# DATA DIVERSITY: AN APPROACH TO SOFTWARE FAULT TOLERANCE

Paul E. Ammann
John C. Knight

Computer Science Report No. TR-87-09
June 10, 1987

Submitted to: IEEE Transactions on Computers
(Revised TR-86-29)

# DATA DIVERSITY: AN APPROACH TO SOFTWARE FAULT TOLERANCE

Paul E. Ammann and John C. Knight

*Affiliation Of Authors*

Department of Computer Science

University of Virginia

Charlottesville

Virginia, 22903

*Financial Acknowledgement*

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

*Index Terms*

Data diversity, design diversity, *N*-copy programming, *N*-version programming, software faults, software fault tolerance, recovery blocks, retry blocks.

*Address For Correspondence*

John C. Knight

Department of Computer Science

Thornton Hall

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. These techniques use *data diversity* to tolerate residual faults.

# I 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 [1] and recovery blocks [9]. 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 use different designs and thereby, it is hoped, contain different faults. The fact that diversity in the design space may provide fault tolerance suggests that 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, that 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 [10], 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 that 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 [4]. 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 [6,7]. The versions use data from adjacent real-time frames rather than the same frame. Since the versions read sensor data at different times, the data tend to 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 of these points. A decision algorithm then determines system output.

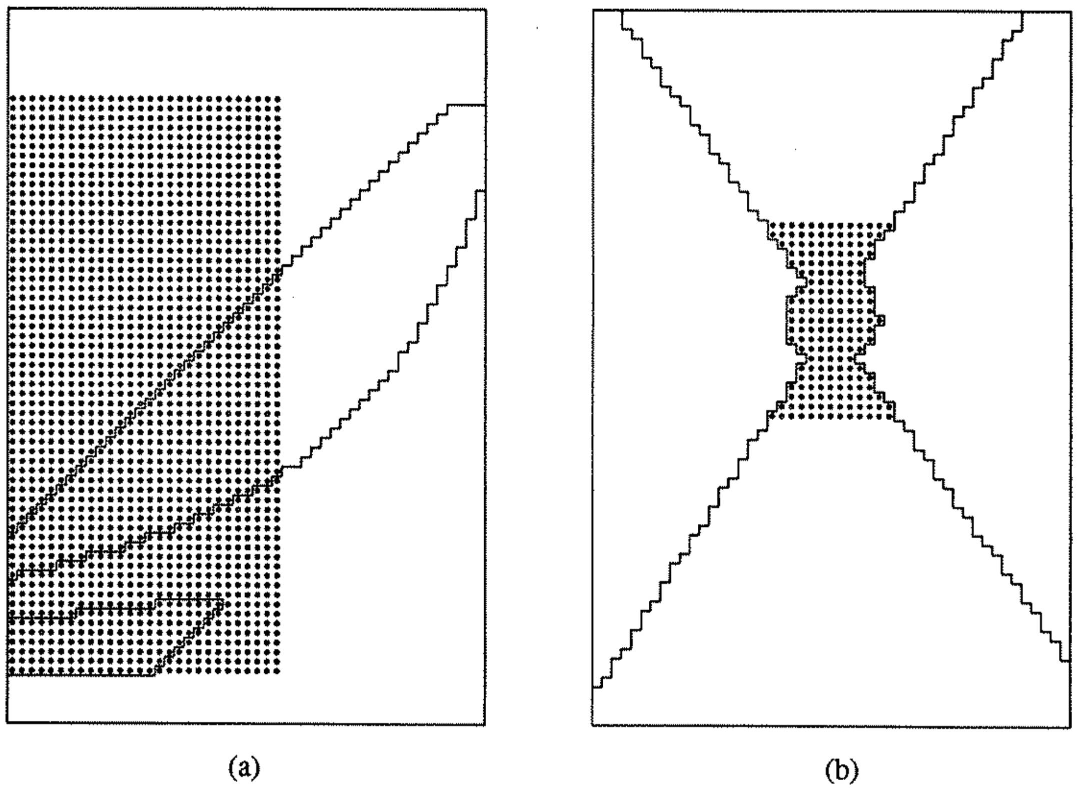
This paper describes data diversity as an approach to fault-tolerant software and presents the results of a pilot study. Section II discusses the regions of the input space that cause failure for certain experimental programs. Section III examines *data re-expression*, the way in which alternate input data sets can be obtained. Section IV describes the *retry block*, the data-diverse equivalent of the recovery block, and presents a model of the retry block together with some empirical results. Section V describes *N-copy programming*, the data-diverse equivalent of *N*-version programming, presents a simple model, and gives some empirical results. Section VI contains conclusions.

## II FAILURE 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 *failure domain* of a program is the set of input points that cause program failure [3]. We call a failure domain along with its geometry a *failure region*. A failure region describes the distribution of points in the failure domain and determines 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 dimension(s), 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 [5]. The programs are simple hypothetical anti-missile missile launch decision programs that determine their output based on relationships



(a) (b)

**Figure 2.1: Two Dimensional Cross Sections of Two Failure Regions.**

among sets of points from an imaginary radar track. For example, one test determines whether certain sets of radar points fit into a circle of given radius.

Figure 2.1 shows cross sections obtained by varying two inputs across a uniform grid while all other inputs remained fixed. The cross sections are 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 produces the wrong output.

The $(x, y)$ coordinates of a set of points in the 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 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 particular failure regions are locally continuous. Second, since the failure regions vary greatly in size, exiting them varies greatly in difficulty.

*The specific faults are 6.2 and 6.3 [2].
**The distribution is not uniform, and is defined in [8].

# III DATA RE-EXPRESSION

At its simplest, *data re-expression* is the generation of logically equivalent data sets. Figure 3.1 illustrates this basic form of data re-expression. An input $x$ given directly to a program $P$ produces an output $P(x)$. Alternatively, a re-expression algorithm $R$ transforms the original input $x$ to produce a new input $y$, where $y = R(x)$. The input $y$ may contain exactly the same information as the input $x$, but in a different form, or $y$ may approximate the information in input $x$. The program $P$ operates on the re-expressed input $y$ to produce $P(y)$. $P$ and $R$ determine the relationship between $P(x)$ and $P(y)$. Data diversity can tolerate faults when $P(y)$ is a useful output but $P(x)$ is not.

## Requirements For The Re-Expression Algorithm

Since it is the outputs that are ultimately important to any given application, the requirements for a re-expression algorithm can be derived from characteristics of the outputs. For a specific input $x$, figure 3.2 shows the sets of input points, $y$, that are of interest given a re-expression algorithm of the type shown in figure 3.1.

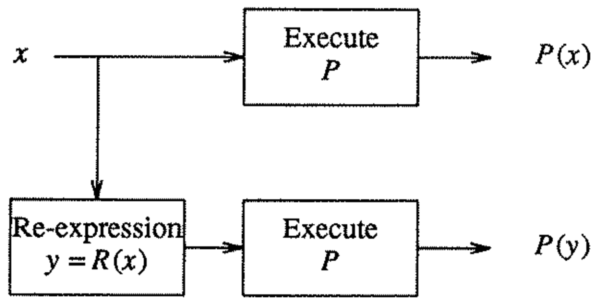


**Figure 3.1. Re-expression.**

One set in figure 3.2, called $I$, the *identical output set,* is defined $I = \{y \mid Correct(x, P_{correct}(y))\}$ [*] and contains all inputs, $y$ for which the correct output is the same as the correct output given $x$ as an input. The identical output set is a desirable set from which to select re-expressed inputs $y$, since the use of data re-expression in such cases is transparent outside the program. Implementing an error detection mechanism is simplified when all re-expressed data points are members of the identical output set.

It is possible that the single output produced by $P_{correct}$ for a specific input $x$ is not the only acceptable output for the application. Different acceptable outputs may arise at boundaries in the output space. For example, if an input $x$ is a real variable of limited precision then different outputs resulting from inputs sufficiently near a boundary are indistinguishable from the viewpoint of the application.

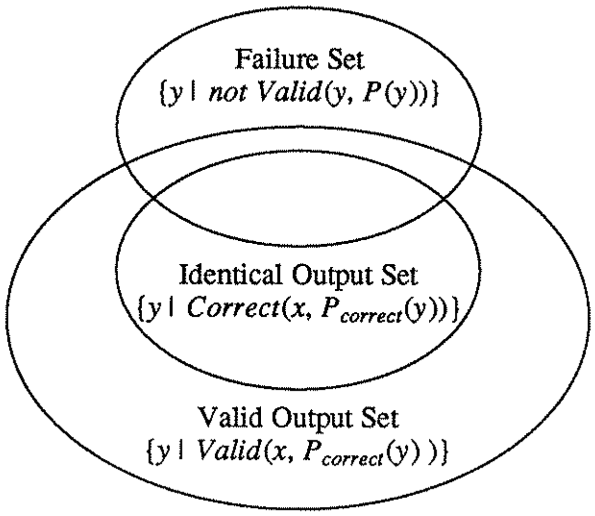


**Figure 3.2. Sets in the Output Space for Given Input $x$.**

[*] The predicate *Correct(in, out)* is true iff *out* is the output demanded by the specification for input *in*. The symbol $P_{correct}$ represents a correctly implemented specification; $P_{correct}(x)$ is the correct output for input $x$.

The identical output set, $I$, is a subset of the *valid output set*, $V$, $V = \{y \mid Valid(x, P_{correct}(y))\}$.* $V$ is the set of all inputs, $y$, for which a correct program produces an acceptable output given $x$ as the actual input. Although it is easier to produce values in $V$ than in $I$, error detection with members of $V$ is more difficult. Also, $V$ is a function of the actual input $x$. The fact that a given $x$ may be re-expressed as $y$ does *not* imply that if $P(x)$ happens to be an acceptable output for some other input, say $z$, then $P(y)$ is also a acceptable output. Formally, $Valid(z,\ P(x))$ and $y = R(x)$ do not imply $Valid(z,\ P(y))$.

The third set, $F$, called the *failure set*, is $F = \{y \mid not\ Valid(y, P(y))\}$, and represents all inputs $y$ for which the program fails to produce a correct or acceptable output. Although elements of $F$ are, by definition, not enumerated, the effectiveness of data diversity is determined by the proportion of re-expressed points that lie in $F$.

Although figure 3.2 describes the sets of interest to a re-expression algorithm, it does not give a construction for the re-expression algorithm. For example, a desirable re-expression algorithm would be able to produce any data point in the identical output set. However, without a correct program, the elements in these sets cannot be enumerated. Even though we are only concerned about relationships among the outputs of a program given various inputs, only inputs that can be produced by a re-expression algorithm are of interest. The re-expression algorithm is limited to making well-defined transformations of the input. There may be a large number of inputs available that produce the desired output, but only those inputs that have a well defined relationship to the original input can be produced by a re-expression algorithm.

*The predicate $Valid(in,\ out)$ is true iff *out* is a valid or acceptable output for input *in*.

## Exact and Approximate Re-expression Algorithms

Figure 3.2 identifies two classes of re-expression algorithms. The first class produces elements in $I$ (up to numerical error); these algorithms are termed *exact.* The second class yields elements in $V$; these algorithms are termed *approximate*. Although exact algorithms are desirable from the viewpoint of error detection, approximate algorithms may be easier to produce and may have a better chance of escaping a failure region. Exact re-expression algorithms may have the defect of preserving precisely those aspects of the data that cause failure.

As an example of an exact re-expression algorithm, suppose that a program processes Cartesian input points and that only the points' relative positions are relevant to the application. A valid re-expression algorithm could translate the coordinate system to a new origin or rotate the coordinate system about an arbitrary point.

Any mapping of a program's data that preserves the information content of the data is a valid re-expression algorithm. A simple approximate data re-expression algorithm for a real variable might alter its value by a small percentage. The allowable percentage by which the variable's value can be altered would be determined by the specification, possibly as a result of a known sensitivity in a particular hardware sensor.

Up to now, we have considered re-expression algorithms that depend on numeric manipulation. It might appear that many non-numeric applications are not candidates for fault tolerance through data diversity because numeric re-expression cannot be used. The source program input to a compiler, for example, is a character string, and any changes to that character string will almost certainly change the meaning of the program.

Data can take other forms, however, and data diversity can be applied successfully to other applications. Consider a compiler with a conventional multi-pass organization. Although the initial representation of a source program is a character string, the program may be represented in many ways during compilation, for example, as a tree. There are many transformations that can be applied to a tree that preserve semantics. A compiler employing data diversity for its later passes could be constructed by executing these later passes on several different copies of the tree obtained by semantics preserving transformations.

Similarly, the order of storage allocation in an activation record is usually determined by the programmer's order of declaration. In practice, this order need not be preserved, and a set of semantically equivalent internal representations of a program can be obtained by shuffling the order of variables in activation records.

Combining tree transformations, data storage reordering, and code storage reordering, *i.e.*, generation of code for subprograms in an arbitrary order, provides considerable diversity in the data processed by large fractions of a conventional compiler. These approaches to re-expression are exact in that, although the code generated by the compiler may be different and therefore not amenable to any simple selection algorithm, the effects of these programs should be identical, and so simple voting can be used for selection during execution of the programs generated by the

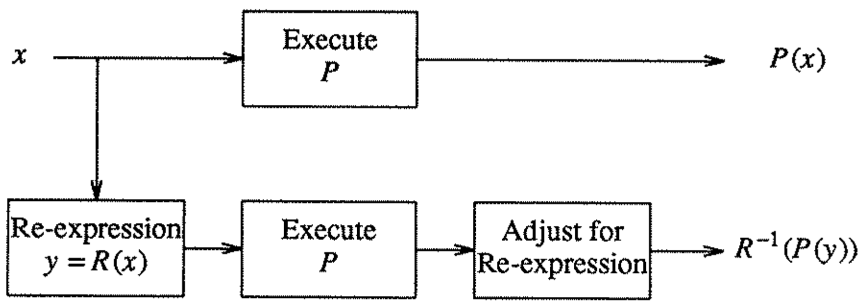


**Figure 3.3. Re-expression with Post Execution Adjustment.**

compiler.

The structure shown in figure 3.3 allows the re-expression algorithm to produce more diverse inputs than is possible with the structure shown in figure 3.1. Some form of correction is performed on the output after the program is run to undo the distortion caused by re-expression. If the distortion induced by $R$ can be removed after execution, the approach permits major changes to the inputs and allows copies of the program to operate in widely separated regions of the input space. As an example, consider a program which computes intersections of line segments. Re-expression might alter the representation of the input by multiplying by a non-singular matrix. The distortion could be recovered by multiplying the program output by the inverse of the matrix.

The approach illustrated in figure 3.3 allows for both exact and approximate corrections to the outputs. The correction might simply reduce the error introduced by $R$. Re-expression algorithms with corrections may be approximate, yet be improved over the approximations possible with re-expression algorithms without post execution adjustment.

Another approach to re-expression is shown in figure 3.4. An input may be decomposed into a related set of inputs. The program is run on each part of the input, and then the results are recombined. Again, as in figure 3.3, figure 3.4 allows for both exact and approximate re-expression algorithms.

As an example of an exact re-expression algorithm of the type shown in figure 3.4, consider a data-diverse computation of the sine function. Assume we have an implementation of the sine function that is known to work over a large percentage of it inputs; the failure probability for the sine function on a randomly chosen input, $x$, is $p$, where $p \ll 1$. To compute $\sin(x)$, we use the two trigonometric identities

$$\sin(a+b) = \sin(a)\cos(b) + \cos(a)\sin(b)$$

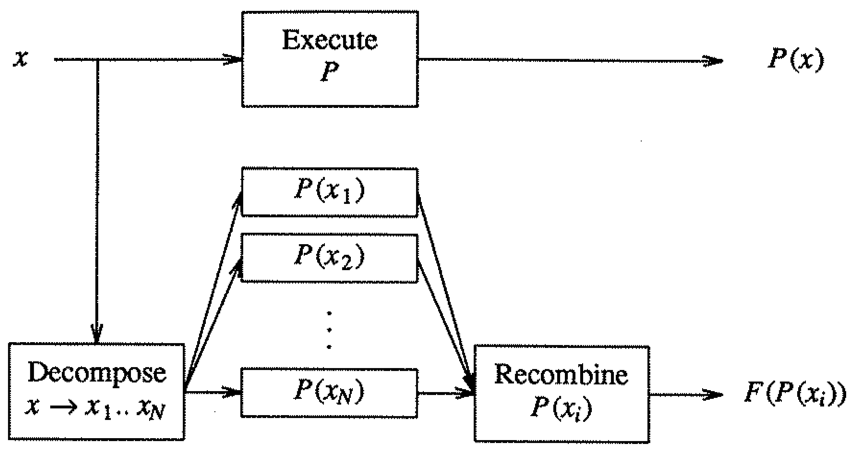


**Figure 3.4. Re-expression via Decomposition and Recombination.**

$$\cos(a) \quad = \sin(\pi/2-a)$$

to rewrite

$$\sin(x) \quad = \sin(a)\sin(\pi/2-b) + \sin(\pi/2-a)\sin(b)$$

in which $a$ and $b$ are any two real numbers such that $a+b = x$. Suppose that $\sin(x)$ is computed using three independent decompositions for $x$ obtained by using three different values each for $a$ and $b$, and that a simple majority voter selects the output. Using the worst case assumption that all incorrect answers appear identical to the voter, a conservative estimate for the probability of computing an incorrect value for $\sin(x)$ can be shown to be on the order of $48p^2$.

In general, a re-expression algorithm must be tailored to the application at hand. Producing a re-expression algorithm requires a careful analysis of the type and magnitude of re-expression appropriate for each candidate datum. Simpler re-expression algorithms are more desirable than complex ones since they are less likely to contain design flaws.

# IV RETRY BLOCKS

A *retry block* is a modification of the recovery block structure that uses data diversity instead of design diversity. Figure 4.1 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.

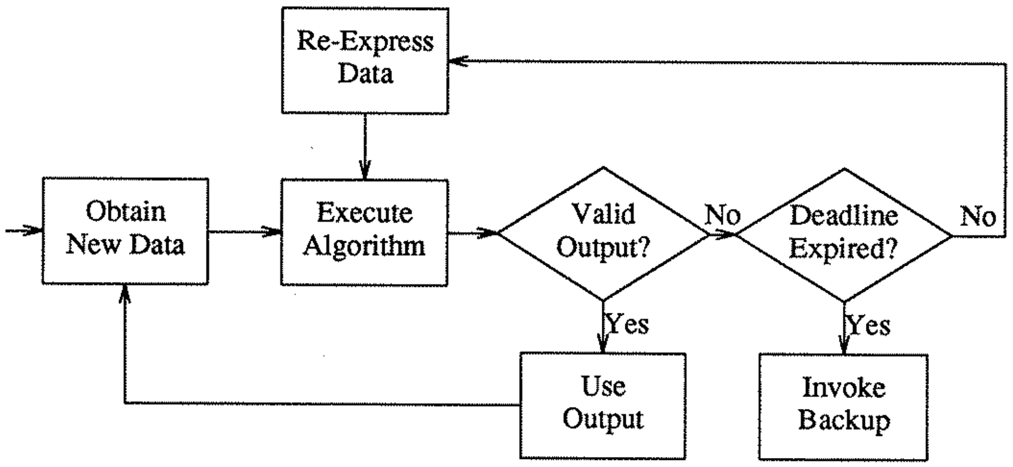


**Figure 4.1: Retry Block.**

## Simple Model for a Retry Block

Figure 4.2 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 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 $F_Q(q) = prob(Q \le q)$. The corresponding density function* for $Q$ is $f_Q(q)$ where $f_Q(q) = \frac{d}{dq} F_Q(q)$.

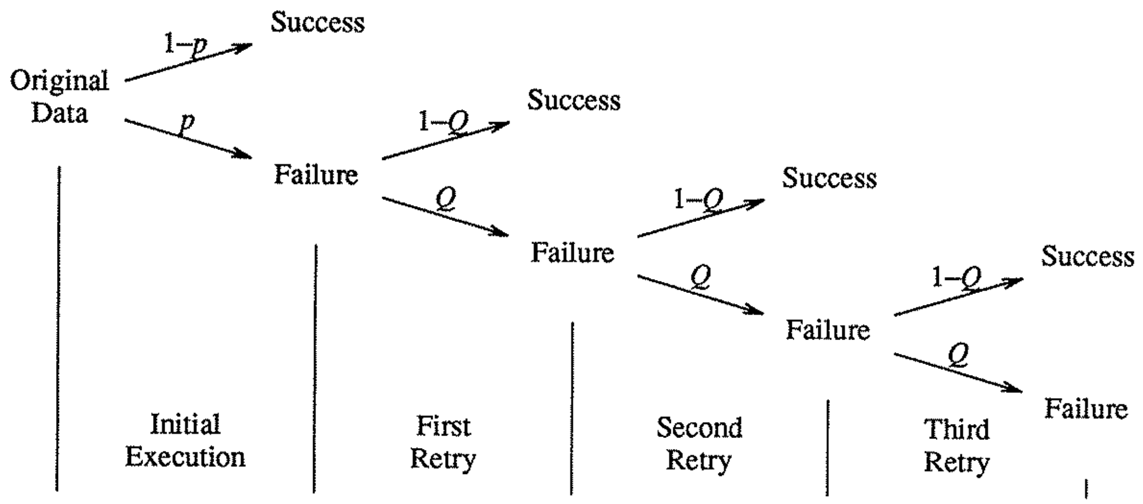


**Figure 4.2: Retry Block Model Assuming A Perfect Acceptance Test.**

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

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 failure. We next estimate $F_Q(q)$ for some of the programs discussed in section 2 and approximate the integral in $E[prob(fail)]$ to predict the reductions in failure probability for retry blocks constructed from those programs.

## Two Forms Of Acceptance Test

In this experiment we have collected data for the performance of the retry block under two different implementations of an acceptance test. First we consider an acceptance test that can determine whether the program has produced $P_{correct}(x)$ given $x$ as input. If the acceptance test rejects the program's output for input $x$ and $x$ is re-expressed as $y$, then the acceptance test demands $P_{correct}(y)$ as the only acceptable output. This acceptance test rejects all outputs for which the program does not satisfy the specification. We call this type of acceptance test a *specification* acceptance test.

The second form of acceptance test that we consider accepts any output that is valid anywhere in the re-expression region of the input space. The re-expression region is that part of the input space from which re-expressed inputs can be selected. The acceptance test determines the validity of the output by examining the output of the $P_{correct}$ for input points in the re-expression region of input $x$. We call this type of acceptance test a *valid output* acceptance test.

The first set of empirical results for the retry block were obtained using a specification acceptance test; the second set were obtained using a valid output acceptance test.

## Empirical Results For A Retry Block - Specification Acceptance Test

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 [5]. As noted in Section II, 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 algorithm 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. The re-expression algorithm is approximate according to the definitions given in Section III. Figure 4.3 shows how this algorithm re-expresses a set of three radar points. Many other re-expression algorithms are possible.

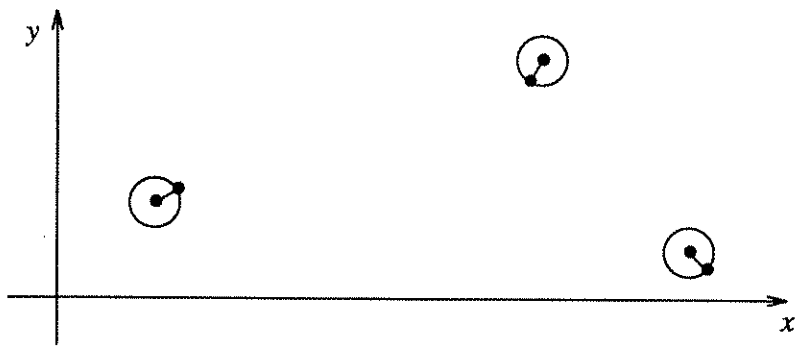


**Figure 4.3: Re-Expression Of Three Radar Points.**

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 4.4 and 4.5 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 containing the point (0.05, 0.95) means that the probability is 0.95 that the re-expressed data

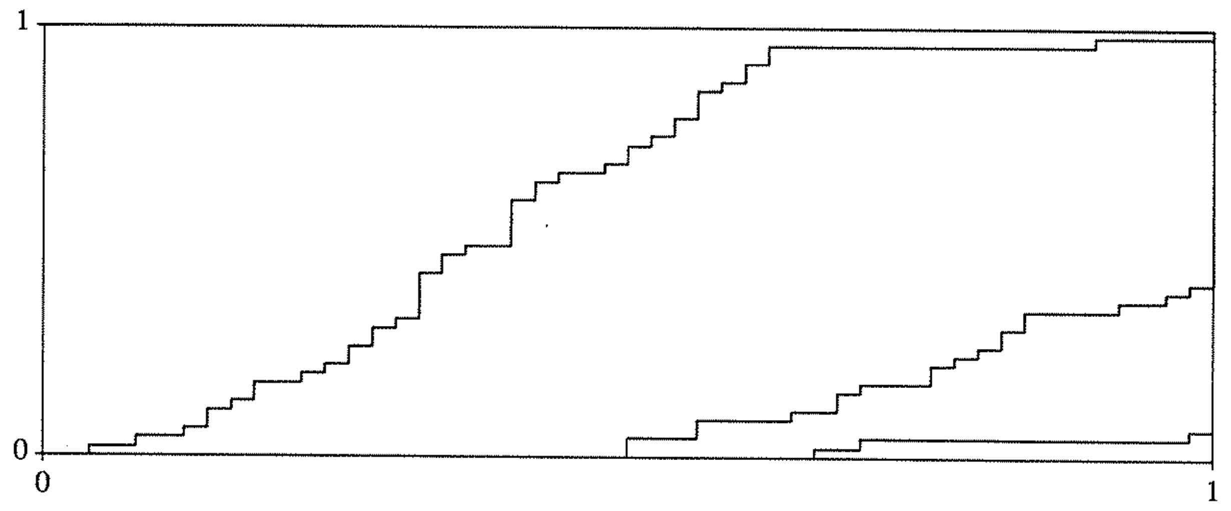


**Figure 4.4: Fault 9.1 Sample $F_Q(q)$ For Three Displacement Values.**

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

point will cause failure with a probability of 0.05 or less. Each data point in figures 4.4 and 4.5 was obtained by averaging the results of 50 applications of the re-expression algorithm to a randomly selected point from a failure region. Figure 4.4 has 47 such data points on each of its three graphs; figure 4.5 has 100, 71, and 100, respectively.

Figure 4.4 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 to the range of values that the radar points could assume.

Figure 4.5 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

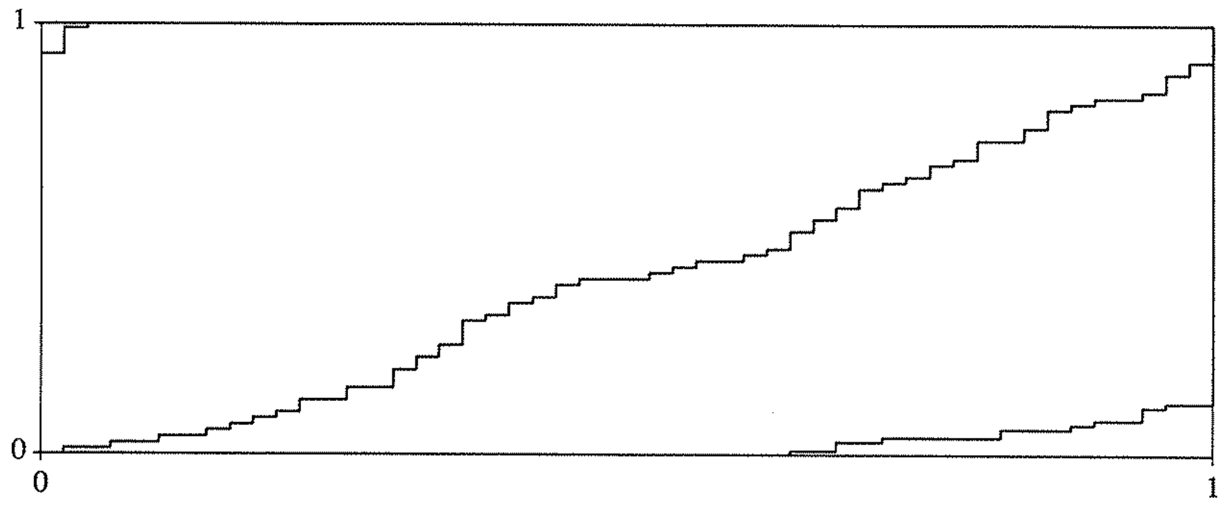


**Figure 4.5: Sample $F_Q(q)$ For Three Different Faults At 0.01 Displacement.**

*From left to right on the graph, the faults are 8.1, 7.1, and 6.2. [2].

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 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 4.6 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 4.6 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

| 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 4.6: Failure Probability Multipliers Using Specification Acceptance Test.**

eliminated, and an entry of 1.00 means that data diversity had no effect.

The model outlined earlier in this section was used to calculate the values shown in figure 4.6. The integral corresponding to the multiplier was approximated using sample $F_Q(q)$, examples of which appear in figures 4.4 and 4.5.

## Empirical Results for a Retry Block - Valid Output Acceptance Test

Figure 4.7 gives the expected value multipliers for the probability of system failure for retry blocks using the valid output acceptance test for 1, 2 and 3 retries. The entries in figure 4.7 were derived in a similar manner to the entries in figure 4.6. The same data yielded both figures, and only the definition of the acceptance test and the meaning of ''failure'' were different. In both cases, the definition of failure was used consistently when comparing single version performance to the retry block performance. When evaluating the retry block using the specification

| 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 | 0.99 | 0.96 | 1.00 | 0.98 | 0.93 | 1.00 | 0.98 | 0.90 |
| 6.3 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 7.1 | 0.96 | 0.83 | 0.26 | 0.93 | 0.71 | 0.12 | 0.90 | 0.63 | 0.07 |
| 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 | 0.93 | 0.70 | 0.98 | 0.89 | 0.54 | 0.97 | 0.86 | 0.44 |

**Figure 4.7: Failure Probability Multipliers Using Valid Output Acceptance Test.**

acceptance test, outputs were considered to be wrong if they disagreed with the specification. Similarly, when evaluating the retry block using the valid output acceptance test, outputs were considered to be wrong if they did not match the output demanded by the specification for any input points in the re-expression region.

The performance of the retry block did not depend significantly on the type of acceptance test used. In both cases, four of the faults were tolerated completely, and three of the faults were not well tolerated. Although the more generous definition of failure used for the valid output acceptance test results in a lower failure rate than the more stringent definition of failure required by the specification acceptance test, the performance improvement attributable to the retry block is comparable for the two cases.

## V *N*-COPY PROGRAMMING

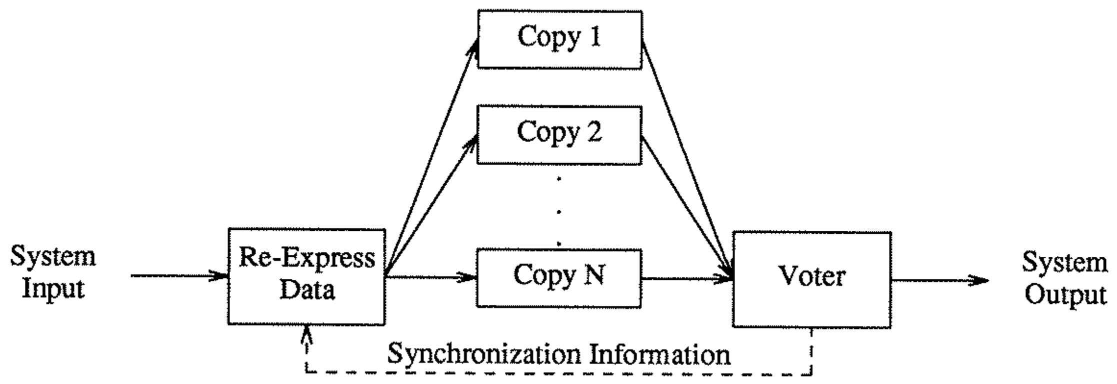


**Figure 5.1: N-Copy Programming.**

An *N-copy* system is similar to an *N*-version system but uses data diversity instead of design diversity. Figure 5.1 shows the structure of an *N*-copy system. *N* copies of a program execute in parallel; each on a set of data produced by re-expression. The system selects the output to be used by an enhanced voting scheme.

Voting in an *N*-copy system is not necessarily straightforward. If the re-expression algorithm is exact, so that all copies should generate identical outputs, then a conventional majority vote can be used. However, if the re-expression algorithm is approximate, the copies could produce different but acceptable outputs. This is very likely to occur near boundaries in the output space. In that case, a simple majority may not exist and the selection process is more involved. If a particular output occurs more than once, then it might be selected. If more than one output occurs more than once or no output occurs more than once, then selection might have to involve an arbitrary choice.

## Simple Model For *N*-Copy Programming

To determine the performance of data diversity as represented by an *N*-copy system, we analyze a *3*-copy system and compare it with a single version. The copies can map their inputs to $R+W$ different outputs. $R$ of the outputs are acceptable to the application, and the probability that a copy produces acceptable output $i$ is $r_i$ for $i$=1..$R$. Similarly, W of the outputs are not acceptable, and the probability that a copy produces unacceptable output $i$ is $w_i$ for $i$=1..$W$. To produce an output, the system generates three re-expressed data sets and executes a copy on each. The system selects the output that occurs most frequently, if there is one. In the case of a tie, the tie is broken by choosing at random. Given the probabilities $r_i$ and $w_i$, we wish to know with what probability the system will select an acceptable output. There are four cases under which the system makes an acceptable choice:

(1) The program maps all three re-expressed data points to acceptable outputs. This happens with probability $\sum_{i=1}^{R}\sum_{j=1}^{R}\sum_{k=1}^{R} r_i r_j r_k$

(2) The program maps two re-expressed data points to the same acceptable output, but maps one to an unacceptable output. This happens with probability $3\sum_{i=1}^{R}\sum_{j=1}^{W} r_i^2 w_j$

(3) The program maps two re-expressed data points to different acceptable outputs, and maps one to an unacceptable output. Two thirds of the time, an acceptable output is selected at random. The probability of this event is $2\sum_{i=1}^{R}\sum_{j=1}^{R}\sum_{k=1}^{W} r_i r_j w_k$, such that $i \neq j$.

(4) Finally, the program maps one re-expressed data point to an acceptable output, but maps the other two to different unacceptable outputs. One third of the time, an acceptable output is selected at random. The probability of this event is $\sum_{i=1}^{R}\sum_{j=1}^{W}\sum_{k=1}^{W} r_i w_j w_k$, such that $j \neq k$.

The sum of these four probabilities gives the probability that a *3*-copy system will yield an acceptable output.

## Empirical Results For *N*-Copy Programming

Figure 5.2 gives the expected value multipliers for the probability of system failure for *N*-copy systems using 3 and 5 copies. For both *3*- and *5*-copy systems, four of the faults were tolerated completely, and three of the faults were not well tolerated. The *N*-copy systems tolerated each fault with about the same success as the retry blocks.

As with the retry block using the valid output acceptance test, the definition of failure for an *N*-copy system was used consistently when comparing single version performance to the *N*-copy system performance. In both cases, outputs were considered to be wrong only if they did not

| Copies | 3 | | | 5 | | |
|---|---|---|---|---|---|---|
| Displacement | 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 |
| 6.2 | 1.00 | 0.99 | 0.97 | 1.00 | 1.00 | 0.97 |
| 6.3 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 7.1 | 0.97 | 0.88 | 0.22 | 0.98 | 0.91 | 0.19 |
| 8.1 | 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 |
| 9.1 | 0.99 | 0.95 | 0.73 | 0.99 | 0.96 | 0.74 |

**Figure 5.2: Multipliers For Probability of System Failure for 3 and 5 Copies.**

match the output demanded by the specification for any input points in the re-expression region.

The data shown here measures the performance of an *N*-copy system only on failure points and not on a representative sample of input data. Although we expect it to be small, in this experiment we did not measure the probability that an *N*-copy system would fail when operating on points for which a single version succeeds.

## VI CONCLUSIONS

We have described the general concept of data diversity as a technique for software fault tolerance and have defined the retry block and *N*-copy programming as two possible approaches to its implementation. We have presented the results of a pilot study of data diversity for both structures. Although the overall performance of data diversity varied greatly, we observed a large reduction in failure probability for some of the faults examined in the study. In several cases data diversity 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 would not affect the application.

An issue in the implementation of a retry block is the need for a suitable acceptance test. This is a well-known problem for the recovery block, and 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.

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. One possible advantage of integrating design and data diversity is that the strategies may remedy different classes of errors. The ways in which data diversity should be used and how it should be integrated with design diversity are open questions.

## 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 thank Doug Miller for providing a valuable critique of an earlier version of this work. 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

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

[2] 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.

[3] F. Cristian, ''Exception Handling'', in *Resilient Computing Systems, Volume 2*, T. Anderson, ed., John Wiley & Sons, New York, (to appear).

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

[5] 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.

[6] 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.

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

[8] "Software Reliability: Repetitive Run Experimentation and Modeling", NASA Report CR-165836, Langley Research Center, February, 1982.

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

[10] "Tutorial: Software Testing and Validation Techniques", 2nd Ed., IEEE Computer Society Press, 1981.