Analysis of Faults in an *N*-Version Software Experiment

SUSAN S. BRILLIANT, MEMBER, IEEE, JOHN C. KNIGHT, AND NANCY G. LEVESON

Abstract—**We have conducted a large-scale experiment in *N*-version programming. A total of 27 versions of a program were prepared independently from the same specification at two universities. The results of executing the versions revealed that the versions were individually extremely reliable but that the number of input cases in which more than one failed was substantially more than would be expected if they were statistically independent.**

After the versions had been executed, the failures of each version were examined and the associated faults located. In this paper we present an analysis of these faults. Our goal in undertaking this analysis was to understand better the nature of the faults. We found that in some cases the programmers made equivalent logical errors, indicating that some parts of the problem were simply more difficult than others. We also found cases in which apparently different logical errors yielded faults that caused statistically correlated failures, indicating that there are special cases in the input space that present difficulty in various parts of the solution. A formal model is presented to explain this phenomenon. It appears that minor differences in the software development environment, such as the use of different programming languages for the different versions, would not have a major impact in reducing the incidence of faults that cause correlated failures.

Index Terms—**Design diversity, fault-tolerant software, multiversion programming, *N*-version programming, software reliability.**

I. INTRODUCTION

DESPITE extensive attempts to build software that is sufficiently reliable for critical applications, faults tend to remain in production software. Although *fault avoidance* and *fault removal* [1] do improve software reliability, new applications for computers in safety-critical systems, such as commercial aircraft and medical devices, have very high reliability requirements. For example, for certain applications in commercial aircraft, no more than a 10^{-9} chance of failure over a ten hour period is permitted. This appears to be beyond the ability of standard software engineering techniques to ensure or even to measure.

Manuscript received September 1, 1986; revised August 29, 1989. This work was supported in part by NASA under Grants NAG-1-242, NAG-1-605, and NAG-1-606, in part by the National Science Foundation under Grant DCR 8406532, and in part by MICRO grants cofunded by the state of California, Hughes Aircraft Company, and TRW.

S. S. Brilliant was with the Department of Computer Science, University of Virginia, Charlottesville, VA 22903. She is now with the Department of Mathematical Sciences, Virginia Commonwealth University, Richmond, VA 23284.

J. C. Knight is with the Department of Computer Science, University of Virginia, Charlottesville, VA 22903.

N. G. Leveson is with the Department of Computer Science, University of California, Irvine, CA 92717.

IEEE Log Number 8932236.

Proposals have been made for building *fault-tolerant* software [1] in an attempt to deal with the faults that remain in operational software. One approach, *N*-version programming [4], requires separate, independent preparation of multiple versions of a piece of software for an application. The versions are executed in parallel, and majority voting selects the results to be used. The amount of reliability improvement achieved is determined by the degree of independence of the failures of the versions [6]. If two versions fail on the same input in a 3-version system, for example, they will either outvote a third correct version or no majority will exist.

Previously, we conducted a large-scale experiment [10] in *N*-version programming. Twenty-seven versions of a program were prepared independently at two universities and then executed one million times. The results of the executions revealed that the individual programs were extremely reliable. However, the number of input cases in which more than one program failed, that is, where failures were coincident, was substantially more than would be expected if the various programs failed in a statistically independent way. The faults responsible for each of the observed failures in the programs written for the experiment have been identified. In this paper, we present an analysis of these faults paying particular attention to those that caused more coincident failures than would occur by chance.

Examination of the faults is important for several reasons. First, it sheds light on the potential value of the technique itself. By analyzing faults such as those described here, methods might be developed to allow the performance of *N*-version systems to be improved. Second, a better understanding of the faults will allow evaluation of techniques that have been suggested for minimizing coincident failures in *N*-version software, such as the use of dissimilar programming languages or development environments [7], [2], [9], [14]. In addition, new development techniques for *N*-version software may be suggested. Finally, a study of the faults made by different programmers on the same problem may provide important information on how to improve the reliability of *single*-version software.

In the next section we summarize the experiment that yielded the twenty-seven programs studied here. A statistical analysis, presented in Section III, is used to determine which faults are responsible for failures that are statistically correlated, without regard to the details of the

faults. The faults themselves are summarized in Section IV, and the interrelationships between the faults are discussed in Section V. It was found that faults that produce statistically correlated failures are *not* necessarily semantically similar, and vice versa. Thus, faults that at first sight seem unrelated sometimes cause coincident failures, and faults that seem very similar sometimes do not cause coincident failures. A formal model to explain this phenomenon is presented. Our conclusions are presented in Section VI.

II. Experiment Summary

Only the major features of the previous experiment are described in this paper since the details have been published elsewhere [10]. The application used in the experiment was a simple (but realistic) antimissile system that came originally from an aerospace company [15], [5]. The program reads data representing radar reflections. Using a collection of conditions, it decides whether the reflections come from an object that is a threat and, if so, a signal to launch an interceptor is generated.

Twenty-seven students in graduate and senior level classes in computer science at the University of Virginia (UVA) and the University of California, Irvine (UCI) wrote programs from a single requirements specification. The programs were all written in Pascal, and developed on a Prime 750 system using the Primos operating system and Hull V Pascal compiler at UVA and on a DEC VAX 11/750 running 4.1 BSD Unix at UCI.

An attempt was made to obtain programmers with varied experience but this was necessarily limited by the need to use students as subjects. Fifteen of the programmers were working on bachelor's degrees and had no prior degree, eight were working on master's degrees, and four were working on doctoral degrees. The graduate students included four with degrees in mathematics, three with degrees in computer science, and one each with degrees in astronomy, biology, environmental science, management science, and physics. The programmers' previous work experience in the computer field ranged from none to more than ten years. There appeared to be no correlation between the programmers' experience levels and the quality of their programs.

Once a program was completed and tested by the programmer, it was subjected to an acceptance procedure that consisted of two hundred randomly-generated input cases. A different set of two hundred inputs was generated for each program in order to avoid a general "filtering" of common faults by the use of a common acceptance procedure. The acceptance procedure was not part of the process of testing the programs. It was a quality filter used to ensure that only programs capable of a minimum level of performance were used in the analysis.

Accepted programs were subjected to one million randomly-generated input cases in order to observe operational behavior. The determination of the success of the twenty-seven individual versions was made by comparing their output with a separate version, referred to as the *gold* program, that had been subjected to extensive previous analysis.

As required by the specification, each program produces a 15 by 15 boolean array, a 15 element boolean vector, and a single boolean launch decision (a total of 241 outputs) on each input case. A *failure* was recorded for a particular version on a particular input case if there was *any* discrepancy between the 241 results produced by that version and those produced by the gold program, or the version causes some form of fatal exception to be raised during execution of that input case.

We define a *fault* formally in Section V and use it to explain the results of the work described in this paper. We define a fault here, informally, to be a defect in the algorithm implemented by a program version that is responsible for at least one failure in the sense that changing the program so as to correct the defect would allow the program to obtain output agreeing with that of the gold program for that input case. For each of the twenty-seven versions, the faults were identified by examining the output of the program for input cases in which failure occurred and analyzing the source text.

Once a fault was located, a correction was devised. The version containing the fault was modified so that either the original faulty code or the corrected code could be executed. The purpose of modifying each version in this manner was to allow the identification of the fault or set of faults responsible for each failure recorded for the version. Each input case that caused the version to fail originally was regenerated. The version was then executed with each individual fault corrected in turn.

For most failures, a version worked correctly when one and only one of its faults was corrected. For these cases, the fault corrected on the execution that gave correct results was assigned sole responsibility for the failure. In a few instances, correcting *either* of two faults gave correct results, so it was recorded that the failure was attributable to either of the two faults. In some cases none of the executions with a single fault corrected yielded correct results. For these failures the version was executed with each pair of faults corrected in turn, then with each set of three faults corrected in turn, and so on, until correct results were obtained. The faults corrected on the execution giving correct results were assigned collective responsibility for the failure.

III. Statistical Analysis of the Failures

For the purposes of discussion in the remainder of this paper, the individual faults are identified by the version number in which the fault occurs concatenated with a sequence number for the faults associated with that version. Thus, for example, fault 3.1 is the first fault associated with version 3. The faults found in each of the twenty-seven program versions and the number of failures attributable to each fault are shown in Table I. Failures associated with more than one fault are counted in the number of failures for each of the associated faults.

TABLE I
FAULT OCCURRENCE COUNTS

Fault	Number of Occurrences	Fault	Number of Occurrences
1.1	2	18.1	8
3.1	700	19.1	264
3.2	1061	20.1	323
3.3	537	20.2	697
3.4	1437	21.1	85
6.1	607	21.2	7
6.2	511	22.1	6551
6.3	32	22.2	1735
7.1	71	22.3	1735
8.1	225	23.1	72
8.2	98	23.2	8
9.1	47	24.1	260
9.2	6	25.1	14
11.1	554	25.2	80
12.1	356	25.3	3
12.2	71	26.1	140
13.1	4	26.2	9
14.1	1297	26.3	1
14.2	71	26.4	6
16.1	28	26.5	4
16.2	34	26.6	368
17.1	201	26.7	243
17.2	76		

The manifestations of a few of the faults were implementation-dependent. When the fault-to-failure identification analysis was performed, a version sometimes executed correctly for an input case on which it had failed originally. This effect was caused by differences in the hardware and compilers used, and it was observed for versions 6, 22, 23, and 26. Analysis of the input cases involved allowed the original failures of versions 6 and 23 to be attributed to faults 6.1 and 23.1 respectively, so these failures were included in the failure counts for the associated faults in Table I. For versions 22 and 26, the original failures could not be associated with specific faults. These original failures are not included in any of the failure counts shown in Table I.

In order to determine which faults caused statistically correlated failures, a statistical test of independence was performed between each pair of faults where the two elements in a pair come from different versions. A matrix C of the coincident failures caused by each pair was constructed. Coincident failure here means that both versions failed and includes failures with both identical and nonidentical outputs. This matrix is indexed in both dimensions by the sequence of fault numbers. Thus, C_{ij} represents the number of test cases in which the two program versions containing faults i and j both failed because of faults i and j. Clearly C is symmetric, and its diagonal represents the failure rates for the individual faults.

For each pair of faults, an approximate χ^2 test [8] was used to test the null hypothesis that the corresponding two faults cause failure independently. The observed value of the χ^2 statistic for each pair i, j of faults causing common failures was calculated, using the following expression for the test statistic:

$$\frac{n(nC_{ij} - C_{ii}C_{jj})^2}{C_{ii}C_{jj}(n - C_{ii})(n - C_{jj})}$$

where

n = total number of input cases = 1,000,000.

Where the observed χ^2 statistic is greater than 7.88, the null hypothesis of independence can be rejected with 99.5 percent certainty. The results of the 945 separate hypothesis tests are shown in Table II. An "R" in Table II indicates that the null hypothesis was rejected for the corresponding pair of faults at the 99.5 percent level. In that case, we define the two faults to be *failure correlated.* The statistical test used here is valid only if the value of C_{ij} is "sufficiently large," and values greater than or equal to five are generally considered to give satisfactory results. A "?" entry in Table II denotes a case in which the value of the χ^2 statistic was large enough to justify rejection of the null hypothesis, but for which the value of C_{ij} is too small to justify reliance on the hypothesis test. Dashes in the table denote entries for which the statistic has no relevance because the faults are in the same program.

The results of these hypothesis tests indicate that 93 of the hypotheses should be rejected; that is, 93 fault pairs found in the experiment are responsible for statistically correlated failures. An additional 67 pairs appear correlated but there is insufficient data to have confidence in this conclusion. The use of a confidence level of 99.5 percent means that the probability that the null hypothesis will be rejected when in fact it is true is 0.5 percent. Thus, if the null hypothesis is in fact true for each of the 945 hypothesis tests that were performed, the expected number of erroneous rejections is approximately five whereas 93 were rejected.

It is clear from the preliminary data that more coincident failures occurred than would be expected by chance [10]. The results of these statistical tests show which faults were responsible for the coincident failures. In the rest of this paper, we examine these faults more carefully.

IV. DESCRIPTIONS OF THE FAULTS

The faults in the programs were examined to determine whether those that are failure-correlated have any unique characteristics. Table III contains the details of the individual faults including the part of the problem involved (LIC number), the type of input that triggers the fault, and a short description of the fault.

Several of the faults involve mistakes in the use of limited-precision arithmetic and require some further explanation. The source text of a function called REALCOMPARE, containing only four executable statements, was supplied to the programmers. They were instructed to use REALCOMPARE for all comparisons of real numbers. This function defines the relational operators for floating-point numbers in the manner described by Knuth [13]. As Knuth points out, operators should be defined for floating-point comparison that allow many of the normal axioms of arithmetic to be assumed.

The REALCOMPARE function performs limited-precision floating-point comparison. It does so by comparing its two floating-point arguments, returning EQ when the difference between the two values is less than 0.000005 of the larger value. Otherwise the function returns LT

TABLE II
RESULTS OF HYPOTHESIS TESTS

Faults	1.1	3.1	3.2	3.3	3.4	6.1	6.2	6.3	7.1	8.1	8.2	9.1	9.2	11.1	12.1	12.2	13.1	14.1	14.2	16.1	16.2	17.1	17.2	18.1	19.1	20.1	20.2	21.1	21.2	22.1	22.2	22.3	23.1	23.2	24.1	25.1	25.2	25.3	26.1	26.2	26.3	26.4	26.5	26.6	26.7
1.1	-	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	?	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
3.1	.	-	-	-	-	.	.	.	.	R	R	.	.	R	.	.	.	.	.	.	.	R	R	.	.	R	R	.	.	R	.	.	.	R	.	.	R	R	.	.	.	.	.	.	.	R	.
3.2	.	-	-	-	-	.	.	.	?	R	R	.	.	R	.	?	.	R	?	.	.	?	R	?	.	R	R	.	.	.	.	.	R	?	.	R	R	.	.	.	.	.	.	.	?	R
3.3	.	-	-	-	-	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
3.4	.	-	-	-	-	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	R	R	?	.	.	.	.	.	.	.	.	?	.	.	.	.	.	.	.	.	.	.	.	.	?	.
6.1	.	.	.	.	.	-	-	-	.	.	.	.	?	.	.	.	.	?	.	.	.	.	.	.	.	?	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
6.2	.	.	.	.	.	-	-	-	.	.	.	R	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
6.3	.	.	.	.	.	-	-	-	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	?	.	.	.	.	.	.	.
7.1	.	.	?	.	.	.	.	.	-	?	.	.	.	?	.	R	.	.	R	.	.	?	.	.	?	?	.	.	.	.	.	.	.	.	?	.	.	.	.	.	.	.	.	.	.	.	.	.
8.1	.	R	R	.	.	.	.	.	?	-	-	.	R	.	?	.	.	?	.	.	R	?	.	?	R	R	.	R	.	.	R	.	.	R	.	.	.	R	.	.	.	.	.	.	.	R	R
8.2	.	R	R	.	.	.	.	.	.	-	-	.	R	.	.	.	R	.	.	.	R	?	.	.	R	R	.	R	.	.	.	.	.	.	R	R	.	.	.	.	.	.	.	.	.	R	R
9.1	.	.	.	.	.	.	R	.	.	.	.	-	-	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
9.2	.	.	.	.	.	.	.	.	.	.	.	-	-	.	.	.	.	.	.	.	.	.	.	.	.	.	.	?	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
11.1	.	R	R	.	.	?	.	.	?	R	R	.	-	-	.	?	.	R	?	R	.	R	?	.	.	R	R	.	.	R	.	.	R	.	.	R	.	.	?	R	.	.	.	.	.	R	R
12.1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	-	-	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
12.2	.	.	?	.	.	.	.	.	R	?	.	.	?	-	-	-	.	R	.	.	.	?	.	.	?	?	.	.	.	.	.	.	.	.	?	.	.	.	.	.	.	.	.	.	.	.	.	.
13.1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	-	.	.	.	?	.	.	.	.	.	.	?	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
14.1	.	.	R	.	.	.	.	.	.	R	.	R	.	.	-	-	.	.	.	.	.	.	.	.	R	R	.	.	.	.	.	.	.	R	.	.	?	?	.	.	.	.	.	.	.	.	R
14.2	.	.	?	.	.	.	.	.	R	?	.	.	?	.	R	.	-	-	.	.	.	?	.	.	?	?	.	.	.	.	.	.	.	?	.	.	.	.	.	.	.	.	.	.	.	.	.
16.1	.	.	.	.	.	?	.	.	.	.	.	.	R	.	.	.	.	.	-	-	.	.	.	.	.	R	.	R	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	?	?
16.2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	?	.	.	-	-	.	.	.	.	.	.	.	R	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
17.1	.	R	?	.	R	.	.	.	.	R	R	.	.	R	.	.	.	.	.	.	.	-	-	?	?	R	R	.	.	R	.	.	?	?	.	?	R	.	.	.	.	.	.	.	.	R	.
17.2	?	R	R	.	R	.	.	.	?	?	?	.	?	.	?	.	.	?	.	.	.	-	-	?	.	R	?	.	.	.	.	.	?	?	.	.	?	.	.	.	.	.	.	.	.	?	?
18.1	.	.	?	.	?	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	?	?	-	.	.	.	.	.	.	.	.	.	R	.	.	.	.	.	.	.	.	.	.	.	.	.	.
19.1	.	.	.	.	.	.	.	.	.	?	.	.	.	.	.	.	.	.	.	.	.	?	.	.	-	?	.	?	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	?
20.1	.	R	R	.	.	.	.	.	?	R	R	.	R	.	?	.	R	?	.	.	R	?	.	.	R	-	-	.	.	R	.	.	R	.	.	R	R	.	.	.	.	.	.	.	.	R	R
20.2	.	R	R	.	.	?	.	.	?	R	R	.	R	.	?	.	R	?	R	.	R	?	.	.	-	-	.	.	R	R	R	R	.	.	?	R	.	.	.	.	.	.	.	.	.	R	R
21.1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	?	.	.	-	-	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
21.2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	?	.	.	.	.	R	.	.	.	.	.	.	-	-	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
22.1	.	R	.	.	.	.	.	.	.	R	R	.	?	R	.	.	.	.	R	.	R	.	.	.	R	R	.	.	-	-	-	?	.	R	?	R	.	.	.	.	.	.	.	.	.	R	R
22.2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	R	.	.	-	-	-	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
22.3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	R	.	.	-	-	-	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
23.1	.	R	R	.	.	.	.	.	.	R	.	.	R	.	.	.	.	.	.	.	.	?	?	.	.	R	R	.	.	?	.	.	-	-	.	R	.	.	.	.	.	.	.	.	.	.	?	R
23.2	.	.	?	.	?	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	?	?	R	.	.	.	.	.	.	.	.	-	-	.	.	.	.	.	.	.	.	.	.	.	.	.	.
24.1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	R	.	.	.	.	.	-	.	.	.	.	.	.	.	.	.	.	?	?
25.1	.	R	R	.	.	.	.	.	.	.	R	.	.	?	.	.	.	R	.	.	.	?	.	.	R	?	.	.	?	.	.	.	.	.	.	-	-	.	.	.	.	.	.	.	.	?	?
25.2	.	R	R	.	.	.	.	.	?	R	R	.	R	.	?	.	.	?	.	.	.	R	?	.	.	R	R	.	.	R	.	.	R	.	.	.	-	-	-	.	.	.	.	.	.	.	R	R
25.3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	-	-	-	.	.	.	.	.	.	.	.	.
26.1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	?	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	-	-	-	-	-	-	-	-	-
26.2	.	.	.	.	.	.	.	?	.	.	.	.	.	.	?	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	-	-	-	-	-	-	-	-	-
26.3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	-	-	-	-	-	-	-	-	-
26.4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	-	-	-	-	-	-	-	-
26.5	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	-	-	-	-	-	-	-
26.6	.	R	?	.	?	.	.	.	.	R	R	.	.	R	.	.	.	.	?	.	R	?	.	.	R	R	.	.	R	.	.	?	.	?	?	R	.	-	-	-	-	-	-	-	-	-	-	-
26.7	.	.	R	.	.	.	.	.	.	R	R	.	.	R	.	.	.	R	.	?	.	.	?	.	?	R	R	.	.	R	.	.	R	.	?	?	R	.	-	-	-	-	-	-	-	-	-	-	-

(GT) if the first argument is less than (greater than) the second. Essentially what REALCOMPARE does is to establish a region around the larger operand in which the operands are considered equal. The size of the region is determined by the size of the larger operand.

Four of the programmers erroneously used limited-precision comparisons with zero to determine *sign*, i.e., they attempted to use REALCOMPARE to determine if a number was negative. It is clear that small negative numbers (i.e., close to zero) will be interpreted mistakenly as nonnegative using this approach to determining sign. This type of fault could have been a result of inexperience on the part of the programmers although one of the participants making this mistake had many years of work experience in real-time, scientific programming. None of these faults were failure-correlated with each other and so do not contribute to our analysis of coincident failures.

Several faults arose from the comparison of the cosines or sines of angles rather than the angles themselves. Although mathematically the comparisons are equivalent, difficulties arise due to the relatively flat shape of the cosine and sine curves near zero and one, respectively. On the flat parts of these curves, angles that are quite different have cosines or sines that are nearly equal, so comparisons using a tolerance find that the angles are different, but their cosines or sines are equal within the tolerance. The specification for the application stated that angles were to be compared, not functions of angles.

An opportunity for multiple correct solutions also arose from our attempt to encourage diversity. Launch conditions 3 and 10 require the determination of whether the angle formed by three points satisfies either of the conditions:

$$\text{angle} < (\pi - \epsilon)$$

or

$$\text{angle} > (\pi + \epsilon)$$

TABLE III
FAULT DETAILS

FAULT	LIC #	INPUT CONDITION	FAULT DESCRIPTION
1.1	LIC 3, 10	Angles near $\pi + \varepsilon$, ε large.	Calculates angles between π and 2π rather than between 0 and π. (Fails because tolerance for comparison with $\pi + \varepsilon$ much greater than for $\pi - \varepsilon$.)
3.1	LIC 3	Three collinear points (subtended angle zero).	Path that handles collinear points always gives π as subtended angle-- misses case in which angle is zero.
3.2	LIC 10	Three collinear points (subtended angle zero).	Similar to fault 3.1.
3.3	LIC 7	Three collinear points (subtended angle zero).	In calculating distance from point to line formed by two other points, calculates distance to nearest point (rather than zero) when points are collinear and first point not between other two.
3.4	LIC 3, 10	Three almost collinear points.	Inaccurate algorithm to determine collinearity; points treated as collinear when just nearly collinear.
6.1	LIC 9, 14	A set of three points in which each pair has a common coordinate.	In determining coincidence, "and" and "or" confused in predicate (results in points described in input condition being treated as if two of the points coincide).
6.2	LIC 2	Three points form acute triangle.	Misses special case of acute triangle when computing radius of smallest circle containing three points.
6.3	LIC 7	First and last of 'N_PTS' consecutive data points coincide.	Array indexing error when calculating distance to coincident points.
7.1	LIC 3, 10	Three almost collinear points (subtended angle near zero).	Cosines rather than angles compared when cosine near 1 or -1.
8.1	LIC 3, 10	Three collinear points (subtended angle zero).	Similar to fault 3.1.
8.2	LIC 3, 10	Three collinear points (subtended angle zero).	Similar to fault 8.1 but on special path to handle horizontal and vertical lines.
9.1	LIC 2, 9, 14	Three points form acute triangle for which length of longest side exceeds perpendicular distance to third vertex.	Incorrect path condition to determine whether three points form an obtuse triangle.
9.2	LIC 4, 11, 15	Three collinear or almost collinear points.	Due to machine roundoff error, negative result obtained when length of one side of triangle subtracted from half its perimeter.
11.1	LIC 3, 10	Three collinear or almost collinear points (subtended angle near zero).	Machine round-off error causes calculated cosine of angle formed by three points to be greater than 1.
12.1	LIC 5	Point on right side of x-axis.	Predicate using = instead of >= results in incorrect assignment to quadrant.
12.2	LIC 3, 10	Three almost collinear points (angle near zero).	Similar to fault 7.1.
13.1	LIC 2, 9, 14	Complicated relationship among three points arising from algorithm used to compute radius of circle containing the points.	Comparison with zero using REALCOMPARE used to ensure that a quantity to be used as an argument to sqrt is nonnegative.
14.1	LIC 10	Two of three points coincide.	Apparent typographical error in array index handling special case of second point coincident with either first or third.
14.2	LIC 3, 10	Three almost collinear points (subtended angle near zero).	Similar to fault 7.1.
16.1	LIC 3, 10	Three collinear points (subtended angle is π).	Angle measurement of 180 rather than π used when point 2 between points 1 and 3.
16.2	LIC 2, 9, 14	See fault 13.1.	Similar in origin and effect to fault 13.1.
17.1	LIC 3	Three almost collinear points (subtended angle near zero).	Cosines rather than angles compared.
17.2	LIC 10	Three almost collinear points (subtended angle near zero).	Similar to fault 17.1.
18.1	LIC 3, 10	Angles near $\pi + \varepsilon$, ε large.	Similar to fault 1.1--occurs on a different set of angles.
19.1	LIC 3, 10	Vertex coincides with an endpoint.	Decides condition not satisfied by any set of three points when one set involves a coincident vertex and endpoint.
20.1	LIC 3, 10	Three collinear points (subtended angle zero).	If tangent is zero, assumes angle is π; misses case in which angle is zero.
20.2	LIC 3, 10	Three collinear or almost collinear points (subtended angle near zero).	In applying formula tan=sqrt(1-sqr(cos))/cos, round-off error causes negative argument to sqrt.
21.1	LIC 3, 10	Coincident points.	Special case for coincident points in algorithm for computing smallest circle containing three points fails when first and second points coincide.
21.2	LIC 2, 9, 14	See fault 13.1.	Similar in origin and effect to fault 13.1.
22.1	LIC 4	Three collinear or almost collinear points.	Similar to fault 9.2.
22.2	LIC 11	Three collinear or almost collinear points.	Similar to fault 9.2.
22.3	LIC 15	Three collinear or almost collinear points.	Similar to fault 9.2.
23.1	LIC 3, 10	Three collinear or almost collinear points (subtended angle near zero).	Uses REALCOMPARE to determine if calculated angle negative; misses some angles near zero.
23.2	LIC 3, 10	Angle near $\pi + \varepsilon$, ε large.	Similar to fault 18.1.
24.1	LIC 2, 9, 14	Coincident points.	Error in arguments to function that determines coincidence (coincidence of points 1 and 2 checked rather than that of 1 and 3).

TABLE III (Continued.)

FAULT	LIC #	INPUT CONDITION	FAULT DESCRIPTION
25.1	LIC 3, 10	Three collinear points (subtended angle zero).	Missing case in computing angle formed by three points when point 1 lies between points 2 and 3 and is closer to point 2 than to point 3.
25.2	LIC 3, 10	Three collinear points (subtended angle zero).	Similar to 25.1, different path.
25.3	LIC 2, 9, 14	Three collinear points (subtended angle zero).	Incorrectly handles case in which three points are collinear and point 1 falls between points 2 and 3.
26.1	LIC 7	Vertical line formed by first and last of 'N_PTS' consecutive data points.	Incorrect algorithm to compute distance on path that handles special case of distance to vertical line.
26.2	LIC 7	LIC 7 satisfied by some point not checked by incorrect algorithm.	Incorrect indexing through points.
26.3	LIC 10	Three points subtend angle near $\pi/2$.	Compares sines rather than angles.
26.4	LIC 3	Three points subtend acute angle and ε near $\pi/2$.	Misses case for which subtended angle acute and ε near $\pi/2$. LIC is met, but assumed not met by default.
26.5	LIC 10	See fault 26.4.	Similar to fault 26.4.
26.6	LIC 3	See fault 9.2.	Similar to fault 9.2. (Occurs in different context.)
26.7	LIC 10	Collinear or almost collinear points.	Similar to fault 26.6.

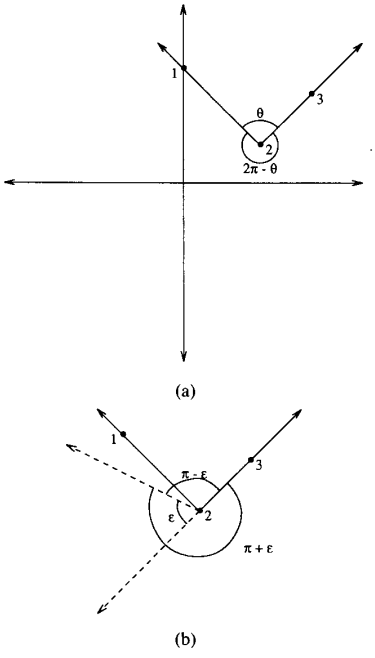


Fig. 1. The angle formed by three points.

where ϵ is a parameter supplied as input. The specification indicates that the second of the three points is the vertex. However, as is illustrated in Fig. 1(a), there is still a choice of angle to be measured. Either the angle marked θ or the angle marked $2\pi - \theta$ could be considered. In absolute terms it makes no difference which angle is measured. Fig. 1(b) illustrates that the smaller of the two possible angles is less than $(\pi - \epsilon)$ if and only if the larger angle is greater than $(\pi + \epsilon)$. However, recall that the tolerance used by REALCOMPARE depends on the *size* of its arguments. Thus, occasionally the function returns EQ for the larger pair when it returns LT for the smaller pair. There is a dilemma here since revising the specification to identify which of the two possible angles is to be measured would reduce the choices available to the programmer, thus reducing the potential diversity among the versions.

V. Discussion

Our goal in analyzing the individual faults in the versions was to attempt to understand the correlated failures that were observed in the experiment. We wanted to determine what other relationships, if any, exist among faults that are failure-correlated.

We define faults to be *logically related* if, in our opinion, they are either the same logical flaw, or they are similar logical flaws and are located in regions of the programs that compute the same part of the application. These assessments are based on our understanding of the application and assumptions about the intentions of the various programmers, and are therefore necessarily subjective.

Initially, we hypothesized that faults that are failure-correlated would be logically related, and vice versa. It seemed intuitively reasonable that there would be certain parts of the problem that would prove to be just more difficult to handle or more "error prone" than others.

This hypothesis does explain some of the observed failure correlations. For example, faults 3.1 and 3.2 involve the calculation of the angle formed by three points as required by launch conditions 3 and 10. In the case in which the three points are collinear, the programmer apparently failed to realize that the angle formed could be zero as well as π. It is easy to explain the failure correlations between these faults and faults 8.1, 8.2, 25.1, and 25.2. The authors of versions 8 and 25 both realized that collinear points could form a zero angle, but failed to consider all of the cases in which such an angle is formed. It is also easy to understand the correlations between all of these faults and fault 20.1. Version 20 takes a slightly different approach, calculating the tangent of the angle formed and mishandling the case in which the tangent is zero. Since a zero tangent indicates that the points are collinear, the

same special case is responsible for the difficulty. Version 20, like version 3, calculates an angle of π for all sets of collinear points, completely overlooking the cases in which the angle formed is zero. Although no two of this set of seven correlated faults are identical, the errors in logic seem to us to be similar.

However, there are faults that we classify as logically related which are not failure-correlated. For example, faults 7.1 and 17.1 both result from comparing cosines of angles rather than the angles themselves in the same part of the application, yet they caused no observed coincident failures. Fault 7.1 causes failure on input cases in which launch condition 3 or 10 is *not* satisfied, but for which there is some angle near zero that almost satisfies the condition. Fault 17.1, on the other hand, causes failure when launch condition 3 or 10 *is* satisfied, and the angle subtended is near zero.

Of more concern, however, is the fact that the hypothesis also fails to explain some of the observed failure correlations. For example, faults 11.1, 20.2, 22.1, 26.6, and 26.7 are all failure-correlated with the faults in versions 3, 8, 20, and 25 that involve the incorrect handling of cases in which collinear points subtend an angle of zero. However, faults 11.1, 20.2, 22.1, 26.6, and 26.7 are of a completely different nature. All of these cause fatal execution errors on calls to the square root function with negative arguments, and result from the failure of the programmers to consider that rounding errors may give an inaccurate computed result. Faults 11.1 and 20.2 both occur when a correctly computed cosine has an absolute value greater than one due to rounding error. The effects of rounding error are quite small so the exact (but unknown) cosine must have been close to one in these cases, and hence the corresponding angle had to be close to zero or π. Thus the failure correlation with other faults that mishandle zero angles is understandable. Similarly, faults 22.1, 26.6, and 26.7 are triggered when the calculated sum of two sides of a triangle is *less than* the length of the third side due to roundoff errors. Once again, this can only occur when the three points forming the triangle are approximately collinear, and the failure correlation is explained.

More difficult to understand is the failure correlation between fault 14.1 and each of faults 3.2, 8.2, 11.1, 20.1, 20.2, and 25.1. Fault 14.1 is the use of an incorrect subscript in a call to a function which determines whether the first or third in a set of three points coincides with the second. The coordinates of the three points are $(x[i], y[i])$, $(x[j], y[j])$, and $(x[k], y[k])$, but an apparent typographical error results in substituting $(x[j], y[i])$ for the second point in considering the special coincident point case. Although this fault apparently does not involve the angle formed by collinear points, an investigation revealed the reason for the observed correlations. Input cases that include a set of three points that form a vertical line and satisfy launch condition 10 trigger faults 3.2, 8.2, 11.1, 20.1, 20.2, and 25.1 due to the collinearity of the points. Fault 14.1 is also triggered because the faulty function call translates the second point such that it coincides with the first. Version 14 finds that no angle is formed, so it concludes that condition 10 is not satisfied by the points.

Based on the examples discussed above, it is clear that there are faults that produce correlated failures but that are not logically related. Thus our initial hypothesis does not explain all the observed failure correlations. They can be explained, however, if we note that faults causing correlated failures involve a mishandling of all inputs having some specific characteristic, i.e., both faults involve handling the same set of input cases incorrectly. This may appear tautological, but serves to emphasize that the similarities are *not* in the errors in the code, but instead are in the inputs. The following model is useful in determining why and when correlated failures occur. A program computes a function P that maps elements in a domain I to a range O. That is:

$$P: I \rightarrow O$$

where

$$I = \{i_1, i_2, \cdots, i_n\}$$

and

$$O = \{o_1, o_2, \cdots, o_n\}.$$

This function P consists of a set of partial functions where each partial function correspond to one of the paths in the program:

$$P = \{P_1, P_2, \cdots, P_m\}$$

where for each P_j, $1 \leq j \leq m$:

$$P_j: I[P_j] \rightarrow O[P_j]$$

$I[P_j]$ is the domain and $O[P_j]$ is the range of the partial function P_j. Clearly:

$$I = I[P_1] \cup I[P_2] \cup \cdots I[P_m]$$

and

$$O = O[P_1] \cup O[P_2] \cup \cdots O[P_m]$$

where for $i \neq j$:

$$I[P_i] \cap I[P_j] = \phi.$$

There is a fault in a program when the function implemented P' is not the function P that is desired, i.e., a mistake exists in the program that implements the function. A fault means that one or more partial functions P_i that make up P' are incorrect. A partial function P_i is incorrect, i.e., faulty, when the computation or the path condition that corresponds to P_i is erroneous, i.e., faulty.

When looking at the relationship between the failures of two programs, we are concerned with two cases:

1) The two programs both have faults but they lie on paths with disjoint input domains. Therefore they will not fail coincidentally since their input domains are disjoint.

2) The two programs each have a fault or faults on a path or paths whose input domain(s) overlap either partially or completely.

In the latter case, three possibilities exist:

1) The partial functions never produce the same wrong output.

2) The partial functions sometimes produce the same wrong output.

3) The partial functions always produce the same wrong output. Note that in this case the faults need not be the same. For example, one may set an output variable to 1 and the other divide the output variable by itself. The faults need not even be logically related, just compute the same erroneous partial function.

Our original hypothesis (and one that appears to be common in the literature) was that faults causing coincident failures would be logically related. The above model shows that this need not necessarily be the case, and it explains the correlations that we found between faults that were not logically related. Therefore, we propose a second hypothesis that faults that result in correlated failures are *input-domain related*, i.e., two faults are triggered by circumstances associated with a particular input whether or not the underlying flaws in the partial functions associated with the faults are related logically. Our second hypothesis can be seen as an extension of the first, since logically-related faults may also be input-domain related. It does, however, explain why logically-related faults sometimes did and sometimes did not cause correlated failures. The extent of the correlation will depend on the proportion of the inputs in the failure-domains of the common and identically wrong partial functions. The actual performance of an *N*-version system will depend on how often inputs from these common failure-domains are encountered in execution.

There are some important implications of this hypothesis in terms of whether "forcing" diversity will be effective. The first hypothesis, that correlated failures are a product of logically-related faults, implies only that separate development may not prevent different implementators from making the same mistake. The second hypothesis, that input-domain related faults may cause correlated failures, implies that correlated failures may occur even if the implementors use entirely different algorithms and make different mistakes.

VI. Conclusions

Our primary goal in this research was to understand what types of faults lead to coincident failures. We conclude that this occurs when the faulty paths have common input-domains. Correlated failures occur when the partial functions computed by the paths are identically wrong. The actual mistakes made, however, need not be similar or logically-related. We did find that programmers often make identical errors in logic. Any given algorithm for solving a problem is likely to involve some computations that are simply more difficult to handle correctly than others, and programmers are more likely to make mistakes on difficult computations than easy ones. We also found, however, that correlated failures arise from logically-*unrelated* faults in different algorithms or in different parts of the same algorithm. It is interesting that the programmers in our experiment did not seem able to identify the difficult parts of the problem or the difficult computations in their algorithms; the faults are not located in the parts of the programs where the programmers expected them to be, as determined by a postexperiment questionnaire.

The Consistent Comparison Problem [3], and other problems that we observed with real number comparisons, illustrate that an understanding of the detailed numerical issues involved in performing such comparisons is particularly important in *N*-version programming. Care must be taken in specifying and implementing *N*-version software to minimize difficulties in reaching a consensus among the versions. However, as has been shown elsewhere [3], these difficulties cannot be eliminated entirely.

Simple methods to reduce correlated failures arising from logically-unrelated faults (i.e., input-domain related faults) do not appear to exist. The faults that induced coincident failures were not caused by the use of a specific programming language or any other specific tool or method, and even the use of diverse algorithms did not eliminate input-domain related faults. In most cases, the failures resulted from fundamental flaws in the algorithms that the programmers designed. Thus we do not expect that changing development tools or methods, or any other simple technique, would reduce significantly the incidence of correlated failures in *N*-version software.

Appendix
Launch Interceptor Conditions

The Launch Interceptor Conditions are defined as follows:

1) There exists at least one set of two consecutive data points that are a distance greater than the length, "LENGTH1", apart.

$$(0 <= \text{LENGTH1})$$

2) There exists at least one set of three consecutive data points that cannot all be contained within or on a circle of radius "RADIUS1".

$$(0 <= \text{RADIUS1})$$

3) There exists at least one set of three consecutive data points which form an angle such that:

$$\text{angle} < (\text{"PI"} - \text{"EPSILON"})$$

or

$$\text{angle} > (\text{"PI"} + \text{"EPSILON"}).$$

The second of the three consecutive points is always the vertex of the angle. If either the first point or the last point (or both) coincides with the vertex, the angle is undefined and the LIC is not satisfied by those three points.

$$(0 <= \text{EPSILON} < \text{PI})$$

4) There exists at least one set of three consecutive data points that are the vertices of a triangle with area greater than "AREA1".

$$(0 <= \text{AREA1})$$

5) There exists at least one set of "Q_PTS" consecutive data points that lie in more than "QUADS" quadrants. Where there is ambiguity as to which quadrant contains a given point, priority of decision will be by quadrant number, i.e., I, II, III, IV. For example, the data point (0, 0) is in quadrant I, the point (−1, 0) is in quadrant II, the point (0, −1) is in quadrant III, the point (0, 1) is in quadrant I and the point (1, 0) is in quadrant I.

$$(2 <= \text{Q\_PTS} <= \text{NUMPOINTS}),$$

$$(1 <= \text{QUADS} <= 3)$$

6) There exists at least one set of two consecutive data points, $(X[i], Y[i])$ and $(X[j], Y[j])$, such that $X[j] - X[i] < 0$ (where $i = j - 1$).

7) There exists at least one set of "N_PTS" consecutive data points such that at least one of the points lies a distance greater than "DIST" from the line joining the first and last of these "N_PTS" points. If the first and last points of these "N_PTS" are identical, then the calculated distance to compare with "DIST" will be the distance from the coincident point to all other points of the "N_PTS" consecutive points. The condition is not met when "NUMPOINTS" < 3.

$$(3 <= \text{N\_PTS} <= \text{NUMPOINTS}), \quad (0 <= \text{DIST})$$

8) There exists at least one set of two data points separated by exactly "K_PTS" consecutive intervening points that are a distance greater than the length, "LENGTH1", apart. The condition is not met when "NUMPOINTS" < 3.

$$1 <= \text{K\_PTS} <= \{\text{NUMPOINTS} - 2\}$$

9) There exists at least one set of three data points separated by exactly "A_PTS" and "B_PTS" consecutive intervening points, respectively, that cannot be contained within or on a circle of radius "RADIUS1". The condition is not met when "NUMPOINTS" < 5.

$$1 <= \text{A\_PTS}, \quad 1 <= \text{B\_PTS}$$

$$\text{A\_PTS} + \text{B\_PTS} <= \text{NUMPOINTS} - 3$$

10) There exists at least one set of three data points separated by exactly "C_PTS" and "D_PTS" consecutive intervening points, respectively, that form an angle such that:

$$\text{angle} < (\text{"PI"} - \text{"EPSILON"})$$

or

$$\text{angle} > (\text{"PI"} + \text{"EPSILON"})$$

The second point of the set of three points is always the vertex of the angle. If either the first point or the last point (or both) coincide with the vertex, the angle is undefined and the LIC is not satisfied by those three points. When "NUMPOINTS" < 5, the conditions is not met.

$$1 <= \text{C\_PTS}, \quad 1 <= \text{D\_PTS}$$

$$\text{C\_PTS} + \text{D\_PTS} <= \text{NUMPOINTS} - 3$$

11) There exists at least one set of three data points separated by exactly "E_PTS" and "F_PTS" consecutive intervening points, respectively, that are the vertices of a triangle with area greater than "AREA1". The condition is not met when "NUMPOINTS" < 5.

$$1 <= \text{E\_PTS}, \quad 1 <= \text{F\_PTS}$$

$$\text{E\_PTS} + \text{F\_PTS} <= \text{NUMPOINTS} - 3$$

12) There exists at least one set of two data points, $(X[i], Y[i])$ and $(X[j], Y[j])$, separated by exactly "G_PTS" consecutive intervening points, such that $X[j] - X[i] < 0$ (where $i < j$). The condition is not met when "NUMPOINTS" < 3.

$$1 <= \text{G\_PTS} <= \{\text{NUMPOINTS} - 2\}$$

13) There exists at least one set of two data points, separated by exactly "K_PTS" consecutive intervening points, which are a distance greater than the length, "LENGTH1," apart. In addition, there exists at least one set of two data points (which can be the same or different from the two data points just mentioned), separated by exactly "K_PTS" consecutive intervening points, that are a distance less than the length, "LENGTH2," apart. Both parts must be true for the LIC to be true. The condition is not met when "NUMPOINTS" < 3.

$$(0 <= \text{LENGTH2})$$

14) There exists at least one set of three data points, separated by exactly "A_PTS" and "B_PTS" consecutive intervening points, respectively, that cannot be contained within or on a circle of radius "RADIUS1". In addition, there exists at least one set of three data points (which can be the same or different from the three data points just mentioned) separated by exactly "A_PTS" and "B_PTS" consecutive intervening points, respectively, that can be contained in or on a circle of radius "RADIUS2". Both parts must be true for the LIC to be true. The condition is not met when "NUMPOINTS" < 5.

$$(0 <= \text{RADIUS2})$$

15) There exists at least one set of three data points, separated by exactly "E_PTS" and "F_PTS" consecutive intervening points, respectively, that are the vertices of a triangle with area greater than "AREA1". In addition, there exist three data points (which can be the same or different from the three data points just mentioned) separated by exactly "E_PTS" and "F_PTS" consecutive intervening points, respectively, that are the vertices of a triangle with area less than "AREA2". Both parts must

be true for the LIC to be true. The condition is not met when "NUMPOINTS" < 5.

$$(0 <= \text{AREA2})$$

ACKNOWLEDGMENT

Thanks go to L. St. Jean, who helped in revising the problem specification and was responsible for much of the early design in the *N*-version experiment. P. Ammann contributed much time and effort in helping with the mammoth job of running the one million tests. We are indebted to J. Dunham and E. Migneault for allowing us to learn from the experience gained in an earlier version of this experiment. It is a pleasure to acknowledge the students who developed the 27 versions: C. Finch, N. Fitzgerald, M. Heiss, E. Irwin, L. Lauterbach, S. Samanta, J. Watts, and P. Wilson from the University of Virginia, and R. Bowles, D. Duong, P. Higgins, A. Milne, S. Musgrave, T. Nguyen, J. Peck, P. Ritter, R. Sargent, R. Schmaltz, A. Schoonhoven, T. Shimeall, G. Stoermer, J. Stolzy, D. Taback, J. Thomas, C. Thompson, and L. Wong from the University of California at Irvine. The Academic Computer Center at the University of Virginia, the AIRLAB facility and the Central Computer Complex at NASA Langley Research Center provided generous amounts of computer time to allow the programs to be tested.

REFERENCES

[1] T. Anderson and P. A. Lee, *Fault Tolerance, Principles and Practice*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[2] A. Avizienis and J. P. J. Kelly, "Fault-tolerant multi-version software: Experimental studies of a design diversity approach," Dep. Comput. Sci., Univ. California, Los Angeles, 1982.

[3] S. S. Brilliant, J. C. Knight, and N. G. Leveson, "The consistent comparison problem in *N*-version software," *IEEE Trans. Software Eng.*, vol. 15, pp. 1481-1485, Nov. 1989.

[4] L. Chen and A. Avizienis, "*N*-version programming: A fault-tolerance approach to reliability of software operation," in *Dig. FTCS-8: Eighth Annu. Int. Symp. Fault-Tolerant Computing*, Tolouse, France, June 1978, pp. 3-9.

[5] J. R. Dunham, J. L. Pierce, and J. W. Dunn, "Evaluating the reliability of *N*-version software subsystems—Some results from an ongoing research project," Research Triangle Inst., Research Triangle, NC, 1983.

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

[7] L. Gmeiner and U. Voges, "Software diversity in reactor protection systems: An experiment," in *Safety of Computer Control Systems*, R. Lauber, Ed. New York: Pergamon, 1980, pp. 75-79.

[8] W. C. Guenther, *Concepts of Statistical Inference*. New York: McGraw-Hill, 1965.

[9] J. P. J. Kelly, "Specification of fault-tolerant multi-version software: Experimental studies of a design diversity approach," Ph.D. dissertation, Univ. California, Los Angles, 1982.

[10] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 96-109, Jan. 1986.

[11] ——, "An empirical study of failure probabilities in multi-version software," in *Dig. FTCS-16: Sixteenth Annu. Int. Symp. Fault-Tolerant Computing*," Vienna, Austria, July 1986, pp. 165-170.

[12] J. C. Knight, N. G. Leveson, and L. D. St. Jean, "A large scale experiment in *N*-version programming," in *Dig. FTCS-15: Fifteenth Annu. Int. Symp. Fault-Tolerant Computing*, Ann Arbor, MI, June 1985, pp. 135-139.

[13] D. E. Knuth, *The Art of Computer Programming, vol. 2, Seminumerical Algorithms*. Reading, MA: Addison-Wesley, 1969.

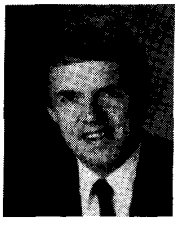
[14] D. J. Martin, "Dissimilar software in high integrity applications in flight controls software for avionics," in *AGARD Conf. Proc.*, Sept. 1982, pp. 36-1-36-13.

[15] P. M. Nagel and J. A. Skrivan, "Software reliability: Repetitive run experimentation and modeling," Boeing Computer Services Co., Seattle, WA, 1982 (prepared for National Aeronautics and Space Administration).

Susan S. Brilliant (S'87-M'88) received the B.S. degree in mathematics from Wake Forest University, Winston-Salem, NC, in 1972, the M.S. degree in accounting from Virginia Commonwealth University, Richmond, in 1977, and the M.S. and Ph.D. degrees from the University of Virginia, Charlottesville, in 1985 and 1988, respectively.

From 1986 to 1989 she was a member of the faculty at the University of Richmond and is presently an Assistant Professor of Mathematical Sciences at Virginia Commonwealth University.

Dr. Brilliant is a member of the Association for Computing Machinery and the IEEE Computer Society.



John C. Knight received the B.Sc. degree in mathematics from the Imperial College of Science and Technology, London, England, and the Ph.D. degree in computer science from the University of Newcastle-upon-Tyne, Newcastle-upon-Tyne, England, in 1969 and 1973, respectively.

From 1974 to 1981 he was with NASA's Langley Research Center and he joined the Department of Computer Science at the University of Virginia, Charlottesville, in 1981. He spent the period from August 1987 to August 1989 on leave at the Software Productivity Consortium in Herndon, VA.

Dr. Knight is a member of the Association for Computing Machinery and the IEEE Computer Society.

Nancy G. Leveson received the B.A. degree in mathematics, the M.S. degree in management, and the Ph.D. degree in computer science from the University of California, Los Angeles.

She has worked for IBM and is currently an Associate Professor of Computer Science at the University of California, Irvine. Her current interests are in software reliability, software safety, and software fault tolerance. She heads the Software Safety Project at UCI which is exploring a range of software engineering topics involved in specifying, designing, verifying, and assessing reliable and safe real-time software.

Dr. Leveson is a member of the Association for Computing Machinery, the IEEE Computer Society, and the System Safety Society.