ANALYSIS OF FAULTS IN A MULTI-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, 12345

Financial Acknowledgement

This work was supported, in part by the National Aeronautics and Space Administration under grant number NAG-1-242, and in part by some UCI grant.

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

The software fault tolerance methodology known as ‘‘*N*-version’’ programming depends for its reliability improvement on the assumption that programs that have been developed independently will fail independently. We have conducted a large-scale experiment in which this fundamental axiom is tested. 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 expected.

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 this analysis was to understand better the nature of faults that lead to coincident failures, and perhaps to determine from this analysis methods of development for multi-version software that would help avoid correlated faults. We found that there are small and quite subtle parts of the problem for which more than one correct solution exists due to the manner in which real numbers are compared. We also found that in some cases the programmers made equivalent logical errors, indicating that some parts of the problem were simply more difficult to get right than others. More surprisingly, there were cases in which apparently different logical errors yielded faults that caused statistically correlated failures, indicating that there are some inputs that present more difficulty than others. We do not think that minor changes in programming languages or similar parameters would have a major impact in reducing the incidence of correlated faults.

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* [] do improve software reliability, new applications for computers in safety-critical systems, such as commercial aircraft and medical devices, have increased the basic 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 despite the use of all possible methods of fault avoidance and removal. One suggested approach, multi-version or *N*-version programming [2], 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 results will be used. If *N* is at least three, the voter is reliable, and a majority of the versions do perform correctly, then the system will be fault tolerant and the correct 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.

A very large improvement in reliability for *N*-version programming requires that the versions fail independently. That is, if two versions fail on the same input in a *3*-version system, 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. We have conducted a large-scale experiment [5] to determine whether programs that are developed independently fail independently. 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 programs were individually 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 independently.

The question that arises immediately from these results is why do so many coincident failures occur? This is an important question for several reasons. In general, if the cause could be identified, perhaps these faults could be avoided or removed. A better understanding of the faults will allow evaluation of techniques that have been suggested for minimizing coincident failures in multi-version software; for example, the use of different programming languages or development environments [Udo et al]. In addition, new development techniques for multi-version software may be suggested from examining the coincident failures. Finally, if after detailed analysis and experimentation multi-version software does not turn out to be a cost-effective way of improving reliability, a study of the faults made by different programmers on the same problem could 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. Our goal in locating and analyzing the faults is to try to understand the underlying reasons for the dependent failures.

Our analysis of the faults is organized as follows. In the next section we summarize the experiment that yielded the twenty-seven programs studied here. Two significant but unexpected problems that were encountered in the course of the experiment and subsequent fault analysis are discussed in Section III. These problems are quite general and associated with several of the faults. They are sufficiently important that they are dealt with separately.

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 IV. 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.

In Section V we describe the faults themselves, and we discuss their interrelationships in Section VI. We show that faults which 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 [.KLS85.]. The application used in the experiment was a simple (but realistic) anti-missile system that came originally from an aerospace company [.ref.]. 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.

Once a program was debugged using fifteen supplied test cases and any other data that the student developed, it was subjected to an acceptance test 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 test, 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 version (referred to as the *gold* program) that had been subjected to extensive previous testing. A program was considered to have failed on a given test case if its results differed from the results generated by the gold program.

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 working on Master's degrees held at least a Bachelor's degree, and four working on Doctoral degrees held at least a Master's degree. 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.

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. Although the launch condition is the only true output in this application, a *failure* was recorded for a particular version on a particular test case if there is *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. The intermediate results were compared because, in a practical multi-version system, votes would be taken on intermediate results. It is more likely that a majority of versions will agree on the correct result of a smaller intermediate computation than that they will agree on the correct result of the entire computation.

We define a *fault* to be the 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 the correct output for that test. Note that nothing is said here about uniqueness. It may be possible to make any one of several different changes to the program that will allow it to work correctly on a given 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 the 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 correction 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 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 GENERAL PROBLEMS

Two problems that were responsible for more than one fault arose while performing the experiment. The first arose because different computers were used for the acceptance testing and for the lifetime simulation, and the second was a consequence of trying to avoid difficulties with round-off errors in floating point computation.

It was discovered that the Pacsal compiler provided with the 4.1 BSD version of Unix for a DEC VAX 11/750 has a built-in square root function (*sqrt*) that does *not* necessarily give a fatal execution error when it is given a negative argument. With checking turned off, it returns the negative of the square root of the absolute value of the argument. Because of this, some of the program versions developed at UCI that appeared to have passed the acceptance procedure had not. Although they produced correct outputs, these versions generated calls to the square root function with small negative arguments in doing so. This was discovered after the programmers had been told that their programs were accepted but before lifetime testing began.

Acceptance testing was repeated on a Prime 750 at UVA. We replaced calls to the square root function that caused execution failures during acceptance testing by calls to a function that would return zero as the value of the square root for small negative arguments. Thus the lifetime-simulation failures that would have resulted from these calls if the version had been used as originally submitted were *not* counted in the failure data. Failures that later resulted from calls that did not cause trouble during acceptance testing were included in the failure data (see Faults 11.1, 20.2, 22.1, 22.2, 22.3, 26.6, and 26.7 described in Section V) although it is possible that the programmers would have corrected these calls as well if they had been presented with the results of the acceptance testing.

This is an important event in the performance of this experiment. It illustrates the problem that faults in the environment in which a program is developed may mask faults that would otherwise be discovered during the development process. This is a situation that might well arise in any form of software development. It is especially serious in the development of multi-version software because undiscovered faults in the environment may be responsible for faults that cause coincident failures. This experience supports very strongly the approach of using different development environments for the different program versions.

The second general problem involved real number comparisons. Most of the launch conditions (see Appendix A) require the computation of a real number. This number is then compared with a fixed parameter value, and the result is a Boolean value based on the comparison. For example, launch condition 2 requires that the program determine whether there exists a set of three consecutive data points that can be contained within or on a circle of radius ‘‘RADIUS1’’. In determining whether the condition is satisfied, the radius of the smallest circle containing a set of three points can be calculated and then compared to ‘‘RADIUS1’’.

Since finite precision arithmetic is being used and computations are sequence dependent, there exist situations in which two computed real numbers are acceptably close to the exact value, but do not have the same order relationship with the parameter. For example, one version might compute for the radius of the circle containing three points a value that is slightly *less* than ‘‘RADIUS1’’ and another, using a *different* algorithm, might compute a value that is slightly *greater* than ‘‘RADIUS1’’. This implies that either Boolean value for launch condition 2 is an

acceptable output for this test case.

It is known that multiple-version software is inappropriate for applications in which multiple correct solutions exist [.Anderson.]. The versions may arrive at different correct solutions and be unable to agree on a solution even though a majority compute correct results. For an application with Boolean results, the existence of multiple correct solutions implies that either Boolean result is correct since there are only two possible solutions.

For the multi-version experiment described here it was important to know whether individual versions had computed an incorrect result so that failures could be recorded accurately. A function called REALCOMPARE was supplied to the programmers in an effort to eliminate spurious failures caused by round-off errors. We sought to eliminate situations in which two versions, arriving at essentially the same real number result, would output *opposite* Boolean results. The function performs limited-precision real comparison. It does so by comparing its two real 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. The programmers were instructed to use REALCOMPARE for all comparisons of real numbers.

In retrospect, it is clear that REALCOMPARE could *never* have succeeded in eliminating the problem it was designed to solve. The effect of the function was merely to change slightly the boundary between the regions of the input space for which the two Boolean results are to be returned. For example, for launch condition 2 the value ‘‘true’’ is to be returned when the computed radius is less than or equal to ‘‘RADIUS1’’ However, REALCOMPARE will return LT or EQ whenever the computed radius is less than (‘‘RADIUS1’’ + 0.000005 * ‘‘RADIUS1’’). The boundary between the regions for which the launch condition is satisfied has been moved by (0.000005 * ‘‘RADIUS1’’), but the same multiple solution problem *still exists*. It is possible that two programs computing nearly identical values for the radius will still output opposite Boolean results for the launch condition.

No alternative implementation of REALCOMPARE could solve the multiple correct output problem without restructuring the application. If it is important to determine whether the real values produced by two versions are acceptably close to each other, then it is necessary to apply a REALCOMPARE function *directly to the real values*.

It is *not possible* to compare each of the two values independently to a third, producing Boolean results, and then to determine whether the original two values are approximately equal based on the Boolean results.

Not only did the function REALCOMPARE fail to accomplish its intended purpose, but its use led to the introduction of some extremely subtle ambiguities in the program specification that were not recognized until the analysis of faults was undertaken. An example occurs in launch conditions 3 and 10. Both of these launch conditions require the determination of whether the angle formed by three points satisfies either of the conditions

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

or

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

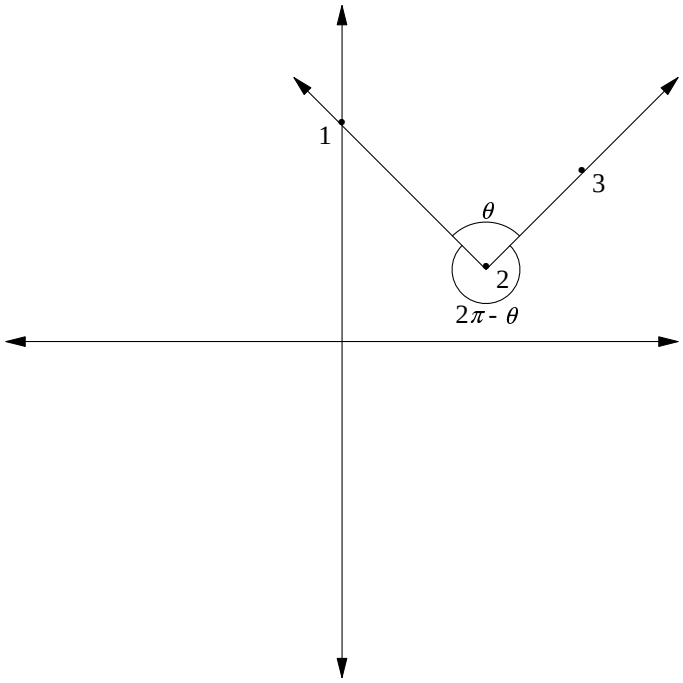
where ε is a parameter supplied as input. The specification indicates that the second of the three points is the vertex. However, as is illustrated in Figure 1(a), there is still an ambiguity as to the angle to be measured. Either the angle marked θ or the angle marked $2\pi - \theta$ could be considered. In absolute terms it makes no difference which angle is measured. Figure 1(b) illustrates that the smaller of the two possible angles is less than $(\pi - \varepsilon)$ if and only if the larger angle is greater than $(\pi + \varepsilon)$. However, recall that the tolerance used by REALCOMPARE depends on the *size of its arguments*. Thus, occasionally, it returns EQ for the larger pair when it returns LT for the smaller pair[1]. Since the problem only occurs on borderline situations, *either* result is acceptable to the application. Thus a single version program would have no difficulty. Also, algebraicly, there is *no* ambiguity. However, on real computers, there are multiple *correct* solutions and this leads to difficulties for multi-version systems.

Other more subtle ambiguities in the requirement also resulted from the failure to specify exactly the way in which REALCOMPARE was to be applied[2]. The requirement that the REALCOMPARE function be applied in all real number comparisons led to its inappropriate application in some flow control situations, with sometimes

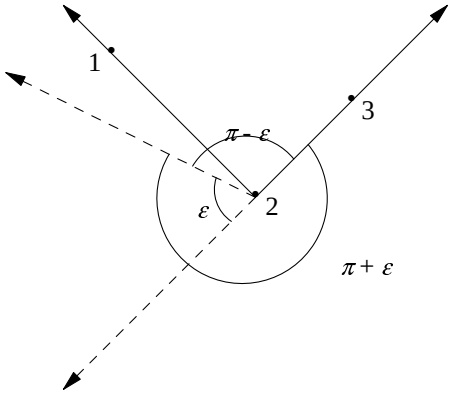
---

[1] This ambiguity in launch conditions 3 and 10 was responsible for Faults 1.1, 18.1, and 23.2.

[2] Faults 7.1, 12.2, 14.2, 17.1, 17.2, and 26.3 all arose from the failure of the requirement to explain that for launch conditions 3 and 10 it is necessary to apply REALCOMPARE to the angles themselves rather than to their *cosines* or *sines*. Fault 3.4 arose when the use of REALCOMPARE in a test of collinearity resulted in insufficient accuracy.



(a)



(b)

Figure 1. The Angle Formed by Three Points

disastrous results[3].

[3] Faults 13.1, 16.2, 21.2, and 23.1 all resulted from inappropriate applications of the REALCOMPARE function.

This problem with real number comparison is unique to multi-version software. It was ‘‘obvious’’ to the authors that approximate comparison was needed for this application but we failed to perceive the subtlety of its use. The problem is unusual in that, although it is an implementation issue, it has a major effect on the way in which the requirements specification should be written for multi-version software.

IV STATISTICAL ANALYSIS OF THE FAILURES

The preliminary results of this experiment revealed that the individual versions were highly reliable, but that the number of test cases on which multiple failures occurred was relatively high. Table 1 shows the observed failure rates of the twenty-seven versions. Table 2 shows the number of test cases in which more than one version failed on the same input.

The faults found in each of the twenty-seven program versions and the number of failures attributable to each fault are shown in Table 3. Failures associated with more than one fault are counted in the number of failures for

Table 1. Version Failure Data

Table 2. Occurrences of Multiple Failures

Table 3. Fault Occurrence Rates

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 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 3. For versions 22 and 26, the original failures were caused by calls to the square root function; it was not considered worthwhile to identify the particular call that was responsible. These original failures are not counted in the any of failure rates shown in Table 3.

In order to determine which faults caused statistically related 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, element i, j of C represents the number of test cases in which the two program versions containing faults i and j both failed. Clearly C is symmetric, and Table 3 is just its diagonal.

For each non-zero off-diagonal entry in C, an approximate chi-square test [.GUE65.] 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 formula: (* Sue fix this in terms of C *)

$$y = \sum_{i=1}^{i=2} \sum_{j=1}^{j=2} \frac{(x_{ij} - n\hat{p}_{i*}\hat{p}_{*j})^2}{n\hat{p}_{i*}\hat{p}_{*j}}$$

where

$$n = total\ number\ of\ test\ cases = 1,000,000$$

$$\hat{p}_{i*} = \frac{x_{i1} + x_{i2}}{n}$$

$$\hat{p}_{*j} = \frac{x_{1j} + x_{2j}}{n}$$

and where x_{11} represents the number of test cases for which both versions failed; x_{12} represents the number of test cases on which version i failed when version j did not; x_{21} represents the number of test cases on which version j failed when version i did not; and x_{22} represents the number of test cases for which neither version failed.

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 4. An 'R' in Table 4 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 x_{11} is ''sufficiently large'', and values greater than or equal to five are generally considered to give satisfactory results. A '?' entry in Table 4 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 x_{11} 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 pairs of the faults found in the experiment are statistically correlated. 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. The results of these statistical tests show that many faults were involved in the coincident failures, and suggest which faults were responsible.

Table 4.a. Results of Hypothesis Tests

Table 4.b. Results of Hypothesis Tests

Table 4.c. Results of Hypothesis Tests

Table 4.d. Results of Hypothesis Tests

V DESCRIPTION OF FAULTS

Once the faults causing statistically correlated failures were identified, it was possible to examine them to determine if they had any unique characteristics. This section gives a short description of each fault.

Fault 1.1 occurs in the evaluation of launch condition 10. The gold version always computes an angle between 0 and π but this version sometimes computes an angle between π and 2π. As explained in Section III, 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 π radians even though the angle is π *only* if point 2 lies *between* points 1 and 3; otherwise the angle is 0.

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. Therefore an angle that is only close to π may be calculated to be exactly π using Version 3's algorithm.

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

```
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

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 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 π, 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 precison, 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 (π-ε) for this case is 0.0027317, the gold version does not consider the launch condition to be satisfied. Version 7, however, found the angle to be zero, which is sufficiently different from the value of (π-ε) value 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 to see 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 π 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 $D \geq (2 * H)$ 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 cases in which the absolute value of

the argument is less that 10E-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 π 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:

```
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

```
(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 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 π.

Fault 16.2 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 π and 2π rather than the angle between 0 and π 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 π. However, the tangent is also zero if the angle is zero. Where the correct angle is zero, the incorrect value π 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 due to a misapplication of the REALCOMPARE function similar to those causing 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 π and 2π rather than an angle between 0 and π. 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 π, 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.

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

```
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

```
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

longest side. The case in which (*blen* ≥ *clen* ≥ *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.

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

One 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 logical relationships, if any, exist among faults that are statistically correlated.

We define faults to be *logically equivalent* 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 the intentions of the various programmers, and are necessarily subjective.

Initially, we hypothesized that faults that are statistically correlated would be logically equivalent, and vice versa. Indeed, this hypothesis does seem to explain some of the observed statistical correlations. For example, faults 3.1, 3.2, 8.1, 8.2, 20.1, 25.1, and 25.2 all involve the calculation of the angle formed by three points as required by launch conditions 3 and 10. The faults all involve mishandling of the case in which the three points are collinear, and the angle formed is zero.

However, there are faults that we classify as logically equivalent but which *do not cause 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 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 the angle satisfying launch condition 3 or 10 is near zero.

The hypothesis also fails to explain some statistical correlations among failures. For example, fault 14.1 is statistically correlated with faults 3.2, 8.2, 20.1, and 25.1. As explained in Section V, fault 14.1 is the use of an

incorrect subscript in a call to a function. The function determines whether a point coincides with either of two others, and this has no obvious similarity with faults 3.2, 8.2, 20.1, and 25.1 which are the incorrect treatment of the angle subtended by collinear points. Collinear points are assumed to subtend an angle of π radians when they may also subtend an angle of zero.

Clearly this initial hypothesis does not explain all of the observed statistical correlations. We propose, therefore, a second hypothesis to explain these additional effects. We define two faults to be *input-related* if they cause the incorrect handling of the same inputs even though the faults are not logically related. We hypothesize that some statistical correlations are caused by input-related faults.

As an example, we note that on two test cases *eight* of the twenty-seven versions failed. On one of these test cases the failures were caused by faults 3.1, 8.1, 11.1, 20.1, 22.1, 23.1, 25.2, and 26.7. These faults are not logically equivalent, but were all triggered by a single set of collinear points. The other test case on which eight failures occurred included a set of collinear points that satisfy launch condition 10; this set of points triggered faults 3.2, 8.1, 17.2, 20.1, and 25.2. But on the same test case, a different set of nearly collinear points triggered faults 7.1, 12.2, and 14.2.

Examination of the correlations among the faults reveals that they are explained by these two hypotheses. The first hypothesis is perhaps obvious and not unexpected. It is the problem that is of most concern to those building multi-version software. The hypothesis reflects the fact that separate development does not necessarily prevent the different implementors from making the same mistake. The second hypothesis is not obvious and is in fact of far greater concern. The interpretation is that apparently unrelated faults may still cause coincident failures because the faults happen to manifest themselves on the same inputs.

VII CONCLUSION

Our primary goal in this research was to understand what types of faults lead to coincident failures. We conclude that there are at least two major causes of this type of fault.

First, programmers often make identical errors in logic. At least in this application, some parts of the problem are simply more difficult for programmers to solve correctly than others. We note, however, 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 [].

The second cause is that some points in the input space are particularly likely to trigger apparently unrelated faults. These faults often involve the handling of special cases, such as, for this application, sets of coincident or collinear points. Data sets that include a set of such points are particularly likely to result in multiple failures.

The difficulties that we experienced with real number comparisons lead us to the conclusion that it is not possible to vote on the Boolean results of such comparisons in an *N*-version system. If it is important to know whether two real results agree, then it is necessary to compare the real numbers directly. In attempting to deal with the anticipated problem of slightly different results being produced by different versions we provided a function to do approximate real comparisons. The result was to introduce situations in which there were multiple correct answers to subparts of the problem.

We were surprised that something as simple as a square root function would be implemented incorrectly in one of the environments that we used. It illustrates clearly the difficulties that environments can produce.

Our findings indicate that the correlated faults found in this experiment result from the nature of the application, from similarities in the difficulties experienced by individual programmers, and by special cases in the input space. Simple methods to reduce these correlated errors do not appear to exist. They were not caused by the use of a specific programming language or any other specific tool or method. Thus we do not expect that changing

development tools or methods, or any other simple technique, would reduce the incidence of correlated faults in multi-version software.