System Reliability Analysis of an N-Version Programming Application

Joanne Bechta Dugan, Senior Member IEEE
University of Virginia, Charlottesville
Michael R. Lyu, Member IEEE
Bell Communications Research, Morristown

Key Words — **N-version programming (NVP), software fault tolerance, fault tree, Markov model**

Reader Aids —
General purpose: Present a system modeling example with experimental data
Special math needed for derivations: Probability
Special math needed to use results: Same
Results useful to: Reliability analysts

Summary & Conclusions — **This paper presents a quantitative reliability analysis of a system designed to tolerate both hardware & software faults. The system achieves integrated fault tolerance by implementing N-version programming (NVP) on redundant hardware. The system analysis considers unrelated software faults, related software faults, transient hardware faults, permanent hardware faults, and imperfect coverage. The overall model is Markov in which the states of the Markov chain represent the long-term evolution of the system-structure. For each operational configuration, a fault-tree model captures the effects of software faults and transient hardware faults on the task computation. The software fault model is parameterized using experimental data associated with a recent implementation of an NVP system using the current design paradigm. The hardware model is parameterized by considering typical failure rates associated with hardware faults and coverage parameters. Our results show that it is important to consider both hardware & software faults in the reliability analysis of an NVP system, since these estimates vary with time. Moreover, the function for error detection & recovery is extremely important to fault-tolerant software. Several orders of magnitude reduction in system unreliability can be observed if this function is provided promptly.**

1. INTRODUCTION

Computer systems used for critical applications, such as flight control, air-traffic control, patient monitoring, or power plant monitoring, are designed to tolerate faults in the software as well as in the hardware. Distinguishing between hardware & software faults can be difficult, because symptoms of transient hardware faults and those of software design faults are often very similar [6]. Fault-tolerant system designers thus now advocate a unified treatment of hardware & software faults.

Acronyms

HECA, SECA	[hardware, software] error confinement area
NVP	N-version programming.

Three recent systems provide an integrated approach to hardware & software fault tolerance. Distributed Recovery Blocks (DRB) scheme [6] combines both distributed processing and Recovery Block (RB) [13] concepts to provide a unified approach to tolerating both hardware & software faults. Architectural considerations for the support of NVP [2] were addressed in [7], in which the FTP-AP system is described. The FTP-AP system achieves hardware & software design-diversity by attaching AP (application processors) to the byzantine resilient hard core FTP (fault-tolerant processor). *N* self-checking programming (NSCP) [8] uses diverse hardware & software in self-checking groups to detect hardware & software induced errors. The NSCP concept forms the basis of the flight control system used on the Airbus A310 & A320 aircraft, and was analyzed in [3].

This paper analyzes a system which uses NVP on redundant hardware. A combination of fault-tree and Markov models provides a framework for the analysis of hardware & software fault-tolerant systems. The system model is Markov in which the states of the Markov chain represent the evolution of the hardware configuration as permanent faults occur. A fault-tree model for each state captures the effects of software faults and transient hardware faults. This hierarchical approach simplifies the development, solution, and understanding of the modeling process. The model is parameterized using actual data derived from an experimental implementation of a real, automatic (computerized) airplane landing system ('autopilot''). The software systems of this project were developed & programmed by 15 programming teams at the University of Iowa and the Rockwell/Collins Avionics Division. A total of 40 students (33 from ECE & CS departments at the University of Iowa, 7 from the Rockwell International) participated in this project to design, code, and test (all independently) the computerized airplane-landing system [11].

2. ASSUMPTIONS & DEFINITIONS

2.1 Definitions

- Coverage: Pr{system can automatically recovery from a permanent hardware fault|a permanent hardware fault occurs}.
- Decider: A computing routine which determines the correct results from the multiple software versions using consensus.
- Error: The manifestation of a fault, *eg*, in the information that is processed or in the internal system state.
- Failure: An unacceptable deviation from the anticipated delivered service.
- Fault: An undesirable imperfection in a hardware or software component of the system, *eg*, a short circuit between 2 leads, or an incorrect program statement.

0018-9529/94/$4.00 ©1994 IEEE

- Related software fault: A software fault that affects at least 2 versions, causing similar erroneous results.
- Unrelated software fault: A software fault that affects only 1 version.

2.2 Assumptions

1. Task Computation. The computation being performed is a task (or set of tasks) which is repeated periodically. A set of sensor inputs is gathered & analyzed, and a set of actuations is produced. All repetitions of a task are mutually *s*-independent. The analysis goal is to find: Pr{a task succeeds in producing an acceptable output, despite the possibility of hardware or software faults}. [More-interesting task-computation processes could be considered using techniques in [9, 16]. We do not address timing or performance issues in this model. See [15] for a performability analysis of fault-tolerant software techniques.]

2. Software Failure Probability. Software faults exist in the code (despite rigorous testing). A fault can be activated by a random input, thus producing an erroneous result. Each instance of a task receives a different set of inputs which are mutually *s*-independent. A software task has a fixed probability of failure when executed. All instances of a particular task are mutually *s*-independent. [We do not assign a failure rate to the software and thus do not consider reliability-growth models.]

3. Coincident Software Failures in Different Versions. If two different software versions fail on the same input, they either produce *similar* or *different* results. *Similar* erroneous results are caused by *related* software faults. *Different* erroneous results which are simultaneously activated are caused by *unrelated* (mutually *s*-independent in the terminology of [1]) software faults. The sets of *related* and of *unrelated* software faults are mutually *s*-independent. [We use the model in [1] for software failures, except that: [1] assumes *related* & *unrelated* faults are mutually exclusive, whereas they are *s*-independent here.] [Our treatment of *related* & *unrelated* differs considerably from faults models for *s*-correlated failures [5, 10, 12] in which *related* & *unrelated* software failures are not differentiated.]

4. Permanent Hardware Faults. The statistical arrival (activation) intensity of *permanent* hardware faults is constant; *ie*, it is a homogeneous Poisson process.

5. Transient Hardware Faults. A transient hardware fault upsets the software running on the processor and produces an erroneous result which is indistinguishable from an input-activated software error. The lifetime of transient hardware faults is short compared to the time for a task computation. Thus a fixed probability is assigned to the occurrence of a transient hardware fault during a single computation.

6. System Maintenance. The systems are not maintained. [Maintainability could be included in the Markov model; we have chosen not to include it in order to make the comparisons clearer.]

7. Coverage Failure. The fault tolerant system fails if it is unable to detect and recover automatically from a permanent hardware fault.

2.3 *Notation*

λ	Poisson arrival intensity for a permanent hardware fault to a single processing element
c	coverage
P_H	Pr{at least one transient hardware fault occurs during a single task computation}
P_V	Pr{an *unrelated* software fault is activated during a task computation}, for each version
P_{RV}	Pr{a *related* fault, common to the version pair, is activated during a task computation}, for each pair of versions
P_{RALL}	Pr{a *related* fault, common to all versions, is activated during a single task computation}
P_D	*Pr*{decider fails, either by accepting an incorrect result or by rejecting a correct result}.

Other, standard notation is given in "Information for Readers & Authors" at the rear of each issue.

3. SYSTEM DESCRIPTION

The software project [1] was scheduled & conducted in 6 phases:

1. Initial design — 4 weeks
2. Detailed design — 2 weeks
3. Coding — 3 weeks
4. Unit testing — 1 week
5. Integration testing — 2 weeks
6. 2-Step formal acceptance testing (AT1 & AT2) — 2 weeks

AT1. Each program was run in a test harness of 4 nominal flight simulation profiles

AT2. One extra simulation profile, representing an extremely difficult flight situation, was imposed. ◄

By the end of the formal acceptance testing phase, 12 of the 15 programs passed the acceptance test successfully and were engaged in operational testing for further evaluation. The average size of these programs was 1564 lines of uncommented code, or 2558 lines of commented code. The mean fault density of the program versions which passed AT1 was 0.48 faults/1000 lines of uncommented code. The mean fault density of the final versions was 0.05 faults/1000 lines of uncommented code.

3.1 NVP Operational Environment

The operational environment for the application was an airplane-autopilot interacting in a simulated environment, as shown in figure 1. Three channels of diverse software autonomously computed a surface command to guide a simulated aircraft along its flight path. To ensure that important command errors could be detected, several levels of random wind turbulences were superimposed to represent difficult flight conditions. The individual commands were recorded, and compared for discrepancies that could indicate the presence of faults.

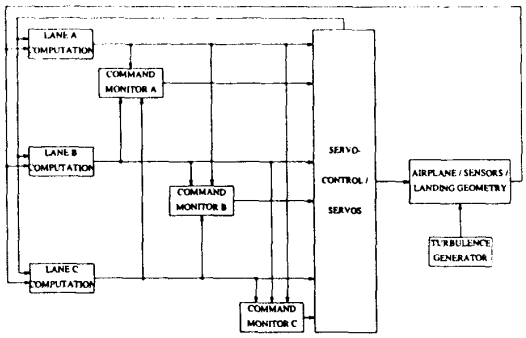

Figure 1. 3-Channel Flight Simulation Configuration

This configuration of a 3-channel flight simulation system consisted of:

- 3 lanes (autonomous software process) of control-law computation
- 3 command monitors
- 1 servo control
- 1 airplane model
- 1 turbulence generator.

The lane computations and the command monitors were the accepted software versions generated by the programming teams. Each lane of computation monitored, but did not interact with, the other two lanes. However, no single lane could decide whether another lane was faulty. A separate servo-control logic function was required to make that decision. The aircraft mathematical model provided the dynamic response of current medium size, commercial transports in the approach/landing flight phase. The 3 control signals from the autopilot computation lanes were inputs to 3 elevator servos. The servos were force-summed at their outputs, so that the mid-value of the 3 inputs became the final elevator command. The Landing Geometry and Turbulence Generator were models associated with the Airplane simulator.

In summary, 1 run of flight simulation was characterized by 5 initial values regarding the landing position of an airplane:

1. initial altitude (about 1500 feet)
2. initial distance (about 52800 feet)
3. initial nose up relative to velocity (range from 0 to 10 degrees)
4. initial pitch attitude (range from -15 to 15 degrees)
5. vertical velocity for the wind turbulence (0 to 10 feet/s).

One simulation consisted of about 5280 iterations of lane command computations (50 ms each) for a total landing time of approximately 264 seconds.

3.2 Operational Error Distribution

During the operational phase, 1000 flight simulations (over $5 \cdot 10^6$ program executions) were conducted. We used the program versions which passed the AT1 for study. Our reason was that had the Acceptance Test not included an extreme situation of AT2, more faults would have remained in the program versions. We were interested in seeing how the remaining faults would be manifested during the operational testing, and how they would be tolerated in various NVP configurations.

TABLE 1
Errors in 3-Version Configurations

Version ID	Number of Failures	Mean Occurence Frequency	
		By Case (10^{-2})	By Frame (10^{-6})
β	510	51	97
γ	0	0	0
ϵ	0	0	0
ζ	0	0	0
η	1	0.1	0.2
θ	360	36	68
κ	0	0	0
λ	730	73	138
μ	140	14	27
ν	0	0	0
ξ	0	0	0
o	0	0	0
Average	145.1	14.5	27.5

Table 1 shows the software failures encountered in each single version, while table 2 shows different software error categories under all combinations of 3-version configurations. We examine 2 levels of granularity in defining software execution errors and s-correlated errors: *by case* or *by frame*.

- *By case* (based on 1000 test-cases). If a version failed at any time in a test case, it was considered failed for the whole case. If two or more versions failed in the same test case (no matter at the same time or not), they were said to have coincident errors for that test case.
- *By frame* (based on 5 280 920 execution time-frames). Errors were counted only in the time-frame upon which they manifested themselves. Coincident errors were defined to be the multiple program versions failing at the same time in the same test case (with or without the same variables and values).

TABLE 2
Error Characteristics for 3-Version Configurations[1]

Category	*By Case*		*By Frame*	
	Number of cases	Frequency	Number of cases	Frequency
F_0 - 0 errors	163370	0.7426	1160743690	0.999089
F_1 - 1 error	51930	0.2360	1056010	0.000909
F_2 - 2 coincident errors	4440	0.0202	2700	0.000002
F_3 - 3 coincident errors	260	0.0012	0	0.0
Total	220000	1.0000	1161802400	1.000000

[1]The number of significant figures is not intended to imply any accuracy in the estimates, but to illustrate the arithmetic.

- *By case*. The mean failure probability for the 1-version (table 1) is 14.5%. For all the 3-version combinations (table 2), the failure probability is 2.1% (sum of error categories 3 & 4), an improvement over the 1-version by a factor of 7.
- *By frame*. The mean failure probability for the 1-version (table 1) is $27 \cdot 10^{-6}$. For all the 3-version combinations (table 2), the failure probability is $2 \cdot 10^{-6}$ (sum of error categories 3 & 4), an improvement over the 1-version by a factor of 13.

These software failure probability estimates were obtained by empirical program-execution results. Section 4 derives a general reliability model for an integrated fault-tolerant system. We verify our reliability model with the empirical results.

4. MODEL DESCRIPTION

A reliability model of an integrated fault-tolerant system must include at least 3 considerations:

- computation errors,
- system structure,
- coverage modeling.

We concentrate on the first two, as coverage modeling has been addressed in detail elsewhere [4].

4.1 Computation Error Model

The task computation process consists of a repeated execution of a fault tolerant software component as described in section 2.2. The software component designed to perform the task is designed to be fault tolerant. During 1 task-iteration, 2 types of events can interfere with the computation:

- the particular set of inputs activates a software fault in at least 1 software-version and/or the decider.
- a hardware transient fault upsets the computation but does not permanently damage the hardware.

The combinations of software faults and hardware transients that can cause an erroneous output for a single computation are modeled with a fault tree.

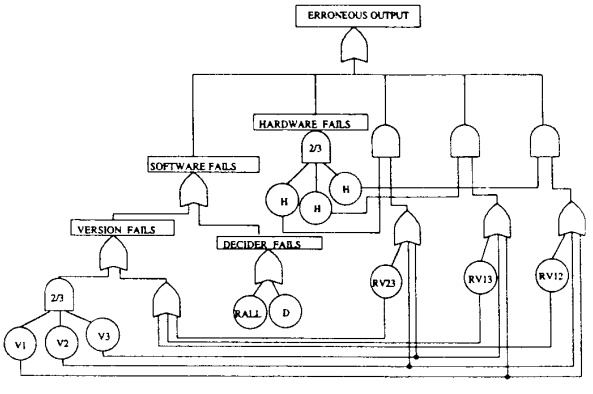

Figure 2. Fault-Tree Model of Computation Errors in Full-Up State

Figure 2 shows the fault-tree model for the computation error process when the system is fully operational. The basic events are labeled with the symbol which represents the probability of occurrence; the symbols are defined in table 3.

TABLE 3
Parameterization of Model

	By Case (%)	*By Frame* (10^{-6})
A. Fault-Tree Basic-Event Probabilities		
P_V	9.6	300
P_{RV}	0.0	0.6
P_{RALL}	0.03	0.0
P_D	0.01	0.1
P_H	0.00073	0.0028
B. Markov Model Parameters		
λ	$1 \cdot 10^{-5}$	
c	0.999	

An erroneous output can result from software failure, hardware failure, or a combination. The software fails if:

- *unrelated* faults are activated in 2 or 3 versions by the same test case,
- a *related* fault is activated between any 2 or 3 versions, or
- a fault in the decider is activated.

The hardware causes a computation error in the resident software if a transient fault occurs during a computation. Any combination of hardware & software faults affecting 2 of the 3 versions leads to an unacceptable output.

If a permanent hardware-fault disables one of the host processors, then the system is reconfigured to a simplex system. In the simplex mode, an unacceptable output results from either an *unrelated* software fault activation, or a hardware transient. Figure 3 shows the fault-tree model for the computation-error process while in the simplex mode.

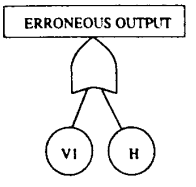

Figure 3. Fault-Tree Model of Computation Errors in Simplex State

4.2 System Structure Model

The longer-term system behavior is affected by the arrival (activation, manifestation) of permanent faults requiring system reconfiguration to a degraded mode of operation. The system structure is modeled by a Markov process, where the Markov states and transitions model the evolution of the system. Each state in the Markov process represents a particular

configuration of hardware & software components and thus a different level of redundancy.

For the NVP-system modeled in this paper, there are 2 operational states and 1 absorbing failure state. The initial state of the system represents the original system configuration, with 3 software versions hosted on 3 different processors. When one of the processors experiences a hard fault, the system is reconfigured to a simplex system, with 1 software version running on 1 processor.

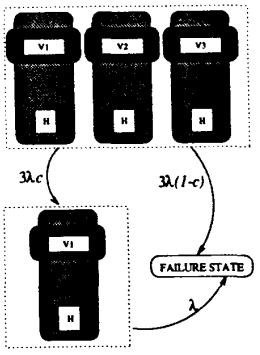

Figure 4. Markov Model of System Structure

Figure 4 shows the Markov model for this system. The 2 operational states show the HECA & SECA associated with the system structure. The HECA is the lightly shaded region; the SECA is the darkly shaded region. The HECA or SECA covers the region of the system affected by faults in that component. For example, HECA covers the software component since the software component fails if that hardware experiences a fault. The SECA covers only the software component since no other components are affected by a fault in that component.

The reconfiguration from the TMR-system to a simplex system is successful with probability c. If the reconfiguration is unsuccessful, the system fails.

4.3 Combining the Models

For each state in the Markov chain, there is a different combination of hardware transients and software faults that can cause an erroneous output. A fault-tree model for each state captures the probability that 1 computation results in an erroneous output.

Notation (for Markov Model)

i	state in the Markov model
q_i	Pr{output error occurs during 1 task-computation}
$P_i(t)$	Pr{system is in state i at time t}
$Q(t)$	Pr{an unacceptable result is produced at time t}.

The fault-tree model solution produces q_i for each i. The Markov model solution produces $P_i(t)$. The model combines these two measures to produce $Q(t)$.

$$Q(t) = \sum_{i=1}^{n} q_i \cdot P_i(t)$$

5. PARAMETERIZATION & RESULTS

This section details the methodology used for estimating the parameters, and discusses the assumptions. Table 3 summarizes the resulting parameter values.

5.1 Software Parameters

The experimental results from [1] in table 2, were used to estimate the probabilities associated with the activation of software faults.

$$F_0 = (1-P_V)^3 \cdot (1-P_{RV})^3 \cdot (1-P_{RALL}) \quad (1)$$

$$F_1 = 3P_V \cdot (1-P_V)^2 \cdot (1-P_{RV})^3 \cdot (1-P_{RALL}). \quad (2)$$

Divide (1) by (2); the result depends only on P_V, and thus can be used to estimate it.

$$P_V = F_1/(3F_0 + F_1). \quad (3)$$

The data from table 2, used in (3), yield an estimate of $P_V = 0.096$ for the probability of activation of an *unrelated* fault in a 3-version configuration. Table 4 compares the observed values with the probability of activation of 1, 2, 3 faults as predicted by a model assuming *s*-independence between versions.

TABLE 4
Comparison of *s*-Independent Model with Observed Data[1]
[N = number of errors activated]

	By Case		*By Frame*	
N	Independent Model	Observed Frequency	Independent Model	Observed Frequency
0	0.7393	0.7426	0.999090	0.7426
1	0.2350	0.2360	0.000909	0.2360
2	0.0249	0.0202	$3 \cdot 10^{-7}$	$2 \cdot 10^{-6}$
3	0.0009	0.0012	$3 \cdot 10^{-11}$	0.0

- *By case*. The observed probability of 2 simultaneous errors is lower than predicted by the *s*-independent model, while the observed probability of 3 simultaneous errors is higher than predicted. For this set of data we conclude that $P_{RV} = 0$; instead we derive an estimate for P_{RALL}. Since $P_{RV}=0$,

$$F_2 = P_V^3 + P_{RALL} - P_V^3 \cdot P_{RALL}, \quad (4)$$

yielding an estimate of $P_{RALL} = 0.03\%$.

- *By frame*. The *by frame* data in table 2, put into (3), estimates $P_V = 0.03\%$. For these data, when the failure probabilities which are predicted by the *s*-independent model are compared to the actual data in table 4, the observed probability of 2 errors is 10 times the predicted probability. There were no cases for which all three programs produced erroneous results. Thus, we estimate $P_{RALL} = 0$ and derive an estimate for P_{RV}. Let 2 errors be produced; this could be caused by either the activation of 2 simultaneous *unrelated* faults, or by the activation of a *related* fault. Also, it depends on the non-failure of the remaining version, either by *unrelated* or *related* fault. Then, consider the 3 combinations of 2 failures which can occur.

$$F_2 = 3(P_V^2 + P_{RV} - P_V^2 \cdot P_{RV}) \cdot (1 - P_V) \cdot (1 - P_{RV})^2 \cdot (1 - P_{RALL}) \quad (5)$$

Eq (3) & (5) are used to estimate $P_{RV} = 6 \cdot 10^{-7}$. ◄

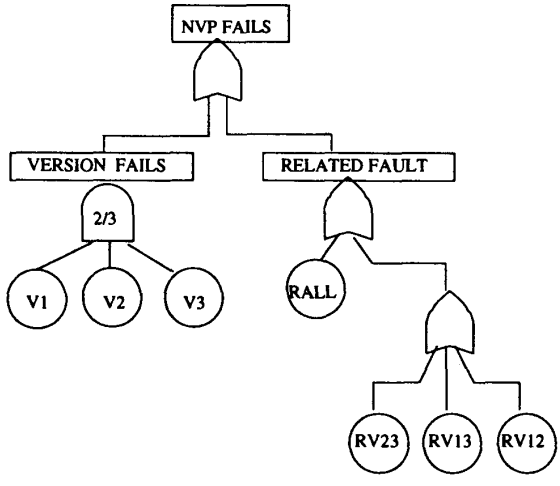

Figure 5. Fault-Tree Model of NVP Software System

For both the *by case* and *by frame* scenarios, the parameters derived from the data were applied to the fault-tree model in figure 5.

- *By case*. The fault-tree model predicts a failure probability of 0.026, while the observed failure probability was 0.0214.
- *By frame*. The fault-tree model predicts a failure probability of $2.07 \cdot 10^{-6}$, while the observed failure probability was $2.3 \cdot 10^{-6}$. ◄

In the model solution section 5.3, these parameters are combined with hardware parameters to predict the NVP system reliability.

5.2 Hardware Parameters

Assumptions

1. [We assume that] A hardware transient that occurs anywhere during the execution of a task disrupts the entire computation running on the host.
2. The lifetime of a transient fault is 1 s.
3. $c = 0.999$; a fairly typical value. ◄

Typical permanent-failure rates for processors are in the 10^{-5}/hour range, with transients perhaps 10 times as large. Thus, for the Markov model, we use:

$\lambda_p = 10^{-5}$/hour,

$\lambda_t = 10^{-4}$/hour.

- *By case*. A typical test case has 5280 frames, each frame being 50 ms; so a typical computation executes for 264 s. The probability that a hardware transient occurs is:

$$1 - \exp(-\lambda_t \cdot 264 \text{ s}) = 7.3 \cdot 10^{-6}. \quad (6)$$

- *By frame*. The probability that a transient occurs is:

$$1 - \exp(-\lambda_t \cdot 0.05 \text{ s}) = 1.4 \cdot 10^{-9}. \quad (7)$$

From assumption #2, a transient can affect as many as 20 time frames. We thus take the probability of a transient to be 20 times the value in (7): $2.8 \cdot 10^{-8}$.

5.3 Model Solution

The full model, including the two fault trees (figures 2 & 3) and the Markov model (figure 4) were solved using the parameters in table 3. The results are shown in figures 6 & 7.

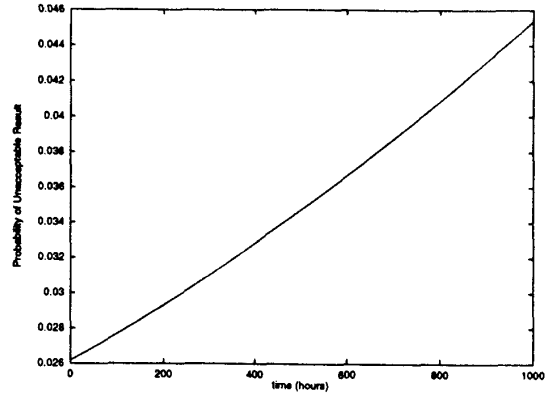

Figure 6. Probability of Producing an Unacceptable Result [during 1 test-case: *by case* data]

- *By case*. Figure 6 shows the time-dependent probability that the NVP system produces an unacceptable result. The length of a typical test case is 264 s (4.4 minutes). Initially, in the full-up state, the probability of an unacceptable result is 2.6%, increasing to 4.5% after 1000 hours, as the probability of operating in the simplex state or in the failure state increases. The probability of producing an unacceptable result while in

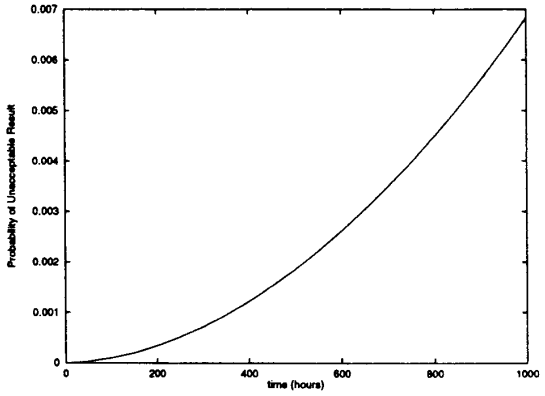

Figure 7. Probability of Producing an Unacceptable Result [during 1 time-frame: *by frame* data]

the simplex mode is 9.6%; and the probability of producing an unacceptable result while in the absorbing failure state is 100%.

- *By frame*. Figure 7 shows the time dependent probability that the NVP system produces an unacceptable result. Initially, in the full-up state, the probability of an unacceptable result is $2.2 \cdot 10^{-6}$, increasing by a factor of 3000 to $6800 \cdot 10^{-6}$ at 1000 hours. The probability of producing an unacceptable result while in the simplex mode is $300 \cdot 10^{-6}$.

ACKNOWLEDGMENT

This work was partially funded by the US NASA AMES Research Center under grant number NCA2-617. The models were solved using SHARPE [14].

REFERENCES

[1] J. Arlat, K. Kanoun, J-C. Laprie, "Dependability modeling and evaluation of software fault-tolerant systems", *IEEE Trans. Computers*, vol 39, 1990 Apr, pp 504-513.

[2] A. Avižienis, "The N-version approach to fault-tolerant software", *IEEE Trans. Software Engineering*, vol SE-11, 1985 Dec, pp 1491-1501.

[3] J.B. Dugan, R. Van Buren, "Reliability evaluation of fly-by-wire computer systems", *J. Systems and Software*, (to appear).

[4] J.B. Dugan, K.S. Trivedi, "Coverage modeling for dependability analysis of fault-tolerant systems", *IEEE Trans. Computers*, vol 38, num 6, 1989, pp 775-787.

[5] D.E. Eckhardt, L.D. Lee, "Theoretical basis for the analysis of multiversion software subject to coincident errors", *IEEE Trans. Software Engineering*, vol SE-11, 1985 Dec, pp 1511-1517.

[6] K.H. Kim, H.O. Welch, "Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications", *IEEE Trans. Computers*, vol 38, 1989 May, pp 626-636.

[7] J.H. Lala, L.S. Alger, "Hardware and software fault tolerance: A unified architectural approach", *Proc. IEEE Int'l Symp. Fault-Tolerant Computing* FTCS-18, 1988 Jun, pp 240-245.

[8] J-C. Laprie, J. Arlat, C. Beounes, K. Kanoun, "Definition and analysis of hardware- and software- fault-tolerant architectures", *IEEE Computer*, vol 23, 1990 Jul, pp 39-51.

[9] J-C. Laprie, K. Kanoun, "X-Ware reliability and availability modeling", *IEEE Trans. Software Engineering*, vol 19, 1992 Feb, pp 130-147.

[10] B. Littlewood, D.R. Miller, "Theoretical basis for the analysis of multiversion software subject to coincident errors", *IEEE Trans. Software Engineering*, vol 15, 1989 Dec, pp 1596-1614,

[11] M.R. Lyu, Y-T. He, "Improving the N-version programming process through the evolution of a design paradigm", *IEEE Trans. Reliability*, vol 42, 1993 Jun, pp 179-189.

[12] V.F. Nicola, A. Goyal, "Modeling of correlated failures and community error recovery in multiversion software", *IEEE Trans. Software Engineering*, vol 16, 1990 Mar.

[13] B. Randell, "System structure for software fault tolerance", *IEEE Trans. Software Engineering*, vol SE-1, 1975 Jun, pp 220-232.

[14] R. Sahner, K.S. Trivedi, "Reliability modeling using SHARPE", *IEEE Trans. Reliability*, vol R-36, 1987 Jun, pp 186-193.

[15] A.T. Tai, J.F. Meyer, A. Avižienis, "Performability enhancement of fault-tolerant software", *IEEE Trans. Reliability*, vol 42, 1993 Jun, pp 227-237.

[16] L. Wei, *A Model Based Study of Workload Influence on Computing System Dependability*, PhD thesis, 1991; University of Michigan.

AUTHORS

Dr. Joanne Bechta Dugan; Dept. of Electrical Engineering; Thornton Hall; University of Virginia; Charlottesville, Virginia 22903-2442 USA.
Internet (e-mail): jbd@Virginia.edu

Joanne Bechta Dugan was awarded the BA (1980) in Mathematics and Computer Science from La Salle University, Philadelphia, and the MS (1982) and PhD (1984) in Electrical Engineering from Duke University, Durham. Dr. Dugan is Associate Professor of Electrical Engineering at the University of Virginia, and was an Associate Professor of Computer Science at Duke University and Visiting Scientist at the Research Triangle Institute. She has performed & directed research on the development & application of techniques for analysis of computer systems which are designed to tolerate hardware & software faults. Her research interests thus include hardware & software reliability engineering; fault tolerant computing; and mathematical modeling using dynamic fault-trees, Markov models, Petri nets, and simulation. Dr. Dugan is an Associate Editor of the *IEEE Trans. Reliability*, a Senior Member of the IEEE, and a member of ACM, Eta Kappa Nu, and Phi Beta Kappa.

Dr. Michael R. Lyu; Bell Communications Research; 445 South St; Morristown, New Jersey 07960 USA.
Internet (e-mail): lyu@bellcore.com

Michael R. Lyu: For biography, see *IEEE Trans. Reliability*, vol 43, 1994 Dec, p 534.

Manuscript received 1994 May 15.

IEEE Log Number 94-06578 ◄TR►