TR-86-29

has been revised

and is now TR-87-09

# DATA DIVERSITY: AN APPROACH TO SOFTWARE FAULT TOLERANCE

Paul E. Ammann

John C. Knight

Computer Science Report No. TR-86-29
December 2, 1986

Submitted to *17th International Symposium on Fault-Tolerant Computing*.

# DATA DIVERSITY: AN APPROACH TO SOFTWARE FAULT TOLERANCE†

Paul E. Ammann John C. Knight‡

Department of Computer Science
University of Virginia
Charlottesville, Virginia, 22903

## ABSTRACT

Crucial computer applications such as avionics systems and automated life support systems require extremely reliable software. For a typical system, current proof techniques and testing methods cannot guarantee the absence of software faults, but careful use of redundancy may allow the system to tolerate them. The two primary techniques for building fault-tolerant software are *N*-version programming and recovery blocks. Both methods rely on redundant software written to the same specifications to provide fault tolerance at execution time. These techniques use *design diversity* to tolerate residual faults.

Nothing fundamental limits diversity to design; diversity in the data space may also provide fault tolerance. Two observations promote this view. First, program faults often cause failure only under certain special case conditions. Second, for some applications a program may express its input and internal state in a large number of logically equivalent ways. These observations suggest obtaining a related set of points in the data space, executing the same software on these points, and then employing a decision algorithm to determine system output. In direct analogy to the *N*-version and recovery block strategies, the decision algorithm uses a voter or an acceptance test. This technique uses *data diversity* to tolerate residual faults.

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

Word Count:
Approximate word count, with figures converted to word equivalents: 4800.

---

†Sponsored in part by NASA grant NAG1-605; paper cleared by affiliation.

‡Presenter at FTCS17 if accepted.

## 1. INTRODUCTION

Researchers have proposed various methods for building fault-tolerant software in an effort to provide substantial improvements in the reliability of software for crucial applications. At execution time the fault-tolerant structure attempts to cope with the effect of those faults that survive the development process. The two best-known methods of building fault-tolerant software are *N*-version programming [AVI 78] and recovery blocks [RAN 75]. To tolerate faults, both of these techniques rely on *design diversity*, the availability of multiple implementations of a specification. Software engineers assume that the different implementations contain different designs and thereby, it is hoped, different faults. Since diversity in the design space may provide fault tolerance, diversity in the data space might also. This paper considers *data diversity*, a fault tolerant strategy that complements design diversity.

*N*-version programming requires the separate, independent preparation of multiple (i.e. "*N*") versions of a program for some application. These versions execute in parallel in the application environment; each receives identical inputs, and each produces its version of the required outputs. A voter collects the outputs, which should, in principle, all be the same. If the outputs disagree, the system uses the results of the majority, provided there is one.

The recovery block structure submits the results of an algorithm to an acceptance test. If the results fail the test, the system restores the state of the machine that existed just prior to execution of the algorithm and executes an alternate algorithm. The system repeats this process until it exhausts the set of alternates or produces a satisfactory output.

It is well known that software often fails for special cases in the data space.† In practice, a program may survive extensive testing, work for many cases, and then fail on a special case. The

†For example, see [TUT 81], pp. 347-348.

special case may take the form of what seems to be an obscure set of values in the data. Testing frequently fails to reveal faults associated with special cases precisely because the test harness does not generate the exact circumstances required. A test data set whose values are merely close to the values which cause the program to fail does not uncover the fault.

These observations suggest that if software fails under a particular set of execution conditions, a minor perturbation of those execution conditions might allow the software to work. Other researchers have exploited this property in specific instances. Gray observed that certain faults that caused failure in an asynchronous commercial system did not always cause failure if the same inputs were submitted to a second execution [GRA 85]. The system succeeded on the second execution due to a chance reordering of the asynchronous events. Gray introduced the term ''Heisenbugs'' to describe these faults and their apparent non-deterministic manifestations.

Shepherd, Martin, and Morris have proposed ''temporal separation'' of the input data to a dual version system [MAR 82, MOR 81]. The versions use data from adjacent real-time frames rather than the same frame. Since the versions read data at different times, the data differ. The system corrects for this discrepancy so that it can vote on the outputs of the versions. It is hoped that the use of time-skewed data will prevent the versions from failing simultaneously.

Each of these approaches attempts to avoid faults by operating software with altered execution conditions. Each approach relies upon circumstance to change the conditions. However, execution conditions can be changed deliberately. For example, concurrent systems need not rely on a chance reordering of events. If reordering events might allow a second execution to succeed, then the system should enforce a reordering. Changing the processor dispatching algorithm after state restoration forces a different execution sequence. Similarly, skewing the inputs to the versions in a *N*-version system does not require the passage of time. Inputs can be manipulated algorithmically. Many real-valued quantities have tolerances set by

their specifications, and all values within those tolerances are logically equivalent.

Data diversity is an orthogonal approach to design diversity and a generalization of the work cited above. A diverse-data system produces a set of related data points and executes the same software on each point. A decision algorithm then determines system output. As in the $N$-version and recovery block strategies, the decision algorithm uses a voter or an acceptance test.

*Data re-expression* is the generation of logically-equivalent data sets. A data re-expression algorithm consults the specifications, and possibly a random number generator, to reassign values to variables. A simple data re-expression algorithm for a real variable might alter its value by a small percentage. Clearly, not all applications can employ data diversity. However, real-time control systems often can. Sensors are noisy and inaccurate, and small modifications of sensor values for fault-tolerant purposes may still allow the software to generate acceptable outputs.

Data re-expression extends beyond perturbing real-valued quantities within specified bounds. Any mapping of a program's data that preserves the information content of those data is a valid re-expression algorithm. For instance, suppose that a program processes Cartesian input points and that only the points' relative positions are of interest. A valid re-expression algorithm could translate the coordinate system to a new origin or rotate the coordinate system about an arbitrary point.

This paper describes data diversity as an approach to fault-tolerant software and presents the results of a pilot study. Section 2 discusses the regions of the input space that cause failure for certain experimental programs. Section 3 describes program structures designed to implement data diversity, and section 4 derives a simple, analytic model for a one of these program structures, the retry block. Results of the pilot experiment using a retry block appear in section 5, and section 6 presents conclusions.

## 2. FAULT REGIONS

The input data for most programs comes from hyperspaces of very high dimension. For example, a program may read and process a set of twenty floating-point numbers, and so its input space has twenty dimensions. In many cases the number of dimensions in the space varies dynamically because the amount of data that a program processes varies for different executions.

The region(s) of the input space that cause program failure are an important characteristic of the program. The volume, shape, and distribution of such regions, which we call *failure regions*, determine both the program's failure probability and the effectiveness of data diversity. The fault tolerance of a system employing data diversity depends upon the ability of the re-expression algorithm to produce data points that lie outside of a failure region, given an initial data point that lies within a failure region. The program executes correctly on re-expressed data points only if they lie outside a failure region. If the failure region has a small cross section in some dimensions, then re-expression should have a high probability of translating the data point out of the failure region.

Knowledge of the geometry of failure regions gives insight into the possible performance of data diversity. We have obtained two-dimensional cross sections of several failure regions for faults in programs used in a previous experiment [KNI.LEV 86]. These cross sections were obtained by varying two inputs across a uniform grid while all other inputs remain fixed. Figure 1 shows cross sections from two separate faults.[†] The solid lines show where the correct output of the program changes, and the small dots show grid points where the faulty program produced the wrong output.

[†]The specific faults are 6.2 and 6.3 [BRI.KNI.LEV 86].

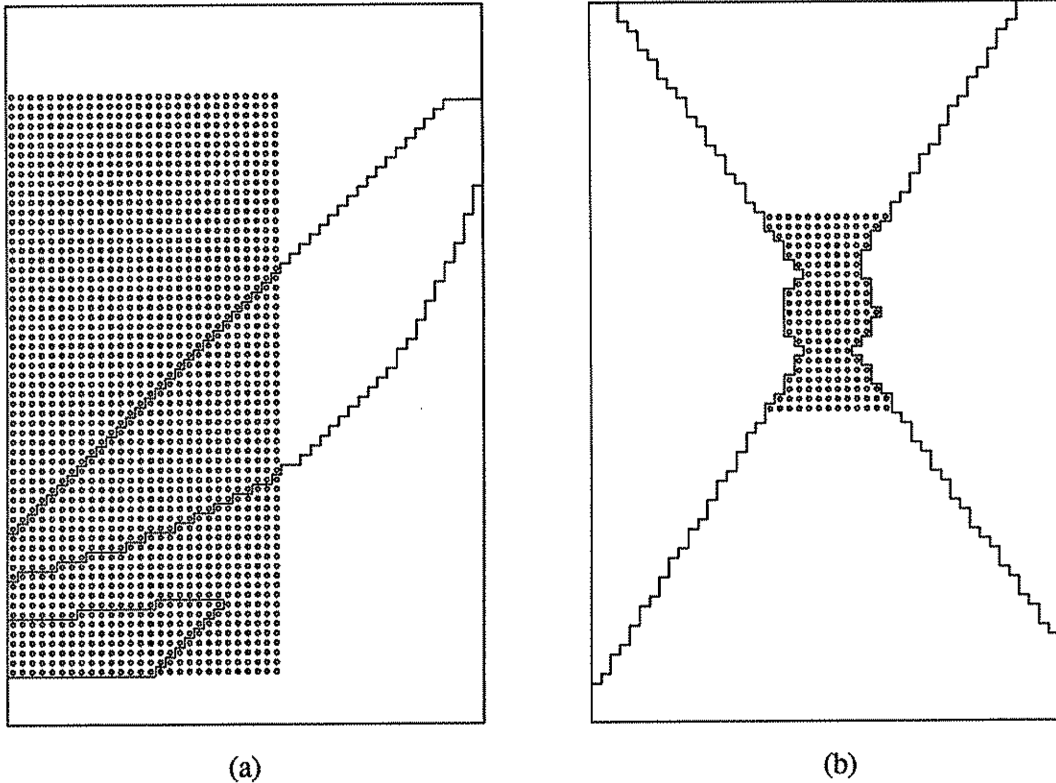


**Figure 1: Two Dimensional Cross Sections of Two Failure Regions.**

The $(x, y)$ coordinates of a set of points in an imaginary radar track form the input space. Since each $(x, y)$ pair supplies two dimensions, the input space has twice as many dimensions as radar points. The radar points are distributed so that the $x$ and $y$ coordinates each span the real range –40..40[†], and all radar points for the original experiment were rounded to the nearest 0.1.

The space from which cross section (a) was taken has 30 dimensions corresponding to 15 radar points. Cross section (a) was obtained by varying coordinates $x_1$ and $x_9$ with a grid point separation of 0.2. The space from which cross section (b) was taken has 18 dimensions

[†]The distribution is not uniform, and is defined in [NAG.SKR 82].

corresponding to 9 radar points. Cross section (b) was obtained by varying coordinates $x_6$ and $y_6$ with a grid point separation of 0.000001. For both, all other inputs were held fixed. The area of cross section (a) is $4 \times 10^{10}$ times larger than the area of cross section (b).

The cross sections shown are typical for these programs. This small sample illustrates two important points. First, at the resolution used in scanning, these failure regions are solid rather than diffuse; for no points interior to the region's boundary will the program execute correctly. Second, since failure regions vary greatly in size, exiting failure regions varies greatly in difficulty.

## 3. PROGRAM STRUCTURES

A *retry block* is a modification of the recovery block structure that uses data diversity instead of design diversity. Figure 2 shows the semantics of a retry block. Rather than the multiple alternate algorithms used in a recovery block, a retry block uses only one algorithm. A retry block's acceptance test has the same form and purpose as a recovery block's acceptance test. A retry block executes the single algorithm normally and evaluates the acceptance test. If the acceptance test passes, the retry block is complete. If the acceptance test fails, the algorithm executes again after the data has been re-expressed. The system repeats this process until it violates a deadline or produces a satisfactory output.

An *N-copy* system is a modification of an *N*-version system that uses data diversity instead of design diversity. Figure 3 shows the semantics of an *N*-copy system. *N* copies of a program execute in parallel; each on a slightly modified set of data. An *N*-copy system votes on results in an analogous manner to an *N*-version system. Reconciling disagreement as copies traverse output space boundaries and preventing divergence as copies follow different execution paths

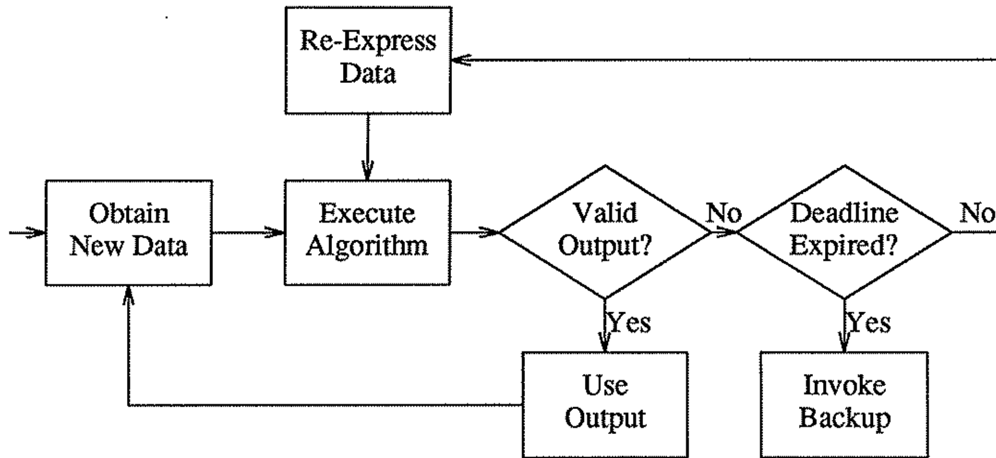


**Figure 2: Retry Block.**

complicate voting in an *N*-copy system. The dashed line in figure 3 labeled "Synchronization Information" symbolizes this difficulty.

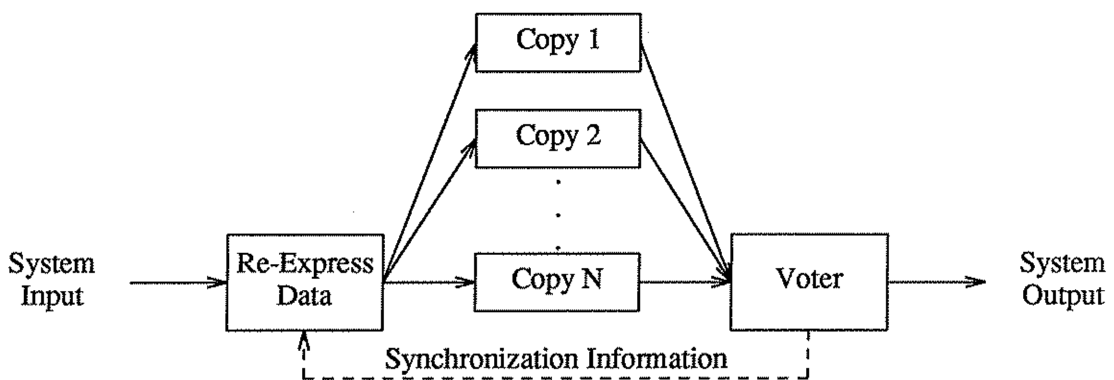


**Figure 3: N-Copy Programming.**

Both retry blocks and $N$-copy systems are substantially less expensive to develop than their diverse-design counterparts. Many hybrid structures incorporating both design and data diversity are possible. The remainder of the paper concentrates exclusively on retry blocks and does not consider hybrid structures or $N$-copy systems further.

## 4. SIMPLE ANALYTIC MODEL FOR A RETRY BLOCK

Figure 4 shows the model we have used to predict the success of a retry block assuming a perfect acceptance test. On a single execution under operational conditions, the program used in the construction of the retry block has a probability of failure, $p$. The random variable $Q$ gives the probability that a re-expressed data point causes failure given that the initial point caused failure. $Q$ is a random variable because its value depends upon the geometry of the failure region

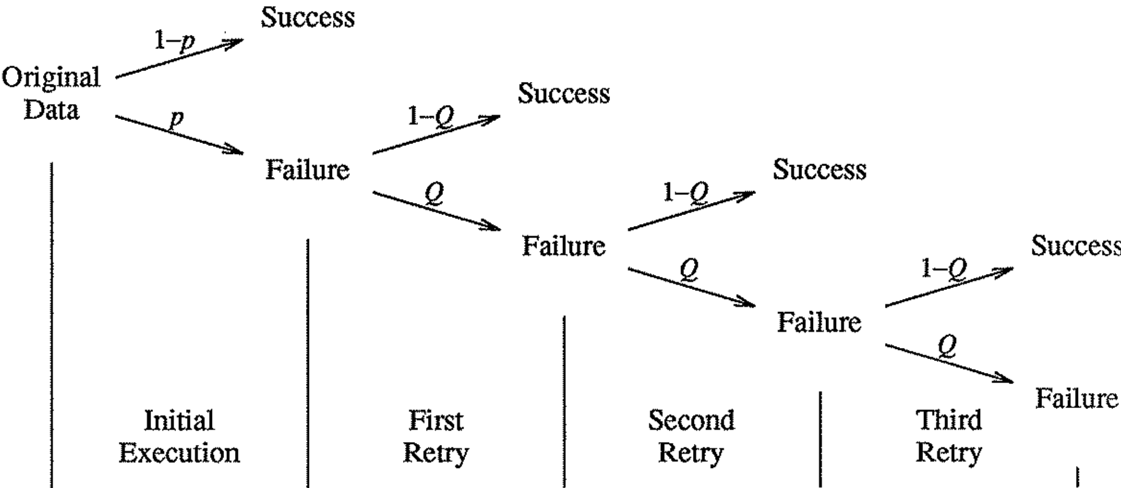


**Figure 4: Retry Block Model Assuming A Perfect Acceptance Test.**

in which the original data point lies, the location of the data point within that region, and the algorithm used to re-express the data. Since $Q$ is a probability, it assumes values in the range 0..1. We denote the distribution function for $Q$ as $F_Q(q)$ where $F_Q(q) = prob(Q \leq q)$. The corresponding density function† for $Q$ is $f_Q(q)$ where $f_Q(q) = \frac{d}{dq} F_Q(q)$.

The probability that the retry block fails when using $n$ retries, denoted $prob(fail)$, is $prob(fail) = pQ^n$. The probability that a retry block will succeed in $n$ or fewer retries is one minus this probability. $prob(fail)$ is a random variable since it is a function of $Q$. Its expected value is:

$$E[prob(fail)] = p \int_0^1 q^n f_Q(q) dq .$$

Since the integral in the expression for $E[prob(fail)]$ is multiplied by $p$, the failure probability of the program, the integral describes how the performance of a retry block improves upon the performance of a program. Thus the quantity $\int_0^1 q^n f_Q(q) dq$ is the average factor by which the use of a retry block reduces the probability of system failure.

## 5. EMPIRICAL RESULTS FOR A RETRY BLOCK

We have obtained empirical evidence of the expected performance of data diversity on some of the known faults in the Launch Interceptor Programs produced for the Knight and Leveson experiment [KNI.LEV 85]. As noted in Section 2, one of the inputs to the programs is a list of $(x, y)$ pairs representing radar tracks. To employ data diversity in this application, we assume that data obtained from the radar is of limited precision. The data re-expression

†We assume that $Q$ is continuously distributed. The extension to to discretely distributed $Q$ is straightforward.

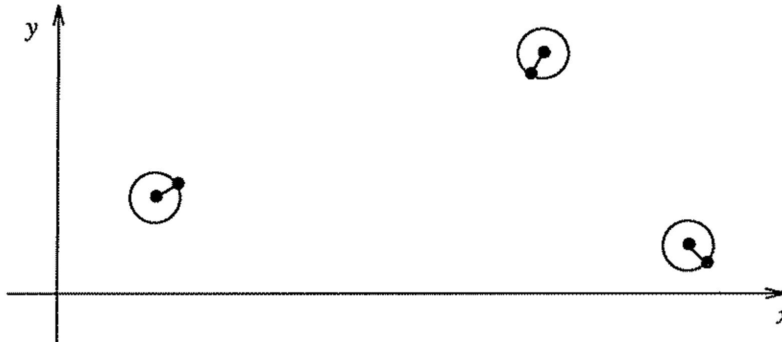


**Figure 5: Re-Expression Of Three Radar Points.**

algorithm that we used moved each $(x, y)$ point to a random location on the circumference of a circle centered at $(x, y)$ and of some small, fixed radius. Figure 5 shows how this algorithm re-expresses a set of three radar points. Many other valid re-expression algorithms are possible.

The experiment measured the performance of data diversity in the form of the retry block on the Launch Interceptor Programs, and how this performance was affected by two parameters. The first parameter studied was the radius of displacement used in the data re-expression algorithm. The second parameter was the effect of the re-expression algorithm on different faults.

Figures 6 and 7 show the effects of these two parameters on estimated $F_Q(q)$ functions.[†] Each distribution function in these figures corresponds to a particular displacement value and fault. A given point, $(q, F_Q(q))$, is the observed probability $F_Q(q)$ that a program executing on a re-expressed data point will have at most a probability of failure, $q$. Distribution functions that rise rapidly imply better performance for data diversity since they indicate a higher probability that re-expression will arrive at a point outside the failure region. For example, a function

[†] As with parameters in many other performance models, $F_Q(q)$ cannot be determined analytically. The functions shown are empirical estimates of $F_Q(q)$.

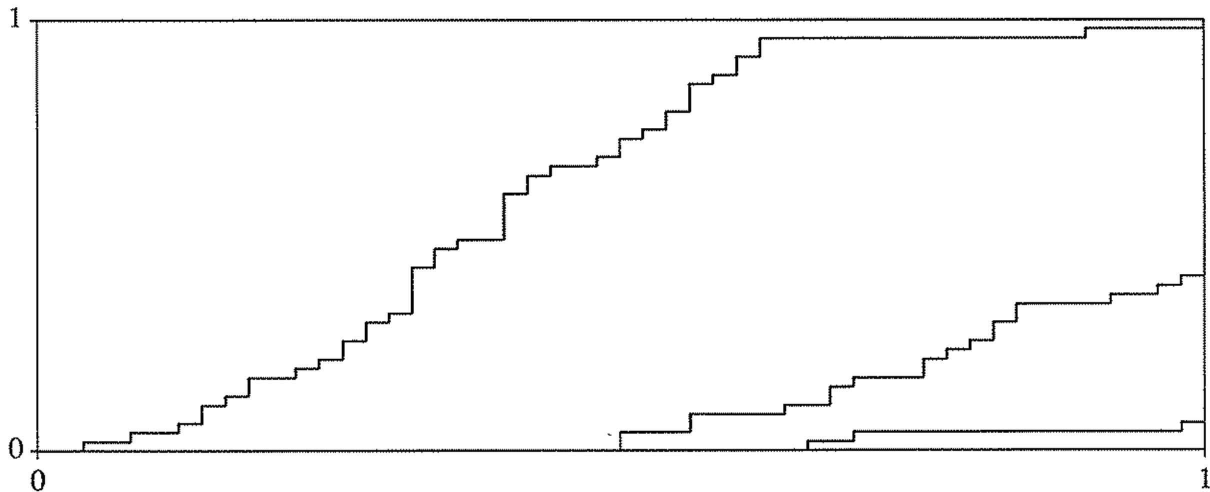


**Figure 6: Fault 9.1 Sample $F_Q(q)$ For Three Displacement Values.**

containing the point (0.05, 0.95) means that the probability is 0.95 that the re-expressed data point will cause failure with a probability of 0.05 or less.

Figure 6 shows the observed $F_Q(q)$ of a single fault for three values for the radius of displacement. From left to right on the graph, the displacement values are 0.1, 0.01, and 0.001. As would be expected, larger displacement values in the data re-expression algorithm have the effect of making the re-expressed data point less likely to cause failure. These displacements are relatively small compared the the range of values that the radar points could assume.

Figure 7 shows the observed $F_Q(q)$ for three different faults† using a fixed displacement of 0.01 in the re-expression algorithm. These three faults were chosen to show the wide variation from fault to fault on the distribution of the probability that a re-expressed data point will cause failure. The leftmost function rises rapidly, which indicates that data diversity will tolerate the

†From left to right on the graph, the faults are 8.1, 7.1, and 6.2. [BRI.KNI.LEV 86].

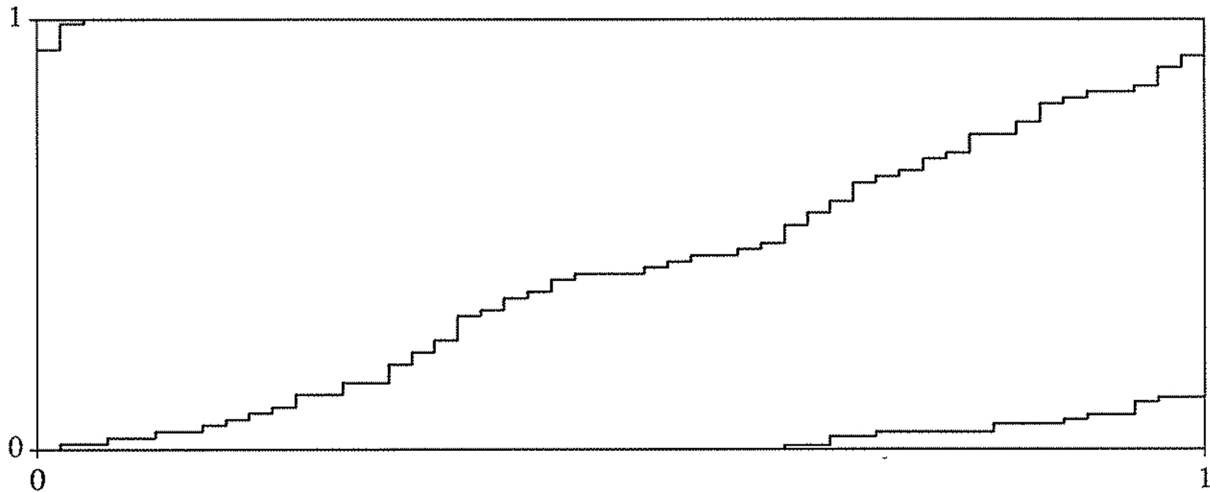


**Figure 7: Sample $F_Q(q)$ For Three Different Faults At 0.01 Displacement.**

associated fault well at the given displacement value. The rightmost function rises slowly, which indicates that data diversity requires a better re-expression algorithm to tolerate the associated fault.

The table in figure 8 shows the performance obtained using retry blocks for various faults under various conditions. Each table entry is an expected value multiplier for the probability of system failure. The table shows results for retry blocks with 1, 2, or 3 retries and displacement values of 0.1, 0.01, or 0.001. An entry in figure 8 means that the failure probability associated with a given fault is reduced by the factor shown. For instance, the value of 0.43 found in the middle of the table means that the failure probability associated with fault 7.1 was reduced by a factor of 0.43 when the displacement in the re-expression algorithm was 0.01 and the number of retries was 2. Similarly, a table entry of 0.00 means that the effects of the associated fault were eliminated and an entry of 1.00 means that data diversity had no effect. These values were obtained from sample distributions such as those illustrated in figures 5 and 6. The model

| Retries | 1 | | | 2 | | | 3 | | |
|---|---|---|---|---|---|---|---|---|---|
| Displacement | 0.001 | 0.01 | 0.1 | 0.001 | 0.01 | 0.1 | 0.001 | 0.01 | 0.1 |
| Fault | | | | | | | | | |
| 6.1 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 6.2 | 1.00 | 0.98 | 0.87 | 1.00 | 0.96 | 0.81 | 0.99 | 0.94 | 0.75 |
| 6.3 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 7.1 | 0.92 | 0.59 | 0.26 | 0.87 | 0.43 | 0.11 | 0.80 | 0.29 | 0.03 |
| 8.1 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 8.2 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 9.1 | 0.99 | 0.90 | 0.39 | 0.97 | 0.83 | 0.19 | 0.97 | 0.74 | 0.07 |

**Figure 8: Expected Value Multipliers For the Probability of System Failure.**

outlined in section 4 was used to calculate the values shown. The integral corresponding to the multiplier was approximated by summing data obtained from a set of points known to be located in the failure region for the particular fault. For each point in the set, a value for $Q$ was estimated from 50 applications of the re-expression algorithm.

Figure 9 displays graphically the non-zero entries from figure 8. For these cases, the extent to which a retry block reduces the failure probability of a given fault depends upon the specific fault, the number of retries, and the displacement value used in the re-expression algorithm.

## 6. CONCLUSIONS

We have described the general concept of data diversity as a technique for software fault tolerance and defined the retry block and $N$-copy programming as two possible approaches to its

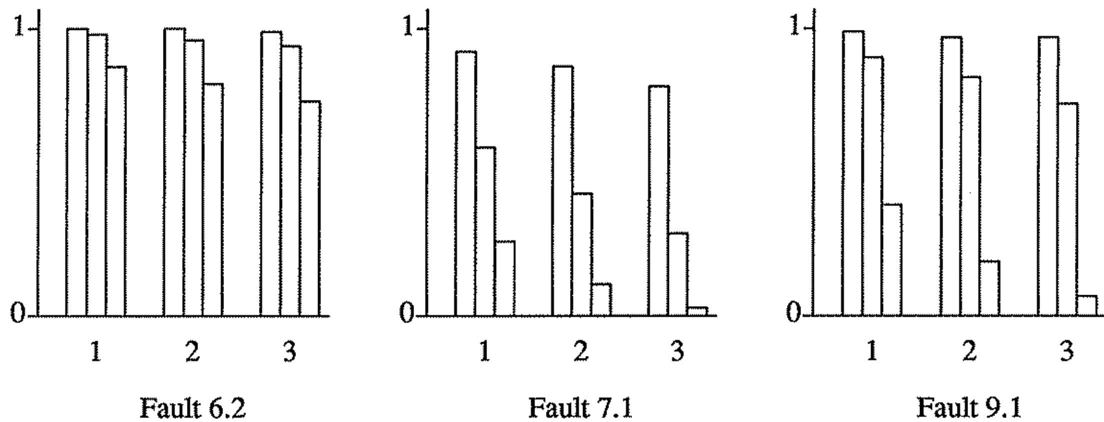


**Figure 9: Non-Zero Values From Figure 8.**

implementation. We have presented the results of a pilot study of data diversity in the form of the retry block. Although the overall performance of the retry block varied greatly, we observed a large reduction in failure probability for some of the faults examined in the study. In several cases the retry block completely eliminated the effects of a fault.

The success of data diversity depends, in part, upon developing a data re-expression algorithm that is acceptable to the application yet has a high probability of generating data points outside of a program's failure region. Many applications could use simple re-expression algorithms similar to the one employed in this study. For example, sensors typically provide data with relatively little precision and small modifications to those data will not affect the application.

Implementing a retry block requires a suitable acceptance test. This is a well-known problem for the recovery block; any techniques developed for the recovery block apply directly to the retry block.

Compared with design diversity, data diversity is relatively easy and inexpensive to implement. Data diversity requires only a single implementation of a specification, although additional costs are incurred in the data re-expression algorithm and the decision procedure.

Data diversity is orthogonal to design diversity. The strategies are not mutually exclusive and various combinations are possible. For example, an *N*-version system in which each version was an *N*-copy system could be built for very little additional cost over an *N*-version system. The way in which data diversity should be used and how it should be integrated with design diversity is an open question.

## 7. ACKNOWLEDGEMENTS

It is a pleasure to acknowledge Earl Migneault for thoughts about the failure regions and data diversity as a general concept. Sue Brilliant's work in identifying the program faults and matching those faults with failure cases made it possible to carry out the empirical parts of this research. We are also pleased to acknowledge Larry Yount for a discussion about time-skewed inputs. This work was sponsored in part by NASA grant NAG1-605.

## REFERENCES


[AVI 78]
A. Avizienis ''Fault-Tolerance: The Survival Attribute of Digital Systems'', *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1109-1124.

[BRI 85]
S.S. Brilliant, ''Analysis of Faults in a Multi-Version Software Experiment'', MS Thesis, University of Virginia, May, 1985.

[BRI.KNI.LEV 86]
S.S. Brilliant, J.C. Knight, N.G. Leveson, ''Analysis of Faults in an *N*-Version Software Experiment'', University of Virginia Technical Report No. TR-86-20, September, 1986.

[GRA 85]
J. Gray, ''Why do Computers Stop and What Can Be Done About It?'', Tandem Technical Report 85.7, June 1985.

[KNI.LEV 86]
J.C. Knight, and N.G. Leveson, ''A Large Scale Experiment in N-Version Programming'' *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1, January 1986.

[MAR 82]
D.J. Martin, ''Dissimilar Software in High Integrity Applications In Flight Control'', 1982 AGARD Conference Proceedings #330, Software for Avionics, pp. 36-1 to 36-13.

[NAG.SKR 82]
''Software Reliability: Repetitive Run Experimentation and Modeling'', NASA Report CR-165836, Langley Research Center, February, 1982.

[MOR 81]
M.A. Morris, ''An Approach to the Design of Fault Tolerant Software'', MSc Thesis, Cranfield Institute of Technology, September, 1981.

[RAN 75]
B. Randell, ''System Structure for Software Fault Tolerance'', *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975.

[TUT 81]
''Tutorial: Software Testing and Validation Techniques'', 2nd Ed., IEEE Computer Society Press, 1981.