AN EXPERIMENTAL EVALUATION OF THE ASSUMPTION OF INDEPENDENCE IN MULTI-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

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 at two universities 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. The results of these tests are presented along with an analysis of some of the faults that were found in the programs. Background information on the programmers used is also summarized. The conclusion from this experiment is that N-version programming must be used with care and that analysis of its reliability must include the effect of dependent errors.

Keywords and Phrases:

Multi-version programming, N-version programming, software reliability, fault-tolerant software, design diversity.

*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.

1. INTRODUCTION

Multi-version or N-version programming [1] has been proposed as a method of providing fault tolerance in software. The approach requires the separate, independent preparation of multiple (i.e. ‘‘N’’) versions of a piece of software for some application. 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 output used by the system.

Separate development can start at different points in the software development process. Since each version of the software must provide the same functional capability, there must exist some common form of system requirements document. Coordination must also exist if the versions are to provide data to the voter, especially if intermediate data is compared as well as the final output data. Obviously, all design specification must be redundant and independent for the versions to have any chance of avoiding common design faults. An interesting approach to dual specification was used by Ramamoorthy *et al.* [2] where two independent specifications were written in a formal specification language and then formal mathematical techniques used to verify consistency between the specifications before the next step in development proceeded. Thus they were able to detect specification faults by using redundancy and then repair them before the separate software versions were produced. Kelly and Avizienis [3,4] also used separate specifications for their N-version programming experiment, but the specifications were all written by the same person so independence was syntactic only (three different specification languages were used).

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 [5] and will not be discussed here.

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, system reliability could be higher than the reliability of the individual components.

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 (for example, incorrect treatment of boundary conditions) even when they are working independently. Some parts of a problem may be inherently more difficult than others. In the experiment described in this paper, 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.

It is interesting to note that, even in mechanical systems where redundancy is an important technique for achieving fault tolerance, common *design* faults 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 [6]. Common Failure Mode Analysis is used in critical hardware systems in an attempt to determine and minimize common failure modes.

If the assumption of independence is not born out in practice for an N-version software system, it would cause the analysis to overestimate the reliability. Recent work [7] has shown that even small probabilities of coincident errors cause a substantial reduction in reliability. This could be an important practical problem since N-version programming is being used in existing crucial systems and is planned for others. For instance, dual programming has been used in the slat and flap control system of the Airbus Industrie A310 aircraft [8]. The two programs are executed by different microprocessors operating

asynchronously. The outputs of the two microprocessors are compared continuously, and any difference greater than a defined threshold causes the system to disconnect after a preset time delay. On the A310, it is sufficient to know that there has been a failure as backup procedures allow the continued safe flight and landing of the aircraft. Dual programming has also been applied to point switching, signal control, and traffic control in the Gothenburg area by Swedish State Railways [9]. In the latter system, if the two programs show different results, signal lights are switched to red. Dual programming has further been proposed for safety systems in nuclear reactors. Voges, Fetsch, and Gmeiner [10] have proposed its use in the design of a reactor shutdown system which serves the purpose of detecting cooling disturbances in a fast breeder reactor and initializing automatic shutdown of the reactor in case of possible emergency. Also, both Ramamoorthy *et al.* [2] and Dahll and Lahti[11] have proposed elaborate dual development methodologies for the design of nuclear reactor safety systems.

A common argument [2,10,12] in favor of dual programming is that testing of safety-critical real-time software can be simplified by producing two versions of the software and executing them on large numbers of test cases without manual or independent verification of the correct output. The output is assumed correct as long as both versions of the programs agree. The argument is made that preparing test data and determining correct output is difficult and expensive for much real-time software. Since it is assumed ‘‘unlikely’’ that two programs will contain identical faults, a large number of test cases can be run in a relatively short time and with a large reduction in effort required for validation of test results.

In addition, it has been argued that each individual version of the software can have lower reliability than would be necessary if only one version were produced. The higher required software reliability is assumed to be obtained through the voting process*. The additional cost incurred in the development of multiple software versions would be offset by a reduction in the cost of the validation process. It has even been suggested [13] that elaborate software development environments and procedures will be unnecessary

and that mail-order software could be obtained from hobbyist programmers.

The important point to note is that all of the above arguments in favor of using redundant programming hinge on the basic assumption that the probability of common mode failures (identical incorrect output given the same input) is very low for independently developed software. Therefore, it is important to know whether this assumption is correct.

Several previous experiments have involved N-version programming, but none have focused on the issue of independence. In two [2,11] independence was assumed and therefore not tested. In each of these, the two versions developed were assumed to be correct if the two outputs from the test cases agreed and no attempt was made to verify independently the correctness of the output. Thus common errors would not necessarily have been detected. In other experiments, common errors were observed but since independence was not the hypothesis being tested, the design of the experiments make it impossible to draw any statistically valid conclusions. Kelly and Avizienis [3,4] report finding 21 related faults, one common fault was found in practical tests of the Halden nuclear reactor project [9], and Taylor [9] reports that common faults have been found in about half of the practical redundant European software systems.

In summary, although there is some negative evidence which raises doubts about the independence assumption, there has been no experiment which attempted to study this assumption in a manner in which clear evidence for or against can be drawn. Because the independence assumption is widely accepted and because of the potential importance of the issue in terms of safety, we have carried out a large scale experiment in N-version programming to study this assumption. 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.

---

*One might note that even in the hardware Triple Modular Redundancy (TMR) systems, from which the idea of N-version programming arises, overall system reliability is not improved if the individual components are not themselves sufficiently reliable [5]. In fact, incorporating redundancy into a system can actually reduce overall system reliability due to the increased number of components [14].

The experiment and its results are presented in the remainder of this paper. In section two we describe the experiment itself, and we review the backgrounds of the programmers and their activities during the experiment in section three. The results of the tests performed on the various versions are presented in section four. Section five contains a model of independence and a statistical test of the hypothesis that the model is valid. Some of the faults that have been found in the programs used in this experiment are described in section six, and various issues arising from this experiment are discussed in section seven. Our conclusions are presented in section eight, and the requirements specification used in the experiment is included as an appendix.

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 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 [15]. It has also been used in a study of N-version programming with N equal to three that was carried out at the Research Triangle Institute (RTI). We chose this problem because of its suitability and because we were able to use the lessons learned in the experiment at RTI 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.

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 fifteen 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. Different data sets were used for each program 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. Although the data was generated randomly, the test case generator was written for, and tailored to this application. Once a program passed its acceptance test, it was considered complete and was entered into the collection of versions. The acceptance test that was used represents a realistic amount of validation for this type of software as is discussed in section seven, and resulted in highly reliable programs as is shown below.

One result of the earlier experiment at RTI 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 RTI experiment and was rewritten in Pascal for this experiment. As part of the 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 gold version was used so that a larger number of test cases could be executed than would be possible if manual checking of the outputs was performed. Naturally, it is possible (but very unlikely) that a common undetected fault existed in all 28 versions, including the gold version. This would have no effect on our final results, however, and any additional undetected common faults would only strengthen our conclusion.

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. PROGRAMMER'S BACKGROUNDS

During this experiment, the programmers were asked to maintain simple work logs, and to fill in questionnaires about their backgrounds. In this section we give general information about the programmers' previous experience, education, and effort level obtained from the logs and questionnaires. This data is provided in summary form only. We choose deliberately not to associate specific background information with the individual versions in order to protect the identity of the programmers. This section is based on twenty six questionnaires. The questionnaire from one programmer could not be obtained for analysis.

Fourteen of the programmers were working on bachelors degrees and had no prior degree, eight on masters degrees and held at least a bachelors degree, and four on doctoral degrees and held at least a

masters degree. Of those who held bachelors degrees, four were in mathematics, three were in computer science, and there was one each in astronomy, biology, environmental science, management science, and physics. All of the programmers had taken a number of undergraduate courses in mathematics and computer science. Most had taken several graduate courses in computer science and some graduate mathematics courses. The number of undergraduate coursework hours varied from six to 45 in computer science, and from 12 to 45 in mathematics. The number of graduate coursework hours varied from zero to 30 in computer science, and from zero to 19 in mathematics.

The programmers' previous work experience in the computer field varied from none at all to more than ten years. Most programmers had only worked for a few months, usually in some form of vacation employment.

The programmers were asked to rate their knowledge of Pascal as either *expert*, *thorough*, *fair*, or *limited*. Of the twenty six, four rated their knowledge as expert, eighteen as thorough, and four as fair. The programmers were asked also to estimate the reliability of their programs. Those who did gave estimates of 0.75, 0.8, 0.85, 0.85, 0.87, 0.9, 0.9, 0.9, 0.95, 0.95, 0.95, 0.97, 0.975, 0.98, 0.98, 0.995, 0.998, 0.998, 0.999, 0.999, and 1.0. Most of the programs were more reliable than the programmers estimated.

The effort-level estimates obtained from the work logs are necessarily approximate since the programers maintained the logs themselves. They were asked to record time spent in reading the requirements specification, designing and implementing the program, and in debugging and testing the program. The reading time varied from one to 35 hours with an average of 5.4; the design time from four to 50 hours with an average of 15.7; and the debugging time from four to 70 hours with an average of 26.6.

4. EXPERIMENTAL RESULTS

For each test case executed, each program produces a 15 by 15 Boolean array, a 15 element Boolean vector, and a single Boolean launch decision, for a total of 241 results. The program calculates these results from the simulated radar tracking data and various parameters, all of which are randomly generated for each test case. The launch condition is the only true output in this application. The other results are really intermediate although they must be produced since the specifications require them as part of the determination of the launch condition. For the programs written for this experiment, all these results must be supplied to the driver program during testing to allow for error detection. We record *failure* 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 exception (such as negative square root) to be raised during execution of that test case.

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. Twenty three of the twenty seven were successful on more than 99.9% of the tests.

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

Table 1 - Version Failure Data

Version	Failures	Pr(Success)	Version	Failures	Pr(Success)
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 - 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

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.

5. MODEL OF INDEPENDENCE

Separate versions of a program may fail on the same input even if they fail independently. Indeed, if they did not, their failures 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 the conditional probability of A occurring given that B has occurred is the same as the probability of A occurring, and vice versa. That is 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.

For any given program, we assume that the probability of failure on each test case is the same. This is reasonable since prior to testing we had no knowledge of the presence of any faults, and all test cases were generated randomly. 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:

$$P_0 = (1 - p_1)(1 - p_2)\ldots(1 - p_N)$$

The probability that exactly 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 with parameter P_{more}. 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 sufficiently large [16], a normal approximation to this binomial distribution can be used. 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, our null hypothesis is that the above is a correct model of the data. We can estimate the quantity P_{more} 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%. We conclude that the model does not hold. However, clearly the only potential problem with the model is that it is derived from the assumption of independent failures. Thus, we reject this assumption.

6. ANALYSIS OF FAULTS

We define a *fault* to be any instance of program text in any particular version that causes that version to fail when that program text is executed on some test case. The various launch conditions that have to be computed are sometimes similar in their description. If a programmer made the same mistake in implementing two different but similar launch conditions, we record that as two different faults.

A total of forty five faults were detected in the program versions used in this experiment. The numbers of faults found in the individual versions is shown in Table 4. All of these faults have been found

Table 4 - Faults Located In Each Version

Version	Faults	Version	Faults
1	1	15	0
2	0	16	2
3	4	17	2
4	0	18	1
5	0	19	1
6	3	20	2
7	1	21	2
8	2	22	3
9	2	23	2
10	0	24	1
11	1	25	3
12	2	26	7
13	1	27	0
14	2		

and corrected. The corrective code was installed so that it could be selectively enabled and an extensive analysis of the faults has been undertaken [17]. Many of the faults were unique to individual versions but several occurred in more than version. We will refer to the former as *non-correlated* and the latter as *correlated*. The details of the faults are quite complex and a complete description is beyond the scope of this paper. We include in this section a description of two non-correlated and two correlated faults for illustrative purposes. Recall that all the versions used in this experiment were required to pass two hundred tests as part of the acceptance procedure. The faults described in this section all survived that acceptance procedure.

The non-correlated faults that we describe here will be recognized as commonly occurring. They are subtle and important nonetheless. The first was an omission by the programmer of the assignment of a value to a function for one path through the function. This was not checked by any of the compilers used in this experiment. The result of executing that particular path through the function was that the function returned whatever happened to be at the memory location allocated for the result. The effect was therefore

implementation dependent since some implementations always initialize storage. The effect was also time dependent since the result obtained was acceptable on some calls and not on others. In the million test cases, this particular fault caused the version containing it to fail only 607 times.

The second non-correlated fault was the use of the wrong expression to index an array. This occurred in several versions. The required expression was usually a single identifier, and the fault usually consisted of using the wrong identifier. A specific example is the following function call:

```
sam3pts(x[i], y[i], x[j], y[i], x[k], y[k]);
```

The wrong index expression has been used for the fourth parameter. The correct function call is:

```
sam3pts(x[i], y[i], x[j], y[j], x[k], y[k]);
```

This particular fault caused the associated version to fail 1297 times during the one million tests. We find it surprising that major faults such as this can occur in programs that are doing extensive manipulation of arrays yet cause relatively few failures.

The correlated faults were, in general, far more obscure. The first example involves the comparison of angles. In a number of cases, the specifications require that angles be computed and compared. As with all comparisons of real quantities, the limited precision real comparison function was to be used in these cases. The fault was the assumption that comparison of the *cosines* of angles is equivalent to comparison of the angles. With arbitrary precision this is a correct assumption of course but for this application it is not since finite precision floating point arithmetic was used and the precision was limited further for comparison. Of the twenty seven versions written, four made this incorrect assumption. In borderline cases this assumption was false, and this caused the associated versions to disagree with the gold program. The number of failures attributable to this fault varied from 71 to 206 in the various versions although this particular fault caused more than one version to fail on the same test case on only eight occasions.

This fault cannot be attributed to the specifications. Rather it was caused by a fundamental lack of understanding of numerical analysis. The solution lies in a thorough analytic treatment of the arithmetic of the machine involved, and the algorithms used in the computation of the angles and their cosines.

The second correlated fault example involved an assumption about the angle subtended by three points. Recall that the program is required to process simulated two-dimensional radar data. The data is presented as points in a plane expressed in Euclidean coordinates. The specifications for the problem require the determination of whether three data points (simulated radar echos) lie on a straight line. It is possible to determine this by examining the angle subtended by the three points, regarding one of them as the vertex of the angle. If the angle is zero *or* the angle is 180 degrees, the points lie on a straight line. Figure 1A shows the general case, and Figures 1B and 1C show the two cases where all three points lie on a straight line. The fault made by more than one programmer is the *omission* of the second case.

Again, this fault cannot be attributed to the specifications. It was caused by a lack of understanding of geometry. It is not clear how such a fault could be prevented since basically it is attributable to an

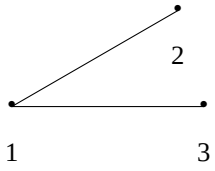

Figure 1A

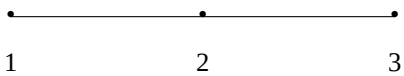

Figure 1B

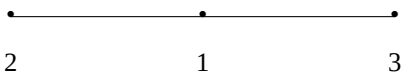

Figure 1C

incomplete case analysis. In fact, although the fundamental fault was the same in more than one version, the effects were different and caused different numbers of failures in the affected versions [18]. One version failed 231 times because of this fault and a second only 37 times. However, whenever the second of the two failed, the first did also. The reason for the difference is the interaction between this fault and the overall algorithms used by the different versions.

7. 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, twenty one had less than one year of programming experience outside their degree programs, three had between two and five years, and two 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 Oak Ridge) 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.

The twenty seven versions ranged in length from 327 to 1004 lines of code. This is much smaller than most real-time systems which may include millions of lines of code. Since many faults occur in the

interconnection between components in a large modular system, results of this experiment relate only to duplication of small pieces of a large system. It would be interesting to do a further experiment with a larger problem. However, from a practical standpoint, economic factors would make it unlikely that many projects could afford complete duplication or triplication of the software. A more likely alternative is that the most critical functions will be identified and separated from the less critical functions and fault tolerance features applied only to those components which have the greatest potential for damage in case of failure. In this respect, the problem used in this experiment is then very realistic.

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.

8. CONCLUSIONS

For the particular problem that was programmed for this experiment, we conclude that the assumption of independence of errors that is fundamental to the analysis of N-version programming *does not hold*. Using a probabilistic model based on independence, our results indicate that the model 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. However, the result does suggest that the use of N-version programming in crucial systems, where failure could endanger human lives for example, should be deferred until further evidence is available.

A second point is that our result does not mean that N-version programming does not work or should never 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 using the coincident errors model [7] to predict reliability.

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 faults or, alternatively, that the common faults 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 faults by different programmers. Thus the fault distribution is more an artifact of the problem itself than the programmer, and thus is *not* random. Another possible hypothesis is that unique (random) faults tend to be those most likely to be caught by a compiler or by testing. Common faults may reflect inherently difficult semantic aspects of the problem or typical human misconceptions which are not easily detected through standard verification and validation efforts.

A final possibility is that common faults 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 specification had been debugged through use in an earlier experiment. Furthermore, the

particular common faults made in this experiment are quite subtle. In our opinion, none involve ambiguity, inconsistency, or deficiency in the specification.

Given that common faults (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 faults 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 faults may be useful. Hardware designers do not rely on simple redundancy or independently generated diverse designs to get rid of common design faults. 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.

9. 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 St.Jean, 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, and to Jo Mahoney for comments on our statistical analysis. 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.

APPENDIX

This is the requirements specification document used in this experiment. It is the version used at UVA. Only minor changes to names and document references were made for the version used at UCI.

LAUNCH INTECEPTOR PROGRAM - REQUIREMENTS SPECIFICATION

INTRODUCTION

As part of a hypothetical anti-ballistic missile system, you will write a parameterless Pascal procedure called DECIDE. It will generate a signal which determines whether an interceptor should be launched based upon input radar tracking information. This radar tracking information is available at the instant the procedure is called. In the following sections, the names of input variables are delimited as 'name'. Terms delimited as #term# are defined in the glossary at the end of this document.

Values of quantities which are parameters of the problem are provided and will determine which combination of the several possible #Launch Interceptor Conditions# (LIC's) are relevant to the immediate situation. The interceptor launch button is normally considered locked; only if all relevant combinations of launch conditions are met will the launch-unlock signal be issued.

Your procedure will determine whether each of fifteen LIC's is true for an input set of up to 100 #planar data points# representing radar echoes. The LIC's are specified in the Functional Requirements section of this document. To indicate which LIC's are satisfied by the set of points, the fifteen elements of a #Conditions Met Vector# (CMV) will be assigned boolean values true or false; each element of the CMV corresponds to one LIC.

Another input, the #Logical Connector Matrix# (LCM), defines which individual LIC's must be considered jointly in some way. The LCM is a 15x15 symmetric #matrix# with elements valued ANDD, ORR, or NOTUSED. CMV elements are combined as indicated by the LCM, and the resulting boolean values are stored in the #off-diagonal elements# of the #Preliminary Unlocking Matrix# (PUM), a 15x15 symmetric #matrix#. Thus, the #off-diagonal elements# of the PUM are an output of your procedure. The PUM's #diagonal elements# are an input to your procedure and represent which LIC's actually matter in this particular launch determination. Each #diagonal element# of the PUM indicates how to combine the off-diagonal values in the same PUM row to form the corresponding element of the #Final Unlocking Vector# (FUV), a 15-element #vector#. If, and only if, all the values in the FUV are true, should the launch-unlock signal be generated.

No actual reading or writing to physical I/O devices will take place. Instead, inputs to your procedure will be available as global variables, and outputs from your procedure will be placed in other global variables.

HARDWARE AND SOFTWARE SUPPORT

Hardware

Your procedure is to be written using, and must execute on, the PR1ME 750 equipment of the Academic Computing Center of the University of Virginia.

Software

The only software facilities you may use to prepare your procedure are:

(1) PRIMOS operating system

(2) Software Tools subsystem (optional)

(3) Any text editor available on the above systems

(4) Hull V-Mode Pascal compiler

References

(1) *PR1ME User's Guide*, Revision 19.0, Third Edition, by Anne W. Patenaude. Published by PR1ME Computer, Inc., Framingham, Massachusetts, 1982.

(2) *PRIMOS Commands Reference Guide*, by Alice Landy. Published by PR1ME Computer, Inc., Framingham, Massachusetts, 1981.

(3) *Software Tools Subsystem User's Guide*, by T. Allen Akin, et.al., Georgia Institute of Technology, Atlanta, Georgia, 1982.

(4) *The Hull V-Mode Pascal Compiler, User Manual Version 3.3*, by Barry Cornelius, Ian Thomas, University of Hull, Hull, England, 1982.

(5) *Pascal User Manual and Report*, Second Edition, by Kathleen Jensen and Niklaus Wirth. Published by Springer-Verlag, New York, 1974.

FUNCTIONAL REQUIREMENTS

All communication with software which calls your procedure is to be accomplished through the global variables and constant defined in this section.

Constant

Available to your procedure is the value of the global constant, PI, representing the number of radians in 180 degrees.

Input Variables

The values of the following global variables are available to your procedure:

NUMPOINTS	The number of #planar data points#.
X,Y	Parallel arrays containing the coordinates of data points.
PARAMETERS	Record holding parameters for LIC's.
LCM	Logical Connector Matrix.
PUM (diagonal elements)	Preliminary Unlocking Matrix.

Output Variables

The values of the following global variables are to be set by your procedure:

PUM (off-diagonal elements)	Preliminary Unlocking Matrix.
CMV	Conditions Met Vector.
FUV	Final Unlocking Vector.
LAUNCH	Final launch/no launch decision.

Global Declarations

The global declarations have been made as follows:

```
const
  PI       = 3.1415926535;

type
  POINTRANGE = 1..100;
  LICRANGE   = 1..15;
  NPOINTS    = 2..100;
  NPTYPE     = 3..100;
  CONNECTORS = (NOTUSED,ORR,ANDD);
  NUMQUADS   = 1..3;
  COORDINATE = array[POINTRANGE] of real;
  CMATRIX    = array[LICRANGE,LICRANGE] of CONNECTORS;
  BMATRIX    = array[LICRANGE,LICRANGE] of boolean;
  VECTOR     = array[LICRANGE] of boolean;
  COMPTYPE   = (LT,EQ,GT);

var
```

```
X         : COORDINATE; {X coordinates of data points}
Y         : COORDINATE; {Y coordinates of data points}
NUMPOINTS  : NPOINTS;    {Number of data points}

PARAMETERS : record
   LENGTH1 : real;       {Length in LICs 1, 8, 13}
   RADIUS1 : real;       {Radius in LICs 2, 9, 14}
   EPSILON : real;       {Deviation from 'PI' in LICs 3, 10}
   AREA1   : real;       {Area in LICs 4, 11, 15}
   Q_PTS   : NPOINTS;    {No. of #consecutive# points in LIC 5}
   QUADS   : NUMQUADS;   {No. of quadrants in LIC 5}
   DIST    : real;       {Distance in LIC 7}
   N_PTS   : NPTYPE;     {No. of #consecutive# pts. in LIC 7}
   K_PTS   : POINTRANGE; {No. of int. pts. in LICs 8, 13}
   A_PTS   : POINTRANGE; {No. of int. pts. in LICs 9, 14}
   B_PTS   : POINTRANGE; {No. of int. pts. in LICs 9, 14}
   C_PTS   : POINTRANGE; {No. of int. pts. in LIC 10}
   D_PTS   : POINTRANGE; {No. of int. pts. in LIC 10}
   E_PTS   : POINTRANGE; {No. of int. pts. in LICs 11, 15}
   F_PTS   : POINTRANGE; {No. of int. pts. in LICs 11, 15}
   G_PTS   : POINTRANGE; {No. of int. pts. in LIC 12}
   LENGTH2 : real;       {Maximum length in LIC 13}
   RADIUS2 : real;       {Maximum radius in LIC 14}
   AREA2   : real        {Maximum area in LIC 15}
end;  {of record PARAMETERS}

LCM       : CMATRIX;    {Logical Connector Matrix}
PUM       : BMATRIX;    {Preliminary Unlocking Matrix}
CMV       : VECTOR;     {Conditions Met Vector}
FUV       : VECTOR;     {Final Unlocking Vector}
LAUNCH    : boolean;    {Decision: Launch or No Launch}

function REALCOMPARE (A,B : real) : COMPTYPE;
{compares real numbers -- see Nonfunctional Requirements}
```

Required Computations

It can be assumed that all input data and parameters that are measured in some form of units use the same, consistent units. For example, all lengths are measured in the same units that are used to define the planar space from which the input data comes. Therefore, no unit conversion is necessary.

Given the parameter values in the global record 'PARAMETERS', the procedure DECIDE must evaluate each of the #Launch Interceptor Conditions# (LICs) described below for the set of 'NUMPOINTS' points:

(X[1],Y[1]) ,...., (X[NUMPOINTS],Y[NUMPOINTS])

where 2 <= NUMPOINTS <= 100

The #Conditions Met Vector# (CMV) should be set according to the results of these calculations, i.e. the global array element CMV[i] should be set to true if and only if the ith LIC is met.

The Launch Interceptor Conditions (LIC) are defined as follows:

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

(0 <= LENGTH1)

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

(0 <= RADIUS1)

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

angle < (’PI’ - ’EPSILON’)

or

angle > (’PI’ + ’EPSILON’)

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

(0 <= EPSILON < PI)

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

(0 <= AREA1)

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

(2 <= Q_PTS <= NUMPOINTS) , (1 <= QUADS <= 3)

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

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

(3 <= N_PTS <= NUMPOINTS) , (0 <= DIST)

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

1 <= K_PTS <= {NUMPOINTS - 2}

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

1 <= A_PTS , 1 <= B_PTS

A_PTS + B_PTS <= NUMPOINTS - 3

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

angle < (’PI’ - ’EPSILON’)

or

angle > (’PI’ + ’EPSILON’)

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

1 <= C_PTS , 1 <= D_PTS

C_PTS + D_PTS <= NUMPOINTS - 3

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

1 <= E_PTS , 1 <= F_PTS

E_PTS + F_PTS <= NUMPOINTS - 3

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

1 <= G_PTS <= {NUMPOINTS - 2}

13) There exists at least one set of two data points, separated by exactly 'K_PTS' #consecutive# intervening points, which are a distance greater than the length, 'LENGTH1', apart. In addition, there exists at least one set of two data points (which can be the same or different from the two data points just mentioned), separated by exactly 'K_PTS' #consecutive# intervening points, that are a distance less than the length, 'LENGTH2', apart. Both parts must be true for the LIC to be true. The

condition is not met when ’NUMPOINTS’ < 3 .

(0 <= LENGTH2)

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

(0 <= RADIUS2)

15) There exists at least one set of three data points, separated by exactly ’E_PTS’ and ’F_PTS’ #consecutive# intervening points, respectively, that are the vertices of a triangle with area greater than ’AREA1’. In addition, there exist three data points (which can be the same or different from the three data points just mentioned) separated by exactly ’E_PTS’ and ’F_PTS’ #consecutive# intervening points, respectively, that are the vertices of a triangle with area less than ’AREA2’. Both parts must be true for the LIC to be true. The condition is not met when ’NUMPOINTS’ < 5 .

(0 <= AREA2)

The #Conditions Met Vector# (CMV) can now be used in conjunction with the #Logical Connector Matrix# (LCM) to form the #off-diagonal elements# of the #Preliminary Unlocking Matrix# (PUM). The

entries in the LCM represent the logical connectors to be used between pairs of LICs to determine the corresponding entry in the PUM, i.e. LCM[i,j] represents the boolean operator to be applied to CMV[i] and CMV[j]. PUM[i,j] is set according to the result of this operation. If LCM[i,j] is NOTUSED, then PUM[i,j] should be set to true. If LCM[i,j] is ANDD, PUM[i,j] should be set to true only if (CMV[i] AND CMV[j]) is true. If LCM[i,j] is ORR, PUM[i,j] should be set to true if (CMV[i] OR CMV[j]) is true. (Note that the LCM is symmetric, i.e. LCM[i,j]=LCM[j,i] for all i and j).

Example

Assume that the given #Logical Connector Matrix# is as shown below:

Logical Connector Matrix (LCM)

LIC	l	2	3	4	5	...	15
l	ANDD	ANDD	ORR	ANDD	NOTUSED	...	NOTUSED
2	ANDD	ANDD	ORR	ORR	NOTUSED	...	NOTUSED
3	ORR	ORR	ANDD	ANDD	NOTUSED	...	NOTUSED
4	ANDD	ORR	ANDD	ANDD	NOTUSED	...	NOTUSED
5	NOTUSED	NOTUSED	NOTUSED	NOTUSED	NOTUSED	...	NOTUSED
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
l5	NOTUSED	NOTUSED	NOTUSED	NOTUSED	NOTUSED	...	NOTUSED

Also assume that the entries in the CMV have been computed as described, giving the following results:

Conditions Met Vector (CMV)

Condition	Value
1	false
2	true
3	true
4	true
5	false
.	.
.	.
.	.
l5	false

The following PUM is generated:

Preliminary Unlocking Matrix (PUM)

LIC	1	2	3	4	5	...	15
1	*	false	true	false	true	...	true
2	false	*	true	true	true	...	true
3	true	true	*	true	true	...	true
4	false	true	true	*	true	...	true
5	true	true	true	true	*	...	true
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
15	true	true	true	true	true	...	*

* Diagonal elements are input, not computed values.

Explanation of selected PUM entries:

(1) PUM[1,2] is false because LCM[1,2] is ANDD, and at least one of CMV[1] and CMV[2] is false.

(2) PUM[1,3] is true because LCM[1,3] is ORR, and at least one of CMV[1] and CMV[3] is true.

(3) PUM[2,3] is true because LCM[2,3] is ORR, and at least one of CMV[2] and CMV[3] is true.

(4) PUM[3,4] is true because LCM[3,4] is ANDD, and both CMV[3] and CMV[4] are true.

(5) PUM[1,5] is true because LCM[1,5] is NOTUSED.

The #Final Unlocking Vector# (FUV) is generated from the #Preliminary Unlocking Matrix#. The input #diagonal elements# of the PUM indicate whether the corresponding LIC is to be considered as a factor in signaling interceptor launch. FUV[i] should be set to true if PUM[i,i] is false (indicating that the associated LIC should not hold back launch) or if all elements in PUM row i are true.

Example

Assume that the PUM now appears as follows:

Preliminary Unlocking Matrix (PUM)

LIC	1	2	3	4	5	...	15
1	true	false	true	false	true	...	true
2	false	false	true	true	true	...	true
3	true	true	true	true	true	...	true
4	false	true	true	false	true	...	true
5	true	true	true	true	false		true
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
15	true	true	true	true	true		false

The FUV generated is:

Final Unlocking Vector (FUV)

Condition	Value
1	false
2	true
3	true
4	true
5	true
.	.
.	.
.	.
15	true

Explanation of selected FUV entries:

(1) FUV[1] is false because PUM[1,1] is true, but PUM[1,2] and PUM[1,4] are false.

(2) FUV[2] is true because PUM[2,2] is false.

(3) FUV[3] is true because PUM[3,i] is true for all i, 1 <= i <= 15.

The final launch/no launch decision is based on the FUV. The decision to launch requires that all elements in the FUV be true, i.e. LAUNCH should be set to true if and only if FUV[i] is true for all i, 1 <= i <= 15. For the example, LAUNCH is false because FUV[1] is false.

NONFUNCTIONAL REQUIREMENTS

(1) The functional requirements are to be implemented by a parameterless Pascal procedure named DECIDE. It will perform no input or output, because the calling program will provide input data through global variables. Likewise, DECIDE should store its results in global variables.

(2) Whenever real numbers must be compared within the procedure DECIDE, that comparison should be made with a fixed amount of precision. The program which calls DECIDE will provide a function called REALCOMPARE.. (See function header in declarations on page 4.) This function compares two real numbers, A and B, with respect to the six most significant digits. REALCOMPARE returns LT if A < B , EQ if A = B , or GT if A > B . DECIDE should call this function for all comparisons of real numbers.

(3) Information contained in the global variables when the subroutine is called will remain valid throughout the execution of the procedure. There are no feedback or time series effects during a call to DECIDE, or from multiple calls to DECIDE.

(4) Do not include input error checking. Assume that the calling program insures inputs are complete and within the specified range.

(5) No double precision or complex variables should be used.

(6) There are no constraints on memory space or execution time, but efficient, well-structured code with descriptive comments is preferred.

(7) In writing the subroutine, do not use any language-dependent software tools other than the Hull V Pascal compiler.

GLOSSARY

angle

An angle is formed by two rays which share a common endpoint called a vertex. If one ray is rotated about the vertex until it coincides with the other ray, the amount of rotation required is the measure of the angle. Three points can be used to determine an angle by drawing a ray from the second point through the first point and another ray from the second point through the third point. Note that different angles are described according to whether the ray is rotated clockwise or counterclockwise. Either can be used in this problem because of the way the LIC's are defined.

CMV

(Conditions Met Vector) The CMV is a boolean vector whose elements have a one-to-one correspondence with the launch interceptor conditions. If the radar tracking data satisfy a certain LIC, then the corresponding element of the CMV is to be set to true.

consecutive

Two points are consecutive if they are adjacent in the input data vectors X and Y. Thus (X[i],Y[i]) and (X[i+1],Y[i+1]) are adjacent.

diagonal element

Consider a matrix M, with n rows and n columns. The diagonal elements of the matrix are M[i,i] , where i=1,..,n.

FUV

(Final Unlocking Vector) The FUV is a boolean vector which is the basis for deciding whether to

launch an interceptor. If all elements of the FUV are true, a launch should occur.

LCM

(Logical Connector Matrix) The LCM describes how individual LIC's should be logically combined. For example, the value of LCM[i,j] indicates whether LIC #i should combine with LIC #j by the logical AND , OR , or not at all.

LIC

(Launch Interceptor Condition) If radar tracking data exhibit a certain combination of characteristics, then an interceptor should be launched. Each characteristic is an LIC.

matrix

For purposes of this problem, a matrix can be considered to be a two-dimensional array.

off-diagonal element

An off-diagonal element of a matrix is any element which is not a diagonal element.

planar data points

Planar data points are points that are all located within the same plane.

PUM

(Preliminary Unlocking Matrix) Every element of the boolean PUM corresponds to an element of the LCM. If the logical connection dictated by the LCM element gives the value "true", the corresponding PUM element is to be set to true.

quadrant

The x and y axes of the Cartesian coordinate system divide a plane into four areas called quadrants. They are labeled I, II, III, IV, beginning with the area where both coordinates are positive and

numbering counterclockwise.

radius

The length of the radius of a circle is the distance from the center of the circle to any point on the circle's circumference.

ray

A ray is a straight line that extends from a point.

vector

For purposes of this problem, a vector may be considered to be a one-dimensional array.

vertex

When two rays originate from a common point to form an angle, the point of their origination is called the vertex of that angle.

REFERENCES

(1) L. Chen 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, pp. 3-9, June 1978.

(2) C.V. Ramamoorthy, Y.R. Mok, E.B. Bastani, G.H. Chin, and K. Suzuki. ‘‘Application of a methodology for the development and validation of reliable process control software,’’ *IEEE Trans. on Software Engineering,* vol. SE-7, no. 6, pp. 537-555, Nov. 1981.

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

(4) J.P.J. Kelly and A. Avizienis, ‘‘A specification-oriented multi-version software experiment,’’ *Digest of Papers FTCS-13: Thirteenth International Conference on Fault Tolerant Computing,* Milan, Italy, pp. 120-125, June 1983.

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

(6) B. Bonnett, ‘‘Software in safety and security critical systems’’ (panel presentation), *COMPCON 84,* Washington D.C., Sept. 1984. (transcription of the panel session available from Albert W. Friend, ELEX 70343, NAVELEX, Washington, D.C. 20363).

(7) D.E. Eckhardt and L.D. Lee, ‘‘A theoretical basis for the analysis of redundant software subject to coincident errors,’’ NASA Technical Memorandum 86369, NASA Langley Research Center, Hampton, Virginia, January 1985.

(8) D.J. Martin, ‘‘Dissimilar software in high integrity applications in flight controls,’’ *Software for Avionics,* AGARD Conference Proceedings, No. 330, pp. 36-1 to 36-9, January 1983.

(9) J.R. Taylor, in ‘‘Letter from the editor,’’ *ACM Software Engineering Notes,* vol. 6, no. 1, pp. 1-2, January 1981.

(10) U. Voges, F. Fetsch, and L. Gmeiner, ‘‘Use of microprocessors in a safety-oriented reactor shut-down system,’’ in E. Lauber and J. Moltoft (eds.) *Reliability in Electrical and Electronic Components and Systems,* North-Holland, pp.. 493-497, 1982.

(11) G. Dahll and J. Lahti, ‘‘An investigation of methods for production and verification of highly reliable software,’’ in L. Lauber (ed.) *Safety of Computer Control Systems (Proceedings of SAFECOMP ’79),* Pergamon Press, pp. 89-94, 1980.

(12) L. Gmeiner and U. Voges, ‘‘Software diversity in reactor protection systems: An experiment,’’ *Safety of Computer Control Systems (Proceedings of SAFECOMP ’79),* Pergamon Press, pp. 75-79, 1980.

(13) A. Avizienis and J.P.J. Kelly, ‘‘Fault tolerance by design diversity: concepts and experiments’’, *IEEE Computer,* vol. 17, no. 8, August 1984.

(14) J.F. Wakerly, ‘‘Microcomputer reliability improvement using triple-modular redundancy,’’ *Proceedings of the IEEE,* vol. 64, no. 6, pp. 889-895, June 1976.

(15) P.M. Nagel 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.

(16) M.S. Raff, ‘‘On approximating the point binomial,’’ *J. Amer. Statist. Ass.*, vol 51, 1956.

(17) S.S. Brilliant, ‘‘Analysis of faults in a multi-version software experiment,’’ M.S. Thesis, University of Virginia, May 1985.

(18) L.D. St.Jean, ‘‘Testing version independence in multi-version programming,’’ M.S. Thesis, University of Virginia, January 1985.