A LARGE SCALE EXPERIMENT IN N-VERSION PROGRAMMING*

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

Lois D. St.Jean
Department of Computer Science
University of Virginia
Charlottesville, Virginia, 22903

ABSTRACT

N-version programming has been proposed as a method of incorporating fault tolerance into software. Multiple versions of a program (i.e. "N") are prepared and executed in parallel. Their outputs are collected and examined by a voter, and, if they are not identical, it is assumed that the majority is correct. This method depends for its reliability improvement on the assumption that programs that have been developed independently will fail independently. In this paper an experiment is described in which the fundamental axiom is tested.

A total of twenty seven versions of a program were prepared independently from the same specification and then subjected to one million tests. The results of the tests revealed that the programs were individually extremely reliable but that the number of tests in which more than one program failed was substantially more than expected. A statistical test of independence of the versions was applied and the hypothesis of independence was rejected.

The conclusion from this result is that N-version programming must be used with care and that analysis of its reliability must include the effect of dependent errors.

1. INTRODUCTION

N-version programming[1] has been proposed as a method of providing fault tolerance in software. The approach requires the independent preparation of several (i.e. "N") versions of a piece of software for some application, usually from the same requirements specifications (although, see Kelly[2]). These versions are executed in parallel in the application environment; each receives identical inputs and each produces its version of the required outputs. The outputs are collected by a voter and, in principle, they should all be the same. In practice there may be some disagreement. If this occurs, the results of the majority (assuming there is one) are assumed to be the correct output and this is the one used by the system.

N-version programming is faced with several practical difficulties in its implementation such as isolation of the versions and design of voting algorithms. These difficulties have been summarized comprehensively by Anderson and Lee[3] and will not be discussed here.

*This work was sponsored in part by NASA grant number NAG1-242 and in part by a MICRO grant cofunded by the state of California and Hughes Aircraft Company. It has been cleared for publication by the sponsoring organizations.

The great benefit that N-version programming is intended to provide is a substantial improvement in reliability. It is assumed in the analysis of the technique that the N different versions will fail *independently*; that is, faults in the different versions occur at random and are unrelated. Thus the probability of two or more versions failing on the same input is very small. Under this assumption, the probability of failure of an N-version system, to a first approximation, is proportional to the N'th power of the probability of failure of the independent versions. If the assumption is true, systems with extremely high levels of reliability could be built with components that are individually of only average quality.

We are concerned that this assumption might be *false*. Our intuition indicates that when solving a difficult intellectual problem (such as writing a computer program), people tend to make the same mistakes even when they are working independently. Some parts of a problem may be inherently more difficult than others. In this experiment, the subjects were asked in a questionnaire to state the parts of the problem that caused them the most difficulty. The responses were surprisingly similar.

If the assumption of independence is not born out in practice, it would cause the analysis to overestimate the reliability of an N-version system. This could be an important practical problem since N-version programming is being used in existing crucial systems[4] and is planned for others.

It is interesting to note that, even in mechanical systems where redundancy is an important technique for achieving fault tolerance, common *design* errors are a source of serious problems. An aircraft crashed recently because of a common vibration mode that adversely affected all three parts of a triply redundant system[5]. Common Failure Mode Analysis is used in critical hardware systems in an attempt to determine and minimize common failure modes.

To test this underlying assumption of independence, we have carried out a large scale experiment in N-version programming. A *statistically rigorous* test of independence was the major goal of the experiment and all of the design decisions that were taken were dominated by this goal.

In section two we describe the experiment itself. The preliminary results obtained so far are described in section three and section four is a discussion of the issues in the experiment. Our conclusions are presented in section five.

0731-3071/85/0000/0135$01.00 © 1985 IEEE

2. DESCRIPTION OF EXPERIMENT

In graduate and senior level classes in computer science at the University of Virginia (UVA) and the University of California at Irvine (UCI), students were asked to write programs from a single requirements specification. The result was a total of twenty seven programs (nine from UVA and eighteen from UCI) all of which should theoretically produce the same output from the same input. Each of these programs was then subjected to one million randomly-generated test cases.

In order to make the experiment realistic, an attempt was made to choose an application that would normally be a candidate for the inclusion of fault tolerance. The problem that was selected for programming is a simple (but realistic) anti-missile system that came originally from an aerospace company. The program is required to read some data that represents radar reflections and, using a collection of conditions, has to decide whether the reflections come from an object that is a threat or otherwise. If the decision is made that the object is a threat, a signal to launch an interceptor has to be generated. The problem is known as the "launch interceptor" problem and the various conditions upon which the decision depends are referred to as "launch interceptor conditions" (LIC's). The conditions are heavily parameterized. For example, one condition asks whether a set of reflections can be contained within a circle of given radius; the radius is a parameter.

The problem has been used in other software engineering experiments[6]. It has also been used in a study of software reliability that was carried out by the Research Triangle Institute (RTI) for NASA Langley Research Center. We chose this problem because of its suitability and because we were able to use the lessons learned in the NASA/RTI experiment to modify our own experiment. RTI had prepared a requirements specification and had experienced some difficulties with unexpected ambiguities and similar problems. We were able to rewrite the requirements specification in the light of this experience. Thus the requirements specification had been carefully "debugged" prior to use in this experiment.

The requirements specification was given to the students and they were asked to prepare software to comply with it. No overall software development methodology was imposed on them. They were required to write the program in Pascal and to use only a specified compiler and associated operating system. At UVA these were the University of Hull V-mode Pascal compiler for the Prime computers using PRIMOS, and at UCI these were the Berkeley PC compiler for the VAX 11/750 using UNIX. No other software tools were permitted.

The students were given a brief explanation of the goals of the experiment and the principles of N-version programming. The need for independent development was stressed and students were carefully instructed not to discuss the project amongst themselves. However, we did not impose any restriction on their reference sources. Since the application requires some knowledge of geometry, it was expected that the students would consult reference texts and perhaps mathematicians in order to develop the necessary algorithms. We felt that the possibility of two students using the same reference material was no different from two separate organizations using the same reference sources in a commercial development environment.

As would be expected during development, questions arose about the meaning of the requirements. In order to prevent any possibility of information being inadvertently transmitted by an informal verbal response, these few questions were submitted and answered by electronic mail. If a question revealed a general flaw in the specifications, the response was broadcast to all the programmers.

Each student was supplied with twelve input data sets and the expected outputs for use in debugging. Once a program was debugged using these tests and any other tests the student developed, it was subjected to an acceptance test. The acceptance test was a set of two hundred randomly-generated test cases; a different set of two hundred tests were generated for each program. This procedure was used to prevent a general "filtering" of common faults by the use of a common acceptance test. An acceptance test was used since it was felt that in a real software production environment potential programs would be submitted to extensive testing and would not be used unless they demonstrated a high level of reliability. Once a program passed its acceptance test, it was considered complete and was entered into the collection of versions.

One result of the earlier NASA/RTI experiment was some difficulty with machine precision differences between versions. Although two programs computed what amounted to the same result, different orders of computation yielded minor differences which gave the impression that one or more versions had failed. To prevent this, all programmers in this experiment were supplied with a function to perform comparison of real quantities with limited precision. The programmers were instructed to use this supplied function for all real-number comparisons.

Once all the versions had passed their acceptance tests, the versions were subjected to the experimental treatment which consisted of simulation of a production environment. A test driver was built which generated random radar reflections and random values for all the parameters in the problem. All twenty seven programs were executed on these test cases and the determination of success was made by comparing their output with a twenty-eighth version, referred to as the *gold* program. This program was originally written in FORTRAN for the NASA/RTI experiment and was rewritten in Pascal for this experiment. As part of the NASA/RTI experiment, the gold program has been subjected to several million test cases and we have considerable confidence in its accuracy. It was also subjected to an extensive structured walkthrough at UVA after translation to Pascal.

A total of one million tests were run on the twenty seven versions written for this experiment and the gold program. Although testing was not continuous on any of the machines, a total of fifteen computers were used in performing these tests between May and September of 1984; five Primes and a dual processor CDC Cyber 730 at UVA, and seven VAX 11/750's and two CDC Cyber 170's at NASA Langley Research Center.

3. EXPERIMENTAL RESULTS

The quality of the programs written for this experiment is remarkably high. Table 1 shows the observed failure rates of the twenty seven versions. Of the twenty seven, no failures were recorded by six versions and the remainder were successful on more than 99% of the tests.

Table 1 – Version Failure Data

Version	Failures	Reliability	Version	Failures	Reliability
1	2	0.999998	15	0	1.000000
2	0	1.000000	16	62	0.999938
3	2297	0.997703	17	269	0.999731
4	0	1.000000	18	115	0.999885
5	0	1.000000	19	264	0.999736
6	1149	0.998851	20	936	0.999064
7	71	0.999929	21	92	0.999908
8	323	0.999677	22	9656	0.990344
9	53	0.999947	23	80	0.999920
10	0	1.000000	24	260	0.999740
11	554	0.999446	25	97	0.999903
12	427	0.999573	26	883	0.999117
13	4	0.999996	27	0	1.000000
14	1368	0.998632			

Table 2 shows the number of test cases in which more than one version failed on the same input. We find it surprising that test cases occurred in which eight of the twenty seven versions failed.

Where multiple failure occurred on the same input, it is natural to suspect that the failures occurred in the versions supplied by only one of the universities involved. It might be argued that students at the same university have a similar background and that this would tend to cause dependencies. However, the exact opposite has been found. Table 3 shows a correlation matrix of common failures between the versions supplied by the two universities. For table 3, and for table 1, versions numbered 1 through 9 came from UVA and versions numbered 10 through 27 came from UCI. A table 3 entry at location i, j shows the number of times versions i and j failed on the same input. In table 3, the rows are labeled with UCI version numbers and the columns with UVA version numbers. Thus, a non-zero table entry show the number of common failures experienced by a UVA version and a UCI version. In the preliminary analysis of common faults, *all* were found to involve versions from both schools.

Table 2 – Occurrences of Multiple Failures

Number	Probability	Occurrences
2	0.00055100	551
3	0.00034300	343
4	0.00024200	242
5	0.00007300	73
6	0.00003200	32
7	0.00001200	12
8	0.00000200	2

Table 3 – Correlated Failures Between UVA And UCI

		UVA Versions								
		1	2	3	4	5	6	7	8	9
UCI Versions	10	0	0	0	0	0	0	0	0	0
	11	0	0	58	0	0	2	1	58	0
	12	0	0	1	0	0	0	71	1	0
	13	0	0	0	0	0	0	0	0	0
	14	0	0	28	0	0	3	71	26	0
	15	0	0	0	0	0	0	0	0	0
	16	0	0	0	0	0	1	0	0	0
	17	2	0	95	0	0	0	1	29	0
	18	0	0	2	0	0	1	0	0	0
	19	0	0	1	0	0	0	0	1	0
	20	0	0	325	0	0	3	2	323	0
	21	0	0	0	0	0	0	0	0	0
	22	0	0	52	0	0	15	0	36	2
	23	0	0	72	0	0	0	0	71	0
	24	0	0	0	0	0	0	0	0	0
	25	0	0	94	0	0	0	1	94	0
	26	0	0	115	0	0	5	0	110	0
	27	0	0	0	0	0	0	0	0	0

4. MODEL OF INDEPENDENCE

Separate versions of a program may fail on the same input even if they are independent. Indeed, if they did not they would be dependent. We base our probabilistic model for this experiment on the statistical definition of independence:

> Two events, A and B, are independent if pr(A|B) = pr(A) and pr(B|A) = pr(B). Intuitively, A and B are independent if knowledge of the occurrence of A in no way influences the occurrence of B, and vice versa.

The null hypothesis that we wish to test is derived from this statement.

By examining the faults (i.e. the flaws in the program logic) that have been revealed by testing, we could determine whether any set of programs contain correlated faults. For this experiment we intend to do that as part of a more extensive analysis. However, from an operational viewpoint, it does not matter *why* programs fail on the same input, it merely matters that they *do*. Thus in examining the hypothesis of independence, we examine the *observed* behavior of the programs during execution. In this paper, our analysis of the hypothesis of independence is based on the results of the tests that have been carried out with no evaluation of the faults in the programs' source text.

The null hypothesis that we wish to test is that the programs fail independently. If the programs fail independently, then, given the individual probabilities of failure $p_1, p_2, ..., p_N$ for N versions, the probability that there are no failures on a given test case is:

$$P_0 = (1-p_1)(1-p_2)...(1-p_N)$$

The probability that only one version fails on a given test case

is:

$$P_1 = \frac{P_0 p_1}{1-p_1} + \frac{P_0 p_2}{1-p_2} + \cdots + \frac{P_0 p_N}{1-p_N}$$

Finally, the probability that more than one of the N versions fails on any particular test case is:

$$P_{more} = 1-P_0-P_1$$

If a total of n test cases are executed, let K be the number of times two or more versions fail on the same input data. Under the hypothesis of independent failures, the quantity K has a binomial distribution. Thus:

$$P(K=x) = \binom{n}{x} (P_{more})^x \ (1-P_{more})^{n-x}$$

$$where \ \binom{n}{x} = \frac{n!}{x!(n-x)!}$$

Since the value of n is very large, the central limit theorem can be applied and a normal approximation to this binomial distribution used in the hypothesis test. If this is done, the quantity:

$$z = \frac{K - nP_{more}}{(nP_{more}(1-P_{more}))^{1/2}}$$

has a distribution that is closely approximated by the standardized normal distribution.

For this experiment, we can estimate the probabilities of failure for the individual versions from the observed probabilities of failure shown in table 1. There were twenty seven versions (i.e. N = 27), one million tests were executed (i.e. n = 1,000,000), and the number of tests in which more than one version failed was 1255 (i.e. K = 1255). With these parameters, the statistic z has the value 100.51. This is greater than 2.33 which is the 99% point in the the standard normal distribution, and so we reject the null hypothesis with a confidence level of 99%.

5. DISCUSSION

An important problem in performing experiments at universities is obtaining programmers with a realistic experience level. An experiment of this size would be extremely expensive to undertake if professional programmers were used as the experimental subjects. Our use of students could be criticized as being unrealistic but we point out that all of the versions were written by graduate students or by seniors with high grade point averages, many of whom had returned to the university after having worked as professional programmers, and all of whom would be entering the professional programming workforce at high levels after graduation. Of the twenty seven programmers, twelve had less than two years programming experience outside their degree programs, ten had between two and five years, and five had more than five years programming experience. We note that the program written for this experiment by the most experienced real-time programmer (who has worked at the Jet Propulsion Laboratory and the Los Alamos National Laboratory) contained multiple faults in common with other programs.

It could also be argued that our results are biased by the fact that the experimental subjects came from similar backgrounds. This in fact is not the case. There is a considerable diversity of education and experience in the students backgrounds. However, the use of two geographically separate universities also contributes to the diversity amongst the subjects.

It might be argued that this experiment does not reflect realistic program development in industry and that one million test cases does not reflect very much *operational* time for programs of this type. In fact, the acceptance test is the equivalent of a very elaborate testing process for production programs of this type. Each of our test cases represents an "unusual" event seen by the radar. Most of the time the radar echoes will be identical from one scan to the next with only an occasional change due to the entry of an object into the field of view. Producing realistic unusual events to test a production tracking program is clearly an expensive undertaking and we feel that two hundred such events would indeed be a realistic number.

One million test cases (several hundred hours of computer time per version) corresponds to dealing with one million unusual cases during production use. In practice once again, these one million events will be separated by a much larger number of executions for usual events. If the program is executed once per second and unusual events occur every ten minutes, then one million tests correspond to about twenty years of operational use.

6. CONCLUSIONS

For the particular problem that was programmed for this experiment, we conclude that the assumption of independence that is fundamental to the analysis of N-version programming *does not hold.* Using a fairly simple probability model of independence, our data indicates that the hypothesis of independence has to be rejected at the 99% confidence level.

It is important to understand the meaning of this statement. First, it is conditional *on the application that we used.* The result may or may not extend to other programs, we do not know. Other experiments must be carried out to gather data similar to ours in order to be able to draw *general* conclusions. Second, the statement above does not mean that N-version programming does not work or should not be used. It means that the reliability of an N-version system *may* not be as high as theory predicts under the assumption of independence. If the implementation issues can be resolved for a particular N-version system, the required reliability might be achieved by using a larger value for N, if this is economically feasible.

Based on a preliminary analysis of the faults in the programs, we have found that approximately one half of the total software faults found involved two or more programs. This is surprisingly high and implies that either programmers make a large number of similar errors or, alternatively, that the common errors are more likely to remain after debugging and testing. Several alternative hypotheses are possible and need to be further explored. One is that certain parts of any problem are just more difficult than others and will lead to the same errors by different programmers. Thus the error distribution is an artifact of the problem itself. Another possible hypothesis is that unique (random) errors tend to be those most likely to be caught by a compiler or by testing.

Common errors may reflect inherently difficult semantic aspects of the problem or typical human misconceptions which are not easily detected through standard verification and validation efforts. Finally, common errors may reflect flaws in the requirements specification document. We do not think this is the case in this experiment since great care went into its preparation and the requirements had been debugged through use in an earlier experiment. However, we plan to conduct further experiments on comprehension of these particular requirements.

Given that common errors (as shown by this and other experiments) are possible and perhaps even likely in separately developed multiple versions of a software system, then relying on random chance to get diversity in programs and eliminate design errors may not be effective. However, this does not mean that diversity is not a possible solution to the software fault tolerance problem. What it does imply is that further research on common errors may be useful. Hardware designers do not rely on simple redundancy or independently generated diverse designs to get rid of common design errors. Instead, they use sophisticated techniques to determine common failure modes and systematically alter their designs to attempt to eliminate common failure modes or to minimize their probability. Perhaps we need equivalent techniques for software. Unfortunately, this will not be simple but perhaps a simple solution just does not exist for what is undoubtedly a very difficult problem.

ACKNOWLEDGEMENTS

It is a pleasure to acknowledge the students who wrote the versions that were tested in this experiment; P. Ammann, C. Finch, N. Fitzgerald, M. Heiss, D. Irwin, L. Lauterbach, S. Samanta, J. Watts, P. Wilson from UVA, and R. Bowles, D. Duong, P. Higgins, A. Milne, S. Musgrave, T. Nguyen, J. Peck, P. Ritter, R. Sargent, R. Schmaltz, A. Schoonhoven, T. Shimeall, G. Stoermer, J. Stolzy, D. Taback, J. Thomas, C. Thompson, L. Wong from UCI. We are also pleased to acknowledge the Academic Computer Center at the University of Virginia, the AIRLAB facility and the Central Computer Complex at NASA Langley Research Center for providing computer time to allow the programs to be tested. Much of the design of the experiment is due to Lois StJean, and Susan Brilliant and Paul Ammann were responsible for much of the testing activities. 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. This work was supported in part by NASA grant number NAG1-242, and in part by a MICRO grant cofunded by the University of California and Hughes Aircraft Company. Finally, none of this work would have been possible and this paper could not have been written without the excellent facilities provided by the ARPA and CSNET computer networks.

REFERENCES

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

(2) Kelly, J.P.J., "Specification of Fault-Tolerant Multi-Version Software: Experimental Studies of a Design Diversity Approach", *Ph.D. dissertation*, University of California, Los Angeles, 1982.

(3) Anderson T., and P.A. Lee, *Fault Tolerance: Principles and Practice*, Prentice Hall International, 1981.

(4) Martin, D.J., "Dissimilar Software In High Integrity Applications In Flight Controls", *Software for Avionics*, AGARD Conference Proceedings, No. 330, January, 1983, pp. 36-1 - 36-9.

(5) Bonnet, B., Panel Presentation, COMPCON 84, Washington D.C., September 1984.

(6) Nagel, P.M., and J.A. Skrivan, *Software Reliability: Repetitive Run Experimentation and Modeling*, prepared for National Aeronautics and Space Administration at Boeing Computer Services Company, Seattle, Washington, 1982.