Data Diversity: A Search For Direction *

Frederick C. Harris, Jr.

Abstract

A method for enhancing the reliability of N-version software is proposed. A pilot study presents experimental results on data diversity, in which retry with a mutation-directed variation in input is attempted after system failure. These results suggest that mutation analysis could be valuable in the design of fault-tolerant software systems.

keywords: software reliability, mutation analysis, data diversity

1 Introduction.

Two of the leading methods proposed in the literature to enhance the reliability of critical software systems such as those used in aircraft and spacecraft flight control, are N-version programming and recovery blocks. N-version programming is defined [2] to be multiple, independent implementations of the same program, where the implementations were coded from the same specifications. When the system is put together, each implementation is given the same input, and it is hopeful that they generate the same output. If they do not, a majority voting scheme goes into effect to determine what the system's response will be. Recovery Blocks are built on the premise of an acceptance tester that can be used in conjunction with multiple implementations of the code. The first implementation is given the system input, and its output is passed to the acceptance test. If the output is rejected, the system is rolled back, and the input is passed to the next implementation.

Ammann and Knight [1] observed that both of these methods are built on the concept of *design diversity*, which is the use of multiple implementations of the same code. They proposed an alternative concept called *data diversity*, which is based on the premise that if software fails a minor change of the input might allow it to succeed. Minor changes in input can be justified in an environment where the input sensors are noisy or have a very limited precision. Their work took

*This work was partially supported by NASA Langley Research Center under grant NAG-1-1024

data diversity and applied it to a recovery block structure called a retry block, which randomly changes the input for one implementation of the code. They suggest that *data diversity* is a fault-tolerant strategy that complements *design diversity*. The major shortcoming of their work was the lack of suggested direction for data variation within the retry block.

This paper presents the results of a pilot study using a retry block equipped with *data diversity* to enhance the reliability of an N-version system. A direction for the data variation is taken from a gradient approximation of a multi-dimensional surface generated using the *kill score* from mutational analysis, described in section 3. The basic structure of this system is presented in figure 1.

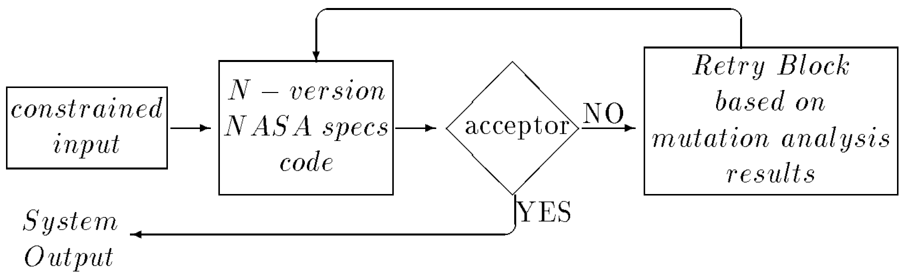

Figure 1: System Model

In section 2 a small experiment in N-version programming is described. Coding is based upon the specifications of NASA's planetary lander control software. Section 3 gives a brief introduction to mutation analysis and describes a surface that is created using it, and section 4 presents three approachs to a retry block which are based on the gradient-approximation of that multi-dimensional surface. Results from the study are presented in section 5, and the conclusions follow in section 6.

2 A Sample Module

Several graduate students were selected in the fall of 1989 to participate in a seminar to study errors that programmers make in Fortran-77 programs. These students were presented with specifications for small, "straightforward" procedures to be completed during the course. The specifications were provided by NASA as part of their Guidance Control Software and were quite detailed. They included a description of the parameters and a specification of the operations in each module [9].

The code for this course was developed in a controlled environment on a series of Sun 3/50's. Whenever a module was compiled the development environment placed a copy of the code in a controlled directory elsewhere on the system. This allowed us to monitor the development of, and corrections to the modules throughout the course. Access to the code also allowed us to determine how well the students were testing their work.

The accelerometer sensor processing module was one of the required procedures, and its specifications are summarized in figure 2. This module takes input from the acclerometer sensor

> Transforming accelerometer data (A_COUNTER) into vehicle accelerations (A_ACCELERATION) requires the following steps:
>
> 1. A_GAIN := A_GAIN_0 + (G1 * ATMOSPHERIC_TEMP) + (G2 * ATMOSPHERIC_TEMP^2)
> 2. A_ACCELERATION_M := A_BIAS + A_GAIN * A_COUNTER
> 3. Shift A_STATUS and A_ACCELERATION right by 1 column.
> 4. A_ACCELERATION := ALPHA_MATRIX * A_ACCELERATION_M
> 5. For each row of A_STATUS and A_ACCELERATION: If any of the previous three values of A_STATUS is *unhealthy*, set A_STATUS(i,0) to *healthy*. Otherwise, for each of the three dimensions in A_ACCELERATION calculate the mean (μ) and the standard deviation (σ). If $|\mu$ - A_ACCELERATION(i,0)$|$ > A_SCALE*σ, then set A_ACCELERATION(i,0) to μ and set A_STATUS(i,0) to *unhealthy*.

Figure 2: ASP Specification Summary.

(A_COUNTER), removes its natural electrical and temperature bias (steps 1 and 2), and then produces actual accelerations in the x, y and z directions.

The Guidance Control Software was designed to operate in an N-version environment, with a typical voting scheme for control of the landing vehicle. With this in mind, five implementations of the procedure were taken from the course and combined into such a system. Initial system test data was generated using Godzilla [8], a software test case generator that is based on mutation analysis

3 Mutation Analysis

Mutation analysis [3] is a software testing technique which is similar to fault injection experiments in circuit design. A collection of mutation operators are applied to the original code, and each yields executable variations of the code called mutants. Some operators explicitly require that the test data meet statement and (extended) branch coverage criteria, extremal values criteria, and domain perturbation; the mutation operators also directly model many types of faults. Some of these mutants are *equivalent* mutants because they are logically equivalent to the original code.

A mutant is said to be *killed* if its output can be made to differ from that of the original code. The goal is to obtain a set of test cases that will "kill" all non-equivalent mutants. The effectiveness of this approach to software testing is based upon a fundamental premise: if the software contains a fault, it is likely that there is a mutant that can only be killed by a test case that also reveals the fault.

Mutation analysis has been implemented in many testing systems, the most recent of which is called Mothra [4]. Mothra was written by students and faculty at Georgia Tech and Purdue and is a testing environment that uses mutation analysis as its underlying method. This system has many integrated modules that allow the user to interactively test Fortran-77 code throughout the development cycle. Mothra allows a tester to examine remaining live mutants and design tests that kill them. The mutation operators used by Mothra [7] represent more than 10 years of refinement through several mutation systems.

Unfortunately, generating mutation-adequate tests can be a labor-intensive task. To solve this problem, Offutt [6] devised an adequacy-based scheme for automatically generating test data through a Constraint-Based Testing (CBT) system. In constraint-based testing, we represent the conditions under which each mutant will die as mathematical constraints on the inputs, and then generate a test case that satisfies the constraint system. An implementation of this technique, Godzilla, has been integrated with Mothra.

Godzilla [8] is one of the most beneficial modules in Mothra for a software tester. Initial modules in Mothra parse the Fortran-77 code and set up path-constraints on the input which are necessary to reach each line of code. Godzilla then generates test data by posing the question, "What properties must the program inputs have to kill each mutant?" The inputs must cause the mutant to have an incorrect program state after some execution of the mutated statement. Godzilla uses the same syntactic information that Mothra uses to force the syntactic change represented by the mutation into making a semantic difference during execution. Since faults are modeled as simple faults on single statements, an initial condition is that the mutated statement must be reached (reachability). A further condition is that once the mutated statement is executed, the test case must cause the mutant to behave erroneously, i.e. the fault that is being injected must result in a failure in the program's output.

Godzilla describes these conditions on the test cases as mathematical systems of constraints. The reachability condition is described by a system of constraints called a *path expression*. For each statement in the program, the path expression contains a constraint system that describes each execution path through the program that will reach that statement.

The condition that the test case must cause an erroneous state is described by a constraint that is specific to the mutation operator. In general, this *necessity constraint* requires that the computation performed by the mutated statement create an incorrect intermediate program state. Although an incorrect intermediate program state is necessary for fault detection, it is clearly not sufficient to guarantee detection. In order to detect the fault, the test case must cause the mutant to produce incorrect output, in which case the final state of the mutant differs from that of the original program. Although deriving a test case that meets this *sufficiency condition* is certainly desirable, it is impractical. Determining the sufficiency condition implies knowing the complete path that the program will take, which is intractable. Thus, the partial solution of relying on the necessity condition is used.

To generate the test cases, Godzilla solves the conjunction of the path expression constraint with the necessity constraint to create a test case consisting of values for the input variables that will make the constraints true. Godzilla consistently generates test cases that kill over 95% of the mutants [5].

Mothra generated $3,778$ non-equivalent mutants when presented with one implementation of the Accelerometer Sensor Processing module. Godzilla then generated 30 test cases which killed all $3,778$ non-equivalent mutants. One test case was particularly troublesome because 4 of the 5 implementations were determined to be incorrect by an oracle module, whose correctness on the

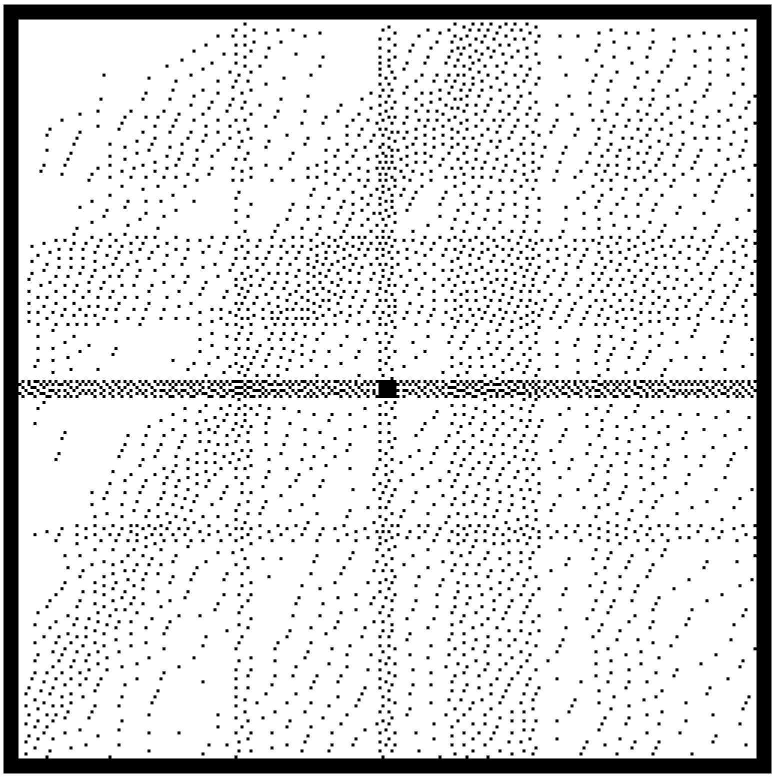

Figure 3: Digitized image of G1 and G2 surface

30 cases was verified by hand.

The Accelerometer module has over 50 input variables which is too many for a pilot study. Therefore, the particularly troublesome test case previously described was selected as a starting point for our data diversity experiment. All input variables were fixed at the values of the troublesome case except G1 and G2, which both have an input domain of $[-5.0, +5.0]$. This was done so that the data gathered could be easily visualized, and because G1 and G2 are launch time constants yet to be determined.

Using the code discussed in section 2, we constructed a surface M of mutation kill scores. $M(G1, G2)$ was calculated by feeding the selected input (G1, G2) to the mutation system and finding out how many mutants were killed by that particular input. We used $1,681$ input values evenly spaced through the G1, G2 domain and found output values of M ranged from 1924 to 2092. A digitized image of this surface is shown in figure 3.

4 Retry Block

A retry block is a modification of the recovery block structure that uses data diversity instead of design diversity [1]. After the code has executed, a typical voting scheme determines whether three or more of the five implementations agree. If they do, the system is assumed to be correct, and the retry block is finished. If the output is incorrect, then the retry block must change the input data and re-execute all versions of the code.

The changing of the input data for this pilot study was based on the concept of a gradient. From basic calculus we know that a gradient of a function ∇M is a vector orthogonal to the level curves of M, i.e. pointing in the "uphill" direction when M is regarded as a terrain map. The gradient is defined only on a smooth surface; therefore, when function evaluation at a finite number of points is our only tool, the gradient must be approximated.

This approximation was done in a simple, straight-forward way. Consider the three points in our lattice of evaluated points, closest to the system failure point in the G1, G2 plane. These three along with their M values are placed into an array which is row-reduced to find the equation of the plane, which is of the form $Ax + By + Cz = 1$. From this equation the gradient for the plane is a simple combination of coefficients, $\nabla M = (-A/C, -B/C)$.

With the pseudo-gradient determined, four retry blocks were designed. The first was designed to change the data in a random direction as it was done in Ammann and Knight's experiment [1]. The next three blocks were based on the pseudo-gradient of the surface at the failure location and changed the data in 3 directions: the direction of the pseudo-gradient (up the surface), the direction of the negative pseudo-gradient (down the surface), and orthogonal to the pseudo-gradient (level with respect to the surface). It was conjectured that regions of lower M score would contain less data sensitivity and hence offer higher probability of success on retry.

5 Results

Using the restricted (G1, G2) input space described in Section 3, random values in that space were given as input to the 5-version system, and the system output was checked for correctness using

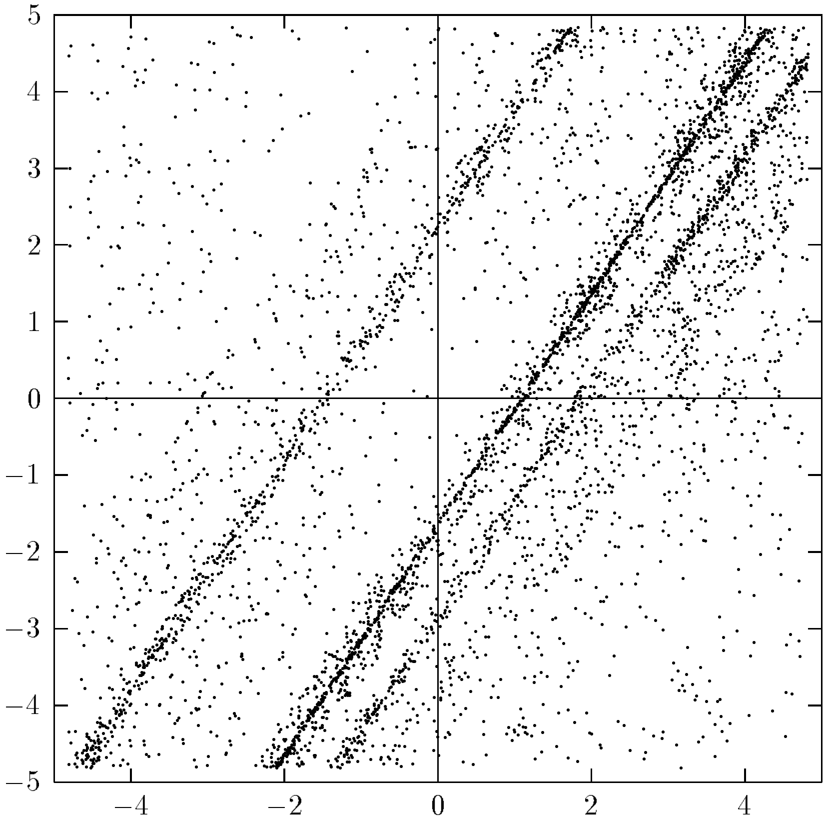

Figure 4: Failure points in (G1,G2) space.

a majority voting scheme. If fewer than 3 of the 5 implementations agreed, the system was ruled to be incorrect. This testing continued until 15,000 system failure points were generated. A graph of the points at which the failures occurred is in figure 4. It is interesting to note the differences between this graph and corresponding graphs in the literature. The failures that Ammann and Knight present are in "failure regions" where points inside that region cause the system to fail and points outside that region cause the system to execute properly. Figure 4 suggests that our failure regions may be more closely approximated by hyperplanes in the input space.

Empirical results were obtained from the implementation of the various retry blocks discussed in the section 4. Each block was given all 15,000 system failure points. These failure points, as input, were modified accordingly, and the data was then passed back to the 5-version system to be re-executed. Figure 5 shows the number of failures surviving the retry as a function of the distance moved. The random retry block corrected over 90% of the original errors, which is quite significant.

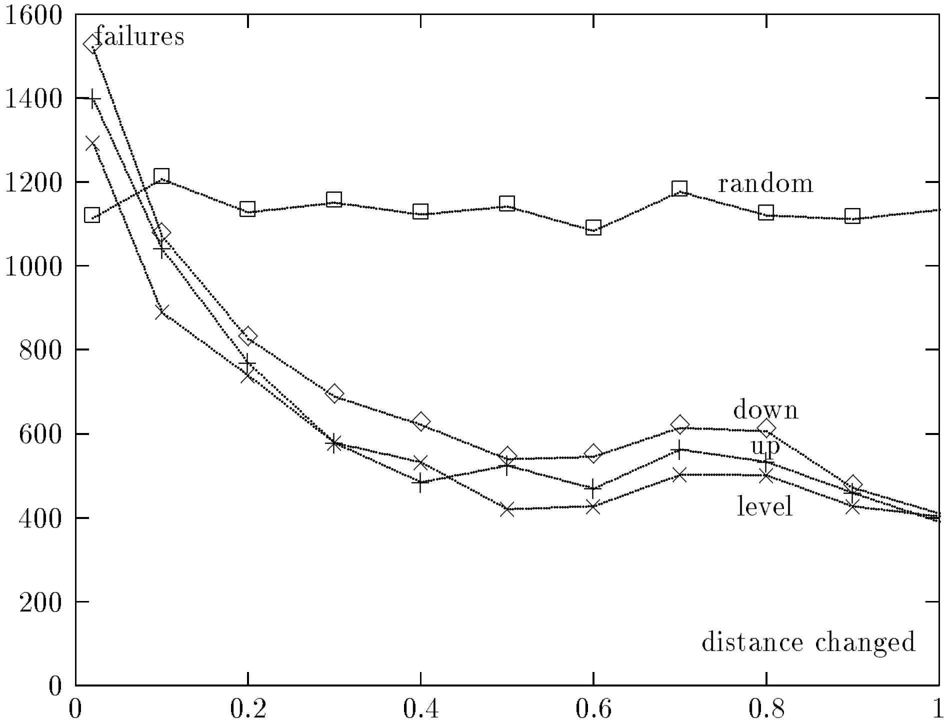

Figure 5: Failures surviving retry.

Yet it is obvious that movement in any particular direction of the gradient is much better than a random retry.

6 Conclusions.

In this pilot study it has been shown that a data diversity approach to software fault tolerance in which input data movement is determined by the pseudo-gradient of the mutation surface yields much better results than movement in a random direction. It is interesting that retry orthogonal to the gradient (level) is usually better than going up or down. This could be due to the fact that the mutation kill score surface is an imperfect indicator of failure regions and is crisscrossed with ravines making movement along the surface in any direction difficult. Such information would be useful in future modifications of the mutation operators.

Further study is definitely needed on this topic. The constraints placed on this pilot study need to be removed by not fixing input variables. It is not clear whether the kill score surface M would differ substantially if the mutation generator were presented with the restricted, two-input code module. Also the effect of multiple retries, subject to real-time constraints, should be explored.

References

[1] P. Ammann and J. Knight. Data diversity: An approach to software fault-tolerance. *Proc. 17th Int. Symp. on Fault-Tolerant Computing (FTCS-17)*, pages 122–126, Pittsburgh, PA, July, 1987.

[2] A. Avižienis. The n-version approach to fault-tolerant software. *IEEE Trans. Soft. Engr.*, SE-11(12):1491–1501, December 1985.

[3] T. A. Budd. *Mutation Analysis of Program Test Data.* PhD thesis, Yale University, New Haven CT, 1980.

[4] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff Alberta, July 1988. IEEE Computer Society Press.

[5] R. A. DeMillo and A. J. Offutt. Experimental results of automatically generated adequate test sets. In *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*, pages 209–151, Portland OR, September 1988. Lawrence and Craig.

[6] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Soft. Engr.*, 17(9):900–910, September 1991.

[7] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software Practice and Experience*, 21(7):686–718, July 1991.

[8] A. J. Offutt. *Automatic Test Data Generation.* PhD thesis, Georgia Institute of Technology, Atlanta GA, 1988. GIT-ICS 88/28.

[9] B. Withers, D. Rich, D. Lowman, and R. Buckland. Software requirements: Guidance and control software development specification. *NASA Contractor Report 182058*, NASA Langley Research Center, June, 1990.