A Semi-Annual Progress Report
NASA Award No. NAG-1-605
July 1, 1985 - June 30, 1987

DETECTION OF FAULTS AND SOFTWARE RELIABILITY ANALYSIS

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

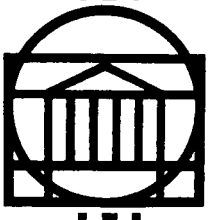
Attention: Mr. Gerard E. Migneault
ISD M/S 130

Submitted by:

John C. Knight
Associate Professor

Report No. UVA/528243/CS87/102

February 1987



SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA 22901

A Semi-Annual Progress Report
NASA Award No. NAG-1-605
July 1, 1985 - June 30, 1987

DETECTION OF FAULTS AND SOFTWARE RELIABILITY ANALYSIS

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: Mr. Gerald E. Migneault
ISD M/S 130

Submitted by:

John C. Knight
Associate Professor

Department of Computer Science
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/528243/CS87/102
February 1987

Copy No. 4

SECTION I

INTRODUCTION

The work being carried out under this grant is an investigation of software faults. The goal is to better understand their characteristics and to apply this understanding to the software development process for crucial applications in an effort to improve software reliability. Some of the work is empirical and some analytic. The empirical work is based on the results of the Knight and Leveson experiment [1] on *N*-version programming. The analytic work is attempting to build useful models of certain aspects of the software development process.

Multi-version or *N*-version programming [2] has been proposed as a method of providing fault tolerance in software. The approach requires the separate, independent preparation of multiple (i.e. "*N*") versions of a piece of software for some application. These versions are executed in parallel in the application environment; each receives identical inputs and each produces its version of the required outputs. The outputs are collected by a voter and, in principle, they should all be the same. In practice there may be some disagreement. If this occurs, the results of the majority (assuming there is one) are taken to be the correct output, and this is the output used by the system.

The major experiment carried out by Knight and Leveson was designed to study *N*-version programming and initially investigated the assumption of independence. In the experiment, students in graduate and senior level classes in computer science at the University of Virginia (UVA) and the University of California at Irvine (UCI), were asked to write programs from a single requirements specification. The result was a total of twenty-seven programs (nine from UVA and eighteen from UCI) all of which should produce the same output from the same input. Each of these programs was then subjected to one million randomly-generated test cases. The

Knight and Leveson experiment has yielded a number of programs containing faults that are useful for general studies of software reliability as well as studies of *N*-version programming.

Our work during the grant reporting period has been mainly in two areas and each area is covered in detail in an appendix of this report. The specific topics are fault tolerance through data diversity which is discussed in appendix A, and analytic models of comparison testing which are discussed in appendix B.

REFERENCES

(1) J.C. Knight and N.G. Leveson, "An Experimental Evaluation Of The Assumption Of Independence In Multi-Version Programming,", *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 1, January 1986.

(2) L. Chen and A. Avizienis, "*N*-Version Programming: A Fault-Tolerance Approach To Reliability Of Software Operation", *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, Toulouse, France, pp. 3-9, June 1978.

APPENDIX A

DATA DIVERSITY: AN APPROACH TO SOFTWARE FAULT TOLERANCE

Paul E. Ammann John C. Knight

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

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.

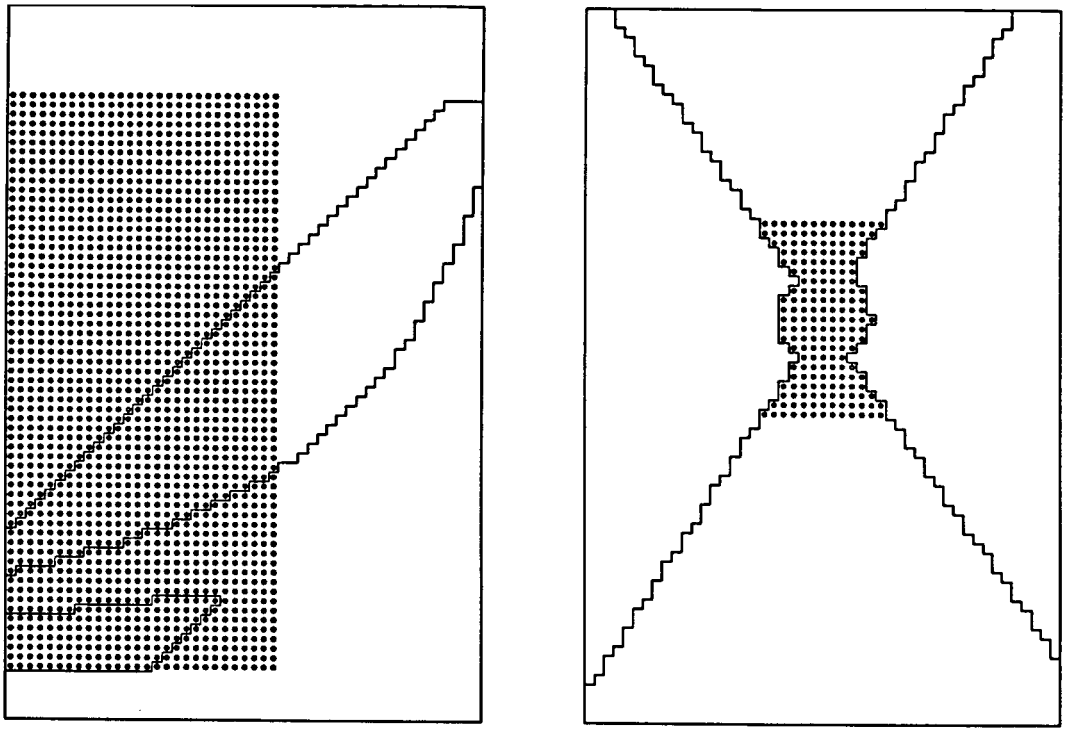
2. FAULT REGIONS

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].



(a) (b)

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 4×10^{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

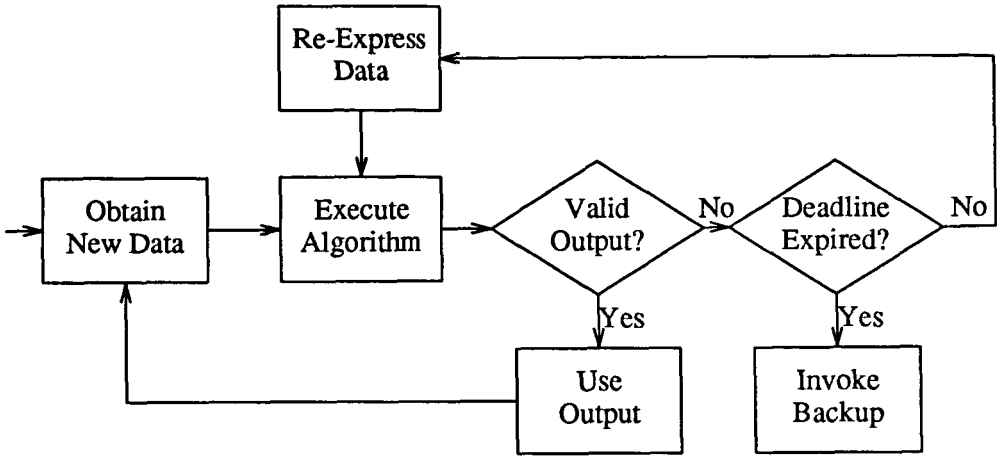


Figure 2: Retry Block.

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

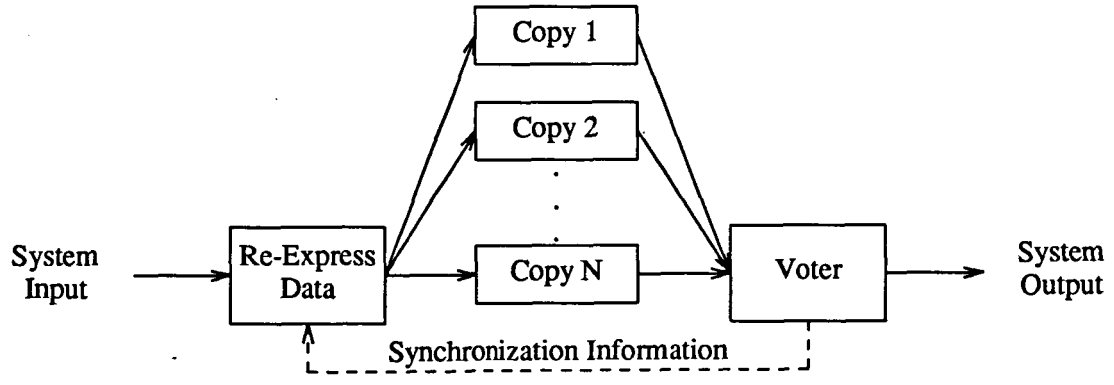


Figure 3: N-Copy Programming.

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

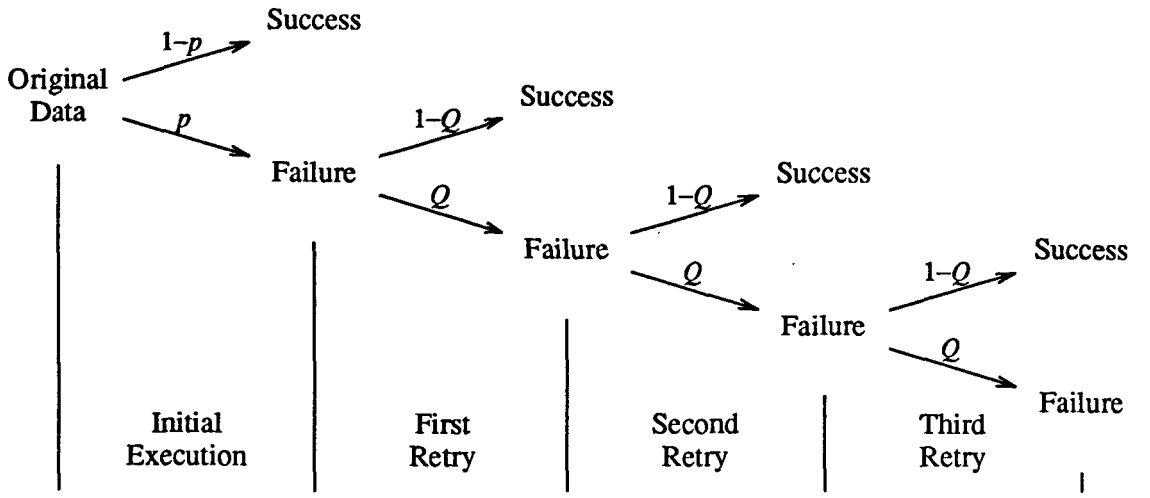


Figure 4: Retry Block Model Assuming A Perfect Acceptance Test.

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.

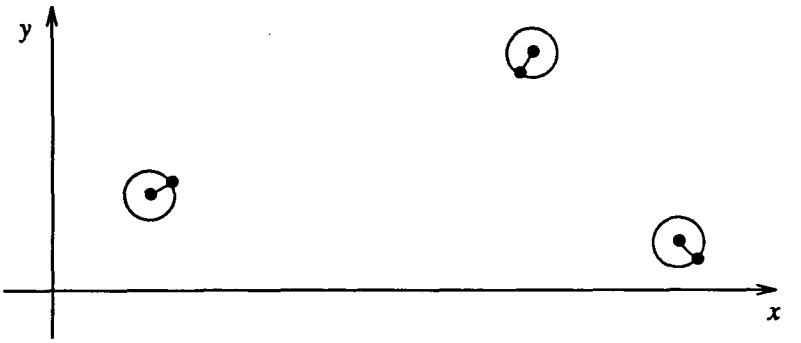


Figure 5: Re-Expression Of Three Radar Points.

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 $F_Q(q)$.

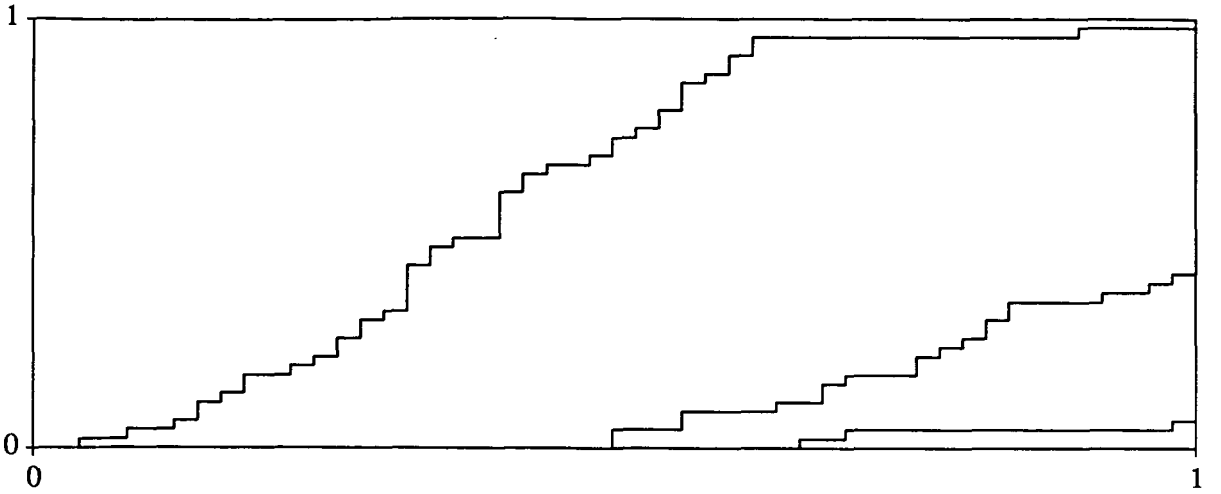


Figure 6: Fault 9.1 Sample $F_Q(q)$ For Three Displacement Values.

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].

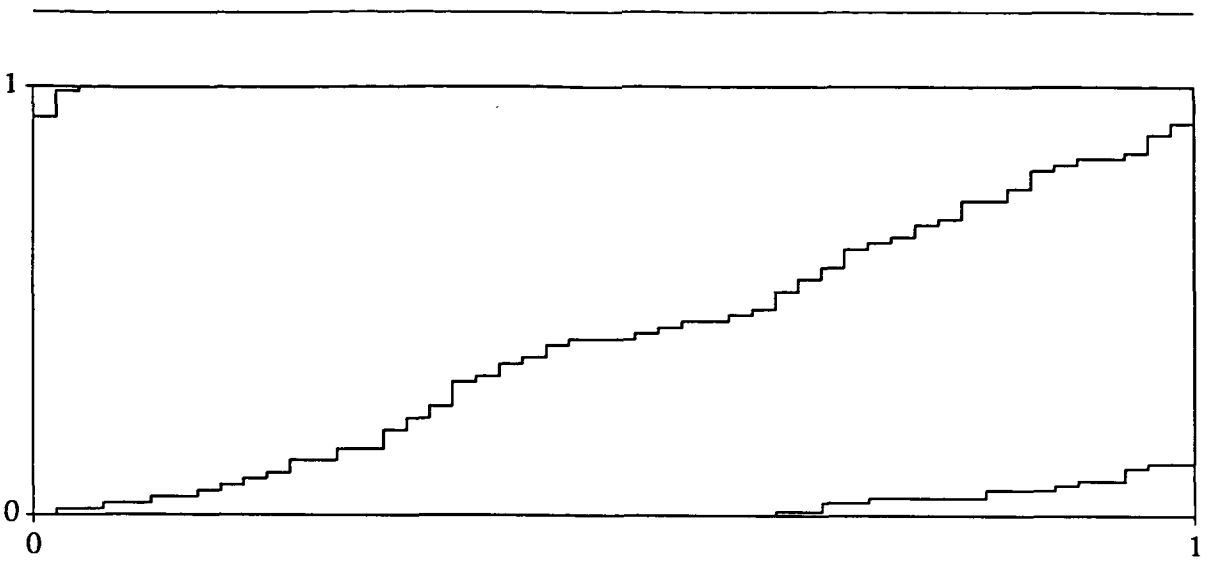


Figure 7: Sample $F_Q(q)$ For Three Different Faults At 0.01 Displacement.

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 6 and 7. 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

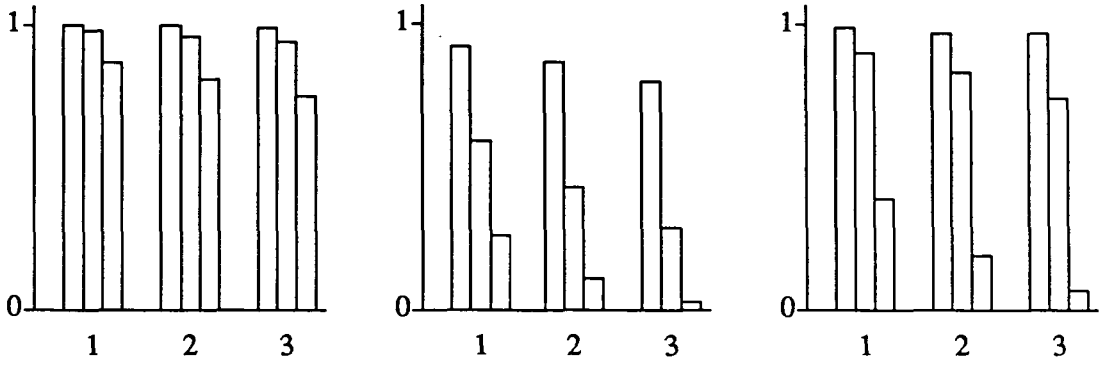


Figure 9: Non-Zero Values From Figure 8.

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.

APPENDIX B

TESTING SOFTWARE

USING MULTIPLE VERSIONS

Susan S. Brilliant John C. Knight

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

TESTING SOFTWARE USING MULTIPLE VERSIONS

A Preliminary Report

Susan S. Brilliant

John C. Knight

Financial Acknowledgement

This work was supported in part by NASA under grant number NAG-1-605.

ABSTRACT

One aspect of the testing process that has been the object of little research is the problem of determining the correct software response for each test input. For many software systems the determination of correct outputs is a difficult and time-consuming task.

Some researchers have suggested that multiple independently developed versions can be used to obviate the need for the *a priori* determination of the correct output. The outputs of the versions can be compared, and any differences can be investigated. We call this method *comparison testing.*

Comparison testing is an appealing approach, because the testing process can be automated easily if there is no need for the independent calculation of outputs. However, the possibility exists that all of the versions could obtain identical incorrect outputs. Some test cases that produce failures, then, will not be investigated.

The purpose of this research is to evaluate comparison testing. Parameterized analytic models have been developed that reveal the effects of fault interrelationships on the ability of comparison testing to reveal a fault. A model of the expected performance of operational single- and multiple-version systems that have been comparison tested is under development, and the effect of the number of versions in a comparison testing system on expected system performance is being studied. Empirical evidence from a multi-version experiment is being analyzed as an example of the parameter values that can be expected to occur.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The purpose of this research is to evaluate a testing method that we call *comparison testing*, which requires the availability of multiple versions of the software to be tested. This testing method has been proposed by other researchers, but its power in revealing the faults in the software to which it is applied has never been evaluated.

1.1. The Need for More Reliable Software

The need for better testing methods grows with the need for reliable software. Computers are now being used to control air traffic, trains, aircraft, manned spacecraft, nuclear power plants, defense systems, and life support systems. In addition to these application areas in which human lives may depend on the reliability of software, there are also a number of application areas in which failure would incur a heavy economic penalty. Communications systems, factory process control, and electronic fund transfers are just a few examples of these application areas [1].

Some applications that are currently being developed require software of extremely high reliability. For example, engineers at the Sperry Corporation in Phoenix are in the process of developing a fly-by-wire system to replace the system of hydraulics and pneumatics currently used to control commercial aircraft [34]. Since the only advantage of the new system is an increase in fuel efficiency, it must be as reliable as existing systems, which have experienced a failure rate of only 10^{-10} failures per hour of flight time. Such reliability rates appear to be beyond the ability of standard software techniques to ensure, or even to measure with a reasonable amount of testing.

1.2. The Operational Use of Multiple Versions

The generation of two or more functionally-equivalent programs, called *versions*, from the same requirements specification, was originally proposed as a method of achieving higher operational software reliability [4]. *N-version programming*, as this approach is called, requires that N versions of critical software be developed by N individuals or groups that do not interact during the development process. These independently developed versions are combined into an N-version unit that can be designed to achieve either *error detection* or *fault tolerance.*

An *error detection* approach is appropriate for situations in which system failure has dire

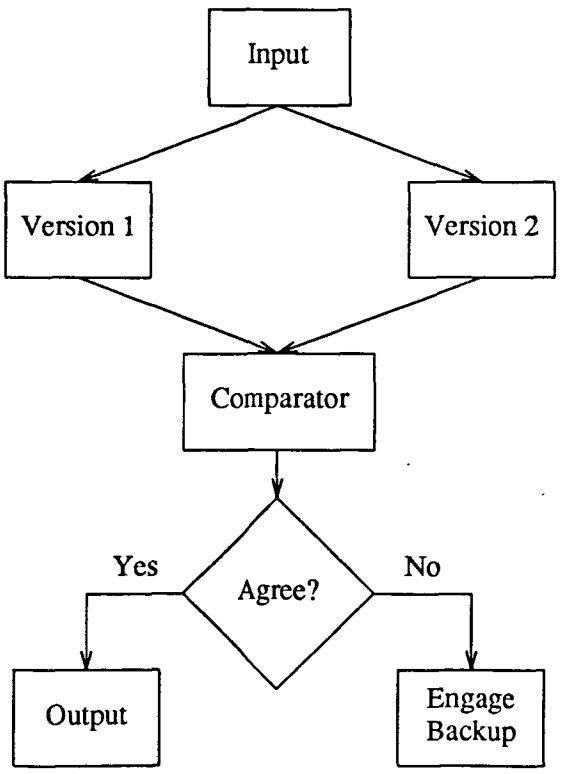


Figure 1.1. Multi-Version Programming to Achieve Error Detection

consequences only if it is *undetected*; in general a backup system is available in the case of primary system failure [23, 30, 34]. Two versions are normally used for fault detection. They are configured as shown in Figure 1.1. If the versions agree then their output is used; otherwise control is transferred to the backup system.

N-version programming can also be used to achieve *fault tolerance*. In most systems, emergency procedures to deal with detected error states, if they exist, are limited to bringing the system into a safe state pending external intervention [1]. It would be preferable if a program were able to continue to function normally despite the presence of faults. Systems with this attribute are said to be *fault tolerant*.

To achieve fault tolerance, an *N*-version system is configured as shown in Figure 1.2. For these applications *N* is generally an odd number greater than or equal to three, and the result achieved by a majority of the versions (if there is one) is used as the system output.

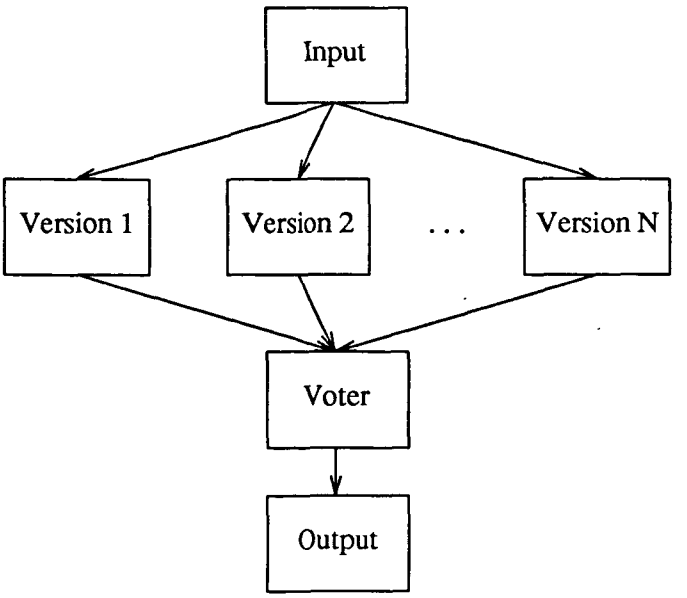


Figure 1.2. Multi-Version Programming to Achieve Fault Tolerance

1.3. The Use of Multiple Versions in Testing

A question that has not been addressed sufficiently in research on testing is that of how the correct response of a software system or subsystem to each test input can be determined, so that failures can be recognized during the testing process. The existence of an "oracle" is often explicitly or implicitly assumed. Yet as a practical matter it is often extremely difficult, if not impossible, to determine the correct outputs for each input.

Several researchers have suggested that the independent development of multiple versions may be useful in the testing process. Ramamoorthy *et al* [27] describe a methodology for the development and validation of reliable process control software in which two versions of the software are developed. They have applied their methodology to the production of pilot software for nuclear power plant safety protection. The authors state that the use of dual development eliminates the need for determining the correct system responses *a priori*, since the results from the two programs can be compared with each other. Similarly, Gmeiner and Voges [14], in their discussion of software diversity in reactor protection systems, present the idea that "diverse" programming, as they call the multi-version approach, makes it unnecessary to compute test outputs manually. Yount, Liebel, and Hill report that the two versions of the SP-300 automatic landing system served as ideal "monitors" for each other during the testing process [34].

We will call the approach described by these researchers *comparison testing*. The approach requires the independent development of two or more versions of the program to be tested. In testing the software, each test input is submitted to all of the versions. The outputs of the versions are compared, and any differences among the outputs are investigated to locate the fault or faults responsible for the discrepancy.

The advantages of such a testing method are obvious. Since there is no need for human examination of the test outputs, the testing process can be automated. Test cases generated randomly or by any other automated method can be executed with no need for human intervention except when a

disagreement among versions indicates the presence of a fault.

When multiple versions have already been developed to be used in an operational N-version unit, it is tempting to test the versions using comparison testing. It may be possible to test the programs much more extensively in this way than would be possible if all test outputs had to be computed manually. Even when multiple versions must be developed for the purpose of comparison testing, the increase in human resources necessary to generate the versions is likely to be more than offset by the reduction in resources needed for the testing process.

The major difficulty with comparison testing is that there may be faults in the individual versions that cause the versions to obtain identical incorrect outputs on some inputs. Test cases on which all versions in the testing system fail identically will not be recognized, so the faults causing such failures will not be investigated. This is a serious problem since it is certainly possible that programmers working independently may make the same mistake. Faults that are common to all of the versions will never be discovered by comparison testing. For an N-version system built for fault tolerance these are exactly the faults that will *not* be tolerated. It therefore seems reasonable to dismiss comparison testing as worthless (or worse).

On the other hand, given the ease with which comparison testing can be automated, it might be worthy of further consideration. In evaluating its usefulness we need to know the likelihood that faults common to all versions in a system will occur.

A multi-version experiment designed to determine whether independently developed versions fail independently in a statistical sense was undertaken jointly by the University of Virginia and the University of California at Irvine [21]. The experiment revealed that coincident failures occurred much more often than would be expected if the versions were statistically independent. Our initial reaction was to conclude that comparison testing would be a dangerous method to use in testing software for critical applications, since it seemed that independent development did *not* prevent programmers from making

the same mistakes.

An investigation of the individual faults that occurred in the multi-version experiment was undertaken in order to determine the reason for the large number of coincident failures that were observed [3]. It turned out that a it is unrealistic to assume that two faults either are unrelated or result from the same programmer error and are essentially identical. Faults that exhibited identical failure behavior were very rare. It was much more common that two faults cause identical failure on some inputs, but that both cause failure separately on other inputs as well.

To facilitate further discussion of the relationships among faults that affect the testing process, we define the *failure subspace* for a fault to be that subset of the input space on which the fault causes failure to occur. The ability of comparison testing to reveal a particular fault will depend on the relationship between that fault's failure subspace and the failure subspaces associated with faults in other versions. If a fault's failure subspace does not overlap with that of any other fault, then comparison testing will be as effective as any testing method can be in finding the fault since each time the fault causes failure, the failure will be detected and investigated. On the other hand, if all other versions contain a fault with an identical failure subspace and causing identical behavior when failure occurs, then none of these faults can ever be detected using comparison testing.

The multi-version experiment indicates that the most likely situation is that intersecting failure subspaces will occur, but that matching failure subspaces are unlikely. The effects of such incomplete overlaps on comparison testing are not as clear, and must be studied in order to evaluate the comparison testing method.

1.4. Organization of Remainder of Report

The purpose of this research is to examine, both analytically and empirically, the potential fault-revealing power of comparison testing. Previous related research is discussed in the next chapter. A

notational framework for this research is developed in Chapter 3.

In Chapter 4, analytic models are developed for the effects of various failure subspace interrelationships on comparison testing. These models are analyzed in Chapter 5.

The concept of the "overlap ratio" for a fault is introduced in Chapter 6, and the expected worst case performance of comparison testing in finding a fault having a given overlap ratio is considered. The variability of the performance of comparison testing is discussed in Chapter 7.

The conclusions drawn from this research are summarized in Chapter 8. The research currently in progress is discussed.

CHAPTER 2

PREVIOUS RESEARCH IN DYNAMIC TESTING

The most widely used method of assuring the quality of software is dynamic testing. This method requires exercising the program using known inputs in a controlled environment. The outputs are examined to determine whether the program processed each input correctly.

This chapter summarizes the research that has been done on the dynamic testing of software. This review is intended to give an overview of the major areas of research on dynamic testing, not a comprehensive coverage of all of the available literature.

Laski and Korel [22] have identified three questions that arise in planning a testing procedure:

(1) Which parts of the program should be tested?

(2) How should the program input data to exercise those parts be determined?

(3) How should the observed intermediate or final results be interpreted to assess the (in)correctness of the program?

Research efforts in testing have been directed toward answering one or more of these three questions.

2.1. Testing Strategies

Laski and Korel's first question indicates the need for a testing *strategy*. Most of the testing strategies that have been developed are *structural*, i.e. based in some way on the structure of the program being tested. This is why Laski and Korel say that the parts of the program to be tested must be determined.

Structural testing strategies began with intuitive notions of what is necessary to ensure that a program is "well-tested". Clearly if a statement in a program has never been executed on any test case, then we can have no assurance at all that the statement is correct. A *statement testing* strategy requires that each statement be executed at least once during the testing process.

Huang [20] illustrates that flow-of-control errors, a class of common programming errors, can be missed if a program is tested only to the extent of executing each statement at least once. He suggests that an obvious solution would be to require the traversal of each control path in the program, but observes that since most programs contain loops, the number of control paths would usually be prohibitively large. Huang recommends a more realistic strategy that has become known as *branch testing*, which requires that every branch in the program's flowchart be traversed.

Although branch testing is a more stringent strategy than statement testing, many researchers consider it to be inadequate. The most thorough flow-of-control testing method is *path testing*, which requires the execution of each control path. Since path testing is infeasible for many programs, many researchers have tackled the problem of attempting to limit the number of test cases required to test a program while attempting to retain, as much as possible, the fault-revealing power of path testing.

Howden [17] describes a *boundary-interior* strategy. A boundary test of a loop requires it to be entered but not iterated; an interior test requires it to be entered and then iterated at least once. Testing both the boundary and interior conditions of a loop permits the selection of a finite but "intuitively complete" set of paths to be tested.

A hierarchy of criteria called *Testing Effectiveness Ratios* are defined by Woodward, Hedley, and Hennell [33]. Rather than providing an absolute decision as to the adequacy of test data, these ratios measure the extent of coverage. Each measures the number of subpaths of a given length that are exercised by the test set as a ratio of the total number of subpaths of the given length.

Zeil and White [35] describe a method for analyzing the effectiveness of individual paths in testing for predicate errors in linearly domained programs. Additional paths should be chosen for testing if they reduce the dimension of the undetected error space. Zeil [36] extends this analysis to determine paths that will be effective in checking for domain errors caused by incorrect computations.

Several researchers have explored the idea of using data flow analysis to aid in the selection of paths to be tested. Laski and Korel [22], for example, define two path selection strategies based on data flow relationships. The first requires that a path set contain at least one subpath between each variable definition and each use reached by that definition. The other criterion is based on the concept of the *data context* of an instruction, which is the set of live definitions for all variables used in the instruction. The strategy requires that each data context of every instruction be tested at least once. A more stringent version of the second criterion imposes an order on the data context, based on the order in which the definitions in the context are encountered. Each ordered data context must be tested at least once.

Ntafos [25] defines a class of path selection criteria that he calls "required k-tuples". The basic structural unit to be tested is a *k-dr* interaction. A *k-dr* interaction consists of k - 1 variables $X_1, X_2,...,X_{k-1}$ and k distinct statements $s_1, s_2,...,s_k$, such that a path visits the statements in the given order. Each s_i contains a definition of a variable X_i that reaches a reference to X_i in s_{i+1}. This reference is used in defining variable X_{i+1}. The required k-tuples strategy requires coverage of all *k-dr* interactions and additional coverage for those having a last reference in a branch predicate or which occur at the beginning or end of a loop. A class of strategies can be obtained by varying k.

A family of data-flow directed path selection strategies are discussed by Rapps and Weyuker [28]. Each variable occurrence is classified as a definitional (*def*), computation-use (*c-use*), or predicate-use (*p-use*) occurrence. The *all-defs* criterion requires that a path set contain at least one subpath from each definition to some use reached by that definition. The *all-uses* criterion requires a subpath to each use reached by each definition. *All-du-paths* requires additionally that each cycle-free and single-cycle path

be included. Several suggested criteria distinguish between computation and predicate uses. The *all-p-uses* criterion requires a subpath between a definition and all p-uses. *All-p-uses/some-c-uses* additionally requires that for any definition reaching no p-uses, a subpath containing the definition and a c-use be selected. The *all-c-uses/some-p-uses* criterion is defined analogously. The authors offer no comment on the relative value of their suggested strategies, except to point out the tradeoff between selecting a "stronger" criterion, which will cause closer scrutinization of the program in an attempt to find faults, and a "weaker" criterion, which can generally be satisfied using fewer test cases.

White and Cohen [32] take a somewhat different approach in their contribution to the literature on path testing. Rather than concerning themselves with the problem of how to select the paths to be tested, they focus on testing each path in a manner designed to detect any *domain* errors. The control flow statements in a program partition the input space into *domains* consisting of input points that cause a particular path to be executed. The *domain testing strategy* requires selection of points on each boundary and a distance of ε from the boundary in order to detect a shift in the boundary or an error in the relational operator. Some alternative domain testing strategies, designed to improve on the error bound for the White and Cohen approach, are discussed by Clarke, Hassell, and Richardson [6].

The testing strategy presented by DeMillo, Lipton, and Sayward [8] relies on what they call the "competent programmer hypothesis," which asserts that programmers produce programs that are "close to" the correct program. Their *mutation testing* approach requires that a test set be able to distinguish between the original program and a set of "mutants". The mutants are created by systematically implanting small defects in the original program, on the theory that these changes model what the programmer might have done wrong.

Howden [19] discusses a related strategy that he calls *weak mutation testing*. This strategy requires the mutation of components of the program. The test set is considered adequate in recognizing a mutant if the mutated component computes at least one "value" different from that computed by the original

component (even if the final program output is the same). The advantage of "weak" over "strong" mutation testing is that it does not require a separate program execution for each mutation, and that the test data necessary to carry out a complete set of weak mutation tests can be more easily determined.

Mills [24] presents a technique that also relies on the introduction of random errors into a program. The goal of this method is to calibrate the testing process to permit statistical inference about the reliability of the tested program.

A number of researchers have pointed out that strategies based only on program structure are inadequate in determining whether the domain partitioning created by the control structures in a program is the same as that inherent in the problem that the program is designed to solve. In particular, no structural strategy is likely to be effective in discovering *missing path* errors.

Goodenough and Gerhart [15] present the *condition table* method for finding the logically possible combinations of conditions that can occur when the program is executed. In creating the condition table, they recommend the examination of the general requirements the program is to satisfy and the program specification in addition to the implementation itself. Similarly, Weyuker and Ostrand [31] suggest a method in which the tester finds the intersection of the domains created by the program and those implicit in the specification. Ntafos [25] incorporates features into his "required element" testing strategy that allows the user to insert input and output assertions based on the specifications.

The simplest of all testing strategies is *random testing.* The test cases to be used are selected randomly from the program's input space. Duran and Ntafos defend this method in [11]. They discuss the question of whether uniform sampling or an operational profile should be used in selecting the test cases, concluding that the operational profile is appropriate for estimating reliability, but that uniform sampling may be more effective for error detection.

The first major theoretical foundations for evaluating testing strategies were provided by Goodenough and Gerhart [15]. Their major contribution was to show that it is possible, if a testing strategy has certain properties, that testing can be used to demonstrate the absence of errors in a program. An *ideal* test, which succeeds only when a program contains no errors, is both *reliable* (consistent in the ability to reveal errors) and *valid* (able to reveal errors).

The Goodenough and Gerhart "reliable" and "valid" criteria are criticized by Weyuker and Ostrand [31] for several reasons. One of the most serious criticisms is that, since *all* test selection criteria are *either* reliable or valid, it is of no value to establish that a criterion has one of the necessary properties. Weyuker and Ostrand present a theory based on the concept of *revealing subdomains*. They define a subdomain as *revealing* if one of its members is processed incorrectly if and only if all of its members are processed incorrectly. The tester's goal, then, is to partition the input space into revealing subdomains.

Howden [18] shows that it is not possible to construct a testing strategy guaranteed to reveal all errors. The reliability of path testing in revealing classes of errors that commonly occur is analyzed. The reliabilities provide an upper bound for the reliabilities of the various path-selection strategies.

Gourlay [16] develops a mathematical framework and a unifying notation that provides a mechanism for comparing the power of testing strategies that have been developed by other researchers. Clarke *et al.* [7] provide a notation for comparing data flow path selection criteria and present a subsumption graph showing the relationships among all the suggested criteria.

2.2. Test Data Selection

The second question posed by Laski and Korel [22] points to the problem of selecting test data to implement a selected testing strategy.

For most of the testing strategies that have been suggested, the problem of determining whether a given test set satisfies the criteria established by the strategy appears to be straightforward. Huang [20] discusses the simple process of instrumenting a program in order to assess the adequacy of test coverage for branch testing. Girgis and Woodward [13] have developed a tool that instruments a program and reports on the completeness of test data with respect to the weak mutation strategy and a family of data flow selection strategies.

A major difficulty that arises in attempting to satisfy any path testing strategy is the possible existence of infeasible paths. Presumably a test will satisfy a given path testing strategy when all the *feasible* paths required by the strategy have been executed. However, the determination of whether a given path is feasible is, in general, unsolvable. Gabow, Maheshwari, and Osterweil [12] prove that even when a tool that automatically generates "impossible pairs" of program statements is available, the problem of determining whether an impossible pairs constrained path exists is NP-complete.

When instrumentation is used to determine the extent to which a test set has satisfied a criterion, the testing strategy actually functions as a stopping criterion rather than a test data selection criterion. It would be preferable if test cases to satisfy a criterion could be automatically generated.

Unfortunately, the problem of finding assignments that will satisfy a given path predicate is, in general, unsolvable, so it is impossible to develop a general algorithm for selecting, *a priori*, test data that will satisfy a given strategy. However, a number of researchers have developed methods that will work with limitations on the program being tested. For example, Clarke [5] has developed a tool that, given a completely specified program path containing only linear path constraints, will either determine that the path is infeasible or will generate a test case that will cause the execution of the path.

2.3. Interpretation of Results

The third problem that arises in planning a test procedure is that of interpreting the results in order to assess the correctness or incorrectness of the program. Although Laski and Korel [22] identify this problem, their paper contains no suggestions regarding its solution. They, like most researchers in testing, do not deal with this problem. Most do not even mention it. One who does is Gourlay [16], who observes that "it should be easier to establish the correctness of a test than of the program as a whole."

While Gourlay's observation is reasonable from a theoretical viewpoint, as a practical matter the determination of correct outputs is often inordinately time consuming. When expected results must be hand-computed, the extent to which it is possible to automate the testing process is limited. Panzl [26] describes several automated testing systems. These systems facilitate retention of software tests (useful for revalidating a system after a change) and free the tester from hand-checking the results as they appear. However, all of the systems require that the tester supply in advance a database of test cases to be executed, along with the expected outputs. The creation of this database is likely to require a large investment in human resources.

A system that could generate inputs and then interpret the results without human intervention would be extremely desirable. Since hardware resources are much less expensive than human resources, many more test cases could be executed for the same cost.

Ramamoorthy *et al.* [27] describe their experience with a completely automatic testing system. It utilized two versions of a nuclear reactor protection system, developed by independent teams at Babcock and Wilcox and at the University of California at Berkeley. The testing system consisted of an automatic test data generator to create inputs and a dual program monitor and analyzer to identify test cases on which failure occurred. If the results obtained by the versions agreed (within a specified numerical threshold) the result was assumed to be correct. Otherwise both programs were analyzed, using the originating requirements, to determine the reason for the deviation.

Gmeiner and Voges [14] also describe their experience in building and testing a two-version reactor protection system. The programs in their system were comparison tested after the completion of all other validation efforts. They report that 18 errors (approximately 14% of all detected errors) were found by their automatic comparison program. They conclude that the automatic comparison program is a valuable tool for the comparison of large amounts of data that would have been impossible to check accurately by hand.

In the development of the dual redundant software for the Airbus Industrie A310 slat and flap system, integration testing was begun without having performed module testing [23]. The explanation given for omitting module testing is that the versions "each provide for the other the most stringent test environments."

"Tracking" tests (automated dual-version comparison tests) were performed by the Commercial Flight Systems Division at Sperry Corporation in validating the SP-300 Critical Flight Control System [34]. Since the system designers were concerned about the possibility of faults common to the two versions, the tracking tests were only a part of the entire validation effort. An analysis of the System Problem Reports (SPR's) that were generated during the development process [10] revealed no common design implementation or coding errors. Also, the authors report that the use of comparison testing significantly contributed to the timely delivery of software change loads during flight testing. Errors were quickly discovered and eliminated during system level debugging.

Comparison testing is the only generally applicable method that has been suggested for automating the process of determining correct program outputs. Although, as the above examples illustrate, comparison testing has often been used or recommended, little research has been directed at evaluating the method.

Saglietti and Ehrenberger [29] offer an analysis of the level of assurance of the reliability of a comparison-tested N-version programming system. Their analysis is based on the assumption that the

probability that two versions will fail identically is less than the probability that they will disagree. They present empirical evidence to show that this assumption is reasonable, and we agree that the assumption is likely to be true when comparison testing is initiated. However, after the versions have been tested using comparison testing, the assumption is much less likely to hold. In fact, after exhaustive comparison testing the probability of disagreement would be zero, but if the versions contain faults that cause only identical failures, the probability of identical failure will be greater than zero. Thus the results presented by Saglietti and Ehrenberger rely on a faulty assumption.

Empirical evidence on the value of comparison testing has been limited to the experience of those who have used the method. The number of versions involved in each such study is two, and no effort has been made in any study to observe the program characteristics that would affect comparison testing. In some cases no testing other than comparison testing was performed, so it is not possible to study the characteristics of faults found by other methods but not by comparison testing. In other cases comparison testing was performed after the completion of conventional testing, allowing no observation of its effectiveness in finding faults eliminated by the other methods. It was observed by many of the researchers using the method that comparison testing was of value in finding faults missed by other methods. The preliminary results indicate that comparison testing is deserving of further consideration.

CHAPTER 3

NOTATION USED TO DESCRIBE FAULT INTERRELATIONSHIPS

In this chapter the fault interrelationships that affect comparison testing are described. A notation that will be used to represent these relationships is given.

3.1. Fault Interrelationships that Affect Comparison Testing

In order to determine the behavior of a comparison testing system in finding a particular fault, we need to look closely at the relationship between that fault's failure subspace and the failure subspaces of other faults in the system. Figure 3.1 shows the intersecting failure subspaces for two faults A and B. For

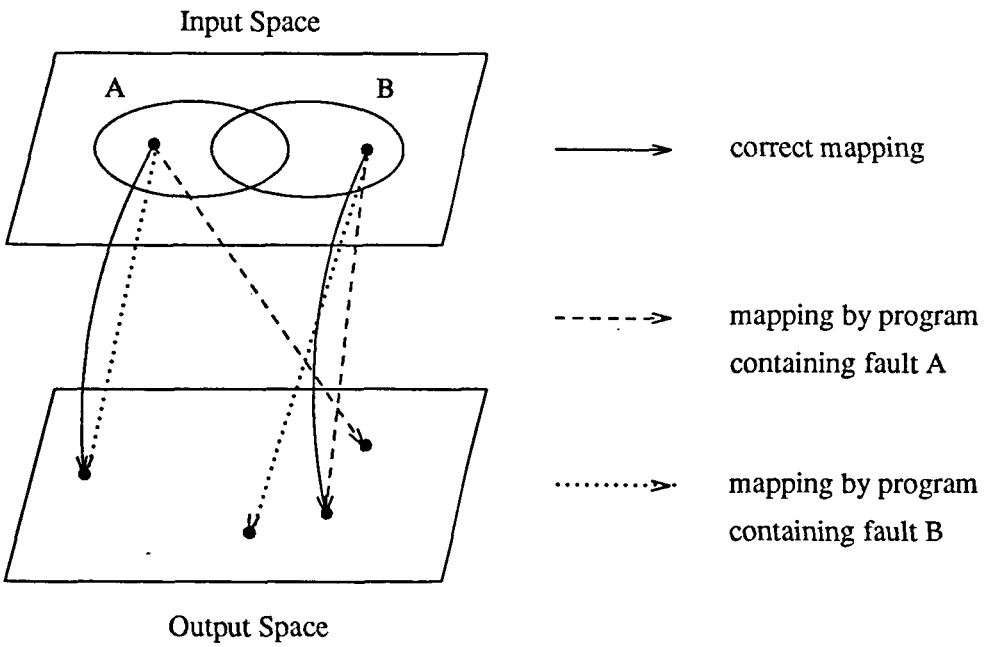


Figure 3.1. Illustration of Failure Subspaces

points that are in each fault's failure subspace, the fault causes an incorrect mapping into the output space.

The points in the intersection of two subspaces can be divided into two classes. Figure 3.2 illustrates one member of each class. The point on the left is a member of the first class, the class of points for which the faults cause non-identical failure. For these inputs, both faults cause failure, but the symptoms of failure differ. The point on the right represents an input on which the versions containing the faults fail identically. For these inputs the versions containing the faults obtain the same incorrect output.

The reason for distinguishing between the two classes of points in the intersection of failure subspaces is that they have different implications in a comparison testing system. If the point on the left

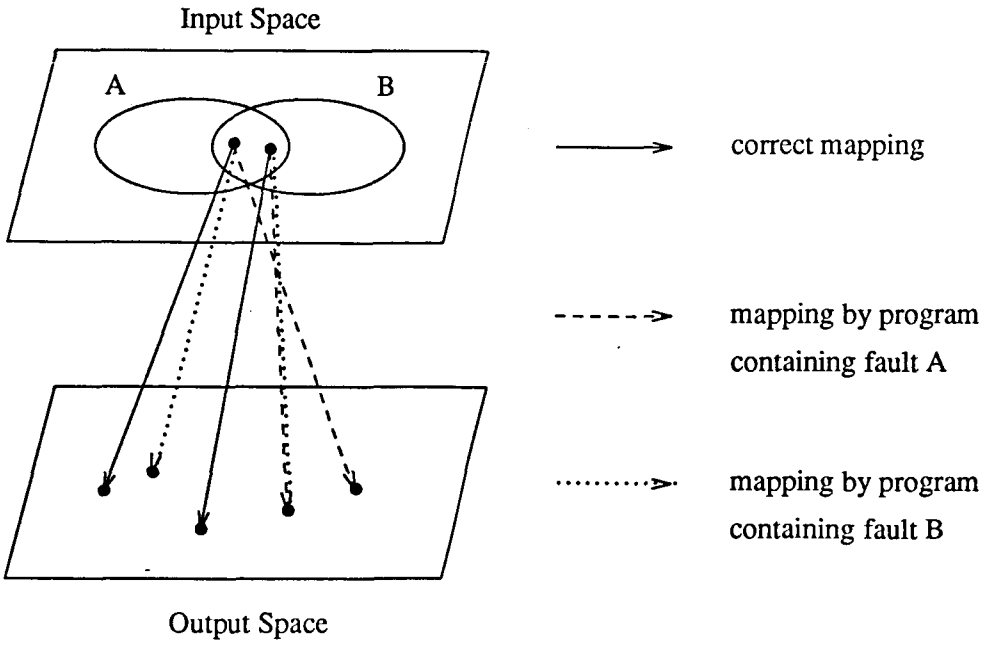


Figure 3.2. Two Classes of Points in Failure Subspace Intersection

is selected as a test case, both faults will be revealed. The point on the right will give identical outputs, so the version failures will not be recognized and investigated. A comparison testing system cannot distinguish such points from those lying outside of both failure subspaces.

In determining the information learned from each test case in a comparison testing system, it is important to know to which of several classes the input belongs. For two faults the relevant classes are:

(1) Fault A causes failure but fault B does not. The point on the left in Figure 3.1 belongs to this class.

(2) Fault B causes failure but fault A does not. The point on the right in Figure 3.1 belongs to this class.

(3) Both faults cause failure, but the outputs obtained by the programs containing the faults are different. The point on the left in Figure 3.2 belongs to this class.

(4) The faults cause identical failure. The point on the right in Figure 3.2 belongs to this class.

3.2. Notation

Each of the classes in the input space will be denoted C_x, where the subscript x denotes all of the faults that cause failure in the class. Commas are used to separate the faults into equivalence classes according to the output produced when failure occurs. For example, the subscript "A,BC" would indicate that faults A, B, and C all cause failure. The versions containing faults B and C obtain identical incorrect outputs, while the version containing fault A obtains a different incorrect output. The probabilities for the relevant classes for a two-fault system are, then:

(1) C_A : A causes failure, B does not

(2) C_B : B causes failure, A does not

(3) $C_{A,B}$: both faults cause failure, but exhibit different symptoms

(4) C_{AB} : A and B cause identical failure

The probability that a randomly selected test case will fall into a particular class C_x depends on the selection distribution. If Q is the selection distribution, then $Q(C_x)$ represents the probability that an input from class C_x is chosen. A shorthand notation, $q_x = Q(C_x)$, will be used to represent these probabilities. So, for example, q_A means the probability that an input will be selected on which fault A is the only fault that causes failure.

These classifications and probabilities are based on the original condition of the versions, and are not changed as the faults are found and corrected. For example, after fault B is corrected, q_B retains its original value and can be interpreted as the probability that fault B would have caused failure if it had not been corrected.

We will be interested in finding the number of test cases necessary to observe a fault. This quantity will be called the *observation time* for the fault. The observation time will be denoted T_S, where the subscript S identifies the testing system to which the parameter applies. This quantity is a random variable. Its distribution measures the effectiveness of a testing system in finding the fault.

We will focus on the observation time distribution for a single fault in one of the versions in a comparison testing system. This fault will always be labeled "A". Since the analysis will not depend on the selected fault, we will be able to determine the effectiveness of comparison testing in locating any fault. Knowledge about the behavior of comparison testing in finding a single arbitrary fault can be extended to determine its effectiveness in finding all faults.

The observation time for fault A, using comparison testing, will depend on the characteristics of faults in the other versions that make up the testing system. These other versions will be called the

comparison versions. The faults in the comparison versions will be labeled with upper-case letters other than A. When there are several faults in the same comparison version, all will be labeled with the same letter, and subscripts will be used to distinguish them. For example, in a two-version system, there is a single comparison version. Its n faults will be labeled $B_1, B_2, \ldots, B_n$.

CHAPTER 4

MARKOV MODELS OF FAULT OBSERVATION TIMES

In this chapter we develop models of the effects of interactions among faults on fault observation time. We will make the following assumptions for all of these models:

(1) Each fault is corrected as soon as it is observed in the testing process.

(2) No new faults are introduced in the correction process.

(3) The other faults in the version containing the fault A on which our attention is focused do not interfere in the observation of fault A.

Recall that in the comparison testing process a test case results in an investigation whenever there is a disagreement among the versions, so test cases on which a fault is triggered result in the observation of that fault unless all comparison versions fail in an identical manner. Thus the only faults which affect the observation time for a fault are faults in comparison versions that cause identical failure symptoms. The effects on comparison testing of faults causing identical failures are modeled here. The existence of other faults that never cause the same failure symptoms as a given fault does not affect the applicability of a model in predicting the observation time for that fault.

4.1. A Single Fault Interaction

In this section we consider a comparison testing system consisting of two versions. We will focus on the problem of determining the observation time for some fault A in one of the versions. A single fault (labeled B) in the comparison version causes identical failure symptoms as fault A.

The comparison testing process for this testing system is a discrete-time stochastic process that can be modeled by the Markov model illustrated in Figure 4.1. Each state in the Markov chain represents a state of knowledge about the faults in the programs during the testing process. Each p_{ij} represents the probability that a single test case will cause a transition from state S_i to state S_j.

Testing begins in state S_1, which represents the condition of having no knowledge of any faults in either program. The objective of the testing process is to reach state S_3, which represents the condition of having found fault A. State S_2 is the state in which fault B has been detected (and therefore corrected) but fault A has not yet been found.

In state S_1 we have no knowledge of any faults, so we can observe the presence of a fault only if either exactly one version fails or both versions fail and obtain different outputs. If no faults are revealed we remain in S_1, so p_{11} is $(1-(q_A+q_B+q_{A,B}))$. Fault A will be revealed if only A causes failure or if both versions fail differently, so p_{13} is $(q_A+q_{A,B})$. If only fault B causes failure, B will be revealed but not A, so p_{12} is q_B.

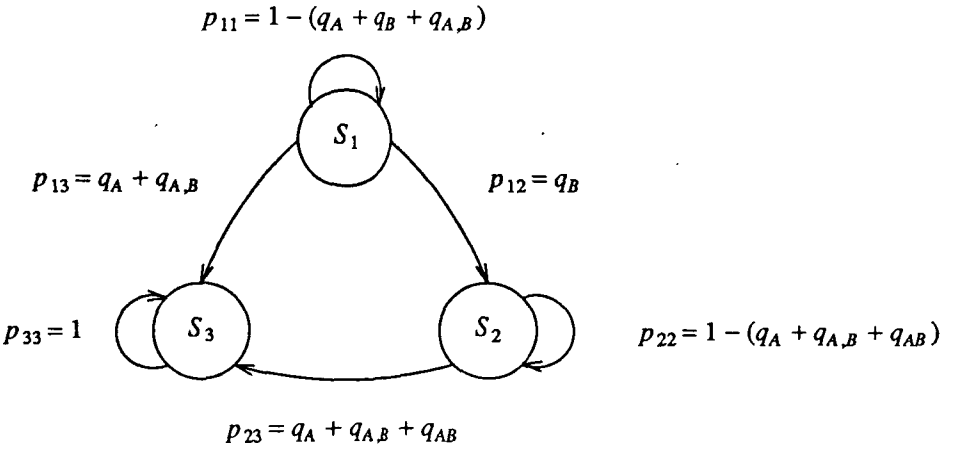


Figure 4.1. Markov Model of the Effects of Single Fault Interaction

Once fault B has been observed and corrected, any failure of the program containing fault A will be detected, so p_{23} is $(q_A+q_{A,B}+q_{AB})$. If A does not cause failure, we remain in state S_2, so p_{22} is $(1-(q_A+q_{A,B}+q_{AB}))$.

State S_3 is an absorbing state. Once we have found A, we have attained all of the knowledge that is of interest, so p_{33} is 1.

The quantity of interest is the observation time for fault A. In terms of the Markov model, this quantity is the *first-passage time* K_{13}. In general the first-passage time K_{ij} is the first time after time 0 at which a process is observed to be in state S_j, after having been initially in state S_i [9]. We are interested in the probability distribution of the discrete random variable K_{13}.

It is possible to arrive at state S_3 for the first time at time t by either of two routes:

(1) Remain in state S_1 for $(t-1)$ test cases, and then move to S_3 on the tth test case. The probability that this will occur is:

$$(1-(q_A+q_B+q_{A,B}))^{t-1}(q_A+q_{A,B}).$$

(2) Remain in state S_1 for $(i-1)$ test cases $(1 \le i \le t-1)$, then move to state S_2, remaining for $(t-i-1)$ test cases, finally moving to S_3 on the tth test case. The probability that this will occur, for each value of i, is:

$$(1-(q_A+q_B+q_{A,B}))^{i-1}(q_B)(1-(q_A+q_{A,B}+q_{AB}))^{t-i-1}(q_A+q_{A,B}+q_{AB}).$$

The probability distribution of K_{13}, then, is:

$$P(K_{13}=t)=(1-(q_A+q_B+q_{A,B}))^{t-1}(q_A+q_{A,B})+$$
$$\sum_{i=0}^{t-2}[(1-(q_A+q_B+q_{A,B}))^{i-1}(q_B)(1-(q_A+q_{A,B}+q_{AB}))^{t-i-1}$$
$$(q_A+q_{A,B}+q_{AB})],\ t=1,2,...$$

Let m_{ij} represent the expected value of the first-passage time from S_i to S_j. Then m_{13}, the expected value of K_{13}, can be found by solving the following set of equations [9]:

$$m_{13} = 1 + p_{11}m_{13} + p_{12}m_{23}$$

$$m_{23} = 1 + p_{21}m_{13} + p_{22}m_{23}$$

$$m_{33} = 1 + p_{31}m_{13} + p_{32}m_{23}$$

Substituting the values of p_{ij} for this Markov chain yields the set of equations:

$$m_{13} = 1 + (1 - (q_A + q_B + q_{A,B}))(m_{13}) + (q_B)(m_{23})$$

$$m_{23} = 1 + (1 - (q_A + q_{A,B} + q_{AB}))(m_{23})$$

$$m_{33} = 1$$

Solving this system of equations yields:

$$E[T_{comparison}] = m_{13} = \frac{q_A + q_B + q_{A,B} + q_{AB}}{(q_A + q_B + q_{A,B})(q_A + q_{A,B} + q_{AB})}.$$

4.2. Multiple Fault Overlaps in a Two-Version System

The behavior of comparison testing in revealing a fault that causes identical failures with more than one fault in the comparison version is considered in this section. The faults in the comparison version causing failures identical to those caused by fault A are faults $B_1, B_2, B_3, ..., B_n$.

We make the following assumptions in addition to those listed at the beginning of this chapter:

(1) When a disagreement among versions arises, an effort will be directed toward finding the cause of the disagreement. The intermediate results of the two versions will be compared to narrow down the source of the disagreement. Thus only the fault or faults responsible for disagreements between the versions will be observed and corrected. For example, if, on a given test case, faults A and B_1, which cause identical symptoms, are both triggered, and fault B_3, which causes distinct symptoms, is triggered as well, then only fault B_3 will be corrected.

(2) Two faults in the comparison version that are triggered on the same test case will cause distinct symptoms. Both faults must be corrected in order to achieve the correct output for the input that triggers them.

(3) If, for a particular test case, none of the individual faults in the comparison version causes identical failure with fault A, then it is not possible for any combination of these faults to result in an identical failure.

The table in Figure 4.2 shows all of the relevant subspaces of the input space, given the above assumptions, for the case in which fault A interacts with two faults, B_1 and B_2. The first column in the table gives the faults triggered by inputs in the subspace. These are assigned to sets according to the symptoms of failure caused by the faults. Faults causing identical failure on an input in the subspace are assigned to the same set. The second column gives the probability that an input from the subspace will be

Faults Triggered	Probability	Faults Observed
$\{A\}$	q_A	A
$\{B_1\}$	q_{B_1}	B_1
$\{B_2\}$	q_{B_2}	B_2
$\{A, B_1\}$	q_{AB_1}	None
$\{A, B_2\}$	q_{AB_2}	None
$\{A\}, \{B_1\}$	q_{A,B_1}	A, B_1
$\{A\}, \{B_2\}$	q_{A,B_2}	A, B_2
$\{B_1\}, \{B_2\}$	q_{B_1,B_2}	B_1, B_2
$\{A, B_2\}, \{B_1\}$	q_{AB_2,B_1}	B_1
$\{A, B_1\}, \{B_2\}$	q_{AB_1,B_2}	B_2
$\{A\}, \{B_1\}, \{B_2\}$	q_{A,B_1,B_2}	A, B_1, B_2

Figure 4.2. Relevant Subspaces for Interaction with Two Faults

selected. The third column shows which of the triggered faults can be observed, assuming no prior knowledge of any of the faults.

The process of finding faults through comparison testing can be modeled by a Markov chain similar to the one discussed in the previous section. The two-overlap model is illustrated in Figure 4.3. In addition to the transitions shown in the figure, there is a transition from each state i to itself, with probability p_{ii}.

As in the Markov model discussed in Section 4.1, each state represents the state of knowledge about the faults in the versions. In the general n-fault case, there are 2^n intermediate states. Each intermediate state is associated with one of the subsets of the set $\beta = \{B_1, B_2, B_3, ..., B_n\}$. Let ψ_i be the set of faults

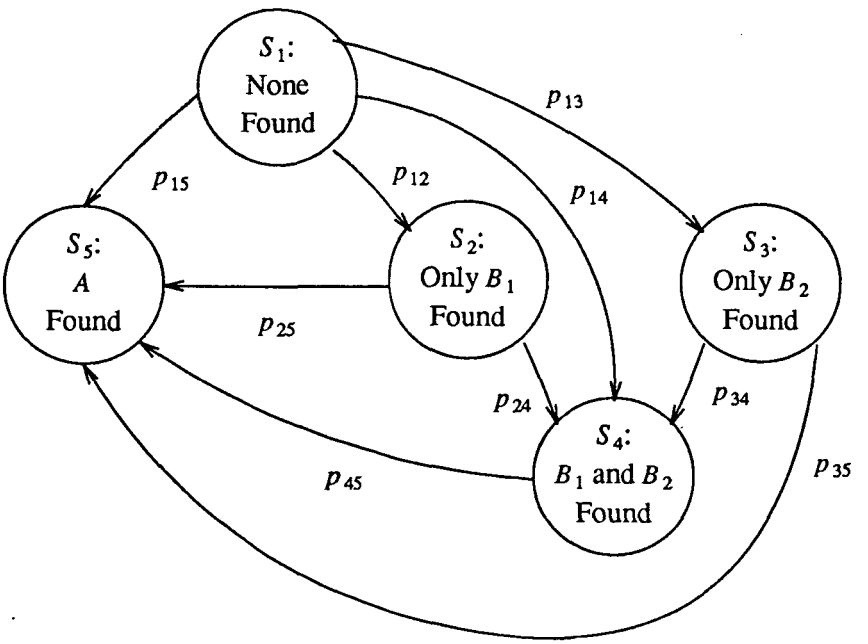


Figure 4.3. Markov Model for Interaction with Two Faults

associated with state S_i. Then state S_i represents the state in which every fault in ψ_i has been found. ψ_1, the set associated with the initial state, is the empty set, since testing begins with no knowledge of any faults. There is a single additional final state S_f ($f = 2^n + 1$), which represents the state in which fault A, the fault of interest, has been found.

To facilitate discussion in defining the transition probabilities p_{ij} ($1 \le i \le 2^n + 1, 1 \le j \le 2^n + 1$), let $L(\psi_k)$ be a listing of the elements in set ψ_k, *i.e.* if

$$\psi_k = \{X_1, X_2, ..., X_n\}$$

then

$$L(\psi_k) = X_1, X_2, ..., X_n.$$

A transition between an intermediate state S_i and the final state S_f occurs whenever fault A is observed. Fault A can be observed whenever both of the following conditions are satisfied:

(1) Fault A causes failure.

(2) Any fault in the comparison version that causes the same symptoms of failure as fault A has already been found.

Thus for ($1 \le i \le 2^n, f = 2^n + 1$) we have:

$$p_{if} = \sum_{\psi \in 2^{\beta}} q_{A,L(\psi)} + \sum_{X \in \psi_i} \sum_{\psi \in 2^{\beta-\{X\}}} q_{AX,L(\psi)}$$

where 2^{β} and $2^{\beta-\{X\}}$ denote the power sets of $\beta = \{B_1, B_2, ..., B_n\}$ and (β - {X}) respectively.

A transition between two intermediate states S_i and S_j occurs when a test case is selected which reveals a previously undiscovered fault (or faults) in the comparison program (but not fault A, since its discovery leads to the final state). Knowledge of a previously discovered fault is never lost. Thus for ($1 \le i \le 2^n, 1 \le j \le 2^n, i \ne j$) we have:

$$p_{ij} = \begin{cases} \sum_{\psi \in \gamma} q_{L(\psi)} + \sum_{X \in \beta - \psi_i} \sum_{\psi \in \gamma} q_{AX,L(\psi)} & \text{if } \psi_i \ \beta \ \psi_j \\ 0 & \text{otherwise} \end{cases}$$

where

$$\gamma = \{\psi \mid \psi \cup \psi_i = \psi_j\}.$$

The probabilities of remaining in each state can be determined in terms of the other transition probabilities:

$$p_{ii} = 1 - \sum_{j \neq i} p_{ij} \ (1 \leq i \leq 2^n + 1, 1 \leq j \leq 2^n + 1).$$

The transition probabilities for the two-fault model shown in Figure 4.3 are, in accordance with the above rules:

$$p_{11} = 1 - (q_{B_1} + q_{AB_2,B_1} + q_{B_2} + q_{AB_1,B_2} + q_{B_1,B_2} + q_A + q_{A,B_1} + q_{A,B_2} + q_{A,B_1,B_2})$$

$$p_{12} = q_{B_1} + q_{AB_2,B_1}$$

$$p_{13} = q_{B_2} + q_{AB_1,B_2}$$

$$p_{14} = q_{B_1,B_2}$$

$$p_{15} = q_A + q_{A,B_1} + q_{A,B_2} + q_{A,B_1,B_2}$$

$$p_{22} = 1 - (q_{B_2} + q_{B_1,B_2} + q_A + q_{A,B_1} + q_{A,B_2} + q_{A,B_1,B_2} + q_{AB_1} + q_{AB_1,B_2})$$

$$p_{24} = q_{B_2} + q_{B_1,B_2}$$

$$p_{25} = q_A + q_{A,B_1} + q_{A,B_2} + q_{A,B_1,B_2} + q_{AB_1} + q_{AB_1,B_2}$$

$$p_{33} = 1 - (q_{B_1} + q_{B_1,B_2} + q_A + q_{A,B_1} + q_{A,B_2} + q_{A,B_1,B_2} + q_{AB_2} + q_{AB_2,B_1})$$

$$p_{34} = q_{B_1} + q_{B_1,B_2}$$

$$p_{35} = q_A + q_{A,B_1} + q_{A,B_2} + q_{A,B_1,B_2} + q_{AB_2} + q_{AB_2,B_1}$$

$$p_{44} = 1 - (q_A + q_{A,B_1} + q_{A,B_2} + q_{A,B_1,B_2} + q_{AB_1} + q_{AB_2} + q_{AB_1,B_2} + q_{AB_2,B_1})$$

$$p_{45} = q_A + q_{A,B_1} + q_{A,B_2} + q_{A,B_1,B_2} + q_{AB_1} + q_{AB_2} + q_{AB_1,B_2} + q_{AB_2,B_1}$$

$p_{55} = 1$

As in the case of a single fault interaction, the quantity of interest is the first-passage time K_{1f}. The expected value of this random variable, m_{1f}, can be found by solving this set of simultaneous equations [9]:

$$m_{1f} = 1 + p_{11}m_{1f} + p_{12}m_{2f} + p_{13}m_{3f} + \cdots + p_{1(f-1)}m_{(f-1)f}$$

$$m_{2f} = 1 + p_{21}m_{1f} + p_{22}m_{2f} + p_{23}m_{3f} + \cdots + p_{2(f-1)}m_{(f-1)f}$$

$$m_{3f} = 1 + p_{31}m_{1f} + p_{32}m_{2f} + p_{33}m_{3f} + \cdots + p_{3(f-1)}m_{(f-1)f}$$

$$\vdots$$

$$m_{nf} = 1 + p_{n1}m_{1f} + p_{n2}m_{2f} + p_{n3}m_{3f} + \cdots + p_{n(n-1)}m_{(n-1)f}$$

CHAPTER 5

AN ANALYSIS OF MODEL IMPLICATIONS

In Chapter 4 models were developed that show how fault interactions affect the performance of comparison testing. These models can be used to predict the performance characteristics of a comparison testing system composed of versions containing faults whose interactions are known. Thus these models can be used to interpret empirical data such as that available from the N-version experiment. However, the expressions for the observation time distributions derived from the models are complex and depend on a large number of parameters. Such expressions do not lend themselves to an intuitive understanding of important relationships and parameter sensitivities.

In this chapter the simplest of our models, the single fault interaction model introduced in Section 4.1, is reparameterized. A simplifying assumption is used to yield a more easily understood formula for the expected value of the fault observation time.

5.1. A Basis for Evaluation: The Ideal Test Bed

It is not possible to evaluate a method adequately without comparing it to the alternatives. The ideal alternative to comparison testing would be a system in which an *oracle* would evaluate the correctness of any result computed by the program being tested. Of course if an automated oracle were available we would not consider using comparison testing.

In applications for which comparison testing would be considered, the alternative test-interpreter would likely be one of the following:

(1) An automatic "envelope oracle" may be available. For example, in testing flight control software a flight simulator is generally used. The flight simulator will detect those incorrect outputs that would

cause an observably incorrect flight pattern. A fault that causes serious consequences during simulation testing will be observed. However, a fault that sometimes has serious consequences may under some circumstances cause incorrect outputs that are "close enough" to cause no observable flight pattern anomaly. Thus there may be test cases for which a serious fault is triggered but not detected.

(2) An unautomated "near oracle" may be available. Test outputs are often evaluated by performing a hand calculation. This type of verification provides at best a "near oracle" since the calculation itself is subject to human error. It is also possible that the algorithm used in the hand calculation contains faults. In a sense, the hand calculation can be viewed as another version, so that a tedious form of comparison testing may be the alternative to an automated comparison testing system.

Whatever the alternatives, they can be no better than an ideal, oracle-based testing system. If the fault-revealing power of comparison testing approaches the ideal, then it provides a valuable alternative or supplementary approach where conventional methods provide only an envelope or an unautomatable, unreliable correctness test.

In an ideal test bed it would be possible to observe a fault on the first test case on which it causes failure. With respect to revealing a particular fault each test case is a Bernoulli trial, with "success" defined as finding a case on which the particular fault is triggered. Therefore the observation time T_{ideal} for fault A in an ideal test bed follows a geometric distribution with parameter $p = q_A + q_{A,B} + q_{AB}$. Its distribution, then, is [2]:

$$P(T_{ideal} = t) = (q_A + q_{A,B} + q_{AB})(1 - (q_A + q_{A,B} + q_{AB}))^{t-1}.$$

Its expected value is:

$$E[T_{ideal}] = \frac{1}{q_A + q_{A,B} + q_{AB}}.$$

5.2. Performance of Comparison Testing

A comparison testing system differs from the ideal in that identical failures of the versions comprising the system hide the faults that cause these failures. The effect of these identical failures will be a (possibly infinite) delay in observing the faults that cause them. The observation times for faults that cause identical failures will be longer in a comparison testing system than in an ideal test bed.

Here we reconsider the model developed in Section 4.1. In this model the fault under consideration (fault A) causes failures identical to those caused by only one fault (B) in the comparison version. The expression derived for the expected fault observation time is:

$$E[T_{comparison}] = \frac{q_A + q_B + q_{A,B} + q_{AB}}{(q_A + q_B + q_{A,B})(q_A + q_{A,B} + q_{AB})}.$$

The model depends on the classification of test cases into one of the four classes C_A, C_B, $C_{A,B}$, and C_{AB}. Notice that test cases in $C_{A,B}$ reveal the same information about fault A as those in C_A. Therefore we can combine the two classes into $\tilde{C}_A = C_A \cup C_{A,B}$. The probability of selecting an input in this class is $\tilde{q}_A = q_A + q_{A,B}$. Thus the expression for the expected fault observation time becomes:

$$E[T_{comparison}] = \frac{\tilde{q}_A + q_B + q_{AB}}{(\tilde{q}_A + q_B)(\tilde{q}_A + q_{AB})}.$$

Let p be the probability that fault A causes failure, and let $r = \frac{q_{AB}}{p}$ be the ratio of the coincident identical failures to the total failures caused by the fault under examination. Using this notation we have:

$$p = \tilde{q}_A + q_{AB}$$

$$q_{AB} = rp$$

$$\tilde{q}_A = p - rp = (1 - r)(p).$$

For the purposes of further analysis, we assume that, on the average, the probability of failure caused by fault B is equal to the probability of failure caused by fault A, so we assume that:

$$q_B = q_A = (1-r)p.$$

The sensitivity of the observation time to this quantity will be considered in Chapter 6. With this assumption we have:

$$E[T_{comparison}] = \frac{2-r}{p(2-2r)}.$$

Applying the same assumptions and notation to compute the expected time to find a fault in an ideal test bed gives:

$$E[T_{ideal}] = \frac{1}{q_A + q_{A,B} + q_{AB}} = \frac{1}{p}.$$

Whether comparison testing or an oracle is used, the observation time for a fault is inversely proportional to p, the probability that the fault manifests itself on a randomly selected test case. The ratio:

$$\frac{E[T_{comparison}]}{E[T_{ideal}]} = \frac{2-r}{2-2r},$$

then, depends only on r and gives a measure of the effect of using comparison testing. The curve in Figure 5.1 shows the functional relationship between r and $\frac{E[T_{comparison}]}{E[T_{ideal}]}$.

It is interesting that, unless r is very large, the observation time for a fault is not much greater than it would be if an oracle were available. As expected, as r approaches 1, $\frac{E[T_{comparison}]}{E[T_{ideal}]}$ becomes infinite. If a fault in one of the two versions in a comparison testing system occurs on the exact subset of test cases as a fault in the other version, then none of the failures caused by either fault is detectable by comparison testing, and neither of the two faults will ever be found.

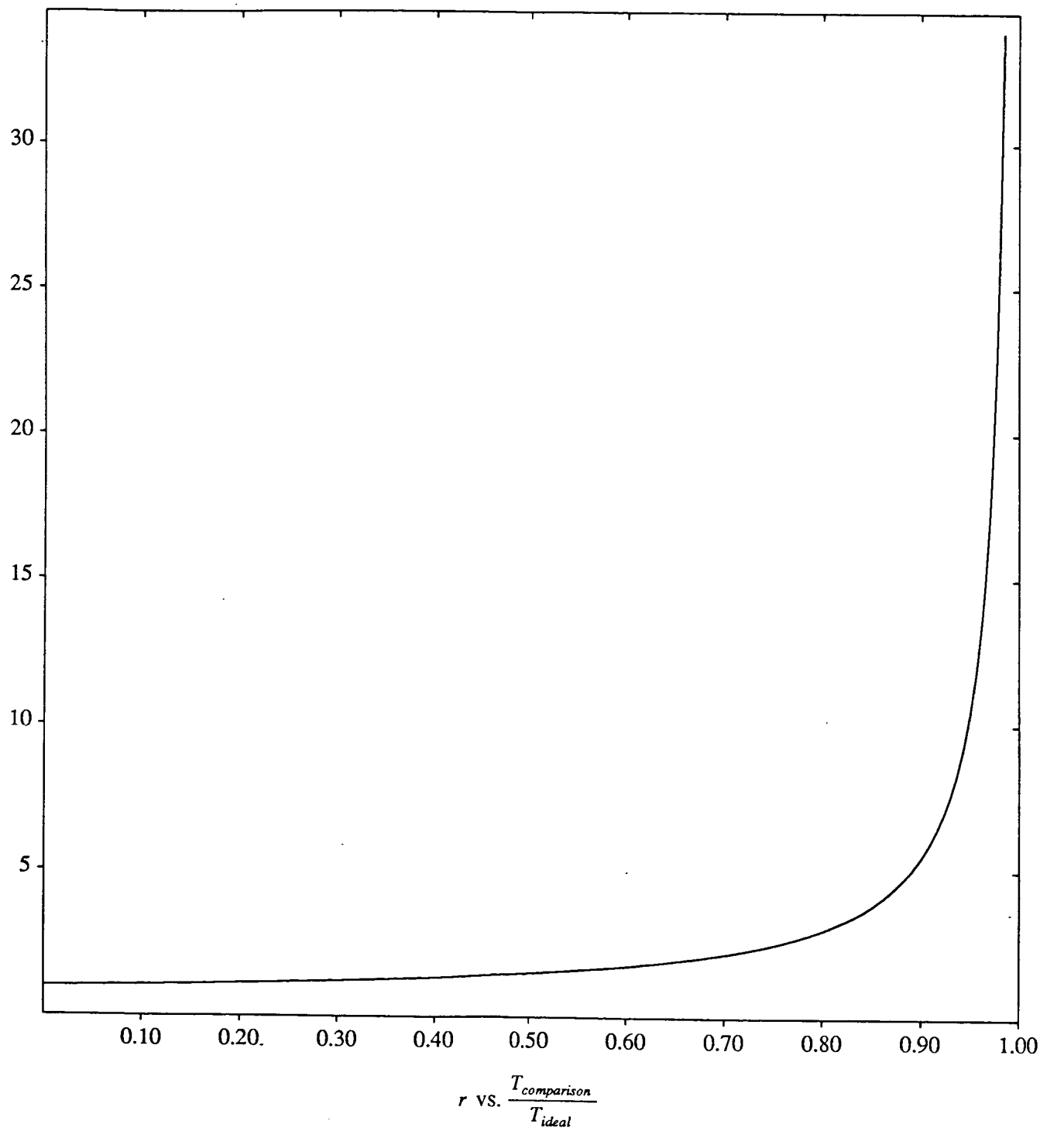


Figure 5.1. Effect of *r* on Fault Observation Time

CHAPTER 6

WORST CASE ANALYSIS

In this analysis of comparison testing we are particularly concerned with finding the maximum penalty that will result from using comparison testing rather than some other method that more closely approaches the ideal. In this chapter we define the concept of the *overlap ratio* for a fault, and obtain bounds on the expected value of the fault observation time for a fault having a given probability of occurrence and overlap ratio.

6.1. Worst Case Performance with Single Fault Overlap

In this section we consider again the model for a two-version system in which a single fault interaction involves fault A. The initial discussion of the model can be found in Section 4.1. Our goal here is to simplify the expression for the expected comparison testing observation time by eliminating q_B as a parameter. In eliminating q_B we want to ensure that we obtain the worst case performance of the testing system, assuming that other parameters remain fixed.

We begin by finding the value of q_B that will maximize:

$$E[T_{comparison}] = \frac{q_A + q_B + q_{A,B} + q_{AB}}{(q_A + q_B + q_{A,B})(q_A + q_{A,B} + q_{AB})},$$

given that all other quantities remain constant.

The partial derivative of this quantity with respect to q_B is:

$$\frac{-q_{AB}}{(q_A + q_{A,B} + q_{AB})(q_A + q_B + q_{A,B})^2}.$$

Each q_x is a probability, and therefore can only take on values in the range $0 \le q_x \le 1$. In determining the meaning of the partial derivative, several cases are relevant:

(1) When either of the quantities $(q_A + q_{A,B} + q_{AB})$ or $(q_A + q_B + q_{A,B})$ is zero, the partial derivative is infinite. But if either quantity is zero then $E[T_{comparison}]$ is infinite, regardless of the value of q_B. For this case, then, *any* value of q_B will maximize $E[T_{comparison}]$.

(2) When q_{AB} is zero the partial derivative is zero for all values of q_B. Therefore the value of q_B has no effect on $E[T_{comparison}]$. This result is expected, since finding B affects the state of our knowledge about A only when q_{AB} is nonzero, and so the probability of finding only B is irrelevant. When q_{AB} is zero the Markov model can be reduced to two states by combining states S_1 and S_2, because the transitions from each of these states to S_3, the state of interest, are equally likely. For this case also, then, any value of q_B will maximize $E[T_{comparison}]$.

(3) When q_B and each of the quantities $(q_A + q_{A,B} + q_{AB})$ and $(q_A + q_B + q_{A,B})$ is positive, the partial derivative is negative, indicating that $E[T_{comparison}]$ decreases as q_B increases. Its maximum value, then occurs at the minimum value of q_B, zero.

Substituting $q_B = 0$ into the expression for the expected observation time derived from the Markov model yields:

$$E[T_{comparison}] = \frac{q_A + q_{A,B} + q_{AB}}{(q_A + q_{A,B})(q_A + q_{A,B} + q_{AB})}.$$

Using the notation of Section 5.2, we have:

$$E[T_{comparison}] = \frac{1}{(1-r)p}$$

and

$$\frac{E[T_{comparison}]}{E[T_{ideal}]} = \frac{1}{1-r}.$$

This is not a surprising result if we examine the way in which faults are found in a comparison testing system. In the two-version, single fault interaction system on which this analysis is based, fault A can be observed by either of two methods:

(1) A test case in C_A or $C_{A,B}$ can be selected, resulting in the immediate observation of fault A.

(2) A test case in C_B can be selected, resulting in the observation and correction of fault B. Subsequently fault A can be observed on test cases in C_{AB}, as well as on those in C_A and $C_{A,B}$.

These two methods correspond to the two routes from state S_1 to state S_3 in the Markov model.

When q_B is zero, the second route is cut off. State S_2, in which B has been observed but A has not, is unreachable. The only way to observe fault A is the direct method of observing a test case in C_A or $C_{A,B}$.

With $q_B = 0$, each test case is a Bernoulli trial, with "success" in observing a fault occurring when a test case in C_A or $C_{A,B}$ is selected. The probability of success on each test case is:

$$q_A + q_{A,B} = (1 - r)p.$$

The observation time would follow a geometric distribution with parameter $\hat{p} = (1 - r)p$:

$$P(T_{comparison} = t) = [(1 - r)p][1 - (1 - r)p]^{t-1}.$$

Its expected value would be:

$$E[T_{comparison}] = \frac{1}{(1 - r)p}.$$

This is the same result obtained by substituting $q_B = 0$ into the expression for $E[T_{comparison}]$ obtained from the Markov model.

The upper curve in Figure 6.1 is the upper bound on $\frac{E[T_{comparison}]}{E[T_{ideal}]}$ that occurs when $q_B = 0$. The basic shape of the curve is the same as that of the lower curve, which is based on the "average" q_B, i.e. $q_B = \bar{q}_A$. This indicates that measuring the ratio:

$$r = \frac{q_{AB}}{p}$$

provides sufficient information to determine whether comparison testing will be an effective approach to finding the fault. We will refer to this quantity as the *overlap ratio* for fault A.

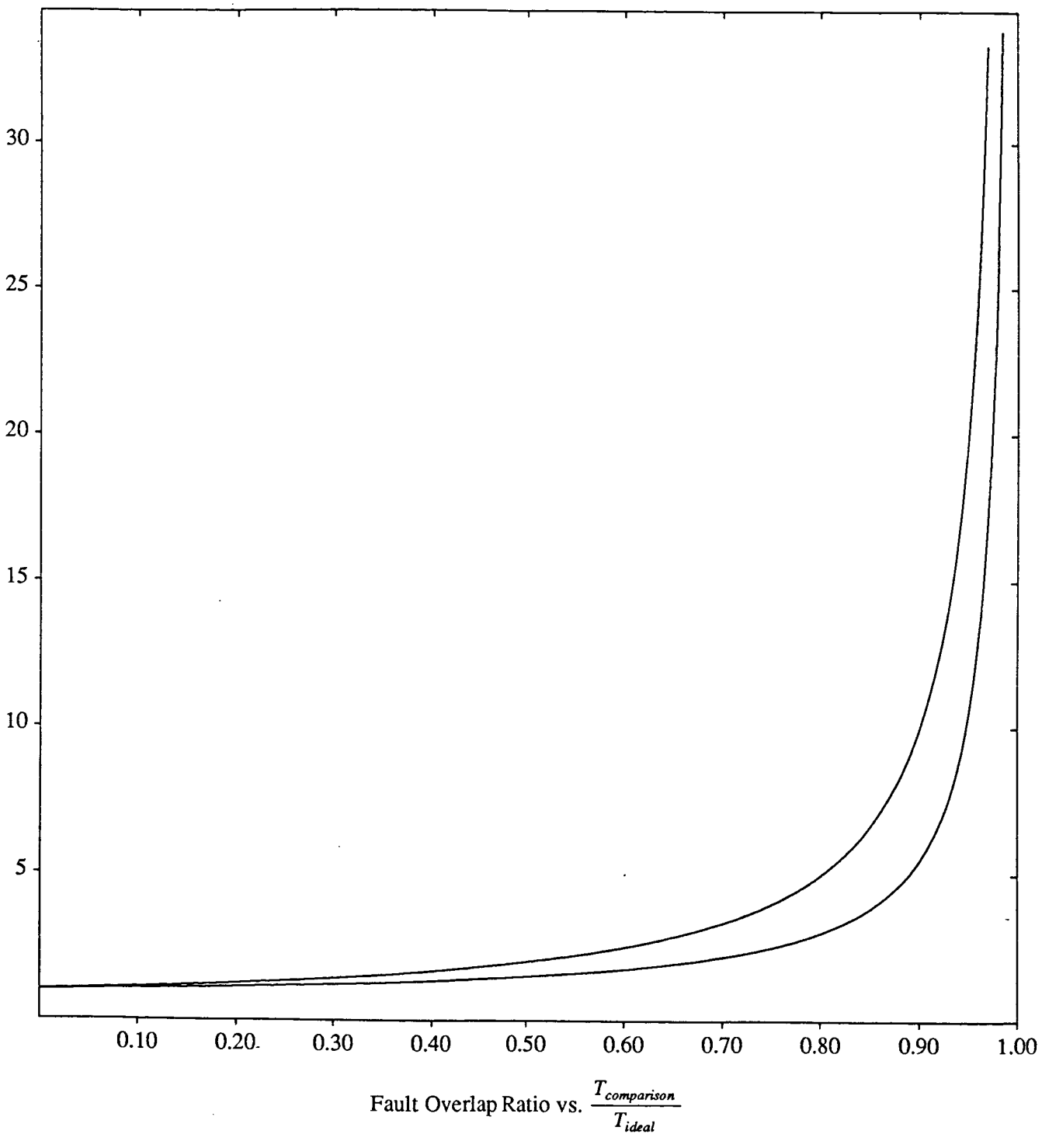


Figure 6.1. Comparison of Worst and "Average" Case

6.2. Extended Definition of Overlap Ratio

The reason that a comparison testing system may be less effective than the ideal in revealing a given fault is that faults in the comparison versions may cause identical failures with the fault, causing that fault to be at least partially "hidden" from the tester. The portion of the input space in which fault A causes failure and all comparison versions contain faults which cause identical failure can be thought of as the *overlap subspace*. The *overlap ratio* for a fault can be defined as the ratio of the probability of selecting a test case in the fault's overlap subspace to the probability of selecting a test case in the fault's failure subspace.

This extended definition for the overlap ratio is consistent with the definition given in the previous section for the overlap ratio in a two-version, single interaction system. The definition for a general two-version system (as described in Section 4.2) is:

$$r = \frac{\sum_{X \in \beta} q_{AX} + \sum_{X \in \beta} \sum_{\Psi \in 2^{\beta - \{X\}}} q_{AX} L(\Psi)}{p}$$

In general a fault A can be found by either of two approaches:

(1) A test case on which fault A is triggered and at least one other version disagrees with the version containing A.

(2) Some fault in a comparison version is observed and eliminated. Subsequently a test case on which that fault would have caused an identical failure with A is selected, resulting in the observation of fault A.

For a fault having a given overlap ratio, the worst performance of a comparison testing system will occur when none of the faults that cause identical failures are independently observable, *i.e.* the first approach is the only approach that can lead to the observation of the fault. In that situation fault A can be

observed only by selecting a test case in which it does not cause identical failure with any other fault. The observation time follows a geometric distribution with parameter:

$$\hat{p} = (1 - r)p$$

as described in the previous section.

CHAPTER 7

DISPERSIONS OF OBSERVATION TIME DISTRIBUTIONS

The analysis up to this point has been based entirely on the relationship between the expected values of $T_{comparison}$ and T_{ideal}. We are also interested in the relative dispersions of the probability distributions, since a wide variability in observation times would indicate a variability in the performance of the related testing method.

7.1. Variances

In order to compare the dispersion of observation times in comparison testing with the ideal, we calculate the variances of the probability distributions.

Recall that T_{ideal} followed a geometric distribution with parameter p. Its variance, then, is [2]:

$$V[T_{ideal}] = \frac{1-p}{p^2}.$$

We are primarily interested in the variance for the worst-case performance of comparison testing. As explained in Chapter 6, the worst-case observation time for a fault having overlap ratio r follows a geometric distribution with parameter $\hat{p} = (1-r)p$. Its variance, then, is:

$$V[T_{comparison}] = \frac{1-[(1-r)p]}{[(1-r)p]^2} = \frac{1-p+rp}{(1-r)^2p^2}$$

Since we assume that faults with very high probabilities of occurrence will be eliminated before testing begins, and since such faults are easier to find by any testing method, we are primarily interested in studying the effectiveness of comparison testing in finding faults with small probabilities of occurrence, *i.e.* faults having small values of p. When p is very small, both p and rp are much smaller than one, so we have:

$$V[T_{comparison}] \approx \frac{1}{(1-r)^2 p^2}$$

and

$$V[T_{ideal}] \approx \frac{1}{p^2}$$

and so, since the standard deviation σ is the square root of the variance [2], we have:

$$\sigma_{comparison} \approx \frac{1}{(1-r)p}$$

and

$$\sigma_{ideal} \approx \frac{1}{p}$$

Both of the standard deviations of the populations are approximately equal to the population means, indicating that both distributions are fairly flat and wide. Both testing methods, then, exhibit a variability in effectiveness. The variability in performance of comparison testing, as well as its expected performance, becomes increasingly worse than the ideal as r increases.

7.2. Confidence Interval Bounds

In order to find a more meaningful measure of the effects of observation time dispersion on the performance of comparison testing, consider the goal of the testing process. Since the purpose of testing software is to assure its quality, we want to ensure that there is a high probability that each fault will be removed. For example, we might want to perform the number of tests necessary to ensure that there is a 0.95 probability that a given fault will be observed. It is necessary, then, to determine the value of U such that:

$$P(T \le U) = 0.95.$$

In general, we want to find U, the upper bound on the one-sided confidence interval, such that:

$$P(T \le U) = C.$$

The probability distribution for T_{ideal} is the geometric distribution [2]:

$$P(T_{ideal} = t) = p(1-p)^{t-1}, t = 1,2,...$$

so

$$P(T_{ideal} \leq U) = C$$

implies that

$$\sum_{t=1}^{U} p(1-p)^{t-1} = C$$

$$p\left[\frac{1-(1-p)^{U+1}}{1-(1-p)}\right] = C$$

$$(1-p)^{U+1} = 1-C$$

$$U+1 = \frac{\log(1-C)}{\log(1-p)}$$

$$U_{ideal} = \frac{\log(1-C)}{\log(1-p)} - 1.$$

For comparison testing, again consider the case in which q_B and $q_{A,B}$ are both zero. For this case the probability distribution for $T_{comparison}$ is:

$$P(T_{comparison} = t) = (1-q_A)^{t-1}(q_A)$$

so

$$P(T_{comparison} \leq U) = C$$

implies

$$U_{comparison} = \frac{\log(1-C)}{\log(1-q_A)} - 1 = \frac{\log(1-C)}{\log(1-p+rp)} - 1$$

Since U_{ideal} and $U_{comparison}$ represent the number of tests necessary to achieve a 95% probability of finding a fault, the ratio:

$$\frac{U_{comparison}}{U_{ideal}} = \frac{\frac{\log(1-C)}{\log(1-p+rp)} - 1}{\frac{\log(1-C)}{\log(1-p)} - 1}$$

provides a measure of the performance of comparison testing as compared to an ideal test bed.

Unfortunately, unlike the ratio of expected values, the ratio of confidence interval bounds depends on the values of p and C as well as r, so the bound ratio is not as easy to interpret.

The number of test cases needed to achieve a given confidence that a fault will be observed increases as the confidence level increases. For example, the table in Figure 7.1 shows the effect of the choice of confidence level on the upper bound of the confidence interval when $p = 10^{-6}$. The confidence interval bounds for an ideal test bed are shown in the second column of the table. The bounds for a comparison testing system were computed for two values of the overlap ratio, 0.50 and 0.98. These bounds are shown in the third and fifth columns of Figure 7.1. The confidence interval bound increases rapidly as the confidence level increases for both the comparison testing system and the ideal test bed. The *ratios* of bounds (see the fourth and sixth columns), however, *remain nearly constant* for each of the values of r, at least for the parameter values used in constructing the table. If the ratio of bounds exhibits the same constant behavior over the relevant ranges for the parameters p and C, then this ratio provides a good measure of the performance of comparison testing.

C	U_{ideal}	$r = 0.50$		$r = 0.98$	
		$U_{comparison}$	$\frac{U_{comparison}}{U_{ideal}}$	$U_{comparison}$	$\frac{U_{comparison}}{U_{ideal}}$
0.50	693,146	1,386,293	2.0000019	34,657,358	50.0000952
0.90	2,302,583	4,605,168	2.0000009	115,129,252	50.0000458
0.95	2,995,730	5,991,462	2.0000008	149,786,611	50.0000408
0.99	4,605,167	9,210,337	2.0000007	230,258,506	50.0000351

Figure 7.1. Effect of Confidence Level on Confidence Interval Bound

Since we are interested in determining the performance of comparison testing in finding faults with realistic occurrence rates, the relevant range for the parameter p is approximately $10^{-10} \leq p \leq 10^{-2}$. A fault with a probability of occurrence greater than 10^{-2} will probably be eliminated in the debugging stage before testing for quality assurance begins. Faults with occurrence rates less than 10^{-10} are unlikely to be found by any testing method, and so will be regarded as irrelevant in evaluating comparison testing.

A testing method should be able to achieve a high level of confidence that a given fault is eliminated, so in evaluating comparison testing the number of tests necessary to achieve relatively high confidence levels is of interest. The relevant range for C, then, is assumed to be $C \geq 0.90$.

In order to determine the behavior of the ratio of confidence interval bounds over the relevant parameter ranges, the ratio was computed for all combinations of these values for p and C:

r	$\frac{E[T_{comparison}]}{E[T_{ideal}]}$	Minimum Ratio of Confidence Interval Bounds			Maximum Ratio of Confidence Interval Bounds		
		$\frac{U_{comparison}}{U_{ideal}}$	C	p	$\frac{U_{comparison}}{U_{ideal}}$	C	p
0.05	1.0526	1.0526	0.99999	10^{-8}	1.0531	0.90	10^{-2}
0.25	1.3333	1.3333	0.99999	10^{-10}	1.3365	0.90	10^{-2}
0.45	1.8182	1.8182	0.99999	10^{-10}	1.8259	0.90	10^{-2}
0.60	2.5000	2.5000	0.99999	10^{-10}	2.5142	0.90	10^{-2}
0.75	4.0000	4.0000	0.99999	10^{-10}	4.0283	0.90	10^{-2}
0.85	6.6667	6.6667	0.99999	10^{-10}	6.7202	0.90	10^{-2}
0.90	10.0000	10.0000	0.99999	10^{-10}	10.0850	0.90	10^{-2}
0.95	20.0000	20.0000	0.99999	10^{-10}	20.1794	0.90	10^{-2}
0.98	50.0000	50.0000	0.99999	10^{-8}	50.4625	0.90	10^{-2}

Figure 7.2. Confidence Interval Bound Ratios Over Relevant Range

$p = 10^{-10}, 10^{-8}, 10^{-6}, 10^{-4}, 10^{-2}$

$C = 0.90, 0.95, 0.99, 0.99999$

The table in Figure 7.2. summarizes the results of these calculations for various values of r. The minimum value for $\frac{U_{comparison}}{U_{ideal}}$ is given in the third column. The next two columns give the values of C and p that gave the minimum value of $\frac{U_{comparison}}{U_{ideal}}$. Similarly, the last three columns give the maximum value of $\frac{U_{comparison}}{U_{ideal}}$ and the values of C and p used in calculating the maximum value.

Figure 7.2. shows that the ratio $\frac{U_{comparison}}{U_{ideal}}$ is almost a function of r over the range of relevant values for p and C, since the minimum value of the ratio is almost equal to its maximum value. Also, the ratio of confidence interval bounds $\frac{U_{comparison}}{U_{ideal}}$ is virtually equal to the ratio of expected values

$$\frac{E[T_{comparison}]}{E[T_{ideal}]} = \frac{1}{1-r}.$$

Since the expected values have the same relationship as the confidence interval bounds, the ratio of expected values gives a complete measure of the effectiveness of comparison testing as compared to an ideal test bed.

CHAPTER 8

CONCLUSIONS

The effectiveness of comparison testing in finding a fault depends on the fault's overlap ratio. The analytic models developed here show that the fault-revealing power of comparison testing is almost as good as that of an ideal test bed for all faults except those having overlap ratios very close to one. The empirical evidence from the multi-version experiment [21] is being analyzed to provide an example of the distribution of overlap ratios that may occur in practice.

Models of two-version comparison testing systems have been presented here. We are in the process of developing models of systems containing more than two versions. The overlap ratio for a fault is a random variable whose value depends on the versions chosen as comparison versions. The effect of the number of versions on the distribution of overlap ratios will also be modeled.

The reliability of single versions and operational N-version systems that have been tested using comparison testing is also of interest. A model for the expected reliability of comparison-tested systems is being developed.

REFERENCES

[1] A. Avizienis, Fault-Tolerance: The Survival Attribute of Digital Systems, *Proceedings of the IEEE*, October 1978, pp. 1109-1124.

[2] L. Blank, *Statistical Procedures for Engineering, Management, and Science*, McGraw-Hill, New York, 1980.

[3] S. S. Brilliant, Analysis of Faults in a Multi-Version Software Experiment, Master's Thesis, University of Virginia, May 1985.

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

[5] L. A. Clarke, A System to Generate Test Data and Symbolically Execute Programs, *IEEE Transactions on Software Engineering*, September 1976, pp. 215-222.

[6] L. A. Clarke, J. Hassell and D. J. Richardson, A Close Look at Domain Testing, *IEEE Transactions on Software Engineering*, July 1982, pp. 380-390.

[7] L. A. Clarke, A. Podgurski, D. J. Richardson and S. J. Ziel, A Comparison of Data Flow Path Selection Criteria, *Proceedings of the 8th International Conference on Software Engineering*, London, August 1985, pp. 244-251.

[8] R. A. DeMillo, R. J. Lipton and F. G. Sayward, Hints on Test Data Selection: Help for the Practicing Programmer, *Computer*, April 1978, pp. 34-41.

[9] C. Derman, L. J. Glesar and I. Olkin, *A Guide to Probability Theory and Application*, Holt, Rinehart and Winston, Inc., New York, 1973.

[10] B. T. Devlin, L. Miller, B. Beuter and E. Swart, *SP-300 Application Software SPR Analysis*, Sperry Corporation, Phoenix, Arizona, May 1985.

[11] J. W. Duran and S. C. Ntafos, An Evaluation of Random Testing, *IEEE Transactions on Software Engineering*, July 1984, pp. 438-444.

[12] H. N. Gabow, S. N. Maheshwari and L. J. Osterweil, On Two Problems in the Generation of Program Test Paths, *IEEE Transactions on Software Engineering*, September 1976, pp. 227-231.

[13] M. R. Girgis and M. R. Woodward, An Integrated System for Program Testing Using Weak Mutation and Data Flow, *Proceedings of the 8th International Conference on Software Engineering*, London, August 1985, pp. 313-319.

[14] L. Gmeiner and U. Voges, Software Diversity in Reactor Protection Systems: An Experiment, in *Safety of Computer Control Systems*, R. Lauber (ed.), Pergamon Press, 1980, 75-79.

[15] J. B. Goodenough and S. L. Gerhart, Toward a Theory of Test Data Selection, *IEEE Transactions on Software Engineering*, June 1975, pp. 156-173.

[16] J. S. Gourlay, A Mathematical Framework for the Investigation of Testing, *IEEE Transactions on Software Engineering*, November 1983, pp. 686-709.

[17] W. E. Howden, Methodology for the Generation of Test Data, *IEEE Transactions on Computers*, May 1975, pp. 554-559.

[18] W. E. Howden, Reliability of the Path Analysis Testing Strategy, *IEEE Transactions on Software Engineering*, September 1976, pp. 208-215.

[19] W. E. Howden, Weak Mutation Testing and Completeness of Test Sets, *IEEE Transactions on Software Engineering*, July 1982, pp. 371-379.

[20] J. C. Huang, An Approach to Program Testing, *ACM Computing Surveys*, September 1975, pp. 113-128.

[21] J. C. Knight, N. G. Leveson and L. D. St. Jean, A Large Scale Experiment in N-Version Programming, *Digest FTCS-15: Fifteenth Annual International Conference on Fault Tolerant Computing*, Ann Arbor, Michigan, June 1985.

[22] J. W. Laski and B. Korel, A Data Flow Oriented Program Testing Strategy, *IEEE Transactions on Software Engineering*, May 1983, pp. 347-354.

[23] D. J. Martin, Dissimilar Software in High Integrity Applications in Flight Controls, *Software for Avionics, AGARD Conference Proceedings*, September 1982, pp. 36-1 - 36-13.

[24] H. D. Mills, On the Statistical Validation of Computer Programs, Technical Report FSC 72-6015, IBM Federal Systems Division, Gaithersburg, MD, 1972.

[25] S. C. Ntafos, On Required Element Testing, *IEEE Transactions on Software Engineering*, November 1984, pp. 795-8033.

[26] D. J. Panzl, Automatic Software Test Drivers, *Computer*, April 1978, pp. 45-50.

[27] C. V. Ramamoorthy, Y. R. Mok, F. B. Bastani, G. H. Chin and K. Suzuki, Application of a Methodology for the Development and Validation of Reliable Process Control Software, *IEEE Transactions of Software Engineering*, November 1981, pp. 537-555.

[28] S. Rapps and E. J. Weyuker, Selecting Software Test Data Using Data Flow Information, *IEEE Transactions on Software Engineering*, April 1985, pp. 367-375.

[29] 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.

[30] J. R. Taylor, Letter from the Editor, *Software Engineering Notes*, January 1981, pp. 1-2. , quote from letter to P. Neumann.

[31] E. J. Weyuker and T. J. Ostrand, Theories of Program Testing and the Application of Revealing Subdomains, *IEEE Transactions on Software Engineering*, May 1980, pp. 236-246.

[32] L. J. White and E. I. Cohen, A Domain Strategy for Computer Program Testing, *IEEE Transactions on Software Engineering*, May 1980, pp. 247-257.

[33] M. R. Woodward, D. Hedley and M. A. Hennell, Experience with Path Analysis and Testing of Programs, *IEEE Transactions on Software Engineering*, May 1980, pp. 278-286.

[34] L. J. Yount, K. A. Liebel and B. H. Hill, Fault Effect Protection and Partitioning for Fly-by-Wire/Fly-by-Light Avionics Systems, *Proceedings of Computers in Aerospace V Conference*, Long Beach, CA, August 1985.

[35] S. J. Zeil and L. J. White, Sufficient Test Sets for Path Analysis Testing Strategies, *Proceedings of the 5th International Conference on Software Engineering*, 1981.

[36] S. J. Zeil, Testing for Perturbations of Program Statements, *IEEE Transactions on Software Engineering*, May 1983, pp. 335-346.

DISTRIBUTION LIST

Copy No.	
1 - 3	National Aeronautics and Space Administration Langley Research Center Hampton, Virginia 23665 Attention: Mr. Gerard E. Migneault ISD M/S 130
4 - 5*	NASA Scientific and Technical Information Facility P.O. Box 8757 Baltimore/Washington International Airport Baltimore, Maryland 21240
6 - 7	J. C. Knight, CS
8	R. P. Cook, CS
9 - 10	E. H. Pancake, Clark Hall
11	SEAS Publications Files

*1 reproducible copy

8716/267R/ald

UNIVERSITY OF VIRGINIA
School of Engineering and Applied Science

The University of Virginia's School of Engineering and Applied Science has an undergraduate enrollment of approximately 1,500 students with a graduate enrollment of approximately 500. There are 125 faculty members, a majority of whom conduct research in addition to teaching.

Research is a vital part of the educational program and interests parallel academic specialties. These range from the classical engineering disciplines of Chemical, Civil, Electrical, and Mechanical and Aerospace to newer, more specialized fields of Biomedical Engineering, Systems Engineering, Materials Science, Nuclear Engineering and Engineering Physics, Applied Mathematics and Computer Science. Within these disciplines there are well equipped laboratories for conducting highly specialized research. All departments offer the doctorate; Biomedical and Materials Science grant only graduate degrees. In addition, courses in the humanities are offered within the School.

The University of Virginia (which includes approximately 1,500 full-time faculty and a total full-time student enrollment of about 16,000), also offers professional degrees under the schools of Architecture, Law, Medicine, Nursing, Commerce, Business Administration, and Education. In addition, the College of Arts and Sciences houses departments of Mathematics, Physics, Chemistry and others relevant to the engineering research program. The School of Engineering and Applied Science is an integral part of this University community which provides opportunities for interdisciplinary work in pursuit of the basic goals of education, research, and public service.