# ANALYSIS OF FAULTS IN AN N-VERSION SOFTWARE EXPERIMENT

Susan S. Brilliant

John C. Knight

Nancy G. Leveson

Computer Science Report No. TR-86-20
September 11, 1986

This paper has been submitted to the *IEEE Transactions on Software Engineering*.

# ANALYSIS OF FAULTS IN AN *N*-VERSION SOFTWARE EXPERIMENT

**Susan S. Brilliant**

**John C. Knight**

**Nancy G. Leveson**

*Affiliation Of Authors*

Susan S. Brilliant    John C. Knight

Department of Computer Science

University of Virginia

Charlottesville

Virginia, 22903

Nancy G. Leveson

Department of Computer Science

University of California

Irvine

California, 92717

*Financial Acknowledgement*

This work was supported in part by NASA under grant numbers NAG-1-242, NAG-1-605 and NAG-1-606, in part by NSF grant DCR 8406532, and in part by MICRO grants cofunded by the state of California, Hughes Aircraft Company, and TRW.

## *Index Terms*

Design diversity, fault tolerant software, multi-version programming, *N*-version programming, software reliability.

*Address For Correspondence*

John C. Knight

Department of Computer Science

University of Virginia

Charlottesville

Virginia, 22903

## *Abstract*

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

After the versions had been tested, 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 faults that lead to coincident failures, and perhaps to use this analysis to determine methods of development for *N*-version software that would help avoid faults that cause correlated failures. 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. More surprisingly, there were 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. We conclude that minor changes 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.

# 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 air transports, 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).

Proposals have been made for building *fault-tolerant* software [1] in an attempt to deal with the faults that remain in operational software. One suggested approach, *N*-version programming [3], requires separate, independent preparation of multiple versions of a piece of software for an application. The versions are executed in parallel, and majority voting is used to determine which result is used. If *N* is at least three, and the voter and a majority of the versions perform correctly, then the required output will be produced. If a majority of the versions provide an incorrect answer, or there is no majority, the software will not perform correctly.

The amount of reliability improvement achieved by N-version programming is determined by the degree of independence of the failures of the versions [5]. 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. Thus, if several versions that are separately very reliable fail on the same inputs, their faults may not be tolerated and reliability may not be improved, although the error will be detected if different outputs are produced.

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 subjected to one million tests. The results of the tests revealed that the individual programs were extremely reliable. However, the number of tests 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.

It is important to determine why so many coincident failures were observed in these results. This is an important question for several reasons. First, it sheds light on the potential value of the technique itself. 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 [2,6,8,13]. 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 an attempt to answer the question of why coincident failures occur~~, 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, organized as follows. In the next section we summarize the experiment that yielded the twenty-seven programs studied here. The discussion of the faults themselves begins with a statistical analysis of the behavior of the failures that they induced. This analysis is presented in Section III. The goal of the statistical analysis is to determine which faults are responsible for failures that are statistically correlated, without regard to the details of the faults. The results of this analysis identify those faults that are causing more coincident failures than would occur by chance, and it is this behavior that needs to be understood.

Two significant but unexpected problems that were encountered in the course of the experiment and subsequent fault analysis are discussed in Section IV. These problems are quite general and associated with several of the faults. They are sufficiently important that they are discussed separately.

The faults themselves are described in Section V, and their interrelationships are discussed in Section VI. 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 actually cause coincident failures, and faults that seem very similar sometimes do not cause coincident failures. Our conclusions are presented in Section VII.

## II EXPERIMENT SUMMARY

Only the major features of the previous experiment are described in this paper since the details have been published elsewhere [9]. The application used in the experiment was a simple (but realistic) anti-missile system that came originally from an aerospace company [4,14]. 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 debugged using fifteen supplied test cases and any other test data that the student developed, it was subjected to an acceptance procedure that consisted of two hundred randomly-generated test cases. A different set of two hundred tests was generated for each program in order to avoid a general "filtering" of common faults by the use of a common acceptance test. Once all the versions had passed their acceptance tests, they were subjected to one million randomly-generated test cases in order to detect as many faults as possible. 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 testing.

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 test case. A *failure* was recorded for a particular version on a particular test 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 test case.

We define a *fault* to be a piece of the source text within a program version that is responsible for at least one failure in the sense that changing that piece of source text in some way would allow the program to obtain output agreeing with that of the gold program for that test. For each of the twenty-seven versions, the faults were identified by examining the output of the program for test 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 test case that caused the version to fail originally was regenerated. The version was then executed with each individual fault corrected in turn.

In most test cases where a version had failed, it 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 cases, 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.

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.

## III STATISTICAL ANALYSIS OF THE FAILURES

The faults found in each of the twenty-seven program versions and the number of failures attributable to each fault are shown in Table 1. Failures associated with more than one fault are counted in the number of failures for each of the associated faults.

The manifestations of a few of the faults were implementation-dependent. When the fault-to-failure identification testing was performed, a version sometimes passed a test case that it had failed when the test was executed originally. This effect was caused by

| 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 | | |

**Table 1. Fault Occurrence Rates**

differences in the hardware and compilers used, and was observed for versions 6, 22, 23, and 26. Analysis of the test 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 1. For versions 22 and 26, the original failures were caused by calls to the square root function (see Section IV); the specific calls responsible were not identified. These original failures are not counted in any of the failure rates shown in Table 1.

In order to determine which faults caused statistically correlated failures, a statistical test of independence was performed between each pair of faults. A matrix, $C$, of the coincident failures caused by each pair of faults was constructed. 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. Clearly $C$ is symmetric, and its diagonal represents the failure rates for the individual faults.

For each non-zero off-diagonal entry in $C$, an approximate chi-square test [7] was used to test the null hypothesis that the corresponding two faults cause failure independently. The observed value of the chi-square statistic for each pair i, j of faults causing common failures was calculated, using the following expression for the test statistic:

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

where

$$n = \textit{total number of test cases} = 1{,}000{,}000.$$

Where the observed chi-square statistic is greater than 7.88, the null hypothesis of independence can be rejected with 99.5 percent certainty. The results of the 990 separate hypothesis tests are shown in Table 2. An 'R' in Table 2 indicates that the null hypothesis was rejected for the corresponding pair of faults at the 99.5 percent level, and the two faults are considered to be *statistically 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 2 denotes a case in which the value of the chi-square 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.

The results of these hypothesis tests indicate that 101 of the hypotheses should be rejected; that is, 101 fault pairs found in the experiment are responsible for statistically correlated failures. 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 990 hypothesis tests that were performed, the expected number of erroneous rejections is *less than five*, whereas 101 were rejected.

It is clear from the preliminary data that more coincident failures occurred than would be expected by chance [9]. The results of these statistical tests show which faults were responsible for the coincident failures.

| 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 | R |  |  |  |  |  |  | R |  |
| 3.2 |  |  |  |  | R |  |  |  | ? | R | R |  |  | R |  | ? |  | R | ? |  |  | ? | R | ? |  | R | R |  |  |  |  |  | R | ? |  | R | R |  |  |  |  |  |  | ? | R |
| 3.3 |  | R |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3.4 |  | R | R |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 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 |
| 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 |
| 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 |  |  |  |  |  |  | R |  |
| 17.2 | ? | R | 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 | R |
| 20.2 |  | R | 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 |  |  |  |  |  |  | R | R |
| 22.2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R |  |  | R |  | R |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 22.3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R |  |  | R | 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 |  |  |  |  |  |  | ? |  |

Table 2. Results of Hypothesis Tests

## IV GENERAL PROBLEMS

Once the statistical correlations between failures were identified, the faults in the programs were examined to determine whether those causing coincident failures had any unique characteristics. Section V contains the details of the individual faults. In this section we describe some general problems that were each responsible for more than one fault.

The first problem occurred because, surprisingly, the Pascal compiler provided with the 4.1 BSD version of Unix for a DEC VAX 11/750 has a built-in square root function that does *not* necessarily give a fatal execution error when it is given a negative argument. With execution-time checking turned off (the default), it returns the negative of the square root of the absolute value of the argument. This caused some of the program versions developed at UCI to pass the acceptance procedure on a DEC VAX 11/750 but to fail quickly on any machine not using the 4.1 BSD version of Unix. During acceptance, the programs produced correct outputs but generated calls to the square root function with very small negative arguments in doing so.

The difficulty with the square root function was discovered after the programmers had been told that their programs were accepted but before testing began. The calls to the square root function that caused execution failures during a repeat of the acceptance procedure on a Prime 750 were replaced by calls to a function that returned zero as the value of the square root for small negative arguments. Thus no failures occurred during random testing from calls that should have caused failure during the acceptance procedure. Calls that caused failure during random testing but that would not have caused failure during the acceptance procedure are identified as faults.[1]

[1]Faults 11.1, 20.2, 22.1, 22.2, 22.3, 26.6, 26.7.

A second general problem in the experiment was related to a function called REALCOMPARE that was supplied to the programmers. This function defines the relational operators for floating point numbers in the manner described by Knuth [12]. 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 (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. The programmers were instructed to use REALCOMPARE for all comparisons of real numbers.

Two factors contributed to the problems associated with floating-point arithmetic. In order to allow as much diversity as possible, the specification did not prescribe exactly which quantities were to be compared using REALCOMPARE. Also, there was a general misunderstanding by the programmers of floating-point comparison. Although they were supplied with REALCOMPARE, they used it without question and, in many cases, the way in which the approximate comparisons were done should have been the subject of analysis by the programmer.

Some of the programmers used limited-precision comparisons with zero to determine *sign*. It is clear that a limited-precision comparison with zero will return equality for positive and negative numbers near zero, since only the most significant digits are being examined. For these values the programmers did not get the answer they expected.

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 REALCOMPARE find that the angles are different, but their cosines or sines are equal within the prescribed tolerance. The specification 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

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

where $\epsilon$ is a parameter supplied as input. The specification indicates that the second of the three points is the vertex. However, as is illustrated in Figure 1(a), there is still a choice of angle to be measured. Either the angle marked $\theta$ or the angle marked $2\pi - \theta$ could be considered. In absolute terms it makes no difference which angle is measured. Figure 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.

The level of understanding of numerical issues by the programmers involved in this experiment was less than desirable, but we suspect that this is a general problem. The

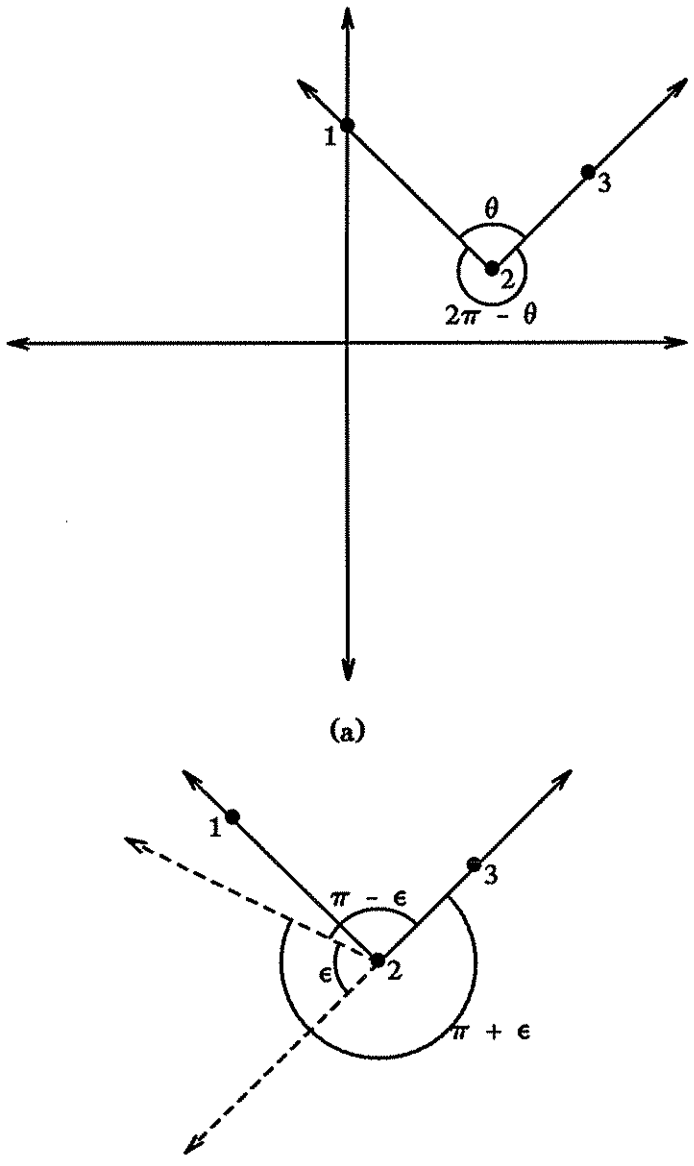

**Figure 1. The Angle Formed by Three Points**

importance of performing limited-precision comparisons correctly is magnified when *N*-version programming is used. For borderline cases it is possible that a single version might arrive at an output acceptable to the application despite the use of an inappropriate tolerance in performing a comparison. However, if the version is part of an *N*-version system, the system might fail because there is no consensus.

The use of floating point numbers may introduce the possibility of multiple correct solutions even for applications for which there do not appear to be such solutions. Previous researchers have pointed out that *N*-version programming is inappropriate for applications that allow multiple correct solutions [1]. It may be necessary to limit diversity deliberately in specifying *N*-version software in order to avoid this difficulty. More investigation is needed to determine whether adequate fault tolerance can be achieved while limiting diversity.

## V DESCRIPTION OF FAULTS

This section gives a short description of each fault.

**Fault 1.1** occurs in the evaluation of launch condition 10 and arises from the opportunity for multiple solutions discussed in Section IV. The gold version always computes an angle between zero and $\pi$ but this version sometimes computes an angle between $\pi$ and $2\pi$. The gold version may conclude that the launch condition is satisfied when this version does not.

**Fault 3.1** occurs in the evaluation of launch condition 3, which requires the calculation of the angle formed by three points. The specification states that point 2 is the vertex of the angle. Version 3 treats a set of three collinear points as a special case and assumes that they form an angle of $\pi$ radians even though the angle is $\pi$ *only* if point 2 lies *between* points 1 and 3; otherwise the angle is zero.

**Fault 3.2** is the same error in logic as the fault 3.1. It occurs in determining whether or not condition 10 is satisfied.

**Fault 3.3** occurs in the determination of the distance of a point from the line formed by two other points, needed in evaluating launch condition 7. Version 3 treats the situation in which all three points are collinear as a special case. The distance of interest in that case is zero, since the point lies on the line. However, version 3's algorithm gives a distance of zero only if the first point lies *between* the other two on the line. Otherwise the distance to the nearest of the other two points is calculated instead.

**Fault 3.4** arises in the determination of the angle formed by three points (launch conditions 3 and 10). As discussed in fault 3.1, version 3 treats collinear points as a special case. The algorithm used to determine collinearity is fairly inaccurate. The lengths of the sides of the triangle having the three points as vertices are calculated. If the sum of the shorter two sides is equal to the longest side, within the tolerance allowed by REALCOMPARE, then the three points are considered to be collinear. Thus an angle that is only close to $\pi$ may be calculated to be exactly $\pi$.

This fault has an interesting relationship with faults 3.1 and 3.2. If a more precise algorithm is used to determine whether three points are collinear, far fewer angles are calculated using the faulty algorithm along this path. Instead the angle is calculated according to the correct algorithm on the alternate path. Therefore there are a large number of cases in which failure can be avoided by correcting *either* the path condition or the algorithm used on the path.

**Fault 6.1** occurs in a programmer-defined function called `rad_circum`. This function is called in evaluating launch conditions 9 and 14 to find the radius of the smallest circle containing three points. The faulty code is shown in Figure 2. The condition is intended to handle as a special case those instances in which any two of the three points coincide. However, the occurrences of `or` in the condition should be replaced by `and` and the `and`'s replaced by `or`'s. This fault is particularly interesting because the test cases on which failure occurs are partially compiler dependent. Note that, on the path that the

```
if ((REALCOMPARE(x1-x2,0.0) = EQ) or
    (REALCOMPARE(y1-y2,0.0) = EQ)) and
   ((REALCOMPARE(x1-x3,0.0) = EQ) or
    (REALCOMPARE(y1-y3,0.0) = EQ)) and
   ((REALCOMPARE(x2-x3,0.0) = EQ) or
    (REALCOMPARE(y2-y3,0.0) = EQ)) then
  begin (*coincident points*)
    if (REALCOMPARE(x1-x2,0.0) = EQ) and (REALCOMPARE(y1-y2,0.0) = EQ) then
      rad_circum := 0.5*distance(x1,y1,x3,y3);
    if (REALCOMPARE(x1-x3,0.0) = EQ) and (REALCOMPARE(y1-y3,0.0) = EQ) then
      rad_circum := 0.5*distance(x1,y1,x2,y2);
    if (REALCOMPARE(x2-x3,0.0) = EQ) and (REALCOMPARE(y2-y3,0.0) = EQ) then
      rad_circum := 0.5*distance(x1,y1,x2,y2);
  end (*coincident points*)
```

**Figure 2. Code Responsible for Fault 6.1**

programmer intended to handle coincident points, each possible combination of points that might be coincident is considered separately. However, the cases that are supposed to follow this path are not the ones that actually do, so in many cases no value is assigned for the function rad_circum. Whether failure results depends on how the particular implementation of Pascal handles this situation. In many cases, a random value contained in the stack location designated for the result will be returned.

**Fault 6.2** occurs in the evaluation of launch condition 2, which requires the determination of the size of the smallest circle containing three points. Rather than calling his rad_circum function, version 6's author includes in-line code to perform the calculation. This code calculates the radius of the circle as half the longest side of the triangle formed by the three points. However, a circle with this radius does not contain all three points if the triangle is acute. No separate path is included to handle acute triangles.

**Fault 6.3** occurs in the evaluation of the special case of launch condition 7 in which the first and last of "N_PTS" consecutive data points coincide. In version 6 the coincident points are (x[i], y[i]) and (x[j], y[j]), hereafter referred to as point i and point j. The index k is used to count through the points between point i and point j. The distance to be calculated should be from point k to either point i or point j; instead the distance from point k to point k+1 is calculated.

**Fault 7.1** occurs in a function called in the evaluation of launch conditions 3 and 10 to calculate the angle formed by three points. The function begins by calculating the cosine of the angle. REALCOMPARE is used to compare the calculated cosine to -1, 1, and 0. If REALCOMPARE returns EQ, the angle is determined to be $\pi$, 0, or $\pi/2$ respectively. This fault is responsible for failure on test 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. For example, given the three points (2.0, 1.9), (-1.5, -4.3), and (2.2, 2.3), both the gold version and this version agree that the cosine of the angle formed is about 0.9999956. Using limited precision, REALCOMPARE determines that this value is equal to 1, so version 7's algorithm computes an angle of zero. The gold version computes the angle to be 0.0029701. Since $(\pi-\epsilon)$ for this case is 0.0027317, the gold version does not consider the launch condition to be satisfied. Version 7, however, finds the angle to be zero, which is sufficiently different from the value of $(\pi-\epsilon)$ to allow REALCOMPARE to return LT. Version 7 concludes that the launch condition is satisfied.

**Fault 8.1** is similar to faults 3.1 and 3.2. In calculating the angle needed for evaluating launch conditions 3 and 10, this version, like version 3, handles collinear points as a special case. Version 8 compares the slopes of the rays that form the angle to establish collinearity. When the points are determined to be collinear, this version checks whether points 1 and 3 coincide. If they do, the version correctly gives the value of the angle as zero. In all other cases, version 8 computes an angle of $\pi$ radians, even though the angle is zero unless point 2 is between points 1 and 3.

**Fault 8.2** is the same error in logic as the fault 8.1, but occurs on a path that handles sets of three points that form either a horizontal or vertical line.

**Fault 9.1** is found in the programmer's function `radius`, which is called in evaluating launch conditions 2, 9, and 14. In calculating the radius of the smallest circle containing three points this version correctly handles on separate paths the cases in which the three points form an obtuse triangle and those cases in which an acute triangle is formed. However, the path condition that determines whether a triangle is obtuse is incorrect. Version 9 calculates `D`, the length of the longest side of the triangle, and `H`, the perpendicular distance to the third vertex. According to version 9, if `D` $\geqslant$ (2 * `H`) then the triangle formed is obtuse. This is not always the case.

**Fault 9.2** occurs in the programmer's function `area`, which calculates the area of a triangle formed by three points (launch conditions 4, 11, and 15). The variables `a`, `b`, and `c` hold the lengths of the sides of the triangle; `s` holds half of their sum. It is geometrically impossible that any of the quantities `s`, (`s - a`), (`s - b`), or (`s - c`) are negative. Nevertheless fatal execution errors occasionally occur in computing the quantity:

```
sqrt (s * (s - a) * (s - b) * (s - c))
```

despite the programmer's attempt to prevent the problem by handling separately the cases in which the absolute value of the argument is less than $10^{-8}$.

**Fault 11.1**, like fault 9.2, is due to machine round-off error. The program experiences fatal execution errors in the programmer's function `angle`, which computes the angle formed by three points as required by launch conditions 3 and 10. The variable `cosine` contains the correctly computed cosine of the angle. However, a call to `sqrt` with the argument `(1 / sqr(cosine) - 1)` results in failures when round-off error has given a calculated cosine having an absolute value greater than 1.

**Fault 12.1** occurs in the programmer's `whichquad` function, which is called in determining whether launch condition 5 is satisfied. Because of an error in a relational operator ("=" is used instead of ">="), the version assigns points on the right side of the x-axis to the second quadrant rather than the first.

**Fault 12.2** is essentially the same as fault 7.1. In a function that evaluates the angle needed for launch conditions 3 and 10, the programmer compares the absolute value of the calculated cosine of the angle to 1 using REALCOMPARE. If this test returns EQ then the angle is calculated as either $\pi$ or zero, depending on the sign of the cosine.

**Fault 13.1** occurs in the code shown in Figure 3. The programmer calls REALCOMPARE to ensure that the quantity:

```
(sqr (testradius) - sqr (0.5 * baselength))
```

is non-negative before the quantity is given as an argument to the built-in `sqrt` function. Due to the tolerance allowed by REALCOMPARE, not all negative values of this quantity are detected, so fatal execution errors sometimes result.

**Fault 14.1** occurs in evaluating the special case of launch condition 10 in which the second in a set of three points coincides with either the first or third. Version 14 calls the programmer-defined function `sam3pts` in order to check for this special case. However, the

```
else if REALCOMPARE(baselength, 2*testradius) = GT then
   pntcirclerltn := GT
else begin
   basebisect := sqrt( sqr(testradius) - sqr(0.5*baselength) );
```

**Figure 3. Code Responsible for Fault 13.1**

programmer made an apparent typographical error in the call:

```
if sam3pts(x[i],y[i],x[j],y[i],x[k],y[k])<>1 then
```

since the second occurrence of "y[i]" in the call should be "y[*j*]".

**Fault 14.2** is the same as fault 7.1.

**Fault 16.1** occurs in handling the special case of the calculation of the angle formed by three points (launch conditions 3 and 10) in which the rays from point 2 through points 1 and 3 are both vertical. If points 1 and 3 are on the same side of point 2 then the angle is correctly determined to be zero. However, if point 2 lies between points 1 and 3 then the function gives an angle of 180 rather than $\pi$.

**Fault 16.2** is similar in origin and effect to fault 13.1. The path condition:

```
if REALCOMPARE( dist(cpoint1,cpoint2), (2*radius) ) = gt
```

is used in an effort to prevent a negative argument to the sqrt function in the sequence:

```
halfchorddist := dist( cpoint1, cpoint2 ) / 2 ;
distmidpointtocenter :=
   sqrt( radius * radius - halfchorddist * halfchorddist );
```

The use of REALCOMPARE allows cases in which dist(cpoint1,cpoint2) is only slightly greater than (2 * radius) to follow the else path to the sqrt function call, causing a fatal execution error.

**Fault 17.1** occurs in the calculation of the angle needed for launch condition 3. Version 17 never calculates the angle itself; instead its cosine is calculated and compared using REALCOMPARE to the cosine of the reference angle ($\pi - \epsilon$). This results in failure in cases in which the angle that satisfies condition 3 is near zero. There are two reasons this occurs. The first is that the size of an angle is much smaller than the corresponding cosine for angles in this range, so that angles must be closer in value than their cosines in order for REALCOMPARE to return EQ. Secondly the cosine curve is relatively flat near zero, so there is a large range of angles with nearly equal cosines.

**Fault 17.2** is the same error in logic as the fault 17.1. It occurs in the calculation of launch condition 10.

**Fault 18.1** occurs in the programmer's `angle` function, which computes the angle formed by three points as needed for launch conditions 3 and 10. This version, like version 1, sometimes computes the angle between $\pi$ and $2\pi$ rather than the angle between zero and $\pi$ and fails for the same reasons. The different algorithm used by this version causes failure to occur on different test cases.

**Fault 19.1** occurs in handling the special case of launch conditions 3 and 10 in which point 2 coincides with either point 1 or point 3. According to the specification, in this situation the angle is undefined and the condition is not satisfied *by those three points*. Whenever such a set of points is found, this program decides that the condition is not satisfied by *any* set of three points.

**Fault 20.1** occurs in calculating the angle formed by three points as required by launch conditions 3 and 10. This version begins by calculating the tangent of the angle. If the angle's tangent is found to be zero, then the version always gives an angle of $\pi$. However, the tangent is also zero if the angle is zero. Where the correct angle is zero, the incorrect value $\pi$ is returned.

**Fault 20.2**, like faults 9.2 and 11.1, results from the programmer's failure to anticipate the effects of imprecision in machine arithmetic. Fatal execution errors sometimes occur on the line:

```
tn := sqrt(1.0 - sqr(cs)) / cs;
```

in which the variable `cs` contains the correctly computed cosine of the angle needed in the evaluation of launch conditions 3 and 10. The case in which `cs` is zero has been separately considered, so division by zero does not occur. In theory the value (1.0 - `sqr(cs)`) should always be non-negative, since the cosine always lies between 1 and -1. However, round-off error sometimes results in a calculated cosine outside of that range, and the program aborts on the call to the sqrt function.

**Fault 21.1** occurs in the programmer's function incircle, which is called in the evaluation of launch conditions 2, 9, and 14. In determining the smallest circle containing a set of three points, this version treats separately cases in which the first or third point coincides with the second. For these cases the distances between points 1 and 3 and between points 2 and 3 are calculated. These distances are added to determine the diameter of the desired circle. This procedure does not work for the case in which points 1 and 2 coincide.

**Fault 21.2** also occurs in the function incircle. This fault is similar to faults 13.1 and 16.2. The condition:

```
if is(length,[gt],2*radius) then
```

is designed to ensure that the argument to the sqrt function in:

```
cdist:=sqrt(sqr(radius)-sqr(length/2));
```

is non-negative. The call to the programmer-defined is function will return true whenever REALCOMPARE returns GT. If the programmer had used strict comparison, this line would guarantee that the else path to the call to sqrt is not followed for cases in which the argument is negative.

**Fault 22.1** occurs in calculating the area of the triangle having three points as its vertices for the evaluation of launch condition 4. The variables d1, d2, and d3 hold the lengths of the sides of the triangle; intvalue holds half of their sum. It is therefore impossible for any of the quantities intvalue, (intvalue – d1), (intvalue – d2), or (intvalue – d3) to be negative, but occasionally round-off error gives a negative value for one of them, resulting in a negative argument to the sqrt function on the sequence:

```
radical := intvalue * (intvalue - d1) * (intvalue - d2) * (intvalue - d3);
triarea := sqrt(radical);
```

**Fault 22.2** is the same error in logic as fault 22.1. It occurs in the evaluation of launch condition 11.

**Fault 22.3** is the same error in logic as faults 22.1 and 22.2. It occurs in the evaluation of launch condition 15.

**Fault 23.1** is an example of a programmer's misuse of the REALCOMPARE function. Launch conditions 3 and 10 specify that if the second point in a set of three coincides with either of the other two, no angle is formed and the points do not satisfy the launch condition. In version 23 the programmer checks for this condition inside the function `angle`, which usually returns the value of the angle formed. In this case the dummy angle value of -1 is returned. To ensure that a negative value is never returned when an angle is formed, the line of code

```
if REALCOMPARE(theta,0.0) = LT  then theta := theta + 2*PI;
```

is used. The use of REALCOMPARE allows some negative angle values to be returned. The code following the call to `angle` treats these negative values as if no angle is formed.

**Fault 23.2** also occurs in the programmer's `angle` function. Like versions 1 and 18, this version often computes an angle between $\pi$ and $2\pi$ rather than an angle between zero and $\pi$. For these cases failure sometimes results because of the greater tolerance allowed by REALCOMPARE in comparing the larger angles.

**Fault 24.1** occurs in this calling sequence:

```
if not issame(x1,y1,x2,y2) and not issame(x3,y3,x2,y2) then
   findcircle :=
        REALCOMPARE(findangle(x1,y1,x3,y3,x2,y2), PI / 2) = LT
```

A call to the function `findangle` will result in a fatal division by zero whenever the point whose coordinates are the third and fourth arguments coincides with either the point having coordinates equal to the first pair or the last pair of arguments. For all except one of the calls to findangle the programmer checks correctly for the coincidence of these points before calling findangle. The function issame is used to check for point coincidence. The first call to issame shown above is given the wrong arguments to prevent failure on the succeeding call to findangle. The coincidence of points 1 and 3 should be checked rather than the coincidence of points 1 and 2.

**Fault 25.1** occurs in the function angle, called in evaluating launch conditions 3 and 10 to find the angle formed by three points. The faulty code is shown in Figure 4. The function lengthline returns the distance between the two points whose coordinates are given to it as arguments. The code following the calls to lengthline handles the case in which the three points fall on a vertical line. In this case the angle formed is either zero or $\pi$, depending on the order of the points on the line. The condition that determines whether the angle formed is zero does not cover the case in which point 1 lies between points 2 and 3, and is closer to point 2 than to point 3.

```
alen := lengthline( x1, y1, x2, y2);
blen := lengthline( x2, y2, x3, y3);
clen := lengthline( x3, y3, x1, y1);

if ((REALCOMPARE (x1, x2) = EQ) and (REALCOMPARE (x2, x3) = EQ)) then
    if REALCOMPARE( clen, alen) = LT then    (* catch angle that
                                                   doubles back *)
        angle := 0.0
    else
        angle := PI
```

**Figure 4. Code Responsible for Fault 25.1**

**Fault 25.2** is the same error in logic as the fault 25.1. It occurs on the path on which the general case of three collinear points is handled.

**Fault 25.3** occurs in the programmer's function `tricirrad`, which is called in determining whether launch conditions 2, 9, and 14 are satisfied. The function determines the radius of the smallest circle containing three points whose coordinates are given as arguments. The faulty code, which occurs in handling the special case in which the area of the triangle formed by the three points (contained in the variable `area`) is zero, is shown in Figure 5. If `area` is zero then the three points are collinear, so the radius of the circle containing the triangle is half the length of its longest side. The case in which (`blen` $\geq$ `clen` $\geq$ `alen`) is not correctly handled. The code shown will return ((`alen` + `blen`) / 2) for these cases, which is correct only when `alen` happens to be zero.

```
alen := lengthline(x1, y1, x2, y2);
blen := lengthline(x2, y2, x3, y3);
clen := lengthline(x3, y3, x1, y1);
                .
                .
                .

if (area = 0) then
    if REALCOMPARE( clen, alen) = LT then   (* catch angle that
                                               doubles back *)
        if REALCOMPARE (alen, blen) = GT then
            tricirrad := alen/2
        else
            tricirrad := blen/2
    else
            tricirrad := (alen + blen) /2
```

**Figure 5. Code Responsible for Fault 25.3**

**Fault 26.1** occurs in the calculation of launch condition 7. Version 26 treats separately the case in which the line joining the first and last of "N_PTS" consecutive data points is vertical. In this case the distance is calculated to be the x-coordinate of the point currently under consideration. The true distance is the difference between the x-coordinate of the point under consideration and the x-coordinate of the points on the vertical line joining the first and last points.

**Fault 26.2** also occurs in the calculation of launch condition 7. In stepping through the sets of points that might satisfy the condition, the programmer initially sets the variable `start` to 1 and the variable `endpt` to "N_PTS". These variables are incremented as each set of points is considered until either the condition is determined to be satisfied or all the possible sets of points have been considered. Within each set of points, the programmer treats as a special case instances in which the first and last points coincide; for these cases the specification states that the distance to the coincident point is to be measured. In stepping through the individual points between the endpoints in the set, an index is started at `start` and incremented until it reaches "N_PTS". This will work properly only for the first set of points. The index should run from `start` to `endpt`, as it does on all of the other branches.

**Fault 26.3** occurs in the evaluation of launch condition 10. Rather than computing the angle formed by three points, this version calculates only the *sine* of the angle. The sine is then compared to the sine of the reference angle, $(\pi - \epsilon)$. Comparison of the sines of the angles using REALCOMPARE gives erroneous results for points on the flat part of the sine curve, at angles near $\pi/2$ (sines near 1).

**Fault 26.4** occurs in the evaluation of launch condition 3. The programmer compares the sines of the relevant angles rather than the angles themselves, as discussed above. This requires a case analysis because the slope of the sine curve changes sign at $\pi/2$. Version 26 begins by determining whether the angle formed by the three points is a right angle, an obtuse angle, or an acute angle. The fault occurs in the code that handles cases in which an acute angle is formed. There is a path to handle acute angles when the parameter $\epsilon$ is greater than $\pi/2$ (as determined by REALCOMPARE) and another to handle acute angles when $\epsilon$ is less than $\pi/2$. However, cases in which REALCOMPARE $(\epsilon,\pi/2)$ returns EQ are not considered on any branch. In these cases the launch condition is satisfied (an acute angle is less than $(\pi - \pi/2)$), but this version never considers the case and assumes by default that the condition is *not* satisfied.

**Fault 26.5** is the same error in logic as fault 26.4. It occurs in the evaluation of launch condition 10.

**Fault 26.6** occurs in the evaluation of launch condition 3. The variables `d1`, `d2`, and `d3` contain the lengths of the sides of the triangle having the three points as vertices, and `sp` contains half of their sum. It is geometrically impossible for any of `sp`, (`sp – d1`), (`sp – d2`), or (`sp – d3`) to be negative, but due to imprecisions in machine arithmetic the argument to the `sqrt` function in the call:

```
sqrt(sp*(sp-d1)*(sp-d2)*(sp-d3))
```

is sometimes negative, so fatal execution errors result.

**Fault 26.7** is the same error in logic as fault 26.6. It occurs in the evaluation of launch condition 10.

## VI DISCUSSION

Our goal in analyzing the individual faults in the versions was to attempt to understand the large number of coincident failures that were observed in the experiment. We wanted to determine what other relationships, if any, exist among faults that are statistically 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 statistically 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 statistical 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 $\pi$. It is easy to explain the statistical 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 $\pi$ 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 but which do not cause statistically correlated failures. For example, faults 7.1 and 17.1 both result from the application of the REALCOMPARE function to the cosines of angles rather than to the angles themselves in the same computation, yet they caused no coincident failures. Fault 7.1 causes failure on test 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 statistical correlations among failures. For example, faults 11.1, 20.2, 22.1, 26.6, and 26.7 are all statistically 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 $\pi$. Thus the statistical 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 round-off errors. Once again, this can only occur when the three points forming the triangle are approximately collinear, and the statistical correlation is explained.

More difficult to understand is the statistical correlation between fault 14.1 and each of faults 3.2, 8.2, 11.1, 20.1, 20.2, and 25.1. As explained in Section V, 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 coincident failures but which are not logically related. Thus our initial hypothesis does not explain all the observed statistical correlations. We propose therefore an alternative hypothesis. We define two faults to be *special-case related* if they both involve a mishandling of all inputs having some specific characteristic, i.e. both faults involve handling the same special case incorrectly. We hypothesize that faults that are statistically correlated are special-case related, and vice versa.

It is clear from our definition of special-case related faults that coincident failures will occur on all inputs having the specific characteristic that identifies the special case relating two such faults. The extent of correlation will depend on the proportion of the inputs which possess the given characteristic.

Our second hypothesis can be seen as an extension of the first, since logically related faults may involve mishandling the same special case in the same way. The occurrence of logically related faults was not unexpected. Our initial hypothesis reflects the possibility that separate development may not prevent different implementors from making the same mistake. This is a possibility that has been of concern to those building *N*-version software. The occurrence of special-case related faults that are not logically related was *not* expected, and is in fact of far greater concern. Even where different programmers have used entirely different algorithms and have made different mistakes during development, the same special input cases may cause their programs to fail.

## VII CONCLUSION

Our primary goal in this research was to understand what types of faults lead to coincident failures. We conclude that special cases in the input space give rise to this type of fault. As anticipated, we found 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. We also found, though, that the same special cases in the input space will cause difficulty in different algorithms or in different parts of the same algorithm. In the light of this, it is interesting to note that the faults are not located in the parts of the programs where the programmers expected them to be, as determined by a post-experiment questionnaire.

We were surprised that something as simple as a square root function would be implemented incorrectly in one of the environments that we used. This problem illustrates clearly the difficulties that environments can produce, and supports the use of diverse environments in developing and operating *N*-version software.

The problems that we experienced 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 avoid introducing difficulties in reaching a consensus among the versions.

Our findings indicate that the statistically correlated faults found in this experiment result from the nature of the application, from similarities in the difficulties experienced by individual programmers, and from special cases in the input space. Simple methods to reduce these correlated 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 the correlations among the faults induced by special cases in the input space. Thus we do not expect that changing development tools or methods, or any other simple technique, would reduce significantly the incidence of statistically correlated faults in $N$-version software.

Despite the existence of statistically correlated faults, $N$-version programming may increase reliability over a single program. In fact, $N$-version systems built using the programs whose faults are analyzed in this paper achieved a reduction in failure probability [11]. By analyzing faults such as those described here, methods might be developed to allow the performance of $N$-version systems to be further improved.

# APPENDIX A

## 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 <= 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 <= RADIUS1 )

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

angle < ('PI' - 'EPSILON')
or
angle > ('PI' + '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 <= EPSILON < 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 <= 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 <= Q_PTS <= NUMPOINTS ) , ( 1 <= 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 <= N_PTS <= NUMPOINTS ) , ( 0 <= 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 <= K_PTS <= {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 <= A_PTS , 1 <= B_PTS
A_PTS + B_PTS <= 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:

angle < ('PI' - 'EPSILON')
or
angle > ('PI' + '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 condition is not met.

1 <= C_PTS , 1 <= D_PTS
C_PTS + D_PTS <= 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.

$1 <= E\_PTS \quad , \quad 1 <= F\_PTS$
$E\_PTS + F\_PTS <= NUMPOINTS - 3$

(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 .

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

(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 .

$( 0 <= LENGTH2 )$

(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 .

$( 0 <= RADIUS2 )$

(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 .

$( 0 <= AREA2 )$

## ACKNOWLEDGEMENTS

Thanks go to Lois St. Jean, who helped in revising the problem specification and was responsible for much of the early design in the *N*-version experiment. Paul Ammann contributed much time and effort in helping with the mammoth job of running the one million tests. We are indebted to Janet Dunham and Earl 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 twenty-seven versions: Carol Finch, Nancy Fitzgerald, Marina Heiss, Edith Irwin, Linda Lauterbach, Shivaji Samanta, Jonathan Watts, and Pat Wilson from the University of Virginia, and Richard Bowles, Dung Duong, Paul Higgins, Alastair Milne, Scott Musgrave, Thieu Nguyen, J. Peck, Phillip Ritter, Robert Sargent, Ronald Schmaltz, A. Schoonhoven, Tim Shimeall, Gary Stoermer, Janice Stolzy, D. Taback, Jeffrey Thomas, Chris Thompson, and Lawrence 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. This work was supported in part by NASA under grant numbers NAG-1-242, NAG-1-605 and NAG-1-606, in part by NSF grant DCR 8406532, and in part by MICRO grants cofunded by the state of California, Hughes Aircraft Company, and TRW.

## REFERENCES

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

[2] A. Avizienis and J. P. J. Kelly, *Fault-Tolerant Multi-Version Software: Experimental Studies of a Design Diversity Approach*, UCLA Computer Science Department, Los Angeles, 1982.

[3] L. Chen and A. Avizienis, N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation, *Digest FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, Tolouse, France, June 1978, pp. 3-9.

[4] J. R. Dunham, J. L. Pierce and J. W. Dunn, *Evaluating the Reliability of N-Version Software Subsystems - Some Results from an On-Going Research Project, Research Triangle Institute, Research Triangle, N.C., 1983.*

[5] D. E. Eckhardt and L. D. Lee, A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors, *IEEE Transactions on Software Engineering*, December 1985, pp. 1511-1517.

[6] L. Gmeiner and U. Voges, Software Diversity in Reactor Protection Systems: An Experiment, in *Safety of Computer Control Systems*, R. Lauber (ed.), Pergamon Press, 1980, 75-799.

[7] W. C. Guenther, *Concepts of Statistical Inference*, McGraw-Hill Book Company, New York, 1965.

[8] J. P. J. Kelly, *Specification of Fault-Tolerant Multi-Version Software: Experimental Studies of a Design Diversity Approach*, PhD Dissertation, University of California at Los Angeles, 1982.

[9] J. C. Knight, N. G. Leveson and L. D. St. Jean, A Large Scale Experiment in N-Version Programming, *Digest FTCS-15: Fifteenth Annual International Conference on Fault Tolerant Computing*, Ann Arbor, Michigan, June 1985.

[10] J. C. Knight and N. G. Leveson, An Experimental Evaluation of the Assumption of Independence in Multiversion Programming, *IEEE Transactions on Software Engineering*,

January 1986, pp. 96-109.

[11] J. C. Knight and N. G. Leveson, An Empirical Study of Failure Probabilities in Multi-Version Software, *Digest FTCS-16: Sixteenth Annual International Conference on Fault Tolerant Computing*, Vienna, Austria, July 1986, pp. 165-170.

[12] D. E. Knuth, *The Art of Computer Programming*, Addison-Wesley, Reading, Massachusetts, 1969.

[13] D. J. Martin, Dissimilar Software in High Integrity Application in Flight Controls, *Software for Avionics, AGARD Conference Proceedings*, January 1983, pp. 36-1 - 36-9.

[14] P. M. Nagel and J. A. Skrivan, Software Reliability: Repetitive Run Experimentation and Modeling, prepared for National Aeronautics and Space Administration, 1982.