System Reliability Analysis of an N-version Programming Application

Joanne Bechta Dugan

Department of Electrical Engineering
University of Virginia
Charlottesville, VA 22903-2442

Michael R. Lyu

Bell Communications Research
445 South Street
Morristown, NJ 07960

Abstract

This paper presents a quantitative reliability analysis of a system designed to tolerate both hardware and software faults. The system being studied achieves integrated fault tolerance by implementing N-Version Programming (NVP) on redundant hardware. The analysis of the system considers independent software faults, related software faults, transient hardware faults, permanent hardware faults, and imperfect coverage. The overall model is a Markov reward model in which the states of the Markov chain represent the long-term evolution of the structure of the system. For each operational configuration, a fault tree model captures the effects of software faults and transient hardware faults on the task computation. The fault tree models define the reward structure for the overall model. The software fault model is parameterized using experimental data associated with a recent implementation of an NVP system using the current design paradigm, in which the predictions of software failures are very close to the empirical data. The hardware model is parameterized by considering typical failure rates associated with hardware faults and coverage parameters. Results from our study show that it is important to consider both hardware and software faults in the reliability analysis of an NVP system, since these estimates increase with time. Moreover, the function for error detection and recovery is extremely important to fault-tolerant software. Several orders of magnitude improvement in the overall system reliability 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 and software faults can be difficult, as symptoms of transient hardware faults and those of software design faults are often very similar [4]. Current fault tolerant system designers thus advocate a unified treatment of hardware and software faults.

Three recent systems provide an integrated approach to hardware and software fault tolerance. The Distributed Recovery Blocks (DRB) scheme [4] combines both distributed processing and Recovery Block (RB) [8] concepts to provide a unified approach to tolerating both hardware and software faults. Architectural considerations for the support of N-version programming (NVP) [1] were addressed in [5], in which the FTP-AP system is described. The FTP-AP system achieves hardware and software design diversity by attaching application processors (AP) to the byzantine resilient hard core Fault Tolerant Processor (FTP). N self-checking programming (NSCP) [6] uses diverse hardware and software in self-checking groups to detect hardware and software induced errors. The NSCP concept forms the basis of the flight control system used on the Airbus A310 and A320 aircraft, and was analyzed in [2].

In this paper we analyze a system which uses N-version programming on redundant hardware. A combination of fault tree and Markov models provides a framework for the analysis of hardware and software fault tolerant systems. The overall system model is a Markov reward model in which the states of the Markov chain represent the evolution of the hardware configuration as permanent faults occur. A fault tree model captures the effects of software bugs and transient hardware faults, and defines the reward structure for the overall model. This hierarchical approach simplifies the development, solution and understand-

0-8186-4010-3/93 $03.00 © 1993 IEEE

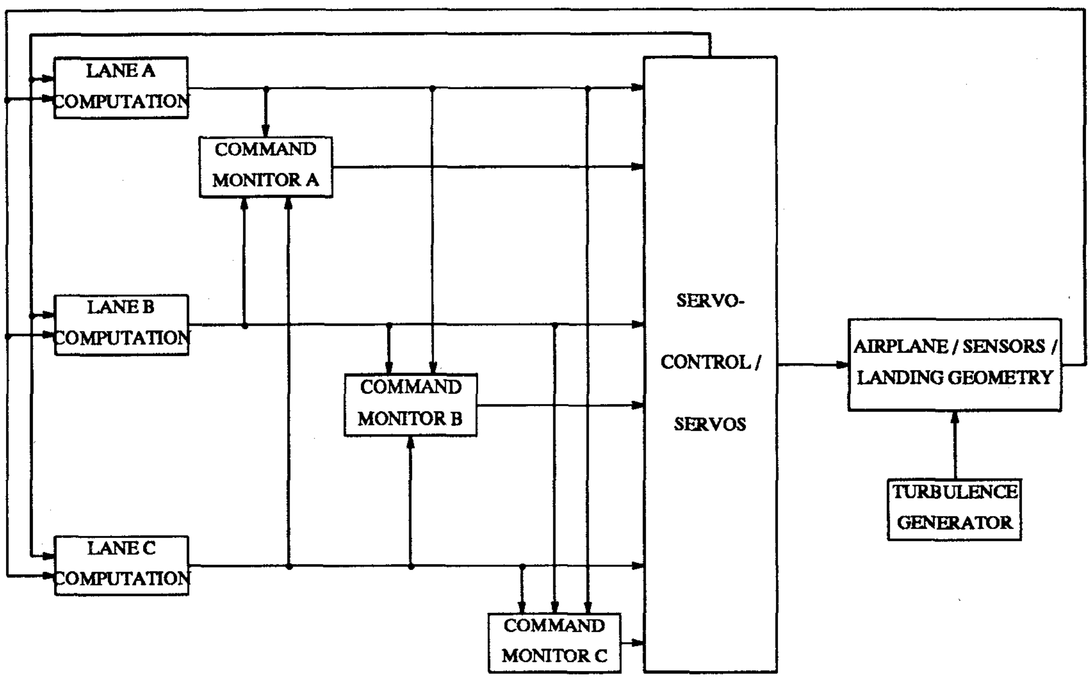

Figure 1: 3-channel flight simulation configuration

ing of the modeling process. The model is parameterized using actual data derived from an experimental implementation of a real-world automatic (i.e., computerized) airplane landing system, or so-called "autopilot." The software systems of this project were developed and programmed by 15 programming teams at the University of Iowa and the Rockwell/Collins Avionics Division. A total of 40 students (33 from ECE and CS departments at the University of Iowa, 7 from the Rockwell International) participated in this project to independently design, code, and test the computerized airplane landing system, as described in the Lyu-He study [7].

2 System Description

The software project in the Lyu-He study was scheduled and conducted in six phases: (1) Initial design phase for four weeks; (2) Detailed design phase for two weeks; (3) Coding phase for three weeks; (4) Unit testing phase for one week; (5) Integration testing phase for two weeks; (6) Acceptance testing phase for two weeks. It is noted that the acceptance testing was a two-step formal testing procedure. In the first step (AT1), each program was run in a test harness of four nominal flight simulation profiles. For the second step (AT2), one extra simulation profile, representing an extremely difficult flight situation, was imposed. By the end of the acceptance testing phase, 12 of the 15 programs passed the acceptance test successfully and were engaged in operational testing for further evaluations. The average size of these programs were 1564 lines of uncommented code, or 2558 lines when comments were included. The average fault density of the program versions which passed AT1 was 0.48 faults per thousand lines of uncommented code. The fault density for the final versions was 0.05 faults per thousand lines of uncommented code.

2.1 The NVP operational environment

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

This configuration of a 3-channel flight simulation

Version Id	Number of failures	Prob. by case	prob. by time
β	510	0.51	0.000096574
γ	0	0.0	0.0
ϵ	0	0.0	0.0
ζ	0	0.0	0.0
η	1	0.001	0.000000189
θ	360	0.36	0.000068169
κ	0	0.0	0.0
λ	730	0.73	0.000138233
μ	140	0.14	0.000026510
ν	0	0.0	0.0
ξ	0	0.0	0.0
o	0	0.0	0.0
Average	145.1	0.1451	0.000027472

Table 1: Errors in three-version configurations

system consisted of three lanes of control law computation, three command monitors, a servo control, an Airplane model, and a turbulence generator. The lane computations and the command monitors would be the accepted software versions generated by the programming teams. Each lane of independent computation monitored the other two lanes. However, no single lane could make the decision as to 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 three control signals from the autopilot computation lanes were inputs to three elevator servos. The servos were force-summed at their outputs, so that the mid-value of the three inputs became the final elevator command. The Landing Geometry and Turbulence Generator were models associated with the Airplane simulator.

In summary, one run of flight simulation was characterized by the following five 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); and (5) vertical velocity for the wind turbulence (0 to 10 ft/sec). One simulation consisted of about 5280 iterations of lane command computations (50 milliseconds each) for a total landing time of approximately 264 seconds.

2.2 Operational error distribution

During the operational phase, 1000 flight simulations, or over five million program executions, were conducted. For a conservative estimation of software failures in the NVP system, we took the program versions which passed the AT1 for study. The reason behind this 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 or would not be tolerated in various NVP configurations.

Table 1 shows the software failures encountered in each single version, while Tables 2 and 3 show different software error categories under all combinations of 3-version configurations. We examine two levels of granularity in defining software execution errors and correlated errors: "by case" or "by time." The first level was defined based on test cases (1000 in total). 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. The second level of granularity was defined based on execution time frames (5,280,920 in total). Errors were counted only at the time frame upon which they manifested themselves, and 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).

In Table 1 we can see that the average failure probability for single version is 0.145 measured by case, or 0.000027 measured by time. Table 2 shows that when measured by case, for all the 3-version combinations the failure probability is 0.0214 (sum of error categories 3 and 4), an improvement over the single

Category	No. of incidents	Probability
1 - no errors	163370	0.7426
2 - single error	51930	0.2360
3 - two errors	4440	0.0202
4 - three errors	260	0.0012
Total	220000	1.0

Table 2: Errors by case in three-version configurations

Category	No. of incidents	Probability
1 - no errors	1160743500	0.999089
2 - single error	1056200	0.000909
3 - two errors	2700	0.000002
4 - three errors	0	0.0
Total	1161802400	1.0

Table 3: Errors by time in three-version configurations

version by a factor of 7.

In Table 3 we see that when measured by time, for all the 3-version combinations, the failure probability is 0.000002 (sum of error categories 3 and 4). This is a reduction of roughly 13 when compared with the single version execution.

The above software failure probability estimations were obtained by empirical program execution results. In the next section we will derive a general reliability model for an integrated fault tolerant system, which gives a more detailed breakdown of the system. We verify our reliability model with the empirical results.

3 Model Description

A reliability model of an integrated fault tolerant system must include at least three different considerations: computation errors, system structure and coverage modeling. In this paper we concentrate on the first two, as coverage modeling has been addressed in detail elsewhere [3].

3.1 Computation error model

The computation process is assumed to consist of a single software task that is executed repeatedly. The software component designed to perform the task is designed to be fault tolerant. During a single task iteration, two types of events can interfere with the computation. First, the particular set of inputs could activate a software fault in one or more of the software versions and/or the decider (a decider is a computing routine which determines the correct results from the multiple software versions using consensus). Second, a hardware transient fault could upset the computation but not cause permanent hardware damage. The combinations of software faults and hardware transients that can cause an erroneous output for a single computation is modeled with a fault tree.

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 4.

An erroneous output can result from software failure, hardware failure or a combination. The software fails if independent faults are activated in two or three versions by the same test case, or if a related fault is activated between any two or three versions, or if 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 and software faults affecting two of the three 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 result results from either an independent software fault activation, or a hardware transient. The fault tree model showing the computation error process while in the simplex mode is shown in figure 3.

3.2 System structure model

The longer-term system behavior is affected by the arrival (activation, manifestation) of permanent faults which require system reconfiguration to a degraded mode of operation. The system structure is modeled by a Markov reward 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 and software components and thus a different level of redundancy.

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

Figure 4 shows the Markov model for this system. The two operational states show the hardware and software error confinement areas associated with the system structure. The hardware error confinement

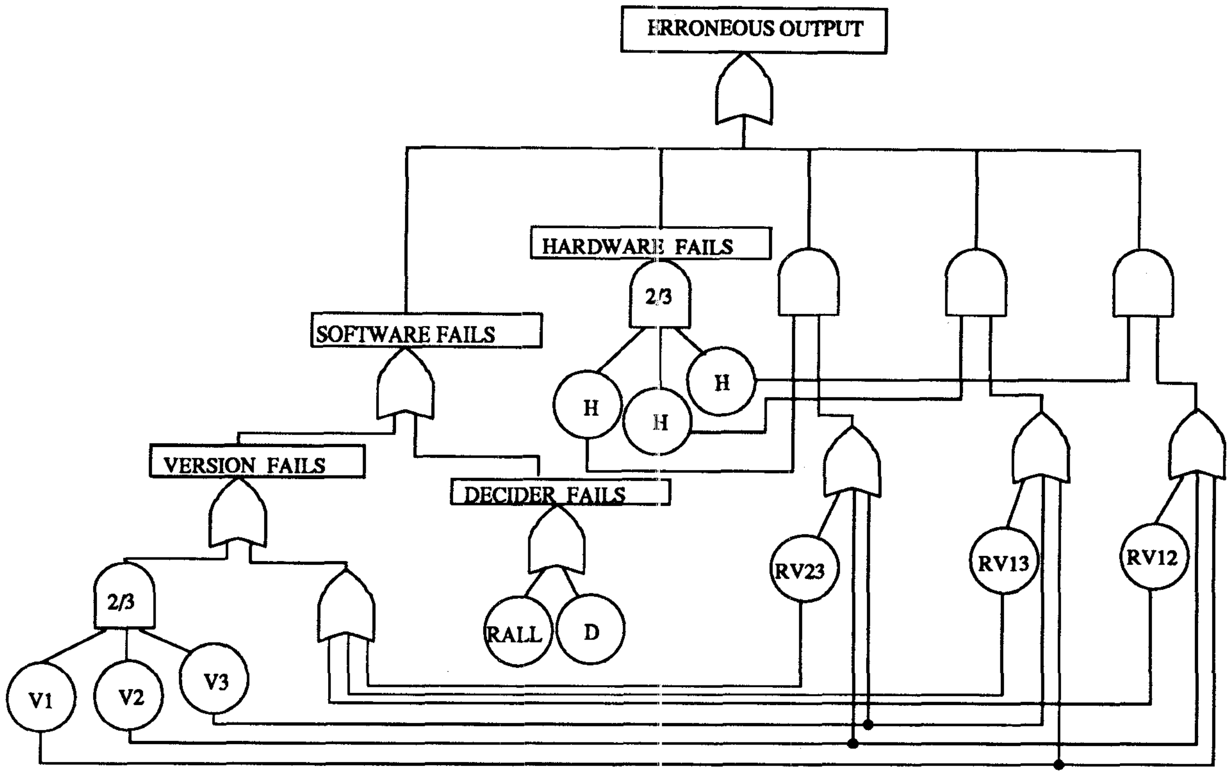

Figure 2: Fault tree model of computation errors in full-up state

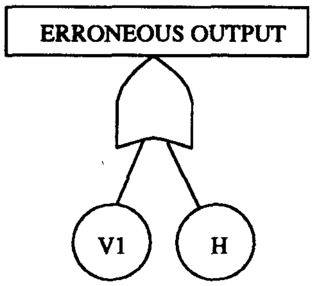

Figure 3: Fault tree model of computation errors in simplex state

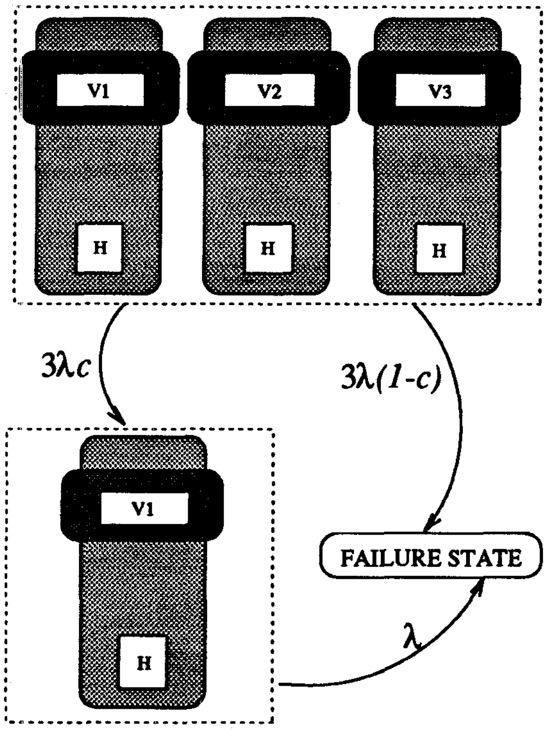

Figure 4: Markov model of system structure

area (HECA) is the lightly shaded region, the software error confinement area (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 will fail if that hardware experiences a fault. The SECA covers only the software component since no other components will be affected by a fault in that component.

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

	Fault Tree basic event probabilities	By case	By time
P_V	Independent software fault in 1 version	0.0958	0.0003
P_{RV}	Related software fault between 2 versions	0.0	6×10^{-7}
P_{RALL}	Related fault between all versions	0.003	0.0
P_D	Independent decider fault activation	0.0001	1×10^{-7}
P_H	Hardware transient fault	7.3×10^{-6}	2.8×10^{-9}
	Markov model parameters		
λ	activation rate of permanent hardware fault	0.00001	0.00001
c	coverage probability	0.999	0.999

Table 4: Parameterization of model

coverage parameter. If the reconfiguration is unsuccessful, the system fails.

3.3 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. The reward structure of the process captures the probability that a single computation will result in an erroneous output. The reward for a given state is the solution of the fault tree model for the computation process in that state.

The fault tree model solution produces, for each state i in the Markov model, the probability q_i that an output error occurs during a single task computation. The Markov model solution produces $P_i(t)$, the probability that the system is in state i at time t. The reward model combines these two measures to produce $Q(t)$, the probability that an unacceptable result is produced at time t.

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

4 Parameterization and Results

The methodology used for estimating the parameters, as well as a discussion of the assumptions made are detailed in the following. A summary of the resulting parameter values is shown in table 4.

4.1 Software parameters

The experimental results from the Lyu-He study [7], shown in tables 2 and 3, were used to estimate the probabilities associated with the activation of software faults.

Let P_V be the probability of an independent fault activation; P_{RV} be the probability of the activation of a related fault between any two versions; and P_{RALL} be the probability of the activation of a related fault affecting all versions. The probability that no errors occur in a three-version configuration, which we will label $C1$ since it relates to Category 1 in tables 2 and 3, is given by

$$C1 = (1 - P_V)^3(1 - P_{RV})^3(1 - P_{RALL}). \quad (1)$$

Similarly, the probability that a single fault is activated is given by the probability that only one independent fault is activated (and no related faults are activated).

$$C2 = 3P_V(1 - P_V)^2(1 - P_{RV})^3(1 - P_{RALL}) \quad (2)$$

Dividing equation (1) by equation (2) yields an equation which is dependent only on P_V, and thus can be used to estimate the probability of the activation of an independent fault.

$$P_V = \frac{C2}{3C1 + C2} \quad (3)$$

The data from table 2 yields an estimate of $P_V = 0.0958$ for the probability of activation of an independent fault in a 3-version configuration. Table 5 compares the probability of activation of 1, 2 and 3 faults as predicted by a model assuming independence between versions, with the observed values. The observed probability of two simultaneous errors is lower than predicted by the independent model, while the observed probability of three simultaneous errors is higher than predicted. For this set of data we will assume therefore that $P_{RV} = 0$ and will instead derive an estimate for P_{RALL}.

Using the assumption that $P_{RV} = 0$, the probability that three simultaneous errors are activated is given by

$$C4 = {P_V}^3 + P_{RALL} - {P_V}^3 P_{RALL}, \quad (4)$$

No. errors activated	Independent model	Observed Probability
0	0.7393	0.7426
1	0.2350	0.2360
2	0.0249	0.0202
3	0.0009	0.0012

Table 5: Comparison of independent model with observed data (by case)

No. errors activated	Independent model	Observed Probability
0	0.999090	0.7426
1	0.000909	0.2360
2	3×10^{-7}	2×10^{-6}
3	3×10^{-11}	0.0

Table 6: Comparison of independent model with observed data (by time)

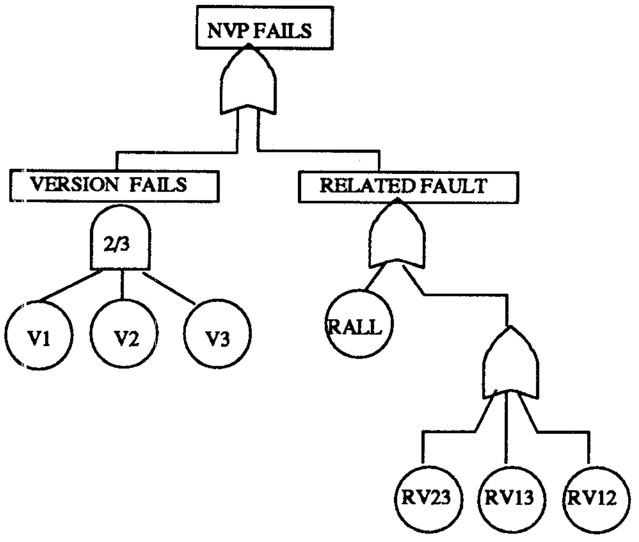

Figure 5: Fault tree model of NVP software system

yielding an estimate of $P_{RALL} = 0.0003$ for the by-case data.

The by-time data in table 3, when used in equation (3) produces $P_V = 0.0003$ as an estimate. For this by-time data, when the failure probabilities which are predicted by the independent model are compared to the actual data (table 6), the observed probability of two errors is an order of magnitude higher than the predicted probability. There were no cases for which all three programs produced erroneous results. Thus, we will estimate $P_{RALL} = 0$ and derive an estimate for P_{RV}. To derive an estimate for P_{RV}, consider the case where two errors are produced. This event could be caused by either the activation of two simultaneous independent faults, or by the activation of a related fault. Also, it depends on the non-failure of the remaining version, either by independent or related fault. Then, considering the three combinations of two failures which can occur, we can use the observed probability of two errors to estimate P_{RV}.

$$C3 = 3(P_V^2 + P_{RV} - P_V^2 P_{RV}) \times \\ (1 - P_V)(1 - P_{RV})^2(1 - P_{RALL}) \quad (5)$$

Equations (3) and (6) can be used to produce the estimate $P_{RV} = 6 \times 10^{-7}$.

For both the by-case and by-time scenarios, the parameters derived from the data were applied to the fault tree model shown in figure 5. For the by-case parameters, the fault tree model predicts a failure probability of 0.0261, while the observed failure probability was 0.0214. Using the by-time parameters, the fault tree model predicts a failure probability of 2.07×10^{-6} while the observed failure probability was 2.3×10^{-6}. In the system analysis section, these parameters will be combined with hardware parameters to predict the overall NVP system reliability.

4.2 Hardware parameters

Typical permanent failure rates for processors range in the 10^{-5} *per hour* range, with transients perhaps an order of magnitude larger. Thus we will use $\lambda_p = 10^{-5}$ per hour for the Markov model.

In the by-case scenario, a typical test case contained 5280 time frames, each time frame being 50 ms., so a typical computation executed for 264 seconds. Assuming that hardware transients occur at a rate $\lambda_t = (10^{-4}/3600)$ *per second*, we see that the probability that a hardware transient occurs during a typical test case is

$$1 - e^{-\lambda_t \times 264 \textit{ seconds}} = 7.333 \times 10^{-6} \quad (6)$$

We conservatively assume that a hardware transient that occurs anywhere during the execution of a task disrupts the entire computation running on the host.

For the by-time data, the probability that a transient occurs during a time frame is

$$1 - e^{-\lambda_t \times 0.05 \textit{ seconds}} = 1.4 \times 10^{-9} \quad (7)$$

If we further assume that the lifetime of a transient fault is one second, then a transient can affect as many as 20 time frames. We thus take the probability of a transient to be 20 times the value calculated in equation 7, or 2.8×10^{-8}.

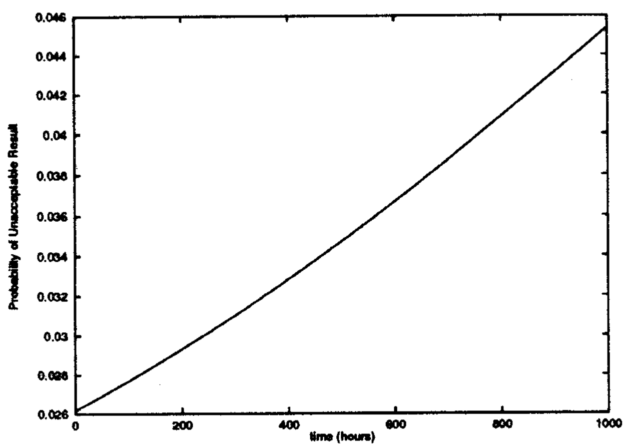

Figure 6: Probability of producing an unacceptable result during a test case, by-case data

Finally, for both the by-case and by-time scenarios, we assume a fairly typical value for the coverage parameter in the Markov model, $c = 0.999$.

4.3 Model solution

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

Figure 6 shows, for the by-case data, the time dependent probability of the NVP system producing an unacceptable result at any time during a test case. The length of a typical test case is 264 seconds, about 4.4 minutes. Initially, in the full-up state, the probability of an unacceptable result is 0.0262, increasing to 0.0454 after 1000 hours, as the probability of operating in the simplex mode or the failure state increases. The probability of producing an unacceptable result while in the simplex mode is 0.0958; and the probability of producing an unacceptable result while in the absorbing failure state is 1.0.

Figure 7 shows, for the by-time data, the time dependent probability of the NVP system producing an unacceptable result during a 50 ms. time frame. In the full up state, the probability of an unacceptable result is 2.17×10^{-6}, increasing three orders of magnitude to 6.83×10^{-3} at 1000 hours. The probability of producing an unacceptable result while in the simplex mode is 3×10^{-4}.

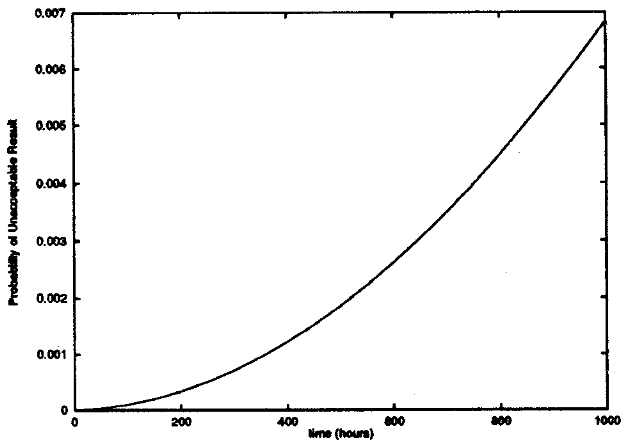

Figure 7: Probability of producing an unacceptable result during a single time frame, by-time data

5 Summary and Conclusions

An integrated model was presented for the analysis of a system intended to tolerate both hardware and software faults. The model included such aspects as software and hardware transients, permanent hardware faults and imperfect coverage.

The software failure parameters for the model were estimated from probabilistic calculations on actual data from an N-version programming experiment conducted at the the University of Iowa and Rockwell. Two scenarios were considered. First, the by-case data counted as a software failure the occurrence of an error in two of the three versions at any point in the 264 second test run. The by-time data, as the other scenario, only counted simultaneous (i.e. in the same frame) errors as a failure.

The rest of the model (i.e. hardware failures) was parameterized by considering fairly typical failure rates. The overall models were solved for time-varying estimates of the probability of producing an unacceptable result. The results show that it is important to consider both hardware and software faults, as the estimates increased with increasing time.

These models were parameterized by considering the results of a single experimental study. The models fit the experimental data in this case, but more data is needed in order to further validate the modeling methodology. Of interest in this particular data set are the relatively low values for the probability of related software faults.

The most interesting result comes from the comparison of the by-case and by-time estimates of the probability of an unacceptable result. The several orders of magnitude difference points to the importance of detecting and correcting errors promptly. If errors are detected only at the end of each simulation run (as in the by-case data) there may be a substantial loss in reliability (four orders of magnitude) as compared with more frequent error detection (as in the by-time data where comparisons were made at each time frame).

6 Acknowledgements

This work was partially funded by NASA AMES Research Center under grant number NCA2-617. The models presented in this paper were solved using SHARPE [9].

References

[1] Algirdas Avižienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[2] Joanne Bechta Dugan and Randy Van Buren. Reliability evaluation of fly-by-wire computer systems. *Journal of Systems and Software*, to appear.

[3] Joanne Bechta Dugan and K. S. Trivedi. Coverage modeling for dependability analysis of fault-tolerant systems. *IEEE Transactions on Computers*, 38(6):775–787, 1989.

[4] K.H. Kim and Howard O. Welch. Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications. *IEEE Transactions on Computers*, 38(5):626–636, May 1989.

[5] Jaynarayan H. Lala and Linda S. Alger. Hardware and software fault tolerance: A unified architectural approach. In *Proc. IEEE Int. Symp. on Fault-Tolerant Computing, FTCS-18*, pages 240–245, June 1988.

[6] Jean-Claude Laprie, Jean Arlat, Christian Beounes, and Karama Kanoun. Definition and Analysis of Hardware- and Software- Fault Tolerant Architectures. *IEEE Computer*, pages 39–51, July 1990.

[7] Michael R. Lyu and Yu-Tao He. Improving the N-version programming process through the evolution of a design paradigm. *IEEE Transactions on Reliability*, June 1993.

[8] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[9] R. Sahner and K. S. Trivedi. Reliability modeling using SHARPE. *IEEE Transactions on Reliability*, R-36(2):186–193, June 1987.