The Effectiveness of Data Diversity as an Error Detection Mechanism

John C. Knight, Paul E. Ammannn, Steven Santos

Computer Science Report No. TR-91-08
April 5, 1991

ON THE EFFECTIVENESS OF DATA DIVERSITY AS AN ERROR DETECTION MECHANISM[†]

John C. Knight
Department of Computer Science
Thornton Hall, University of Virginia
Charlottesville, Virginia, 22903
(804) 924-7605 FAX: (804) 982-2214
knight@virginia.edu

Paul E. Ammann
Department of Information Systems and Systems Engineering
George Mason University, Fairfax, VA 22030
(703) 764-4664 FAX: (703) 323-2630
pammann@gmuvax2.gmu.edu

Steven J. Santos
Department of Computer Science
University of Virginia, Charlottesville, Virginia, 22903
(804) 924-7605 FAX: (804) 982-2214
sjs2g@cs.virginia.edu

ABSTRACT

We have proposed previously an approach to software fault tolerance based on diversity in the data space; a technique we term *data diversity*. In a fault-tolerant system employing data diversity, the *same* software is executed on several set of points in the data space, each of which should produce the same (or simply related) output. Error detection in data diversity is achieved by observing unexpected differences between the outputs of the software on the different data points. This approach to error detection can be employed in software testing as well as fault tolerance, and an important question in both applications is the error detection performance of data diversity. We report in this paper on a study in which several pieces of commercial software were obtained and into which faults were injected. These faulty programs were tested using data diversity for error detection. The results of this study show that data diversity can be an effective error detection technique.

Subject Index:
Fault-tolerant software, software reliability, design diversity, data diversity.

[†]Sponsored in part by NASA grant NAG_1-1123-FDP; paper cleared by affiliation; approximately 6000 words.

1. INTRODUCTION

Error detection is the determination that the output of a software system is unacceptable to its application, i.e., the software is not in compliance with its specification. Clearly, error detection is a critical element of both software testing and software fault tolerance. In software testing, for example, if a test input has been created that causes a fault to manifest itself, this will be of little value unless the erroneous software operation can be detected. Similarly, it is not possible to mask the effects of a fault during execution, and thereby tolerate it, if the erroneous software operation is not detected. In some systems, detection of errors during operation rather than fault masking is the only goal [1].

Most existing approaches to software fault tolerance are based on the notion of *design diversity* [2]. These approaches employ some form of software redundancy, and depend for their operation upon differences in the various redundant designs. In the *N*-version approach to software fault tolerance, for example, independent implementations of the software are prepared [2, 3]. Although various error detection mechanisms are employed in such systems, an unexpected difference in the outputs of these different implementations is the primary mechanism for error detection.

In testing software systems, many different forms of error detection are used including replication checks, reversal checks, timing checks, and acceptability checks. The approach to error detection in which the output of a software system is desk checked by a human is actually a replication check. Similarly, human examination of the output to make sure that it "looks right" is actually an informal acceptability check. Many checking techniques are actually applications of design diversity.

We have proposed previously an approach to software fault tolerance based on diversity in the data space; a technique we term *data diversity* [4]. In contrast to systems employing design diversity, a system employing data diversity executes the *same* software on several *different* (but related) sets of points in the data space. If no fault

manifests itself, each input data set should produce the same (or simply related) output, and a decision algorithm is employed to determine system output. As with design diversity, it is an unexpected difference in the outputs that signals an error in a data-diverse system. Where there is such a difference, at least one of the executions of the software must have involved the manifestation of a fault. Error detection is followed in a fault-tolerant system by the selection of a suitable system output by a decision algorithm. In direct analogy to the *N*-version [3] and recovery block [5] strategies of design diversity, the decision algorithm in a data-diverse system uses a voter or an acceptance test. Data diversity was shown to provide a useful degree of software fault tolerance in a pilot study [6].

The fact that data diversity supports error detection is what permits its use in software fault tolerance. However, this ability to detect errors suggests that data diversity can be employed in software testing also. In practice, what is required in both cases is for errors to be detected with probability one when they occur and with probability zero otherwise. Not detecting an error when it occurs is potentially catastrophic in a fault-tolerant system. Signalling an error when none has occurred is also serious in a fault-tolerant system since this also could lead to failure. Not detecting an error that occurs when testing is less serious. Provided the fault responsible for the undetected error subsequently causes another error that *is* detected, the presence of the fault will be revealed. At test time, the important thing is that the errors produced by a given fault have non-zero probabilities of manifestation and of detection. The larger the probabilities the better, but non-zero probabilities ensure that the probability of finding the fault can be made as close to one as desired by increasing the number of tests. During testing, false error signals lead to wasted human effort.

The performance of data diversity as an aid to testing and as a technique for fault tolerance hinges on its error detection performance. As part of a long-term research project on software dependability, we are assessing the performance of various aspects of data diversity. In this paper we report on a study of its effectiveness in error detection.

We have performed an experiment in which the ability of a data-diverse system to detect the effects of injected faults was determined. In the next section, we describe the concepts of data diversity and the mechanisms of error detection in more detail. In section 3 we review the experiment that was performed and in section 4 we present the results of the experiment. Section 5 contains our conclusions.

2. ERROR DETECTION AND DATA DIVERSITY

The idea of using data diversity as a technique for achieving software fault tolerance was based on two observations. First, anecdotal evidence suggests that software tends to fail on special input cases and unusual combinations of events, and second, in many cases, software is a many-to-one mapping. Every programmer is familiar, for example, with the situation in which a program fails because of the incorrect processing of unexpected or obscure data cases. Similarly, programmers are aware that the exact order of the data read by a program is frequently irrelevant.

The first observation leads to the idea that software would not fail because of a specific special case if the special case were avoided, and the second observation suggests that the special case might be avoided by using a different input data set. Clearly, this different data has to be chosen so that an acceptable output is produced. A fault-tolerant software structure called an *N*-copy system [6] that is designed to take advantage of these ideas is shown in Figure 1.

The process of deriving input data sets that are equivalent to the original yet different in some way is termed *data reexpression*. There are two forms of data reexpression - *exact* and *approximate*. Exact data reexpression takes the supplied input data set and produces an alternative input data set that will produce either the same output as the original (perhaps up to the limits of numeric accuracy) or an output that is simply related to the original. An example of exact reexpression for a sort function is a random permutation of the input. Clearly the reexpressed input is different from the original input yet should produce identical output. Another example of exact

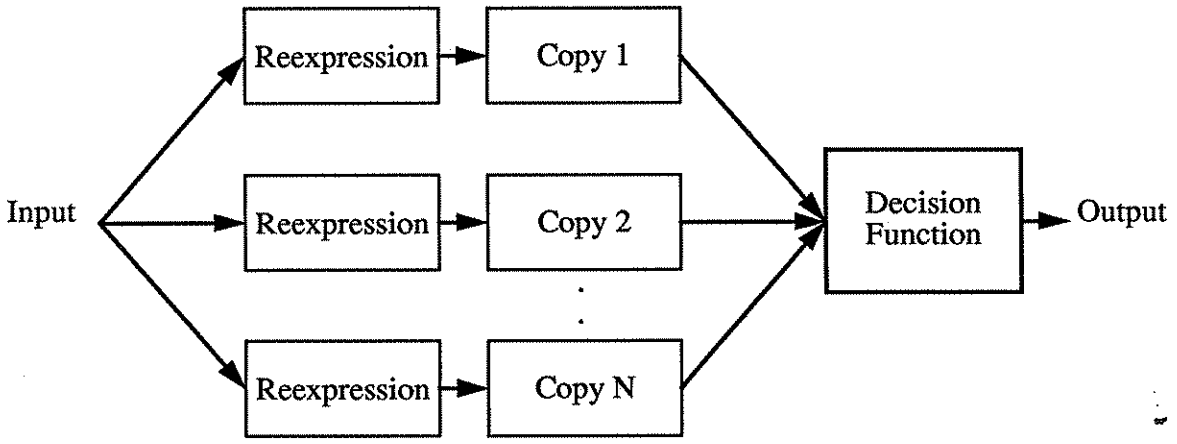

Figure 1 - *N*-Copy System Structure

reexpression for a sort function is to subtract each input data value from a value larger than all the input data values. Again, the reexpressed input data set is different from the original but the output will be different also. However, the output is simply related to the original and the original output is easily recovered.

Approximate data reexpression produces an alternative input data set that will produce an output acceptable to the application but possibly not the same nor simply related to that produced by the original data set. An example is the introduction of a low intensity noise term into the sensor values used by a control system. This should have little or no impact on the application since sensor data is usually of limited accuracy. However, the reexpressed data might avoid a failure arising from a special case in the data, such as a division by zero resulting from the subtraction of two identical data values.

Clearly, data reexpression algorithms are application-dependent. There is no general rule that permits reexpression algorithms to be derived for all applications, although this can be done in some special cases (see [7], for example). Our experience to date is that data reexpression algorithms exist for a surprisingly wide range of applications [8]. The efficacy of data reexpression is an important factor in the success of data diversity. It is data reexpression that yields the multiple data sets which ultimately lead to the unexpected difference in the outputs and thereby to error detection.

In summary, error detection using data diversity involves execution of the same software with different data sets where the different data sets should produce identical or related outputs. The different data sets are produced by deliberate manipulation in a process called reexpression. An error is signalled, and hence the manifestation of a fault observed, if there is an unexpected difference in the outputs from the various executions of the software. A potentially valuable benefit of detecting errors at test time in this way is that it permits automation of error detection in much the same way as comparison checking does [9, 10]. This facilitates automation of the whole test process. An automated test system employing data diversity is shown in Figure 2.

3. EXPERIMENTAL ASSESSMENT

3.1. Overview

The overall goal of this study was to assess the error detection performance of data diversity. Clearly, an analytic approach in which quantifiable characteristics of a software system are used to predict error detection performance would be ideal. However, such an approach is infeasible in general, and we resorted to an experimental approach in which we measured the ability of data diversity to detect errors in sample programs containing known faults. Informally, the approach was to execute samples of commercial software in an *N*-copy structure with input data taken from the appropriate operational distribution.

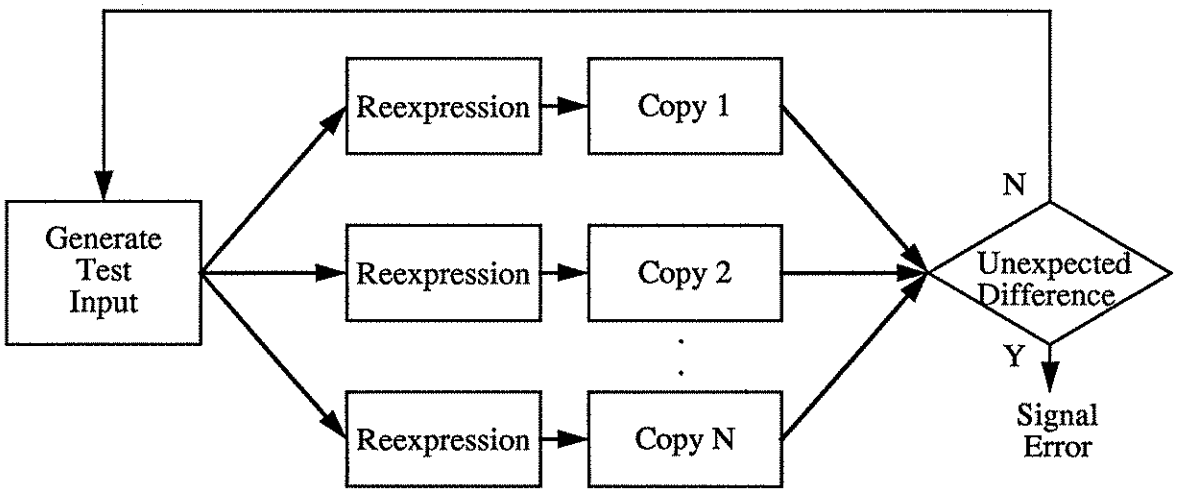

Figure 2 - Automated Test System Using Data Diversity

An experimental assessment of this type yields the most useful results when it is based on production software with faults left undetected by the development process. Statistically valid general results are obtained when large numbers of faults covering the known spectrum of fault types can be studied. Unless a large number of programs with a wide range of characteristics and a large number of faults of various types can be studied, the observed behavior has little statistically predictive meaning. Merely observing the performance of one or a small number of what would amount to randomly selected programs with essentially random faults does not necessarily lead to any useful general indication of the error detection performance of data diversity. Issues that would remain include the relevance of the results to other faults and fault types, the applicability of the results to software for different application areas, the effect of the skills of the programmers involved and their backgrounds, the effect of the specific data reexpression algorithm used, the effect of the software development environment, etc.

The size of the experimental study that the above implies is beyond our resources. Also, unfortunately† the production software that was available for study contained no known flaws. To mitigate the effect of the inevitable resource limitation and to obtain more than anecdotal evidence of performance, we chose to perform an experiment in which we constrained most of the independent variables in the experiment. That is, we fixed most of the factors that could influence the results and varied one factor completely and another somewhat. Thus our results are conditional on the values of the fixed factors but complete for one variable and indicative for the other. Specifically, we focused on a small number of samples of software written in C that were written by the same programming organization and that come from the same application domain. The domain used was geometric calculations associated with maritime navigation.

The study was performed in two phases. In the first phase, we explored the effects of a set of different fault types using fault injection. This study was complete in that all possible faults of certain types were generated and the resulting programs tested. The goal was to acquire information about the performance of data diversity for all possible instances of a set of fault types in a program. For example, each arithmetic operator in a given test program was systematically changed to every other possible arithmetic operator. This was repeated for all the arithmetic operators in the program and the resulting series of programs containing injected faults, or mutants [11], were tested. The same procedure was followed in phase one for all the relational operators and all the logical operators in the available test programs. Since all the arithmetic, logical, and relational operators were changed and each was changed into all possible alternative operators, this phase of the experiment provides a general indication of the ability of data diversity to detect faults characterized by the incorrect use of these operator types.

This kind of systematic fault injection can only be done for small test programs since the resulting number of mutant programs is very large and grows rapidly with the size of the initial test program. We generated all possible mutants in certain classes for

† Or fortunately, depending on your viewpoint.

two small programs. Each mutant was executed with realistic inputs using data diversity for error detection. Some of the mutants were benign, i.e., the change to the original program did not cause a fault, and others did not fail on every test case. The original software was used as an oracle to detect the cases where an error occurred, and hence when data diversity should have detected the error.

In the second phase of the study, we sought information on the effect of program size. The central question was whether the performance observed in the first phase was likely to scale up to larger programs. Again, resources were limited and only a single larger program was studied. For that program, we generated a random sample of the mutants since generating and testing all possible mutants was infeasible.

In both phases of the study each mutant was tested many times but the entire test process itself was also repeated many times. This was to obtain some indication of the variance in the observed performance. Testing a mutant with a single set of test cases reveals a certain performance level but this observation is a random sample subject to statistical variance. Repeating the test process many times with different test data each time gave insight into this variance which proved to be quite wide.

3.2. Subject Programs

The programs used in this study were supplied by Sperry Marine, Inc., of Charlottesville, Va. Sperry Marine manufactures computerized maritime electronic systems. The two programs used in the first phase of the study compute heading and distance from a ship's present position to its desired destination. Both present position and desired destination are supplied as latitude/longitude pairs. The outputs are the heading that must be steered and the distance to the destination in nautical miles. The first of the two test programs computes the great-circle course and the second computes the rhumb-line course. The great-circle course is that which minimizes the distance traveled between two points. Following a great-circle route requires constant adjustment of the heading. The rhumb-line course is the course that maintains a constant heading.

Although the traveled distance is longer, the ability to steer a fixed heading is a considerable benefit.

The second phase of the study used a program that computes the orbital position of a satellite in the Global Positioning Satellite System (GPSS). The GPSS is used in navigation systems to determine the location of an observer. The orbital position that is computed is the location of the satellite at some specified time in a Euclidean frame of reference based on the center of the Earth. The computations are based on orbital parameter information for a particular reference time that includes the inclination of the orbit above the equatorial plane, the orientation of the orbit relative to the zero meridian, and a large number of data elements used in correcting anomalies to increase the accuracy of the final position estimate.

3.3. Reexpression Algorithms

In phase one, three reexpression algorithms were used. The first depends upon the fact that the output of the two programs was a course heading and distance with no dependence on the actual values of latitude and longitude that were input. The reexpression algorithm rotated the input points around the Earth by an arbitrary amount. Essentially this amounts to changing the longitude but fixing the latitude. Clearly, this should have no effect on the output and so this is an exact reexpression algorithm.

The second reexpression algorithm was approximate. The supplied latitude/longitude pairs were modified by adding a small random offset to each. The resulting courses and headings produced should be very similar, and this required similarity was used as the error indicator.

The third reexpression algorithm was also approximate. It took the supplied latitude/longitude pairs and generated inputs that should produce almost identical outputs by shifting each location by one degree latitude East and one degree latitude West. These reexpressed locations along with the original data produce a total of six different

required courses, each of which should have very similar headings and distances. By arranging the data sets in an appropriate way, the output was forced to be ordered and this required ordering was used as the error indicator.

In the second phase of the experiment, four reexpression algorithms were used. Three of the four were exact and one was approximate. In the first exact reexpression algorithm the inclination of the satellite's orbit with respect to the equatorial plane (an input) was reversed. The effect of this reexpression on the output should be to reverse the sign of the output z coordinate. The second and third reexpressions were similar but involved the orientation of the orbit to the zero meridian (another input). Finally, the approximate reexpression algorithm perturbed the inclination of the orbit and its orientation to the zero meridian by a small random amount. The effects of these changes on the individual Euclidean coordinates are complex but the distance of the satellite from the center of the Earth should be virtually unchanged. An unexpected change in this distance was used as the error indicator.

3.4. Experimental Procedure

In the first phase of the study, each of the arithmetic operators (+, −, ×, /), the relational operators (<, ≤, =, ≥, >, ≠), and the logical operators (*and*, *or*) were changed into every each other operator in the set. Thus a single + operator generated three mutants with the + operator replaced by −, ×, and / respectively. Some of the resulting mutants failed to compile and were discarded.

In the second phase of the study, operators were selected at random for mutation and the change made was selected at random from the available changes. The total number of mutants generated this way is a very small fraction of the possible mutants for the program. For each of the test programs used in the study the numbers of each type of mutant that survived compilation are shown in Table 1. Table 1 also shows the size of each test program in lines where lines are non-comment source lines of code.

	Program 1	Program 2	Program 3
Lines	34	38	977
Arithmetic	48	45	5 (Fixed)
Relational	25	40	5 (Fixed)
Logical	1	3	0

Table 1 - Numbers Of Mutants and Program Sizes

The error detection performance of data diversity was determined for each mutant. This was done by executing the mutant with test data in an *N*-copy software structure and comparing its error detection performance with that of the original program acting as an oracle. This process was repeated for each available reexpression algorithm. In phase one, the exact reexpression algorithm (rotation about the Earth) was parameterized by the amount of rotation. The resulting *N*-copy system was executed with *N* equal to 32, i.e., the mutant was executed separately with the original data rotated by each of 32 different amounts. Error detection then amounted to checking for any unexpected difference in the 32 different output which should have all been the same.

The general form of the test harness used to evaluate the mutant programs is shown in Figure 3. The purpose of the comparators in the test harness is to determine whether there was any error to detect and to identify false alarms. Although the mutants contained faults, it is not necessarily the case that the fault would manifest itself on each test. Similarly, an error detected by the *N*-copy system did not mean there was an error.

For each mutant, a set of 100 test cases was executed by the *N*-copy system and the oracle, and the error detection performance measured by comparing the number of times the *N*-copy system detected an error to the number of times the oracle detected an error. Finally, this whole process was then repeated 200 times in order to get some information about how the data diverse system's error detection performance varied. During the

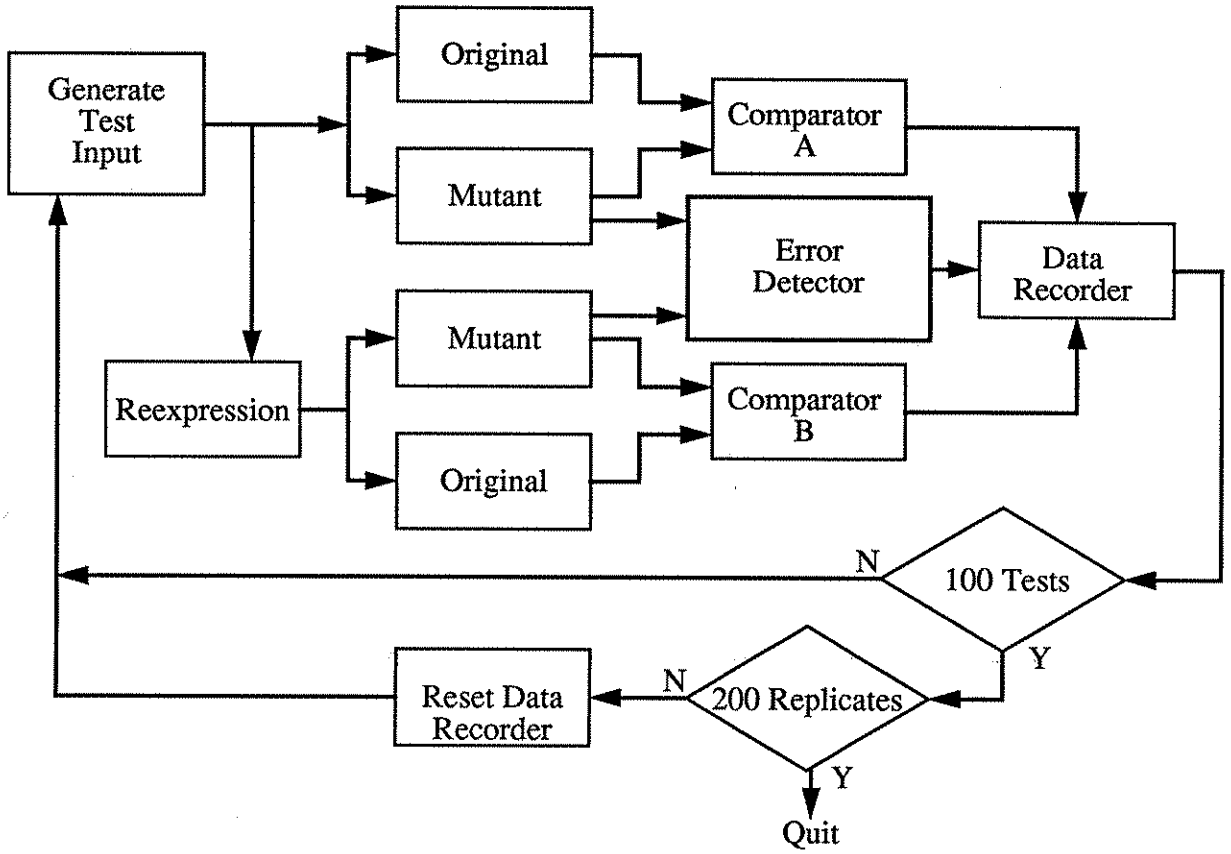

Figure 3 - Evaluation Test Harness

execution of these 200 replicates, the best, worse, and average detection rates for the data-diverse error detector were determined.

4. EXPERIMENTAL RESULTS

4.1. Phase One - Maritime Course Computation

Too many mutants were generated and tested in phase one to permit the results for each mutant to be included in this paper. We have selected from the phase one results

the data for those mutants with the best and the worse error detection performance for each reexpression algorithm. We define best and worst here to be the highest and lowest *average* detection rate, averaged over all 200 replicates. For the first reexpression algorithm, the best and the worst were the same since this reexpression algorithm performed similarly on each replicate. Tables 2 and 3 show the results of testing the arithmetic mutants, and Tables 4 and 5 show the results of testing the relational mutants. In each of these tables, entries expressed as ratios show the number of test cases in which the *N*-copy system detected an error over the number of test cases in which an error occurred. Thus, for example, in Table 2 the maximum data for reexpression algorithm 2 and mutant 11 (center of the table) is 28/40 meaning that of all the replicates that were run, the best detection rate observed for that mutant and reexpression algorithm was 28 detected out of 40 that occurred. Recall that in each replicate 100 tests were run and so the mutant actually failed on 40 out of the 100 tests. A table entry of "NA" means that

		Reexpression Algorithm 1	**Reexpression Algorithm 2**	**Reexpression Algorithm 3**
Mutant 2 (− → +)	Maximum	40/40	6/40	40/40
	Minimum	40/40	0/40	38/40
	Average	1.000	0.046	0.993
Mutant 10 (× → −)	Maximum	40/40	38/40	40/40
	Minimum	40/40	20/40	37/40
	Average	1.000	0.762	0.995
Mutant 11 (× → +)	Maximum	40/40	28/40	40/40
	Minimum	40/40	11/40	37/40
	Average	1.000	0.472	0.995
Mutant 13 (− → +)	Maximum	40/40	28/40	40/40
	Minimum	40/40	11/40	37/40
	Average	1.000	0.465	0.994
Mutant 15 (− → ×)	Maximum	40/40	14/40	19/40
	Minimum	39/40	1/40	4/40
	Average	0.999	0.183	0.226

Table 2 - Great Circle Computation, Arithmetic Mutants

		Reexpression Algorithm 1	Reexpression Algorithm 2	Reexpression Algorithm 3
Mutant 1 (− → ×)	Maximum Minimum Average	48/48 48/48 1.000	13/13 6/10 0.876	1/10 0/17 0.003
Mutant 2 (− → +)	Maximum Minimum Average	48/48 48/48 1.000	2/7 0/13 0.054	4/18 0/5 0.034
Mutant 5 (+ → ×)	Maximum Minimum Average	100/100 100/100 1.000	28/99 0/17 0.183	100/100 88/100 0.941
Mutant 19 (− → +)	Maximum Minimum Average	62/62 62/62 1.000	3/3 3/6 0.893	1/11 0/9 0.002
Mutant 36 (/ → ×)	Maximum Minimum Average	100/100 100/100 1.000	80/100 54/100 0.696	100/100 97/100 0.997

Table 3 - Rhumb Line Computation, Arithmetic Mutants

		Reexpression Algorithm 1	Reexpression Algorithm 2	Reexpression Algorithm 3
Mutant 21 (< → >)	Maximum Minimum Average	NA NA NA	5/40 0/40 0.047	6/40 0/40 0.036
Mutant 22 (< → =)	Maximum Minimum Average	NA NA NA	4/18 0/21 0.042	3/17 0/22 0.039
Mutant 24 (< → ≥)	Maximum Minimum Average	NA NA NA	5/40 0/40 0.047	6/40 0/40 0.036
Mutant 25 (< → ≠)	Maximum Minimum Average	NA NA NA	4/19 0/19 0.044	5/23 0/19 0.039

Table 4 - Great Circle Computation, Relational Mutants

		Reexpression Algorithm 1	Reexpression Algorithm 2	Reexpression Algorithm 3
Mutant 2 (= → >)	Maximum	20/20	3/3	NA
	Minimum	20/20	0/1	NA
	Average	1.000	0.438	NA
Mutant 4 (= → ≥)	Maximum	40/40	35/40	1/40
	Minimum	35/40	19/40	0/40
	Average	0.973	0.688	0.0004
Mutant 16 (> → <)	Maximum	NA	8/39	4/40
	Minimum	NA	0/40	0/40
	Average	NA	0.085	0.019
Mutant 18 (< → =)	Maximum	NA	5/16	2/13
	Minimum	NA	0/17	0/21
	Average	NA	0.089	0.021
Mutant 39 (< → >)	Maximum	40/40	30/35	1/32
	Minimum	35/40	12/29	0/37
	Average	0.973	0.655	0.001

Table 5 - Rhumb Line Computation, Relational Mutants

for the data produced by that reexpression algorithm, the particular mutant never failed. The mutations used to produce the mutant programs are shown in each table in the leftmost column beneath the mutant number.

Only a single logical operator was present in each of the great circle and rhumb line programs, and the single mutant generated from each did not fail in any of the tests executed.

False alarms, i.e., signaling an error when none was present, occurred very rarely, and the rate was directly determined by the tolerance used in the comparison. Inevitable differences in numeric results sometimes triggered false alarms because the differences were somewhat larger than expected yet still acceptable.

The results of phase one suggest that error detection by data diversity was very effective for arithmetic mutants (Tables 2 and 3). There were considerable differences

between the reexpression algorithms, and, in some cases, differences between the mutants for a given reexpression algorithm. The detection of errors caused by faults in relational operators (Tables 4 and 5) was less effective but the average over all replicates was still nonzero indicating that the fault would eventually be detected, at least during testing. Again there was considerable difference between the reexpression algorithms.

4.2. Phase Two - GPSS Orbit Computation

In phase two, a total of five arithmetic mutants and five relational mutants were generated that survived compilation. There were no logical operators in the program. The results of the testing of the arithmetic mutants is shown in Table 6.

		Reexpression Algorithm 1	**Reexpression Algorithm 2**	**Reexpression Algorithm 3**	**Reexpression Algorithm 4**
Mutant 1 (+ → ×)	Maximum	100/100	0/100	100/100	4/100
	Minimum	98/100	0/100	100/100	0/100
	Average	0.996	0.000	1.000	0.017
Mutant 2 (/ → ×)	Maximum	100/100	0/100	100/100	4/100
	Minimum	99/100	0/100	100/100	0/100
	Average	0.994	0.000	1.000	0.020
Mutant 3 (× → /)	Maximum	100/100	100/100	100/100	32/100
	Minimum	97/100	100/100	100/100	19/100
	Average	0.990	1.000	1.000	0.259
Mutant 4 (× → /)	Maximum	35/35	0/43	38/38	2/46
	Minimum	28/29	0/43	38/38	0/43
	Average	0.986	0.000	1.000	0.0094
Mutant 5 (× → +)	Maximum	100/100	0/100	100/100	3/100
	Minimum	99/100	0/100	100/100	0/100
	Average	0.997	0.000	1.000	0.0140

Table 6 - GPSS Computation, Arithmetic Mutants

Each mutant that was generated by changing a relational operator either failed on every test case that was run or did not fail at all. For those that failed on every test case, none of the failures was detected by the *N*-copy structure. Investigation of this situation revealed that these mutants were generating output that was wildly incorrect and doing so no matter how the data was reexpressed. The test harness was modified to include a simple acceptability check, i.e., that the satellite was actually above the surface of the Earth, and this acceptability test failed on every occasion. Thus although the *N*-copy system was defeated, the software had failed so badly that a trivial additional check detected all of the errors.

The error detection performance observed on the larger program used in phase two was similar to that observed during phase one. The arithmetic mutations caused errors that stood a high chance of being detected but there were considerable differences between reexpression algorithms. The performance on relational mutations, i.e., no detection of programs failing every test case, was surprising and suggests that data diversity might be more useful in detecting errors caused by more obscure faults. Combining data diversity with other error detection techniques, such as the reasonableness check that we used, seems to be a fruitful approach.

5. CONCLUSIONS

We conclude from this study that data diversity has considerable potential for error detection but it is quite variable in its performance. This is confirmation of the same conclusion reached in the pilot study [6]. In the present study, however, we see that the variation occurs across different fault classes and across reexpression algorithms. The fact that for many mutants the *N*-copy systems were able to detect the errors generated with very high probability is encouraging.

The variation in performance can be reduced if more than one reexpression algorithm is used, and by using multiple instantiations of the same reexpression algorithm if it is parameterized. It also appears to be beneficial to combine the method

with other error detection methods. Selection of methods that can be easily automated, such as reasonableness checks, is particularly appropriate. The combined error detection performance of a set of methods would be a profitable area to study.

It is important to note the conclusions we have reached is based on a very small-scale study with many experimental variables deliberately fixed. We cannot extrapolate our results to other application domains or in any other way. General results about the performance of data diversity await further experimentation by ourselves and others.

6. ACKNOWLEDGEMENTS

It is a pleasure to thank Sperry Marine, Inc. of Charlottesville, VA. for permitting us to use samples of their software for this evaluation. John Yancey provided us with substantial technical assistance in understanding the algorithms used. This work was sponsored in part by NASA grant NAG_1-1123-FDP;

REFERENCES

[1] 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.

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

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

[4] P.E. Ammann and J.C. Knight, "Data Diversity: An Approach To Software Fault Tolerance", *Digest FTCS-17: Seventeenth International Symposium on Fault Tolerant Computing*, Pittsburgh, PA, July, 1987, pp. 122-126.

[5] B. Randell, "System Structure For Software Fault Tolerance", *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975.

[6] P.E. Ammann and J.C. Knight, "Data Diversity: An Approach To Software Fault Tolerance", *IEEE Transactions On Computers*, Vol. 37, No. 4, April, 1988.

[7] P.E. Ammann, D.L. Lukes, and J.C. Knight "Applying Data Diversity To Differential Equation Solvers", *submitted to FTCS-21: Twenty First International Symposium on Fault Tolerant Computing*, Montreal, Canada, June, 1991.

[8] P.E. Ammann and J.C. Knight "Data Reexpression Techniques For Fault-Tolerant Systems", Technical Report Number TR90-32, Department of Computer Science, University of Virginia, November, 1990.

[9] S.S. Brilliant, J.C. Knight, and P.E. Ammann, "On The Performance Of Software Testing Using Multiple Versions", *Digest FTCS-20: Twentieth International Symposium on Fault Tolerant Computing*, Newcastle Upon Tyne, UK, June, 1990.

[10] F. Saglietti and W. Ehrenberger, "Software Diversity - Some Considerations About Its Benefits And Its Limitations", *Digest of Papers: SAFECOMP '86, 5th International Workshop on Achieving Safe Real-Time Computer Systems*, France, October, 1986.

[11] T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Theoretical And Empirical Studies On Using Program Mutation To Test The Functional Correctness Of Programs", *Proceedings of the Seventh Conference on Principles of Programming Languages*, January, 1980.