In *Proc. of Pacific Rim Intl. Symposium on Fault-Tolerant Systems'97*, Taipei, Taiwan

Reliability Simulation of Fault-Tolerant Software and Systems

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

[1] Dept. of Electrical and Computer Engg, Center for Advanced Comp. & Comm.
Duke University, Durham, NC 27708-0291 - USA
E-mail: {ssg,kst}@ee.duke.edu
Phone: (919) 660-5269, Fax: (908) 660-5293.

[2] Room No. 2A-413, Lucent Technologies, Bell Laboratories
600, Mountain Avenue, Murray Hill, NJ 07974
E-mail: lyu@research.bell-labs.com
Phone: (908) 582-5366, Fax: (908) 582-5809.

Abstract

Fault tolerance is a survival attribute of complex computer systems and software in their ability to deliver continuous service to their users in the presence of faults. Formulating an analytic model for dependability and performance evaluation of hardware/software fault tolerant architectures can be quite cumbersome. Also, in practice, isolating the effect of various parameters on a system, while holding the others constant requires exploring a variety of scenarios. It is economically infeasible to build several such systems. Simulation offers an attractive mechanism for dependability evaluation and the study of the influence of various parameters on the failure behavior of the system. In this paper, we develop algorithms to simulate the failure behavior of three commonly used fault tolerant architectures, viz., Distributed Recovery Block (DRB), N-Version Programming (NVP) and N-Self Checking Programming (NSCP). We demonstrate the ability of the approach to simulate complex failure scenarios with various dependencies using some illustrative numerical examples.

*Supported in part by Bellcore as a core project in the Center for Advanced Computing and Communication

†Supported in part by National Science Foundation under grant number EEC-9418765.

1 Introduction

The size and complexity of modern software systems embedded in sophisticated hardware has grown more rapidly in the past decade, than our ability to design, implement, test and maintain them. Faults in a computer system are inevitable as the system complexity grows, and hence computer systems are often designed to tolerate both software and hardware faults, by configuring multiple software versions on redundant hardware. Fault tolerance is the survival attribute of computing systems or software in their ability to deliver continuous service to their users in the presence of faults [2].

Dependability and performance modeling of fault tolerant software has been done extensively [1, 6], for the quantitative evaluation of their relative and absolute merits. Most of these techniques do not explicitly consider hardware failures. Dugan et al. [4] model fault tolerant architectures providing a unified tolerance to both hardware and software faults in a hierarchical manner. Formulating an analytical model of a system, which employs both hardware and software fault tolerance, can be quite cumbersome. Rate-based simulation can offer an attractive mechanism to study the combined influence of hardware and software failures, and the possible interactions between them on the overall failure behavior of a system. Also, the ultimate success of modeling is governed by the availability of comprehensive, complete and consistent data sets. Fault tolerant systems are inherently complex due to various dependencies between the software and hardware components, and hence collection of comprehensive and homogeneous data sets for such systems is a formidable task. Also, in practice, isolating the effect of various parameters on a system, while holding the others constant requires exploring a variety of scenarios. It is economically infeasible to build several systems with different values/levels of the factors of interest. Simulation can also provide a viable mechanism to supply carefully controlled, homogeneous data sets, and to study the overall failure behavior of a system as well as the influence of various parameters/factors on the failure behavior.

The layout of the paper is as follows: Section 2 presents an overview of rate-based simulation technique for non-homogeneous continuous time Markov chains (NHCTMCs) and briefly describes the three fault tolerant architectures studied here, Section 3 discusses various assumptions based on which the algorithms are developed, Section 4 describes combinations of software and hardware failures that could lead to a system failure, Section 5 presents some illustrations to demonstrate the ability of simulation to study the failure behavior, and Section 6 concludes the paper and discusses directions for future research.

2 Background

2.1 Simulation for NHCTMCs

In this section we provide an overview of rate based simulation technique which forms the basis of this paper. The failure behavior of an individual component can be described by a process belonging to a class of non-homogeneous continuous time Markov chains (NHCTMCs). The stochastic process of interest, $\{X(t)\}$, is the number of faults activated in the component, and depends only on a rate function $\lambda(n, t)$, where n denotes the state of the system[1] and is the number of faults detected upto time t. The conditional probability that an event occurs in the infinitesimal interval $(t, t+dt)$ is given by $\lambda(n, t)dt$. If we assume that the number of faults detected at $t = 0$ is 0, then the state of the system is 0 at time $t = 0$, the fault detection rate is given by $\lambda(0, t)$, and the probability that a fault will not be detected in the time interval $(0, t)$, denoted by $P_0(t)$, is given by:

$$P_0(t) = e^{-m_0(0,t)} \tag{1}$$

[1]System in this section is a single component

```
/* 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(n,t) * dt))
                event ++;
        t += dt;
        }
        return t;
}
```

Figure 1. Single Event Simulation Routine

where

$$m_0(t_0, t) = \int_{t_0}^{t} \lambda(0, \tau) d\tau \tag{2}$$

$\lambda(0, t)$ is often referred to as failure intensity, since the events of interest are failures, and $m_0(0, t)$ is the mean value function. The subscript 0 on $m_0(0, t)$ indicates that no failure have occurred prior to time $t = 0$. 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 [13]:

$$F_1(t) = 1 - P_0(t) = 1 - e^{-m_0(0,t)} \tag{3}$$

and

$$f_1(t) = \frac{d}{dt} F_1(t) = \lambda(0, t) e^{-m_0(0,t)} \tag{4}$$

Expressions for occurrence times of further events are rarely analytically tractable [13]. These processes are also known as conditional event-rate processes [11].

The occurrence time of the $(n+1)^{st}$ event of the NHCTMC based process described above can be generated (sampled) using the C-like routine shown in Figure 1 [11], The function *single_event*() returns the occurrence time of the $(n+1)^{st}$ event. In the routine above, $occurs(x)$ compares a random number with x, and returns 1 if $random() < x$, and 0 otherwise. This routine is the basis of all the algorithms developed in this paper.

2.2 Fault Tolerant Architectures

In this section we briefly discuss the three system architectures, viz., Distributed Recovery Block(DRB), N-Version Programming(NVP) and N Self-Checking Programming (NSCP), studied in this paper. Each system is characterized by the number of software variants, the number of hardware replications, and the decision algorithm.

2.2.1 Distributed Recovery Block (DRB)

The recovery block (RB) [10] approach to software fault tolerance consists of a set of diverse program versions called alternates, along with an error detection routine known as the Acceptance Test (AT). The acceptability of a computation performed by the primary is determined by an acceptance test. If the results are deemed unacceptable, the state of the system is rolled back to that on entry to the RB and a spare is executed. This process is repeated until an acceptable result is delivered or no more alternates are available. Alternates are designed to provide the same functionality as the primary but deliberately as independent as possible. The Distributed Recovery Block (DRB) proposed by Kim et. al [7] provides a way of combining hardware redundancy with recovery blocks. The RB/1/1 [8] structure used in this study and is obtained by the duplication of RB composed of two alternates and an acceptance test on two hardware components.

2.2.2 N-Version Programming (NVP)

The NVP method employs N independently developed, functionally equivalent software versions, from the same initial specification, to perform the same task [3]. The programming efforts are carried out by N individuals or groups that do not interact with respect to the programming process, so that the versions are as diverse as possible. These versions are executed in parallel using identical inputs, and their outputs are collected and evaluated by a decider/voter/adjudicator. In the event that all the outputs do not match, the output produced by the majority of the versions is taken to be correct. The NVP/1/1 [8] system studied here is assumed to have three identical hardware components, each running a distinct software version.

2.2.3 N Self-Checking Programming (NSCP)

The NSCP/1/1 [8] architecture considered in this study is comprised of four software versions and four hardware components, each grouped in two pairs, essentially dividing the system in two halves. The hardware pairs operate in hot standby redundancy with each hardware component supporting one software version. The version pairs form self-checking software components, so that error detection is done by comparison. The four software versions are executed and the results of the two versions executing in each half of the system are compared. If either pair of results do not match, they are discarded and only the remaining two are used. If the results do match, the results of the two pairs are then compared. A hardware fault causes the software version running on it to produce incorrect results, as would a fault in the software version itself. This results in a discrepancy in the output of the two versions, causing that pair to be ignored.

3 Simulation Assumptions

In this section, we describe the assumptions regarding the failures of the software versions, permanent and transient hardware failures, failures of the acceptance test/voter, and coincident failures among versions. The simulation algorithms are based on these assumptions.

- **Task Computation:** We assume that the computation being performed is a task or (a set of tasks) that is repeated periodically. A set of sensor inputs is gathered and analyzed and a set of actuations are produced. Each repetition of the task is independent. We do not address timing or performability issues in this study. The interested reader is referred to [12] for a performability analysis.

- **Failures of software versions and AT/Decider:** We assume that the failure process of the versions/alternates and acceptance test/voter can be described by the failure intensity function associated with one of the six software reliability growth models [11]. Most of the existing approaches to dependability modeling of fault tolerant systems either assign a fixed failure probability or a constant failure rate to the software, except the one by Kanoun et al [6]. Simulation can easily accommodate reliability growth of the software versions, as we will see in the sequel.

- **Coincident errors:** When two software versions fail, they produce either similar or different erroneous results. We use the Arlat/Kanoun/Laprie [1] terminology for software failures and assume that similar erroneous, or identical-and-wrong (IAW) results [9] are caused by related software faults, and different erroneous results are caused by unrelated or independent software faults. We also assume that related and unrelated software faults are mutually exclusive.

- **Permanent hardware faults:** The rate of occurrence of a permanent hardware fault is assumed to be time independent.

- **Transient hardware faults:** They are modeled separately from permanent hardware faults. A transient hardware fault is assumed to upset the software running on the processor and produce an erroneous result which is indistinguishable from an input activated software error. We assume that the lifetime of a transient hardware fault is shorter when compared to the length of task computation. We assume that a hardware transient fault occurs with a fixed probability during each time frame.
- **Fault Treatment:** No fault treatment mechanisms are employed to make a faulty software version passive. Should a version produce an incorrect result as detected by the acceptance test/voter, it is still kept in the system architecture and supplied with new input data [1].

Most of the assumptions described above, except the one which assumes that related and unrelated faults are mutually exclusive, are the same as in [4]. Software error detection is performed at the end of each time frame of fixed duration.

4 Failure Scenarios

In this section, we describe various combinations of software and hardware failures for the three fault tolerant architectures that could lead to a system failure. Simulation programs for these failure scenarios have been developed.

4.1 DRB System

The recovery block is executed on redundant hardware in the initial configuration, and can lead to an unacceptable result if the software recovery block fails, or a transient fault occurs in both the hardware hosts. The software RB can fail as follows: the execution of the primary can (1) result in a success, (2) activation of an independent fault, (3) activation of a related fault between primary and secondary, or (4) the activation of a related fault between primary and acceptance test. An independent fault can be activated in the acceptance test after the activation of an independent fault in the primary. The activation of a related fault between primary and secondary or primary and AT leads to a failure. Thus the secondary alternate is executed only when an independent fault has been activated either in the primary and/or AT [1]. The activation of a fault in the secondary alternate leads to an unacceptable result, and hence an unreliable operation of the RB. The activation of an independent fault in the acceptance test after the successful execution of the secondary also leads to a failure. Further distinction of the fault activated in the secondary into related/independent is necessary from the point of view of safety analysis, since they lead to undetected/detected failures respectively. After the occurrence of a permanent hardware fault, the DRB is reconfigured and a single copy of the RB is executed. An unacceptable result in the reconfigured mode of operation can be caused by an error in the RB, or a transient failure of the hardware host on which the software is executing. Thus the key difference between the initial and the reconfigured mode is the reduction in hardware redundancy.

4.2 NVP System

The 3-version programming system consists of three software versions running on three different processors, and hence different failure scenarios, including related and unrelated software faults, hardware transients, and combination of hardware and software faults must be considered. The NVP system in its initial configuration can fail from several causes: (1) if two of the three versions activate unrelated faults, or if any related fault between two versions is activated; (2) if the input activates a fault which affects all three versions or a fault in the voter; (3) two of the three processors experience faults; and (4) if a hardware host fails and one of the software version on the other host also fails (via an unrelated or related fault) [4]. Thus the activation of either an independent or a related fault between two or three software versions leads to an unreliable behavior of the NVP system. The activation of an independent fault leads to a detected failure, whereas the activation of a related fault leads to an undetected failure. We assume that the system is reconfigured to the simplex mode after the first permanent hardware fault. In this reconfigured state, an unacceptable result is produced by either a hardware transient or a software fault activation.

4.3 NSCP System

The NSCP system is vulnerable to related faults, whether they involve versions running in the same or different half of the system. We have ignored the possibility of a related fault among all three versions to enable comparisons with NVP and DRB systems. The various causes due to which NSCP system can fail in its initial configuration are: (1) any two versions activate related faults; (2) activation of a related fault among all four versions, or voter failure; (3) activation of independent faults among two versions, if the versions are running on two hardware hosts in two different halves; (4) activation of an independent fault in two versions in the same half of the system and the activation of a transient fault in the hardware host in the other half; and (5) activation of a transient hardware fault in each half of the system. The key difference between NVP, DRB and NSCP systems is that in case of NSCP, two independent faults in the software versions can be tolerated as long as they occur in the same half of the system, and the hardware host in the other half does not fail.

5 Numerical Results and Discussion

In this section, we describe the results of the simulation of the failure behavior of the three systems. The failure profile is expressed in terms of the expected number of failures experienced by the system over a period of time. The rate functions and the values of the parameters chosen are merely to demonstrate the utility of simulation, and are not based on any systematic experimental study.

Without loss of generality we assume that the failure intensities of the versions / alternates / voter / AT are given by the failure intensity of the Goel-Okumoto model [5]. Thus $\lambda(n,t) = abe^{-bt}$, where a is the expected number of faults that would be detected given infinite testing time, and b is the failure occurrence rate per fault. The failure intensities used in this study are summarized in Table 1. The parameters of Failure Intensity # 1 are estimated from NTDS data [5]. Initially, we study the vulnerability of the fault tolerant architectures to related faults among software versions. The failure intensity of the AT / voter is assumed to be Failure Intensity # 4 in this case. The effect of the failure behavior of AT/voter on the overall failure behavior of the fault tolerant architectures was studied next. The probability of a related fault among the software versions was set to a very low value in this case. Simulations were carried out by setting the failure intensities of the acceptance test / voter to all the four intensities in Table 1. Figure 2, Figure 3 and Figure 4 show the expected number of failures for various values of correlation and failure intensities of the acceptance test/voter, for DRB, NVP and NSCP, respectively. The figure also shows the expected number of faults that would be detected from a single version with the same failure intensity, for the sake of comparison. Initially we assume that the hardware hosts are perfect, by setting the probability of activation of

Table 1. Failure Intensities of AT/Voter/Versions/Alternates

Failure Intensity # 1	$34 * 0.0057 * e^{(-0.0057*t)}$
Failure Intensity # 2	$34 * 0.0020 * e^{(-0.0020*t)}$
Failure Intensity # 3	$3.4 * 0.0020 * e^{(-0.0020*t)}$
Failure Intensity # 4	$0.34 * 0.0020 * e^{(-0.0020*t)}$

a transient hardware fault, and rate of occurrence of a permanent hardware fault to 0.0. Figures 2, 3 and 4 depict that for a given value of the probability of a related fault, the expected number of failures is highest for NSCP, followed by NVP, followed by DRB. This could be attributed to the fact that NSCP has four software versions executing in parallel, NVP has three, while two versions execute sequentially in case of DRB. Also, as the probability of a related fault increases, related fault increases, the expected number of failures increases, and after a certain threshold probability, the single version software is in fact less failure-prone than the fault tolerant software. The expected number of failures increases as the failure intensity of the AT / voter ranges from Failure Intensity #4 to Failure Intensity #1.

We then compared the failure profiles of DRB, NVP, NSCP and a single version. An extreme case of an acceptance test is another module. The expected number of failures observed by DRB and NVP systems in this extreme situation (assuming that NVP system has a perfect voter), was comparable. The expected number of failures of NSCP system is higher than NVP system, followed by the DRB system. The probability of a related fault among two versions is assumed to be 0.1, and the probability of a related fault among all versions is assumed to be 0.0. For a low probability of a related fault among software versions, fault tolerance does improve the reliability of a system over a single version. NSCP system experiences a larger number of failures than NVP, and hence is more unreliable than NVP.

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. Simulation was used to compute the percentage of undetected faults for NVP and NSCP systems, for different values of the probability of a related fault among two versions, which is the most comon source of undetected failures. The effect of the probability of a related fault was studied assuming a perfect voter. The results

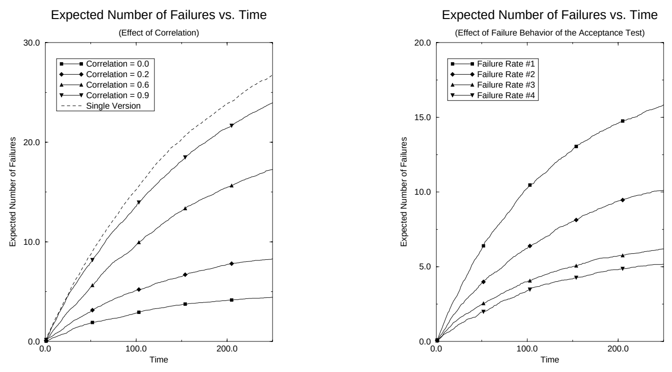

Figure 2. DRB - Failure Behavior in the Absence of Hardware Failures

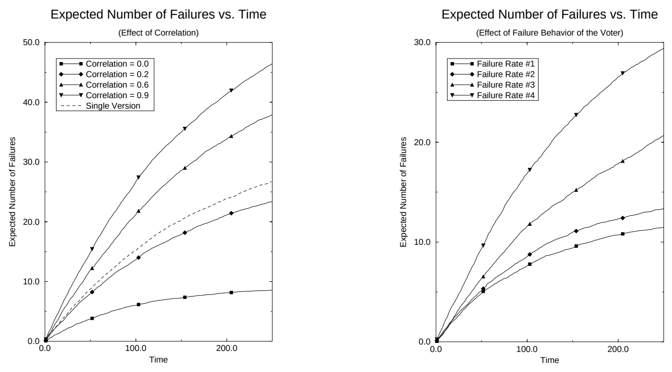

Figure 3. NVP - Failure Behavior in the Absence of Hardware Failures

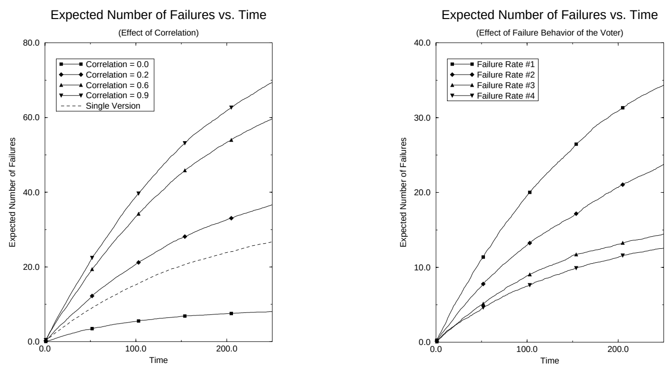

Figure 4. NSCP - Failure Behavior in the Absence of Hardware Failures

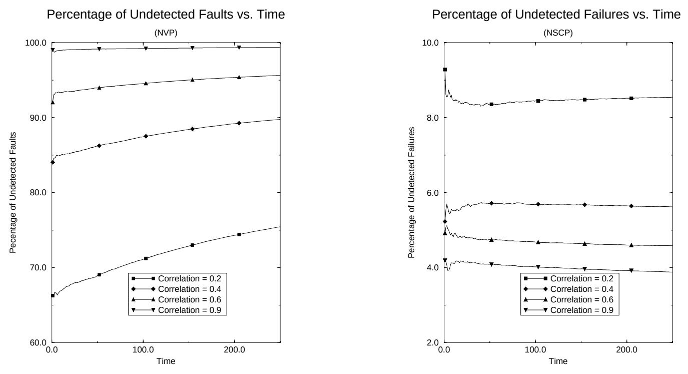

Figure 5. NVP & NSCP - Percentage of Undetected Software Failures

are shown in Figure 5.

Similar experiments can be conducted under the influence of permanent and transient hardware faults. The expected number of failures in case of all the three systems will increase, due to the contribution of hardware faults.

6 Conclusions and Future Research

In this paper we have explored simulation technique to study the failure behavior of the three commonly used fault tolerant architectures, viz., Distributed Recovery Block (DRB), N-Version Programming (NVP) and N-Self Checking programming (NSCP). We have demonstrated the ability of simulation to study complex failure scenarios with interactions among the various components comprising the system, by choosing the rate functions to describe the reliability growth of the software versions, and failure probabilities and rates for the hardware hosts. The simulations have been used to study the effect of various parameters like the probability of a related fault, failure behavior of the acceptance test/voter, etc. on the expected number of failures of the system. Simulation can also be used to compute other metrics of interest like the mean time between failures (MTBF), expected number of hardware failures, expected number of failures caused by related and independent software faults, expected number of failures of the acceptance test/voter etc,. Simulations have been developed specifically for 2 alternates in case of DRB, 3 versions in case of NVP, and 4 versions in case of NSCP, and are not scalable. Future work involves developing scalable simulations, and studying the influence of the other parameters like coverage etc. on the failure behavior of the system, along with faster and better simulation techniques.

References

[1] J. Arlat, K. Kanoun, and J. C. Laprie. "Dependability Modeling and Evaluation of Software Fault Tolerant Systems". *IEEE Trans. on Comp.*, 39(4):504–512, April 1990.

[2] A. Avižienis. "Fault Tolerance: The Survival Attribute of Digital Systems". *Proc. of IEEE*, 66(10):1109–1125, Oct. 1978.

[3] A. Avižienis. "The N-Version Approach to Fault-Tolerant Software". *IEEE Trans. on Soft. Eng.*, SE-11(12):1491–1501, Dec. 1985.

[4] J. B. Dugan and M. R. Lyu. *Software Fault Tolerance, M. R. Lyu, Ed.,*, chapter Dependability Modeling for Fault-Tolerant Software and Sytems, pp 109–138. John Wiley, New York, 1995.

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

[6] K. Kanoun, M. Kaaniche, C. Beounes, J. C. Laprie, and J. Arlat. "Reliability Growth of Fault-Tolerant Software". *IEEE Trans. on Rel.*, 42(2):205–219, June 1993.

[7] K. H. Kim and H. O. Welch. "Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications". *IEEE Trans. on Comp.*, 38(5):626–636, May 1989.

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

[9] D. F. McAllister and M. A. Vouk. *Handbook of Software Reliability Engineering, M. R. Lyu, Ed.,*, chapter Fault-Tolerant Software Reliability Engineering, pp 567–614. McGraw-Hill, New York, NY, 1996.

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

[11] R. C. Tausworthe and M. R. Lyu. *Handbook of Software Reliability Engineering, M. R. Lyu, Ed.,*, chapter Software Reliability Simulation, pp 661–698. McGraw-Hill, New York, 1996.

[12] L. A. Tomek and K. S. Trivedi. *Software Fault Tolerance, M. R. Lyu, Ed.,*, chapter Analyses Using Stochastic Reward Nets, pp 139–165. John Wiley, New York, 1995.

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