THE FAILURE CHARACTERISTICS OF UNBIASED FAULTS AND THEIR RELATION TO MULTI-VERSION SOFTWARE

Paul E. Ammann
John C. Knight

Computer Science Report No. TR-87-06
April 23, 1987

Submitted to: IEEE Transactions on Software Engineering

THE FAILURE CHARACTERISTICS OF UNBIASED FAULTS AND THEIR RELATION TO MULTI-VERSION SOFTWARE

Paul E. Ammann and John C. Knight

Affiliation Of Authors

Department of Computer Science

University of Virginia

Charlottesville

Virginia, 22903

Financial Acknowledgement

This work was supported by the National Aeronautics and Space Administration under grant number NAG-1-605.

Index Terms

N-version programming, multiversion programming, fault tolerant software, independent failures, error seeding, software reliability.

Address For Correspondence

John C. Knight

Department of Computer Science

University of Virginia

Charlottesville

Virginia, 22903

Abstract

Coincident failure of several versions can cause a multi-version software system to fail to tolerate the associated faults. Multiple programs may contain faults that cause coincident failures because the programmers made similar mistakes or because the specification supplied to all programmers contained mistakes. The introduction of such faults is *biased* by the application. The use of a separate development environment, in which the teams producing the programs are kept in mutual isolation, aims to reduce or eliminate such faults. This paper investigates a special class of faults termed *unbiased* faults. The term *unbiased* refers to the relation of a fault to the specific application. The introduction of unbiased faults into software does not depend upon knowledge of the application. We describe an experiment in which unbiased faults were introduced into multiple programs written to identical specifications. The faults were introduced mechanically and so there was no possibility of any similarity in their form due to programmer thought processes or inadequacies in the specification. We found that subsets of unbiased faults caused programs to fail coincidentally. The results indicate that faults in multiple programs written to the same specification are likely to cause coincident failures even if the separate development process is successful in isolating the programming efforts.

I INTRODUCTION

N-version programming [2] is an approach to building fault-tolerant software in which several (i.e. *N*) programs, or versions, are separately developed, usually from the same specification. In operation, they execute in parallel and submit their outputs to a voter. The voter selects the majority output as the system output. Provided the voter is correct, such a system can tolerate software faults if the majority computes the correct output.

The appeal of *N*-version programming is the possibility of using it to improve reliability for crucial applications such as avionics, nuclear reactor control, and life support systems. Assuming a correct voter, the increase in reliability of an *N*-version system over a single version depends upon the extent to which a majority of the versions fail coincidentally.

Separate development is intended to reduce the sharing of (mis)understanding of the application, and thereby, it is hoped, the incidence of coincident failures. Although experiments have shown that separately developed programs do not necessarily fail in a statistically independent way [4, 9], empirical studies have shown that multiple-version systems can provide a decrease in failure probability over single version systems [6, 3, 12, 10]. Theoretical models predict the extent of the possible improvement [7]. Thus, at least some failures in *N*-version systems are due to faults which cause more coincident failures than would be expected by chance. It has been hypothesized that faults in two different versions of an application might cause coincident failures because the faults are associated with the same difficult part of the application [5].

In contrast, certain kinds of faults are introduced into software during its development as a result of the process of programming rather than in response to a specific challenge posed by the application. For example, in preparing a particular program, a programmer may inadvertently misorder certain steps in a computation despite being well aware of the correct order or perhaps

even type a predicate incorrectly while entering the source text for a conditional expression. We term faults of this type *unbiased*; that is, unbiased faults are the product of the mechanics of programming in general and do not result from an influence of the semantics of the application, such as a difficulty or misunderstanding on the part of the programmer. Similarly, we define *biased* faults to be faults introduced because of some influence of the application, such as a misunderstanding. The kinds of faults that can be detected at compile time in strongly typed languages tend to be unbiased faults.

In the context of *N*-version programming, concern has focused on biased faults. The separate development strategy is the manifestation of such concern. Clearly, although biased faults require attention, *N*-version programming contains the implied hypothesis that unbiased faults will not cause statistically significant numbers of coincident failures. That is, unbiased faults cause independent failures. If unbiased faults do result in more coincident failures than might be expected by chance, then even versions produced in an ideal separate development environment would be expected to fail coincidentally more often than an independent failures model predicts. In this paper we describe an experiment which examines the failure characteristics of mechanically generated unbiased faults.

The paper is organized as follows. Section 2 provides the definition of independent failures that we use to evaluate the unbiased faults. Section 3 describes the the structure of the experiment. Section 4 presents results, and section 5 gives conclusions.

II DEFINITION OF INDEPENDENCE

Before examining the performance of programs, it is necessary to have a clear definition of what is meant by independence. In principle, the performance of programs on specific inputs is all that matters. Which fault is responsible for failure is irrelevant if there is more than one fault.

However, in the experiment described in this paper, we are concerned with a particular type of fault, and we need to examine the performance of programs that contain this kind of fault. Thus, we are concerned with which fault caused failure as well as the existence of the failure.

We adopt a statistical definition of independence for failures of two programs. Informally, if two programs fail on the same input significantly more often than would be expected given their individual probabilities of failure then they are considered to contain faults which cause dependent failure. How such faults were introduced does not affect the *operational* view of the programs. Whether two programs tend to fail on the same inputs for one reason rather than another is not relevant to their operational performance.* Formally, the approach used here is to form a hypothesis that two programs fail independently and then compute a statistic that has a χ^2 distribution under the assumption of independence. We reject the hypothesis if the statistic exceeds the 99.5% point in the χ^2 distribution. Thus, in the event that the hypothesis is rejected, we are using a 99.5% confidence level that we are correct in that rejection. The independence hypothesis is formed and tested for all possible pairs of programs.

III EXPERIMENT DESCRIPTION

In a previous experiment [9], 27 Pascal programs were written separately by different programmers to a single requirements specification. Thus all 27 are intended to perform the same function, the processing of radar data in a simple antimissile system. As part of the previous experiment, the programs have been subjected to one million tests, and a great deal is known about the indigenous faults present in the programs. Details of the application, the backgrounds of the programmers, and more information about the testing and its results are available in [9] and [5].

* We note that other authors use different definitions of independence; see, for example, [2] p. 1493.

Version	Source Lines	Indigenous Faults	Failure Probability	for statements	if statements	assignment statements
1	855	1	0.000002	29	16	186
2	327	0	0.000000	9	21	108
3	764	4	0.002297	8	35	295
4	815	0	0.000000	20	22	126
5	950	0	0.000000	10	9	155
6	643	3	0.001149	7	23	216
7	654	1	0.000071	10	64	257
8	598	2	0.000323	4	39	213
9	1004	2	0.000053	7	17	161
10	837	0	0.000000	21	21	126
11	604	2	0.000427	5	22	218
12	502	1	0.000004	8	16	157
13	604	2	0.001368	10	43	159
14	520	0	0.000000	7	16	147
15	967	2	0.000062	25	22	199
16	488	2	0.000269	25	17	120
17	395	1	0.000115	4	28	141

Figure 1. Program Characteristics.

In a subsequent fault-seeding* experiment [11, 1], 17 of the 27 programs were selected at random, faults were seeded into all 17, and the resulting programs were tested. Figure 1 presents some characteristics of the 17 selected programs. The algorithms used for seeding faults were simple syntactic manipulations: 2 algorithms modified the bounds on **for** statements, 3 algorithms modified the Boolean expression in **if** statments, and 1 algorithm deleted assignment statements. The particular algorithms are shown in figure 2. The seeded faults were introduced mechanically

*_Error seeding_ is the term used by the originator of the concept [11].

without using any semantic knowledge of the programs. To introduce one seeded fault, a syntactic structure was selected at random and the seeding algorithm was applied. The seeded faults are, therefore, unbiased according to the definition given in section 1. Each of these fault-seeding algorithms was applied 4 times to each of the 17 programs for a total of 408 modified programs, each containing one seeded fault.

For the experiment described in this paper, 45 of the programs containing mechanically seeded faults were selected from the 408 seeded programs available. These programs were selected because they satisfied an acceptance procedure that required programs to fail, on average, on less than one in 200 test cases. This acceptance procedure is similar to the one used in [9]. All of the indigenous faults in the versions* were corrected before the seeded faults were installed in the programs, and only one seeded fault was installed in a program at a time. Thus

Grammar Rule	Syntactic Alteration
for_stmt ::=	1. *expr1* -> *expr1* + 1
for *var* := *expr1* **to** *expr2* **do** *stmt*	2. *expr2* -> *expr2* - 1
if_stmt ::=	3. *condition* -> true
if *condition* **then** *stmt* **else** *stmt*	4. *condition* -> false
	5. *condition* -> not *condition*
assignment_stmt ::=	6. *statement deleted*
var := *expr*	

Figure 2. Seeding Algorithms.

*Since the test cases used in this experiment are a subset of those used in [9], all of the indigenous faults triggered by these test cases are assumed to be known.

each failure of a version is due to precisely one seeded fault.

Once the programs with seeded faults had been prepared, they were tested. The test cases were randomly generated and correctness was determined by comparison with a *gold* program. Each program was executed on a total of 45,000 test cases that were a subset of the one million randomly generated test cases used in [9].

IV EXPERIMENT RESULTS

Figure 3 summarizes the results of executing the 45,000 test cases on the programs containing seeded faults. The figure is a 45 by 45 symmetric matrix, the upper half of which is shown. The (i, j) entry in the matrix describes failure on common test cases between program i and program j.

The programs are divided into three categories according to the type of seeded fault used to generate the program. The divisions are shown by lines on the figure. The first 13 programs contain **for** statement seeded faults, the next 29 contain **if** statement seeded faults, and the last 3 programs contain assignment statement seeded faults.

The programs are ordered within the divisions by type so as to make the nonzero elements cluster near the diagonal. The reordering results in "blocks" of entries on the diagonal. Each block indicates that the corresponding programs all fail together on a certain subset of test cases. Note that the reordering has no effect on the entries in the matrix; only the visual appearance has been altered. The reordering shows that common failures can be grouped together.

For every combination of two programs, a χ^2 test was used to test the hypothesis that the two programs fail independently. The confidence level used for the χ^2 test is 99.5%. The results of these tests appear in Figure 3. On the figure, an "R" indicates that the independence

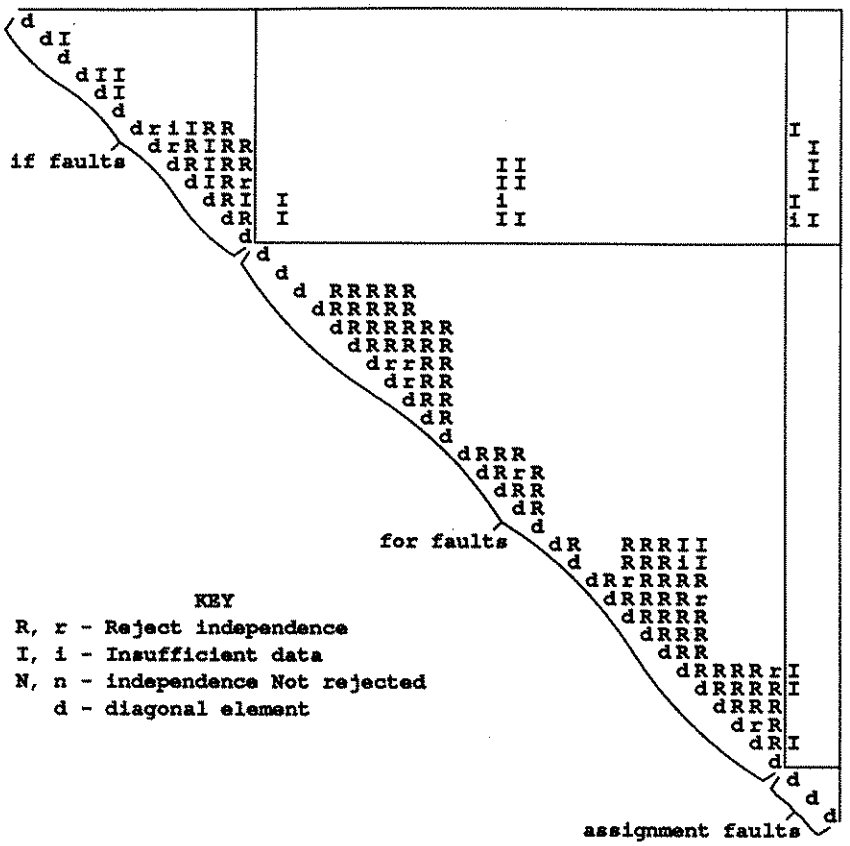

Figure 3: Results of Independence Hypothesis Tests.

hypothesis is rejected. An "I" indicates that insufficient data existed to make the χ^2 test meaningful (at least 1 but less than 5 common failures). An "N" would indicate that the independence hypothesis is not rejected. Although the figure does not contain any "N" entries, blank entries, which represent zero common failures, may be interpreted as likely "N" entries.

Since all 45 of the seeded programs were produced by seeding faults into 17 original programs, there are entries in the matrix corresponding to two distinct seeded faults being evaluated in the same base program. This situation does not accurately reflect a scenario that will occur in the separate development of multiple versions, and these entries are distinguished by the

lower case letters "r", "i", and "n", respectively. These data are included for completeness and show that different unbiased faults in the *same* program may cause dependent failures.

V CONCLUSION

An examination of figure 3 shows that, for a given category of seeded faults, statistical dependence among faults is common. If the faults were independent, at most 5 of the 990 possible hypothesis tests* would be expected to fail at a confidence level of 99.5%. In fact, as can be seen from figure 3, 96 of the hypothesis tests failed.** We conclude that unbiased faults do not, in general, cause independent failures.

All of the hypotheses that were formally rejected were between pairs of faults in the same category. Although there were coincident failures among different categories of seeded faults, there was insufficient data to permit the use of the χ^2 hypothesis test and so no statistical dependence was demonstrated.

Informally, unbiased faults are the "best" faults that can be hoped for in a separate development environment since they model program defects that are free from influence propagated among the development teams. Although the assumption that unbiased faults cause independent failures may seem quite reasonable, this experiment has shown that the assumption is false. We hypothesize that the reason for this effect is that, although the versions are developed separately by separate teams, the various versions are solving the *same* problem using basically the *same* programming "primitives", i.e. sequence, selection, and iteration. The misuse of similar programming primitives results in programs with similar faults.

---

*Of the 990 possible hypothesis tests, 924 are between seeded programs derived from distinct original programs and the remaining 66 are from seeded programs derived from the same original program.

**The value 96 counts both "R" and "r" entries from figure 3. The count of failed hypothesis tests between seeded programs derived from distinct original programs is 85, leaving 11 failed hypothesis tests from seeded programs derived from the same original program.

The implication for *N*-version programming is that these "best" faults may produce coincident failures even if the specification is error free and the separate development environment is entirely successful in isolating the programming efforts. It is clear that *N*-version system development and analysis must ensure that this source of coincident failures is taken into account.

ACKNOWLEDGEMENTS

It is a pleasure to thank Earl Migneault and Susan Brilliant for their comments on the experiment described in this paper and its results. This work was supported in part by NASA grant number NAG-1-605.

REFERENCES

[1] P.E. Ammann, and J.C. Knight, "An Experimental Evaluation of Simple Methods For Seeding Program Errors", Submitted to: *IEEE Transactions on Software Engineering*, July, 1986.

[2] A. Avizienis "The *N*-Version Approach to Fault Tolerant Software", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12 (December 1985).

[3] A. Avizienis, J.P.J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments", *IEEE Computer*, Vol. 17, No. 8 (August 1984).

[4] S.S. Brilliant, "Analysis of Faults in a Multi-Version Software Experiment", MS Thesis, University of Virginia (May, 1985)

[5] S.S. Brilliant, J.C. Knight, and N.G. Leveson, "A Fundamental Difficulty with N-Version Programming", Submitted to *IEEE Transactions on Software Engineering*, July, 1986.

[6] L. Chen, and A. Avizienis, "N-version programming: A Fault Tolerance Approach to Reliability of Software Operation", *Digest of Papers FTCS-8: Eighth Annual Symposium on Fault Tolerant Computing*, Toulouse, France, pp. 3-9, June 1978.

[7] 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*, Vol. SE-11, No. 12 (December 1985).

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

[9] J.C. Knight, and N.G. Leveson, "A Large Scale Experiment in N-Version Programming" *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1 (January 1986).

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

[11] H.D. Mills, "On The Statistical Validation of Computer Programs", in *Software Productivity*, Little Brown, Toronto, 1983.

[12] R.K. Scott, J.W. Gault, D.F. McAllister, and J. Wiggs, "Experimental Validation of Six Fault-Tolerant Software Reliability Models", *Proceedings IEEE Conference on Fault Tolerant Computing*, 1984, pp. 102-107.