lead to an unsafe operation of the system, and it could be highly undesirable if the system is a part of a safety-critical application. The two sources of undetected failures are: a related fault among two or more autopilot versions, and a related or an independent fault in the AT/voter. Next we study the effect of a related fault among two or more software versions on the percentage of undetected faults for the DRB, NVP and NSCP systems. The failure probability of the voter is assumed to be $0.001$ and that of the versions is assumed to be $0.095$. The probability of a related fault among any two versions, and among all the versions was set to $0.0167$, $0.2$, $0.4$, $0.6$, and $0.8$ respectively. The percentage of undetected faults increases significantly in case of NVP, as the correlation increases, whereas the increase in the percentage is not so dramatic in case of NSCP. Thus, NVP is more vulnerable to related faults from the safety view point than NSCP. Comparison with the results obtained for the DRB system also indicates that the NSCP exhibits least vulnerability to related faults among the three fault tolerant systems.

We then study the combined influence of two parameters. viz., the reliabilities of an individual version, and the probability of a related fault among two versions on the failure behavior of the DRB, NVP, and NSCP systems. A single experiment in this case consisted of varying the reliability of the individual versions from $0.90$ to $0.995$ for a fixed value of the probability of a related fault among two $autopilot$ versions. The failure behavior was simulated with correlation among two versions set to $0.01$, $0.05$ and $0.1$, for $1000$ runs of the $autopilot$. Since the AT in case of the DRB system is significantly more complex than the voter in case of the NVP/NSCP system, and in the extreme case could be another alternate, we assume the failure probability of the AT to be the same as one of the alternates. The prob-

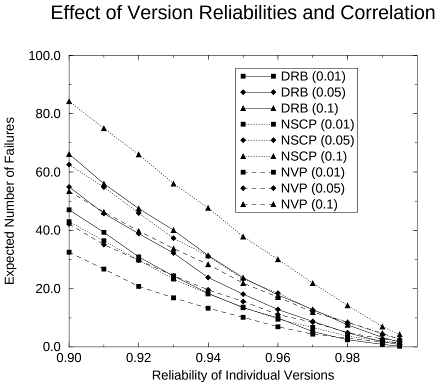


**Figure 7. Effect of reliability and correlation**

ability of a related fault among the AT and the alternates in case of a DRB is set to the same value as the probability of a related fault among the alternates. The probability of a related fault among all the three versions in case of NVP, and three and four versions in case of NSCP is set to $0.01$. Figure 7 shows the effect of these two parameters on the failure behavior of the three fault-tolerant architectures. As expected, for a fixed probability of a related fault among the alternates/versions, and alternates/AT, the expected number of failures for $1000$ runs decreases, as the alternates/versions and AT become more reliable. The figure indicates that the NVP system is more reliable followed by the DRB system, followed by the NSCP system. It is noted however, that the AT in case of the DRB system is assumed to have the same failure probability as that of the alternates, which is a very pessimistic view. From safety point of view, NSCP would be better than DRB and NVP.

Suppose the reliability objective is specified as the expected number of failures that would be tolerated in $1000$ runs of the $autopilot$ system, and that this objective could be achieved by adjusting two parameters, viz., the reliabilities of the individual versions, and the correlation among the versions. Let us assume that the versions are available with a basic reliability of $0.90$. For every additional $1\%$ increase in the reliability, we incur a cost of $c_{1,DRB}$ in case of DRB, $c_{1,NVP}$ in case of NVP, and $c_{1,NSCP}$ in case of NSCP, for every alternate/version. Similarly, the basic probability of a related fault among the two alternates, and alternates and

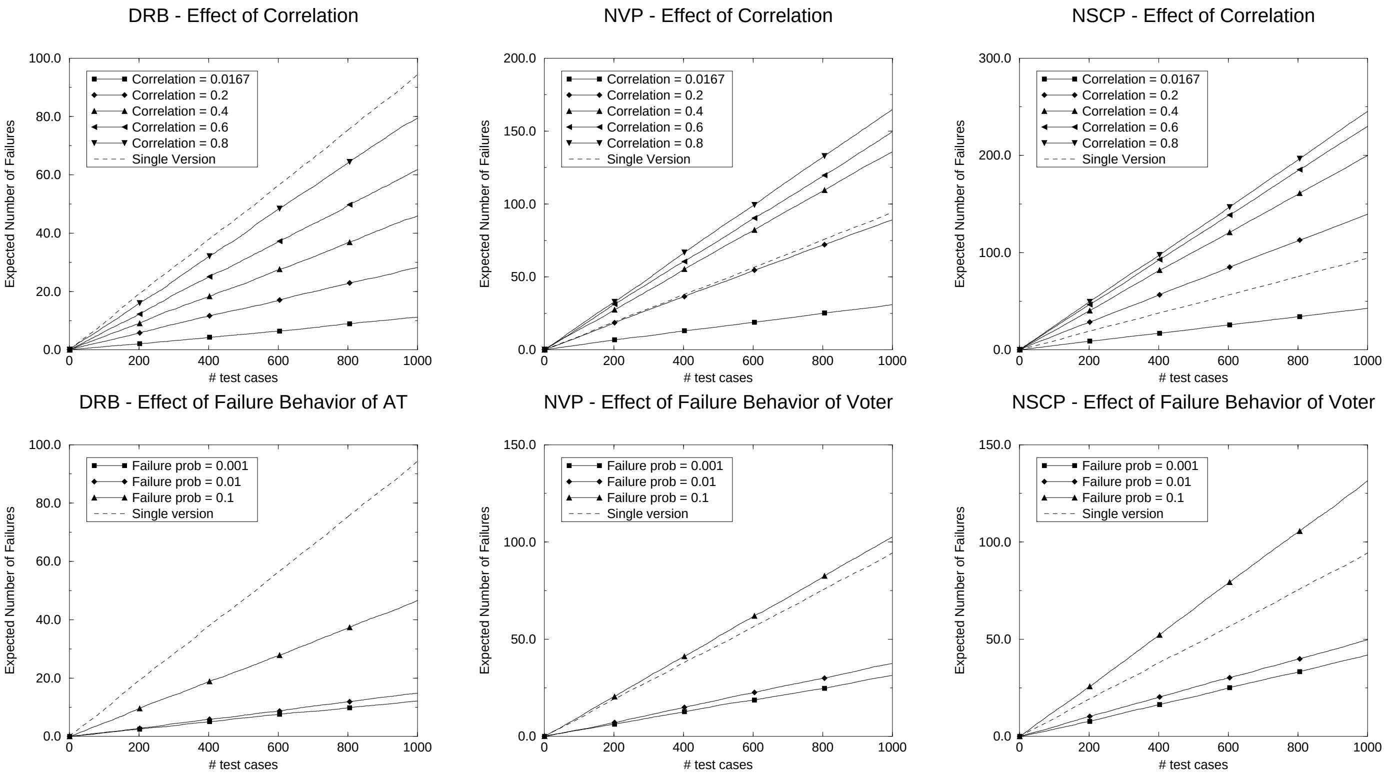


**Figure 6. Effect of correlation and failure behavior of AT/Voter**

AT in case of DRB, and any two versions in case of NVP and NSCP is $0.2$, and it costs $c_{2,DRB}$ for DRB, $c_{2,NVP}$ for NVP, $c_{2,NSCP}$ for NSCP, for every additional reduction by $0.01$. In addition, it costs $c_{AT}$ to build the AT in case of the DRB system. Thus, the cost of the DRB system denoted by $c_{DRB}$, cost of the NVP system denoted by $c_{NVP}$, and the cost of the NSCP system denoted by $c_{NSCP}$ are given by:

$$c_{DRB} = 2 * (R_{DRB} - 0.90) * c_{1,DRB} + c_{AT} + (C_{DRB} - 0.30) * c_{2,DRB} \tag{1}$$

$$c_{NVP} = 3 * (R_{NVP} - 0.90) * c_{1,NVP} + (C_{NVP} - 0.30) * c_{2,NVP} \tag{2}$$

$$c_{NSCP} = 4 * (R_{NSCP} - 0.90) * c_{1,NSCP} + (C_{NSCP} - 0.30) * c_{2,NSCP} \tag{3}$$

In Equations (1), (2), and (3), $R_{DRB}$ denotes the reliability of a single alternate of the DRB system, $R_{NVP}$ denotes the reliability of a single version of the NVP system, and $R_{NSCP}$ denotes the reliability of the individual version of the NSCP system. Similarly, $C_{DRB}$, denotes the correlation among the two alternates, and the alternates and the AT in case of the DRB system, and $C_{NVP}$ and $C_{NSCP}$ denote the correlation among any two versions of the NVP and NSCP systems respectively. For example, if the expected number of allowable faults in 1000 runs is specified to be approximately 20, and we have to choose among three options, viz., (1) DRB with individual version reliabilities of $0.939$, and a correlation of $0.01$ among the alternates, and the alternates and the AT, (2) NVP with version reliabilities of $0.922$ and a correlation of $0.01$ among any two and all three versions, and (3) NSCP with version reliabilities of $0.96$ and a correlation of $0.05$ among any two versions, and a correlation of $0.01$ among all four versions, as indicated in the Figure 7. We note that many more possibilities exist than the ones listed here, however, we consider these three possibilities merely to illustrate the use of simulation to aid decision making under such scenarios. Thus for costs summarized in Table 3, the cost of the DRB alternative is $146.0$ units, NVP alternative is $90.0$ units, and NSCP alternative is $165.0$ units. The choice according to a minimum cost criteria would then be the NVP alternative. This is a very simplistic cost model, and can certainly be enhanced to include other costs. However, we do emphasize, that this was used merely as an illustration to demonstrate the fact that studies like the one described cannot be easily obtained analytically, but can be incorporated into the basic simulation procedures very easily.

**Table 3. DRB, NVP and NSCP systems' costs**

| System | $R_*$ | $C_*$ | $c_{1,*}$ | $c_{2,*}$ | $c_{AT/Voter}$ | Total Cost |
|---|---|---|---|---|---|---|
| DRB | $0.939$ | $0.01$ | $500$ | $300$ | $50$ | $127.5$ |
| NVP | $0.922$ | $0.01$ | $500$ | $300$ | $0$ | $63.0$ |
| NSCP | $0.96$ | $0.05$ | $500$ | $300$ | $0$ | $165$ |

## 5 Conclusions and Future Research

In this paper, we have demonstrated the utility of simulation to study the failure behavior of a software system based on its architecture through two case studies. One case study was of a terminating application and the other was of a real-time application with feedback control. Simulation was also used to assess the sensitivity of the three fault tolerant systems, viz., DRB, NVP, NSCP to various parameters such as the probability of a related fault among the two or more versions, failure behavior of the AT/voter, etc. Combined influence of two parameters on the failure behavior was also studied using simulation. A simple cost model was developed to demonstrate the effectiveness of simulation to study trade-offs and to choose among a set of competing alternatives. Results demonstrate the flexibility offered by simulation to study the influence of various factors on the failure behavior of the applications for single as well as fault-tolerant configurations. Simulation thus holds a lot of promise for modeling a detailed system structure, however, it may be expensive in case of some systems.

As demonstrated in this paper, simulation techniques are not restricted to the assessment of fully functional systems. In fact, one of the biggest advantages of these techniques, is that they can be used to evaluate the reliability and performance, as early as the architecture phase in the life-cycle of the software. This can aid in decision-making such as which components should be re-used, and which should be developed in-house, and allocation of reliabilities to the individual components so that the overall reliability objective is met. It can also help in the identification of reliability and performance bottlenecks, so that remedial actions can be taken before it is too late/ too expensive.

In order for these techniques to be used widely on large scale systems, they need to be encapsulated and made available in a systematic, user-friendly form. Future research includes the design and implementation of a tool, to facilitate reliability and performance assessment of component-based software systems.

## References


[1] R. C. Cheung. "A User-Oriented Software Reliability Model". *IEEE Trans. on Software Engineering*, SE-6(2):118–125, March 1980.

[2] J. Dugan and M. R. Lyu. "System Reliability Analysis of an N-version Programming Application". *IEEE Trans. on Reliability*, R-43(4):513–519, 1994.

[3] 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.

[4] 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.

[5] S. Gokhale. *"Analysis of Software Reliability and Performance"*. PhD thesis, Duke University, Durham, NC, June 1998.

[6] S. Gokhale, M. R. Lyu, and K. S. Trivedi. "Reliability Simulation of Fault-Tolerant Software and Systems". In *Proc. of Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS '97)*, pages 167–173, Taipei, Taiwan, December 1997.

[7] S. Gokhale, T. Philip, and P. N. Marinos. "A Non-Homogeneous Markov Software Reliability Model with Imperfect Repair". In *Proc. Intl. Performance and Dependability Symposium (IPDS '96)*, pages 262–270, Urbana-Champaign, IL, September 1996.

[8] S. Gokhale and K. S. Trivedi. "Structure-Based Software Reliability Prediction". In *Proc. of Fifth Intl. Conference on Advanced Computing (ADCOMP '97)*, pages 447–452, Chennai, India, December 1997.

[9] J. R. Horgan and A. P. Mathur. *Handbook of Software Reliability Engineering, M. R. Lyu, Editor*, chapter Software Testing and Reliability, pages 531–566. McGraw-Hill, New York, NY, 1996.

[10] D. E. Eckhardt. Jr. and L. D. Lee. "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors". *IEEE Trans. on Software Engineering*, SE-11(12):1511–1517, December 1985.

[11] J. C. Laprie, J. Arlat, C. Beounes, and K. Kanoun. "Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures". *IEEE Computer*, 23(7):39–51, July 1990.

[12] J. C. Laprie and K. Kanoun. *Handbook of Software Reliability Engineering, M. R. Lyu, Editor*, chapter Software Reliability and System Reliability, pages 27–69. McGraw-Hill, New York, NY, 1996.

[13] J. C. Laprie, K. Kanoun, C. Beounes, and M. Kaaniche. "The KAT (Knowledge-Action-Transformation) Approach to the Modeling and Evaluation of Reliability and Availability Growth". *IEEE Trans. on Software Engineering*, SE-17(4):370–382, 1991.

[14] B. Littlewood. "A Semi-Markov Model for Software Reliability with Failure Costs". In *Proc. Symp. Comput. Software Engineering*, pages 281–300, Polytechnic Institute of New York, April 1976.

[15] B. Littlewood and D. R. Miller. "Conceptual Modeling of Coincident Failures in Multiversion Software". *IEEE Trans. on Software Engineering*, 15(12), December 1989.

[16] M. R. Lyu and Y. He. "Improving the N-Version Programming Process Through the Evolution of a Design Paradigm". *IEEE Trans. on Reliability*, 42(2):179–189, June 1993.

[17] B. Randell. "System Structure for Software Fault Tolerance". *IEEE Trans. on Software Engineering*, SE-1(2):220–232, June 1975.

[18] 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.

[19] L. A. Tomek and K. S. Trivedi. *Software Fault Tolerance, Edited by M. R. Lyu*, chapter Analyses Using Stochastic Reward Nets, pages 139–165. John Wiley and Sons Ltd., New York, 1995.

[20] 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.

## 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: Simulation Procedure for Terminating Application

```
double time_to_completion(double dt, double w)
{
    int curr_comp = 1; double t = 0;
    while (curr_comp != n)
    {
        generate_failure(curr_comp,dt);
/* Calls procedure A and checks if a failure
occurs based on Lambda(n,tau,t) */
        t += phi;
        temp = random(); sum = 0.0;
        for (i=1;i<=n;i++)
        {
            sum += w(curr_comp,i);
            if (temp <= sum)
                break;
        }
        curr_comp = i;
    }
    return t;
}
```

### Procedure C: Simulation Procedure for Real-Time Application with Feedback Control

```
void control_appl(double dt)
{
    int time_frame_num = 0;
    while (time_frame_num <= n)
    {
     /* Check for failure in mode 1*/
       if (n_1 <= time_frame_num < n_1
                            + n_2)
           check_fail_mode_1(dt);
   /* Check for failure in mode 2 through k*/
    }
}
```