NASA Technical Memorandum 102613

An Experimental Evaluation of Software Redundancy As a Strategy for Improving Reliability

Dave E. Eckhardt, Jr.

Alper K. Caglayan

John C. Knight

Larry D. Lee

David F. McAllister

Mladen A. Vouk

John P. J. Kelly

May 1990

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

I. INTRODUCTION

The use of multiple versions of independently developed software has been advocated as a means of coping with residual design faults in operational software. This approach is characterized by two fault-tolerant software structures known as N-version programming [1] and recovery blocks [2] that are modeled after hardware approaches to fault tolerance known respectively as N-modular redundancy and stand-by sparing. Although redundant hardware structures can cope with independent hardware failures resulting from physical wearout, the effectiveness of redundancy applied to software is unclear. Analytical arguments [3,4] and empirical evidence [5,6,7] indicate that failures in redundant software components are unlikely to be independent. The degree to which design faults manifest themselves as dependent failures determines the effectiveness of redundancy as a strategy for improving software reliability.

An important question is whether the multi-version software structure can be *depended on* to provide a sufficient measure of increased reliability to warrant its use in critical applications. The magnitude of reliability gain that can be achieved using software redundancy is generally unclear. Several studies have concluded that reliability was improved although the conclusions were based on only a few versions (e.g., two versions [8], three versions [9,10] and six versions [11]). Other experiments have produced more versions but under less rigorous development methods [12,6,7,13,14] and in some studies relatively low quality codes were produced. There are other issues that might influence the use of redundant software. For example, multiple versions generally require more resources during development and they pose new problems for maintenance personnel. However, the fundamental issue is reliability improvement; none of these other issues need be addressed at all unless multi-version software can be depended on to produce reliable systems for critical applications.

A key assumption is that design diversity will result in software versions that have sufficiently different failure characteristics such that fault-tolerant structures can provide "continued service" [15] in the presence of failures of the component versions. Although models of fault-tolerant software have been proposed in the literature [3,4,16,17,18,19], Littlewood and Miller [4] point out that there does not seem to exist any precise definition, much less measures of the "degree of diversity." Nevertheless, a number of ways have been suggested to enhance the diversity of developed software; e.g., different programming environments, programming languages, and algorithms [20,21]. Avizienis [11] suggests that the use of independent programming teams is the "baseline dimension for design diversity". The present study investigates the gain in reliability achieved under this baseline dimension. The approach taken is to consider a large set of programs that have been developed and validated to a common specification using independent programming teams. On average, these programs will fail with some probability p for the given application, development process, and cost constraints. Based on a model of dependent failures [3], we examine the change in the failure probability P_N that can be obtained with N of these programs arranged in a redundant structure.

The development process by which the versions are produced is central to the issue of software redundancy as a reliability enhancing technique. A basic question that arises is to what extent and over what phases of the development are the programming teams isolated in order to increase the independence of the resulting software? For example, are open discussions of software specifications allowed? At what point are the versions integrated into a redundant structure; after each version has been subjected to an independent validation or at an earlier stage to take advantage of multi-version testing as a validation procedure? Any development process tailored to a fault-tolerant software project would likely produce feedback to all software developers based on information gained from individual development teams. As a general rule, it would seem that the earlier the independent teams come into contact, with the potential of directly or indirectly influencing each other, the greater the possibility that common design faults would ultimately reside in the redundant software versions. For this experiment, as much independence as possible was maintained between the programming teams. In addition, a separate and independent validation was used for each version of software.

This paper describes a large-scale experimental evaluation of software redundancy. Forty programmers from four universities produced, in teams of two, twenty versions of software for a rather complex aero-space application. An additional twenty programmers performed an independent certification of the versions. The versions were then submitted to an independent organization for evaluation in an operational setting. In Section II, the phases of the experiment are described. The procedures used to evaluate the results of the experiment are developed in Section III and the results are presented in Section IV.

II. EXPERIMENT PHASES

Application Selection

After establishing the objective and the protocol for the experiment, the application was selected. An application was sought that could be programmed within time constraints, presented a programming challenge that was neither too simple nor too difficult, and was a realistic application requiring high reliability so as to be a potential candidate for fault tolerance. The goal was to obtain a representative set of high quality programs for such an application. Initially an application was sought for which a good specification existed so that the investigation could focus on the effectiveness of redundancy, given a high-quality specification from the outset. Because we chose, instead, what we felt to be a good application that required a specification to be developed, the experiment required an additional certification phase to correct deficiencies in the software resulting from inadequacies in the specification.

The selected application was the sensor management of a Redundant Strapped Down Inertial Measurement Unit (RSDIMU) [22]. An RSDIMU is a component of a modern inertial navigation system and provides acceleration data that is integrated to determine velocity and position. Acceleration is measured by a set of eight accelerometers that are mounted, in pairs, on four faces of a semioctahedron (Figure 1). During flight the accelerometers are read and the vehicle acceleration computed at regular intervals. The accelerometers have many sources of inaccuracy (e.g., temperature, noise, misalignment) and are subject to mechanical failure. Accelerometer failures manifest themselves as inaccuracy beyond that expected and deciding which of eight accelerometers has failed involves complex computations.

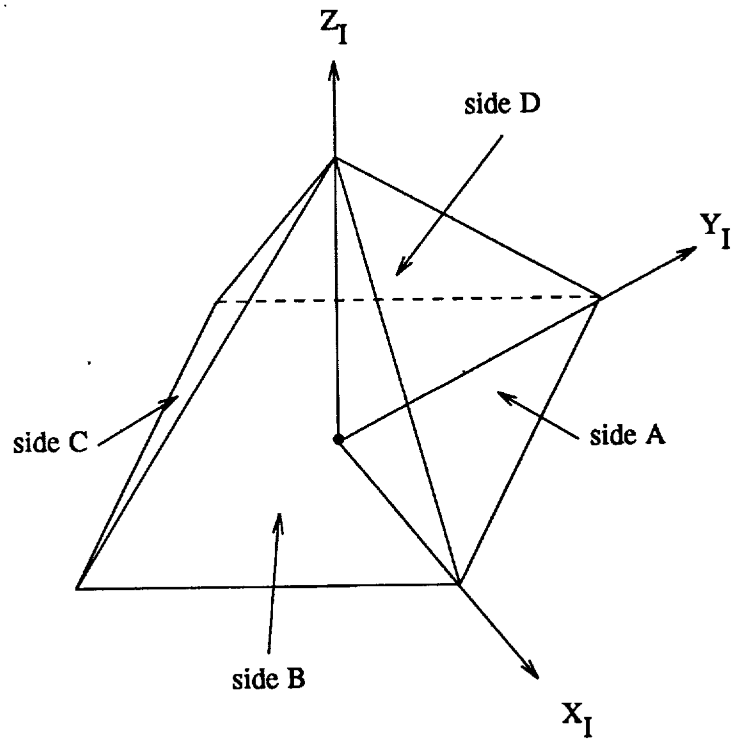


Figure 1. RSDIMU Geometry Showing Instrument Frame of Reference

The application involves calibration of the accelerometers with a known acceleration (gravity), analysis of noise mixed in with the data from the accelerometers, sensor Fault Detection and Isolation (FDI) and, using only those accelerometers deemed to be good, computation of the acceleration of the vehicle in which the instrument is located. The acceleration computation is complicated by the existence of many frames of reference, most of which are orthogonal but some are not. Each sensor operates in its own frame of reference and there is a vehicle frame of reference, an instrument frame of reference, and a frame of reference through which the vehicle is moving. For navigation purposes, this last frame of reference is the one in which the acceleration has to be computed. However, the accelerometers provide data in their own frames of reference and a number of transformations are required. The application is further complicated by a temperature sensitivity in the sensors, known errors in the positioning of the sensors on the semioctahedron during manufacture, and sensor data being supplied in a form characteristic of the sensor hardware rather than in engineering units that can be used for navigation.

In summary, the various attributes of this application are typical of flight instruments involving physical vectors, transformation between various coordinate systems, operation in a noisy environment, and discrete modes. A requirements specification (in [23]) was prepared jointly by staff at the Research Triangle Institute who have conducted experiments in fault-tolerant software, and by the staff of Charles River Analytics, a company with expertise in the development and analysis of flight-critical software for aircraft applications. The specification was reviewed by researchers actively involved in fault-tolerant software research at the four universities participating in this study: the University of Virginia, the University of Illinois, North Carolina State University, and the University of California. Based on their reviews, several revisions were made to the specification. The final form of the specification is sixty-five pages in length. It is written in English and includes a considerable amount of mathematics.

Software Development

The four universities were involved in the software development phase of the experiment to obtain as many software versions as possible, within budget constraints, and to enhance the prospect of achieving independent development of these versions. They provided a large population of programmers from which to choose the development teams and also provided a population with diverse academic and cultural backgrounds. Because the selected application required substantial mathematical knowledge, graduate students were hired in both computer science and applied mathematics. Programming teams were formed such that each team had at least one member with experience in each discipline.

Development started with a briefing to the programmers of the goals of the experiment, the protocol to be followed, and the distribution of documents. The preparation of the versions for this experiment followed a rigorous development schedule. Although not equivalent to an industrial process, it was as close as can be reasonably achieved in a university environment. The development protocol required the teams to generate several work products according to a set of deadlines. The development occurred over a ten-week period during the summer so that the teams had full time to devote to the project. Two weeks after receiving the specification the designs were due and design reviews had to be held. The reviews were monitored unobtrusively by the researchers to ensure compliance with the protocol. After a further two weeks, the source code was required and it too was the subject of a formal review. At the end of eight weeks the tested programs had to be delivered and subjected to the acceptance procedure. This procedure consisted of one hundred test cases, fifty of which were functional tests and the remaining randomly generated. The final two weeks were provided to permit the teams to make any necessary changes to complete the acceptance procedure.

During the development phase, as much independence as possible was maintained between the programming teams. Direct communication between teams was forbidden, and this was ensured to some extent by using four geographically-separate universities to supply and host the teams. There was also the need, however, for all participants to have access to the same information base so as not to confound the results of the experiment. No additional information was provided for questions from the programming teams that suggested a particular design approach. However, for those questions that generated corrections and clarifications to the specifications, both the questions and the responses were broadcast to all development teams. These question/answer pairs were viewed as part of the specification.

Independent Certification

The versions produced during the development phase [24] proved to be insufficiently reliable and contained common errors resulting from inadequacies in the software specifications. Since the study was examining the redundancy issue, this was considered to be an unfair evaluation and, since there would be no compromise of the independence of the development process, an additional *certification* phase was initiated. During the certification phase, the original specification was restored to a single document by incorporating clarifications and removing specification ambiguities revealed by the questions from the original development teams. Based on this modified specification, an elaborate acceptance procedure was developed for the certification phase. This acceptance procedure consisted of 1,196 input cases of which 796 were systematic, functional tests based on the specification and 400 were randomly generated tests. Three of the four universities supplied programmers who modified the versions to meet the criteria of the new acceptance procedure. One programmer was allocated to each version for certification and, to maintain independence of the versions of software, none of the programmers involved in the original development were involved in the certification. The twenty versions were allocated at random to the certification sites and modified over an eight-week period. In total, seven man-months of effort were allocated to the development and certification of each version.

Independent Operational Evaluation

The certification sites submitted the programs to Charles River Analytics for an independent evaluation of the programs in an operational environment. This operational environment simulated the motion of a commercial aircraft operating over short-hauls with a mean flight time of fifty minutes. The simulated environment included an aircraft simulation with a guidance control law, an RSDIMU simulation, and a means for asserting correctness of the submitted software.

The aircraft simulation, which implemented differential equations for translational dynamics, rotational kinematics, and an aircraft guidance law, was required to generate specific force and aircraft attitude variables from a feasible input space; that is, one that exhibited both static and temporal physical consistency. Flights included typical take-off, cruise, and landing patterns with wind gusts.

The RSDIMU simulation used the specific force and attitude information and generated the sensor outputs for the versions of sensor redundancy management software being evaluated. Sensor errors were introduced by misalignment, noise, and temperature effects.

The aircraft simulation was randomized over trajectory segments and maneuver parameters. To test the software under a variety of operating conditions, the RSDIMU simulation parameters were randomly changed from flight to flight. These parameters accounted for possible operational differences across flights such as differences in instrument mounting geometry, alignment, ambient temperature, sensor noise statistics, etc. At intervals in the flights the acceleration estimates and the state of the sensors (failed or operational) were checked for correctness for each of the twenty versions of redundancy management software. The results presented in Section IV were obtained after approximately 921,000 of these interval checks from the simulated flights.

III. ANALYSIS METHOD

Failure Probability

We define failure of an N-version system on a given input to be the event that a majority of the versions produce incorrect output. For this study, we do not distinguish between identical failures and dissimilar failures of the versions. Although it is true that the latter type of failure could be detected by a voter and an alarm raised, we were not investigating the error-detection capabilities of N-version systems. The viewpoint taken here is that if there is a requirement for fault tolerance, then there is also a requirement for the system to provide a continuation of service in the presence of software failures.

In this section we present the basic ideas used to analyze the reliability improvement of multiversion software. An intensity function $\theta(x)$ represents the probability that a program, randomly chosen out of a population of versions that are developed according to a common set of requirements, will fail on a particular input x. It is assumed that the versions are tested on a random input series selected according to a usage distribution Q that reflects the actual operational use of the software. The usage distribution gives the probabilities $Q(A)$ that inputs occur in subsets A of the input set Ω. The simulated flight conditions described previously provided this distribution. Although programs are not actually picked at random from a large population of programs, the input sets on which the different versions fail will likely vary. Thus, to evaluate a multi-version structure, one must either physically examine the failure probability of every possible system that can be formed from subsets of the available versions or make use of a failure intensity distribution, which we describe next, to determine an average failure probability.

A general model for the probability P_N that a majority $m=(N+1)/2$ of the components fail in a system of N versions ($N=1,3,5,\ldots$) is given by [3]

$$P_N = \int_{[0,1]} \sum_{l=m}^{N} \begin{bmatrix} N \\ l \end{bmatrix} [\theta(x)]^l [1-\theta(x)]^{N-l} dQ(x). \tag{1}$$

This can be written in the form

$$P_N = \int_{[0,1]} \int \sum_{l=m}^{N} \begin{bmatrix} N \\ l \end{bmatrix} z^l (1-z)^{N-l} dG(z) \tag{2}$$

where

$$G(z) = \int_{\{x \mid \theta(x) \le z\}} dQ$$

is the cumulative failure-intensity distribution on the interval [0,1]. $G(z)$ gives the probability that an input is chosen so that the proportion $\theta(x)$ of versions that fail is in the range $0 \le \theta(x) \le z$ where $0 \le z \le 1$.

The assumptions giving rise to (1) are that (a) corresponding to the different development processes, the version failures are conditionally independent, given a particular input, and (b) the versions fail on input x with the same probability, namely $\theta(x)$. Unconditionally, since inputs are not fixed quantities in an operational situation, version failures are dependent whenever $\theta(x)$ varies for different inputs. These and other aspects of this model have been discussed in some detail in [3] and [25].

Estimation of N-Version Failure Probability P_N

Suppose that n software versions are tested on an input series $x_1, x_2, \ldots, x_k$ of length k. Let $u_y(x)=1$ if y out of n versions fail on input x and $u_y(x)=0$, otherwise. The quantity $g(y)$, defined by

$$g(y) = k^{-1} \sum_{i=1}^{k} u_y(x_i),\ y=0,1,2,\ldots,n, \tag{3}$$

gives the empirical proportion of inputs for which y out of n versions fail.

An unbiased estimator of P_N can be constructed by considering all possible N-version systems that can be formed by selecting subsets of size N out of the n available versions. Let Ψ denote all subsets of size N. For J, an element of Ψ, define the indicator function $u_J(x,l)=1$ if l versions in the set J fail on input x and $u_J(x,l)=0$, otherwise. Then $k^{-1} \sum_{i=1}^{k} \sum_{l=m}^{N} u_J(x_i,l)$ is the proportion of inputs on which the system of N versions fails. Averaging over all possible subsets gives the following unbiased estimator of P_N

$$\tilde{P}_N = k^{-1} \binom{n}{N}^{-1} \sum_{\Psi} \sum_{i=1}^{k} \sum_{l=m}^{N} u_J(x_i,l). \tag{4}$$

For $u_y(x_i)=1$, $\sum_{\Psi} u_J(x_i,l) u_y(x_i)$ is the number of ways of selecting l versions out of y that fail on input x_i and $N-l$ versions out of $n-y$ versions that do not fail on input x_i. We have

$$\sum_{\Psi} u_J(x_i,l) = \sum_{y=0}^{n} \sum_{\Psi} u_J(x_i,l) u_y(x_i) = \sum_{y=0}^{n} \binom{y}{l} \binom{n-y}{N-l} u_y(x_i). \tag{5}$$

Substituting (5) into (4) and using the definition of $g(y)$ given in (3) we obtain

$$\tilde{P}_N = \sum_{y=0}^{n} \binom{n}{N}^{-1} \sum_{l=m}^{N} \binom{y}{l} \binom{n-y}{N-l} g(y) \tag{6}$$

where $\binom{a}{b}=0$ if b>a.

The assumption that $\theta(x)$ does not vary with different inputs implies that the versions fail independently. In this case (1) yields the classical estimator of P_N given by

$$\bar{P}_N = \sum_{l=m}^{N} \binom{N}{l} \hat{p}^{\,l}(1-\hat{p})^{N-l} \tag{7}$$

where $\hat{p}$ is an estimate of the failure probability of a single version obtained by setting N=1 in (6).

Variation in the estimate of P_N will occur when testing the versions on a different input series. To obtain the theoretical variance of $\tilde{P}_N$, let Y_i be the number of versions out of n that fail on input x_i so that

$$\tilde{P}_N = k^{-1} \sum_{i=1}^{k} \sum_{y=0}^{n} a_{ny} I(Y_i = y) \tag{8}$$

where

$$a_{ny} = \binom{n}{N}^{-1} \sum_{l=m}^{N} \binom{y}{l} \binom{n-y}{N-l}$$

and $I(Y_i=y)$ is the indicator function of the event $(Y_i=y)$. Under the assumption that $Y_1, Y_2, \ldots, Y_k$ are independent random variables, the expression in (8) is a linear function of independent random variables $u_i = \sum_{y=0}^{n} a_{ny} I(Y_i=y), i=1,2,\ldots,k$. Thus the variance of $\tilde{P}_N$ is τ^2/k where

$$\tau^2 = \sum_{y=0}^{n} (a_{ny})^2 \phi(y) - \left(\sum_{y=0}^{n} a_{ny}\phi(y)\right)^2 \tag{9}$$

and $\phi(y) = P(Y_i = y), y=0,1,2,\ldots,n$. This variance can be estimated by replacing $\phi(y)$ in (8) by $g(y)$. Although the use of (9) requires the extra assumption that $Y_1, Y_2, \ldots, Y_k$ are independent, we note that this assumption is implicitly made in replicated software testing experiments [26]. It should also be noted that (9) does not measure the variability due to the process of developing a different set of versions. The latter source of variation is influenced by the number of versions and this is generally limited by the development costs.

The significance of (6) and (7) to the present study is that they suggest different ways of analyzing the effectiveness of the multi-version software approach. Comparing $\tilde{P}_N$ with the estimated failure probability $\hat{p}$ of a single version indicates the magnitude of the decrease in failure probability that can be achieved using N versions rather than a single version. Additionally, since $\bar{P}_N$ results from the general model by making an assumption that versions fail independently, comparing $\tilde{P}_N$ with $\bar{P}_N$ indicates how close version failures are to being independent.

Partitioned Usage Distribution

Based on functional performance requirements, the RSDIMU's software reliability is measured under various operating conditions. That is, the RSDIMU as a system, including all mechanical and electronic hardware and software, is required to remain operational for up to two sequential accelerometer failures, and must provide the indication of three sequential accelerometer failures. This is known as fail-op/fail-op/fail-safe operation. These cases are included in the following mutually exclusive conditions defining six system states

$S_{i,j}$ = { *i sensors previously failed* and *j of the remaining sensors fail* | $i=0,1,2\,;j=0,1$ }.

The software failure probability under each case is defined as the conditional probability $P(F \mid S_{i,j})$ where F is the event that a version's output is incorrect. The conditional probabilities are used rather than the unconditioned failure probability

$$P(F)=\sum_{i=0}^{2}\sum_{j=0}^{1}P(F \mid S_{i,j})P(S_{i,j})$$

since the latter would be misleading from a functional performance view. That is, a low hardware failure rate reduces the significance of the software failure rate. For example, let $P_{HF} \ll 1$ be the hardware failure rate for the RSDIMU accelerometers (typically, $P_{HF} \leq 10^{-4}$ failures/flight hour). Then, for the prior probabilities $P(S_{0,0}) \approx 1$ and $P(S_{0,1}) \ll 1$, the software failure probability under a single sensor failure $P(F \mid S_{0,1})$ would not significantly contribute to the overall failure probability even though it is crucial that the sensor FDI software works correctly under this failure condition.

Correctness Determination

In fault-tolerant software experiments, a problem that arises is how to efficiently detect software failures during the evaluation or simulated operational phase, typically involving many test cases. Correctness can be determined using either (1) comparison of outputs of a large number of versions, (2) a golden version, or (3) a consistency relationship between inputs and outputs that is necessary and sufficient to assert correctness. The first approach requires an assumption that, for the number of versions being evaluated, identical and incorrect answers of all the versions is a remote possibility while the second approach assumes that the golden version is not incorrect in the same way as the versions under evaluation (presumably an error in the golden program that causes a conflict with the versions being evaluated could be attributed to the golden program). For the present study, the correctness of a software version's acceleration estimate makes use of the third approach. Environmental information (i.e., noise added to the sensor readings, quantization and misalignment induced errors, true vehicle state) is used to compute a correctness criterion. This information is not available to the RSDIMU redundancy management software. An acceleration estimate $\hat{x}$ for sensor measurement vector y is given by

$$\hat{x} = [C^T C]^{-1} C^T y$$

where C is the transformation matrix from the instrument frame to the navigation frame of reference. Since sensor measurements are related to the true acceleration by

$$y = Cx + \tilde{y}$$

where $\tilde{y}$ is the sensor inaccuracy caused by noise, misalignment and quantization, then

$$C^T C(\hat{x} - x) = -C^T \tilde{y}$$

is a necessary and sufficient criterion to assert correctness for acceleration estimates. Note that x and $\tilde{y}$ are not available to each version's estimation software. Also, this criterion is simplier to implement than another version of the estimation software.

IV. EXPERIMENT RESULTS

Version Failures

The number of failures of each of the twenty versions is given in Table 1 for the six operating conditions discussed in Section III. The programs were generally more reliable at filtering false alarm conditions due to noise, temperature variations, etc.(i.e., states $S_{0,0}$, $S_{1,0}$ and $S_{2,0}$) than cases where a sensor failure occurred in the presence of these conditions (states $S_{0,1}$, $S_{1,1}$ and $S_{2,1}$). This was because of the added complexity of processing the failed sensor and because some programs did not detect or identify the failed sensor.

As stated previously, the goal of the present study was to obtain a representative set of high quality programs for a realistic and complex application. Seven of the twenty versions did not exhibit any failures over 920,746 test cases and an additional three versions exhibit only a single failure. At the other

Version Number	Number of failures under state $S_{i,j}$ where i sensors have previously failed and j of the remaining sensors fail					
	$S_{0,0}$	$S_{0,1}$	$S_{1,0}$	$S_{1,1}$	$S_{2,0}$	$S_{2,1}$
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	2	4	51	9	63
5	0	1	0	0	0	0
6	1	0	0	0	0	0
7	0	3	0	12360	110	41032
8	196	561	14	128	7	89
9	187	279	0	0	0	0
10	0	2	21	80	10	67
11	14	179	20	155	13	41662
12	0	0	0	0	0	0
13	0	1	0	0	0	0
14	0	0	13	45	7	36
15	2	46	0	0	0	0
16	0	0	0	0	0	0
17	0	0	14	47	7	42
18	53	194	13	55	7	40
19	0	0	0	0	0	0
20	0	0	0	0	0	0
Total number of test cases	309,442	134,135	129,999	101,151	102,510	143,509

Table 1. Version Failure Counts

extreme, however, versions 7 in states $S_{1,1}$ and $S_{2,1}$ and version 11 in state $S_{2,1}$ were substantially less reliable than the other versions. The failure events of Table 1 result in the failure intensity distributions shown in Table 2. This latter table represents the probabilities of versions failing on the same input. The average of these distributions provides an estimate of the single version failure probability p.

Faults Causing Coincident Failures

Software faults that result in coincident failures of the program versions are given in Table 3. Fault numbers 1 through 3 are due to a common and apparently inadequate understanding of the different properties of nonorthogonal coordinate systems. In this application there were four nonorthogonal coordinate systems, i.e., measurement frames of reference. Each of these measurement frames is defined by a small misalignment transformation with respect to an orthogonal sensor frame of reference. The

Proportion of versions failing together	Probabilities of coincident failures under state $S_{i,j}$ where i sensors have previously failed and j of the remaining sensors fail					
	$S_{0,0}$	$S_{0,1}$	$S_{1,0}$	$S_{1,1}$	$S_{2,0}$	$S_{2,1}$
0/20	.999364	.995707	.999722	.875038	.998800	.568626
1/20	.000032	.002214	.000154	.124040	.001044	.285362
2/20	.000433	.000626	.000008	.000386	.000078	.145684
3/20	.000126	.000171	.000008	.000089	.000010	.000028
4/20	.000039	.000939	.000008	.000020	.000000	.000028
5/20	.000006	.000343	.000069	.000109	.000010	.000084
6/20	0	0	.000023	.000189	.000039	.000111
7/20	0	0	.000008	.000119	.000010	.000049
8/20	0	0	0	.000010	.000010	.000028
9/20	0	0	0	0	0	0
10/20	0	0	0	0	0	0
11/20	0	0	0	0	0	0
12/20	0	0	0	0	0	0
13/20	0	0	0	0	0	0
14/20	0	0	0	0	0	0
15/20	0	0	0	0	0	0
16/20	0	0	0	0	0	0
17/20	0	0	0	0	0	0
18/20	0	0	0	0	0	0
19/20	0	0	0	0	0	0
20/20	0	0	0	0	0	0
Average Probability	.000073	.000472	.000038	.006387	.000083	.028928

Table 2. Failure Intensity Distributions

specification explicitly highlights the nonorthogonality of the measurement frames and also supplies the transformation matrices between the corresponding orthogonal and nonorthogonal frames. Although the faults in these six programs are not all identical, they do result in coincident version failures. Since the misalignment angles are small, these faults do not normally induce failures. Only for the rarely occurring input conditions when the misalignment angles and sensor noises are large and the measured acceleration is large and oriented along one of the misaligned axes do these faults induce coincident failures. The six versions in this set were fairly evenly distributed across the development sites, two sites each producing one version with these faults and the other two sites each producing two versions with these faults.

Fault numbers 4 and 5 are logical faults producing most of the coincident failures in states $S_{0,0}$ and $S_{0,1}$. For the four faces (A, B, C, D) of the RSDIMU, there are six edge relations (AB, AC, AD, BC, BD,

Fault number	Software Faults	Version Numbers
1	A unit vector in an orthogonal coordinate system was apparently assumed to remain a unit vector after a nonorthogonal transformation	4, 10
2	Failure isolation algorithm was implemented in a coordinate system other than specified	8, 14, 18
3	Vector components were apparently assumed to remain the same after a small angle transformation	17
4	Three edge out-of-tolerance edge relations were apparently assumed to have a face common to all of the out-of-tolerance edge relations	8, 9, 18
5	Four out-of-tolerance edge relations were incorrectly processed	8, 9, 11, 15, 18
6	Test threshold computed incorrectly	11
7	Variable initialized incorrectly	7

Table 3. Faults Causing Coincident Failures

CD) to be evaluated. The specification states that a face is failed if and only if all (three in this case) edge relations involving that face are out of tolerance. Versions containing fault number 4 seem to have incorrectly assumed that there will always be a face common to all the out-of-tolerance edge relations whenever any three edge relations are out of tolerance. The specification also states that at most a single face will fail on a given execution (i.e., five or six edge relations will not be out of tolerance in states $S_{0,0}$ and $S_{0,1}$). Five versions seem to be based on the incorrect assumption (fault number 5) that four out-of-tolerance edge conditions will not occur on the same input.

Version 11 also contains a different type of fault in its FDI module, namely, in the computation of the test threshold used in the FDI test. The specification calls for converting the units of the input sensor noise standard deviation, from meters/sec^2 to integer counts using the average of the working accelerometer slopes. Version 11 performs the conversion using the slope of the particular sensor being tested, not the average of all slopes. Given the noise distributions involved, the slope of a single sensor does not greatly differ from the average of all sensors. Hence, this fault causes failures in cases when the noise level of a suspect sensor is significantly different from the average. Although this fault is different from the previously described faults, the failures resulting from this fault were coincident with other version failures.

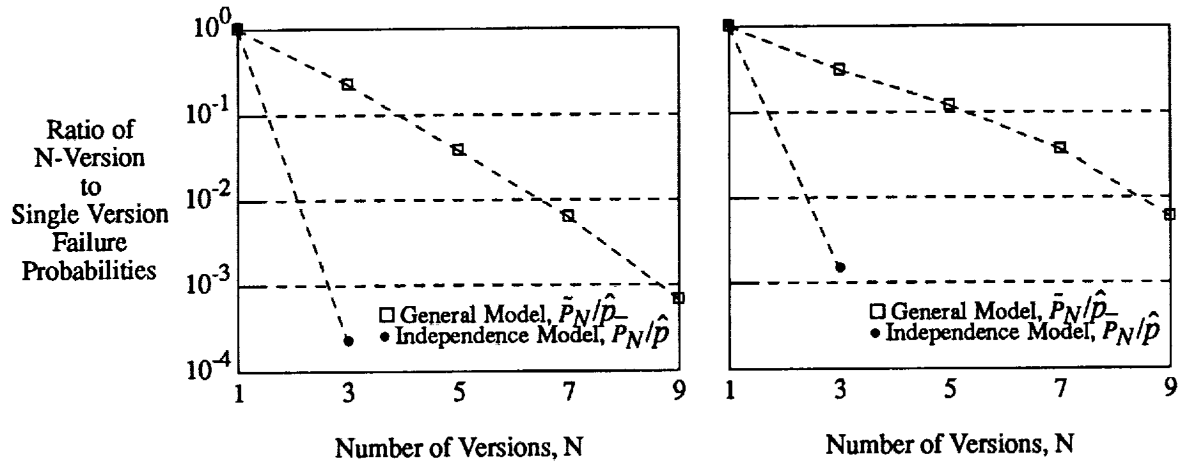
Version 7 exhibits erratic failures in its FDI module because of a failure to initialize a variable before using it in a conditional statement. The fault leads to frequent failures in the FDI related outputs in state $S_{2,1}$.

In summary, six versions have mathematical faults arising from the same point in the specification, although the specification indicated the proper course of action. The specification is judged to be clear and this assessment is further justified by the fact that the remaining fourteen of the twenty versions did not contain these faults. However, when the software developers departed from the specification either through oversight or taking a more independent approach, their lack of depth in understanding this aspect of the specification became apparent. Five versions contain logical faults causing coincident failures. Of these, only two versions were in the set of six having the mathematical faults. Dissimilar (logically unrelated) faults causing coincident failures were also observed in this experiment. Such failure behavior has been previously observed [27] and suggests that similar design faults are only part of the problem that must be solved for redundant software systems. In total, these errors induce coincident failures with failure intensities ranging from 2/20 through 8/20 as shown in Table 2.

Effectiveness of Redundancy

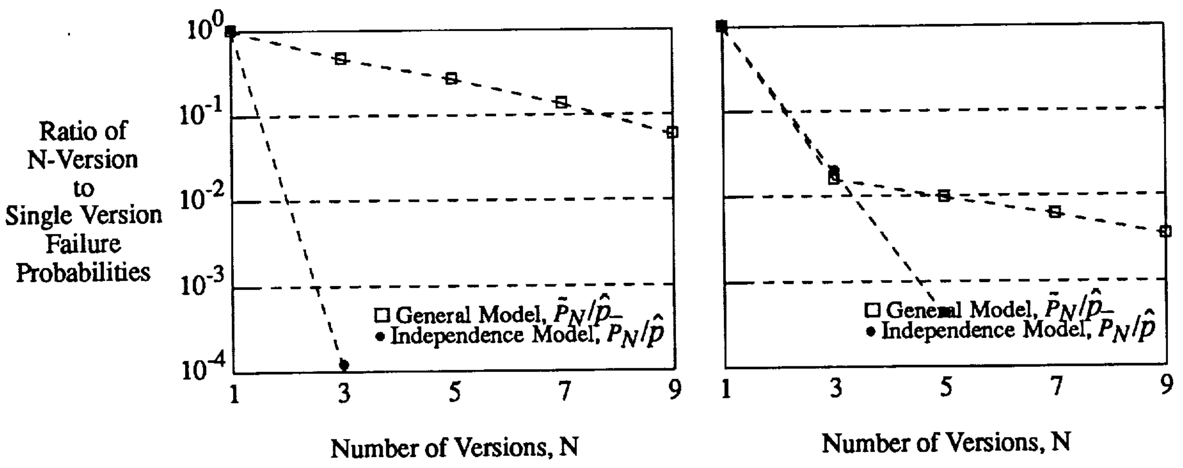
In Figures 2-4, N-versions systems are compared to single versions by plotting the ratio of the estimated failure probability $\tilde{P}_N$ for an N-version system obtained under the general model to the estimate $\hat{p}$ obtained for a single version system. The assumption of independence is clearly not justified under this model as is evident from the plot of $\bar{P}_N/\hat{p}$, also shown in these figures. The independence assumption leads to predictions of a substantial improvement in failure probability when comparing an N-version system with a single version. For example, in Figure 2(a) the plot of $\bar{P}_N/\hat{p}$ indicates that P_3 is only a small fraction (0.00022) of the failure probability of a single version. In comparison, estimates based on the general model are far less favorable to N-version programming. For example, the failure probability of a 3-version system is only about four times smaller (i.e., the fraction 0.23) than the failure probability of a single version. This is typical of all the states except states $S_{1,1}$ (Fig 3b) and $S_{2,1}$ (Fig 4b) where the plots indicate that the failure probabilities of 3-version systems are twelve and sixty-four times smaller, respectively, than the failure probability of single versions. This is due to the effect of program version 7 in state $S_{1,1}$ and program versions 7 and 11 in state $S_{2,1}$ (see Table 1). For these cases, the larger reductions are expected since not only is *N*-version a good technique to protect against the effects of versions that fail infrequently when other versions do not fail but also to protect against a few bad versions that fail often, as is the case here.

For the other four states, however, the programs are uniformly more reliable and, there are less single program failures. When the programs did fail there was a high intensity (up to 8/20) of coincident failures (Table 2). The multi-version approach is not as effective under these conditions; that is, the average failure probabilities of 3-version systems is only two to five times smaller than the average failure probabilities of single version systems.



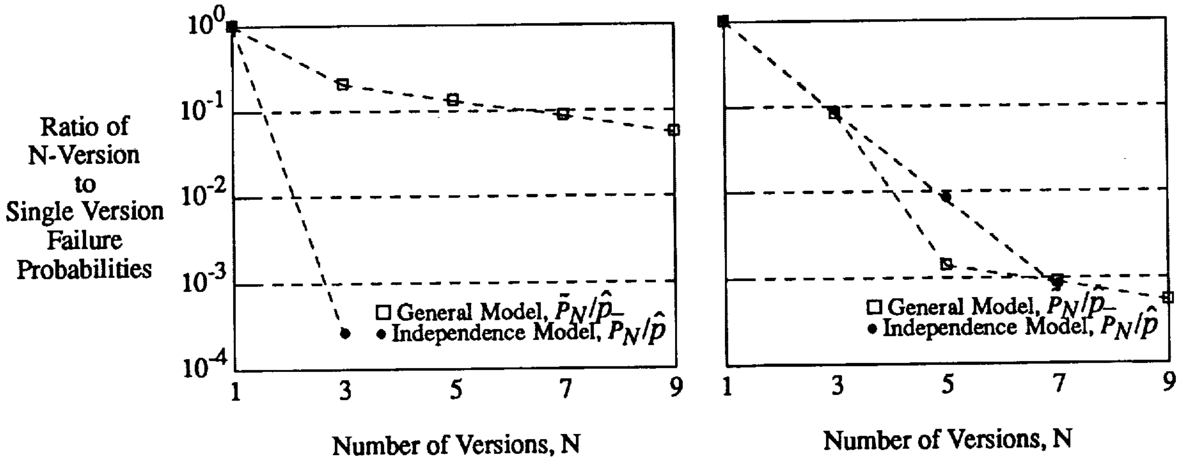
(a) State $S_{0,0}$, no additional sensors fail (b) State $S_{0,1}$, one additional sensor fails

Figure 2. Ratio of the failure probability of an *N*-Version system to the failure probability of a single version estimated under the general model and the independence model. The RSDIMU initial state is fail-op/fail-op/fail-safe (i.e., no failed sensors).



(a) State $S_{1,0}$, no additional sensors fail (b) State $S_{1,1}$, one additional sensor fails

Figure 3. Ratio of the failure probability of an *N*-Version system to the failure probability of a single version estimated under the general model and the independence model. The RSDIMU initial state is fail-op/fail-safe (i.e., one failed sensor).



(a) State $S_{2,0}$, no additional sensors fail (b) State $S_{2,1}$, one additional sensor fails

Figure 4. Ratio of the failure probability of an *N*-Version system to the failure probability of a single version estimated under the general model and the independence model. The RSDIMU initial state is fail-safe (i.e., two failed sensors).

Variability

Table 4 shows the standard errors of the estimates of P_3 for each of the six operating conditions. The standard errors measure variability across repetitions of an input sequence and indicate, for the particular set of twenty versions, that P_3 is estimated with high precision. The standard errors in Table 4 are

System State	$S_{0,0}$	$S_{0,1}$	$S_{1,0}$	$S_{1,1}$	$S_{2,0}$	$S_{2,1}$
Estimate, $\tilde{P}_3$	1.7×10^{-5}	1.4×10^{-3}	1.8×10^{-5}	1.0×10^{-4}	1.7×10^{-5}	2.4×10^{-3}
Standard error, $k^{-1/2}\tau$	1.6×10^{-6}	1.0×10^{-4}	4.8×10^{-6}	1.4×10^{-5}	6.0×10^{-6}	1.8×10^{-5}

Table 4. Standard Errors of Estimates

smaller than estimates of P_3 suggesting that these estimates would change little if there were a larger number of inputs.

In addition to the variability across repetitions of the input sequence, the other source of variability in this experiment is the variability across repetitions of the development process. This latter source of variability is, of course, the primary motivation for this study. It is clear from the individual failure events of Table 1 that the distribution of failure probability is highly skewed with the bulk of the probability mass at the origin because of the large number of versions that do not exhibit failures. Table 5 shows, for each of the twenty versions, the percentage of the 171 triples containing a particular version that have fewer failures than the version itself. Thus, Table 5 indicates the *frequency* with which triples improve on the failure characteristics of a single version while Figures 2-4 show the average *magnitude* of the improvement.

The zero entries in Table 5 reflect the fact that it is not possible for a triple to improve upon a version that does not fail. For a version that does exhibit failures, the percentage of triples having fewer failures than the version itself ranges from 59.6% to 100% over the six system states. On the other hand, if we *do* consider that a high number of versions do not fail, then there is a less than even chance (24.7%-42.4% over the six system states) that a randomly selected version from this set can be improved by including it in a triple. However, not shown in Table 5 is the fact that there are a large number of triples made up entirely of versions that do not fail so that, in effect, there is also a high probability (89.2%-95.3%) that the triple will not be worse than the single version.

Version Number	Percentage of triples containing version number having fewer failures than the version itself -- measured for system states $S_{i,j}$					
	$S_{0,0}$	$S_{0,1}$	$S_{1,0}$	$S_{1,1}$	$S_{2,0}$	$S_{2,1}$
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	94.2	91.2	99.4	98.8	99.4
5	0	68.4	0	0	0	0
6	94.2	0	0	0	0	0
7	0	94.2	0	100.0	100.0	100.0
8	100.0	100.0	78.4	100.0	59.6	99.4
9	89.5	89.5	0	0	0	0
10	0	94.2	100.0	100.0	99.4	99.4
11	70.2	98.2	100.0	100.0	100.0	100.0
12	0	0	0	0	0	0
13	0	68.4	0	0	0	0
14	0	0	61.4	68.4	59.6	80.7
15	61.4	61.4	0	0	0	0
16	0	0	0	0	0	0
17	0	0	78.4	87.7	59.6	97.1
18	79.5	79.5	61.4	99.4	59.6	76.0
19	0	0	0	0	0	0
20	0	0	0	0	0	0
Average	24.7	42.4	28.5	37.7	31.8	37.6

Table 5. Percentage of triples containing an individual version that have fewer failures than the version itself.

V. CONCLUSIONS

The results presented in this paper generally show a modest gain in reliability using redundant software configured in a *N*-version structure. This observation is based on twenty versions of a complex aerospace application developed using independent programming teams from four geographically-separate development sites and subjected to an independent certification.

The results suggest that the *N*-version systems are effective at coping with the failures of a few bad programs. However, under operating conditions where the programs were uniformly more reliable, the failures that did occur tended to be coincident and have a magnitude that substantially decreased the effectiveness of *N*-version systems. Indeed, the present study suggests that independent development alone is not sufficient to achieve high reliability gains over using a single version. More is required to reduce, if possible, the intensity of coincident failures. Coincident failures, as seen in this experiment and which have been observed in other experiments, do not necessarily result from similar design faults! Input domain related faults may prove to be much more difficult to prevent since there is not a logical relationship between the faults. Rather, it is a matter of determining those rarely occurring input conditions that trigger the coincident failures of the dissimilar faults.

While there are analytical arguments and strong empirical evidence that the failures of functionally equivalent component versions will be dependent, their nature, prevalence, and intensity is a matter of considerable controversy. For the twenty versions in this experiment, coincident failures occurred at rates that greatly exceed the rates expected by chance under the assumption of independence. The magnitude of the intensity of coincident version failures and the absence of independent failure behavior observed in this experiment is similar to that of the Knight and Leveson experiment [5].

Our results indicate that lack of understanding by the programmers of key points in the specification was a major contributor to faults that caused coincident failures. This resulted in a *diversity of understanding* by the programmers. It is not clear what form of diversity could have eliminated or mitigated this problem.

If redundancy can routinely provide substantial improvement in reliability, then the technique is worthy of further consideration. The present work was undertaken with the understanding that comparing the failure probability of the individual versions with N-version systems built from them would not be a complete test of separate development methods. We acknowledge that if *equal* resources were dedicated to the development of a single version one obtains a better measure of the effectiveness of using N-version software. In view of the modest gain in reliability that has been obtained with multiple versions in this experiment, future experiments might well address the cost issue.

ACKNOWLEDGEMENT

Many individuals and organizations have contributed to the success of this experiment: specifically, graduate student programmers at North Carolina State University, the University of Illinois, the University of Virginia, and the University of California at Los Angeles; John McHugh, Linda Lauterbach, Janet Dunham, John Pierce, and other staff at the Research Triangle Institute; Paul Lorczak, Greg Zacharias, Les Walker and other staff at Charles River Analytics; Roy Campbell at the University of Illinois, and anonymous referees of this paper.

REFERENCES

1. Avizienis, A. and Chen, L., "On the Implementation of N-Version Programming for Software Fault Tolerance During Program Execution," Proc. of COMPSAC 77, Chicago, IL (Nov. 1977).
2. Randell, B., "System Structure for Software Fault Tolerance," *IEEE Trans. of Software Eng.*, SE-1(2) (June 1975).
3. Eckhardt, D. E. and Lee, L. D., "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors," *IEEE Trans. Software Eng.*, **11**(12), pp. 1511-1517 (December 1985).
4. Littlewood, B. and Miller, D.R., "A Conceptual Model of Multi-Version Software," Proc. of the 17th Symposium on Fault-Tolerant Computing (June 1987).
5. Knight, J. C. and Leveson, N. G., "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming," *IEEE Trans. Software Eng.*, **12**(1), pp. 96-109 (January 1986).
6. Scott, R.K., Gault, J.W., McAllister, D.F., and Wiggs, J., "Investigating Version Dependence in Fault-Tolerant Software," *AGARD 361*, pp. 21.1-21.10 (1984).
7. Shimeall, T.J. and Leveson, N.G., "An Empirical Comparison of Software Fault Tolerance and Fault Elimination," Proceedings of the Second Workshop on Software Testing, Verification and Analysis, Banff, Canada (July 1988).
8. Anderson, T., Barret, P.A., Halliwell, D.N., and Moulding, M.R., "Software Fault Tolerance: An Evaluation," *IEEE Trans. Soft. Eng*, **SE-11**(12), pp. 1502-1510 (1985).
9. P.G., Bishop, D.G., Esp, M., Barnes, P., Humphreys, G., Dahl, and J., Lahti, "PODS--A Project on Diverse Software," *IEEE Trans. of Software Eng.*, **SE-12**(9) (September 1986).
10. Bishop, P.G. and Pullen, F.D., "PODS Revisited--A Study of Software Failure Behaviour," *Proc. of the 18th Symposium on Fault-Tolerant Computing*, pp. 2-8 (June 1988).
11. Avizienis, A., R., Lyu, Michael, and Werner, Schutz, "In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software," *Proc. of the 18th Symposium on Fault-Tolerant Computing* (1988).
12. Kelly, J.P.J. and Avizienis, A., "A Specification-Oriented Multi-Version Software Experiment," *Proc. of the 13th Symposium on Fault-Tolerant Computing* (June 1983).
13. Knight, J. C. and Leveson, N. G., "An Empirical Study of Failure Probabilities in Multi-Version Software," *Proc. of the 16th Symposium on Fault-Tolerant Computing*, pp. 165-170 (July 1986).
14. Tso, K.S. and Avizienis, A., "Community Error Recovery in N-Version Software: A Design Study with Experimentation," *Proc. of the 17th Symposium on Fault-Tolerant Computing* (July 1987).
15. Anderson, T. and Lee, P.A., in *Fault Tolerance Principles and Practice*, ed. Prentice Hall International (1981).

16. Hecht, H., "Fault-Tolerant Software for Real-Time Applications," *ACM Computing Surveys*, **8**(4), pp. 391-407 (December 1976).

17. Gmarov, A., Arlat, J., and Avizienis, A., "On the Performance of Software Fault Tolerance Strategies," *Proc. of the 10th Symposium on Fault-Tolerant Computing*, pp. 251-253 (October 1980).

18. J.C., Laprie, "Dependability Evaluation of Software Systems in Operation," *IEEE Trans. Software Eng.*, **10**(6), pp. 701-714 (November 1984).

19. Scott, R.K., Gault, J.W., and McAllister, D.F., "Fault Tolerant Software Reliability Modeling," *IEEE Trans. Software Eng.*, **13**(5), pp. 582-592 (May 1987).

20. Fischler, M.A., Firschein, O., and Drew, D.L., "Distinct Software: An Approach to Reliable Computing," Proceeding 1975 USA-Japan Computer Conf., pp. 573-579.

21. Avizienis, A., "Design Diversity - The Challenge of the Eighties," *Proc. of the 12th Symposium on Fault-Tolerant Computing*, pp. 44-45 (August 1984).

22. Systems, Litton Inc., "Preliminary Design of an RSDIMU Using Two-Degree-of-Freedom Tuned- Gimbal Gyroscopes," NASA CR-145035 (October 1976).

23. Lauterbach, L., "Development of N-Version Software Samples for an Experiment in Software Fault Tolerance," NASA Contractor Report 178363 (September 1987).

24. Kelly, J.P.J., Eckhardt, D.E., Vouk, M.A., McAllister, D.F., and Caglayan, A.K., "A Large Scale Second Generation Experiment in Multi-Version Software: Description and Early Results," *Proc. of the 18th Symposium on Fault-Tolerant Computing* (June 1988).

25. Eckhardt, D.E. and Lee, L.D., "Fundamental Differences in the Reliability of N-Modular Redundancy and N-Version Programming," *Journal of Systems and Software*, **8**, pp. 313-318 (1988).

26. Nagel, P. M. and Skrivan, J. A., "Software Reliability: Repetitive Run Experimentation and Modeling," NASA CR 165836, NASA Langley Research Center, Hampton, Virginia (February 1982).

27. Brilliant, S. S., Knight, J. C., and Leveson, N. G., "Analysis of Faults in an N-Version Software Experiment," *IEEE Trans. Software Eng.*, **16**(2) (February 1990).

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No. NASA TM-102613	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle An Experimental Evaluation of Software Redundancy As a Strategy for Improving Reliability		5. Report Date May 1990
		6. Performing Organization Code
7. Author(s) Dave E. Eckhardt, Jr, Alper K. Caglayan, John C. Knight, Larry D. Lee, David F. McAllister, Mladen A. Vouk, John J. Kelly		8. Performing Organization Report No.
		10. Work Unit No. 505-65-11
9. Performing Organization Name and Address NASA Langley Research Center Hampton, Virginia 23665		11. Contract or Grant No.
		13. Type of Report and Period Covered Technical Memorandum
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546		14. Sponsoring Agency Code

15. Supplementary Notes

D.E. Eckhardt, Jr: Langley Research Center, Hampton, Virginia. A.K. Caglayan: Charles River Analytics, Cambridge, Massachusetts. J.C. Knight: University of Virginia, Charlottesville, Virginia. L.D. Lee: Old Dominion University, Norfolk, Virginia. D.F. McAllister & M.A. Vouk: North Carolina State University, Raleigh, North Carolina. J.P.J. Kelly: University of California, Santa Barbara, California.

16. Abstract

The strategy of using multiple versions of independently developed software as a means to tolerate residual software design faults is suggested by the success of hardware redundancy for tolerating hardware failures. Although, as generally accepted, the independence of hardware failures resulting from physical wearout can lead to substantial increases in reliability for redundant hardware structures, a similar conclusion is not immediate for software. The degree to which design faults are manifested as independent failures determines the effectiveness of redundancy as a method for improving software reliability. Interest in multi-version software centers on whether it provides an adequate measure of increased reliability to warrant its use in critical applications. The effectiveness of multi-version software is studied by comparing estimates of the failure probabilities of these systems with the failure probabilities of single versions. The estimates are obtained under a model of dependent failures and compared with estimates obtained when failures are assumed to be independent. The experimental results are based on twenty versions of an aero-space application developed and certified by sixty programmers from four universities. Descriptions of the application, development and certification processes, and operational evaluation are given together with an analysis of the twenty versions.

17. Key Words (Suggested by Author(s)) *N*-version programming; multi-version programming; fault-tolerant software; software reliability	18. Distribution Statement Unclassified - Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	23	A03

NASA FORM 1626 OCT 86