TESTING SOFTWARE
USING MULTIPLE VERSIONS

Susan S. Brilliant

John C. Knight

Computer Science Report No. RM-86-07
December 10, 1986

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

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

Abstract i
Table of Contents ii

1. Introduction 1
   1.1. The Need for More Reliable Software 1
   1.2. The Operational Use of Multiple Versions 2
   1.3. The Use of Multiple Versions in Testing 4
   1.4. Organization of Remainder of Report 6

2. Previous Research in Dynamic Testing 8
   2.1. Testing Strategies 8
   2.2. Test Data Selection 13
   2.3. Interpretation of Results 15

3. Notation Used to Describe Fault Interrelationships 18
   3.1. Fault Interrelationships that Affect Comparison Testing 18
   3.2. Notation 20

4. Markov Models of Fault Observation Times 23
   4.1. A Single Fault Interaction 23
   4.2. Multiple Fault Overlaps in a Two-Version System 26

5. An Analysis of Model Implications 32
   5.1. A Basis for Evaluation: The Ideal Test Bed 32
   5.2. Performance of Comparison Testing 34

6. Worst Case Analysis 37
   6.1. Worst Case Performance with Single Fault Overlap 37
   6.2. Extended Definition of Overlap Ratio 41

7. Dispersions of Observation Time Distributions 43
   7.1. Variances 43
   7.2. Confidence Interval Bounds 44

8. Conclusions 49

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

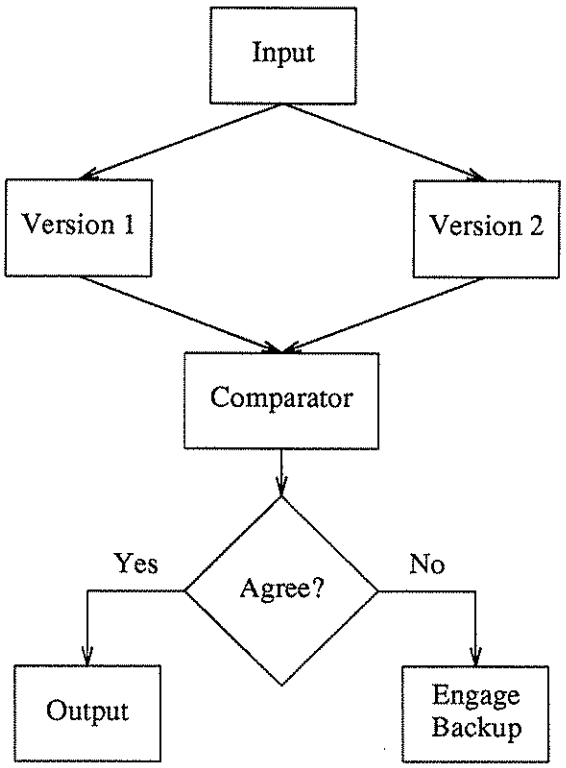

Figure 1.1. Multi-Version Programming to Achieve Error Detection

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.

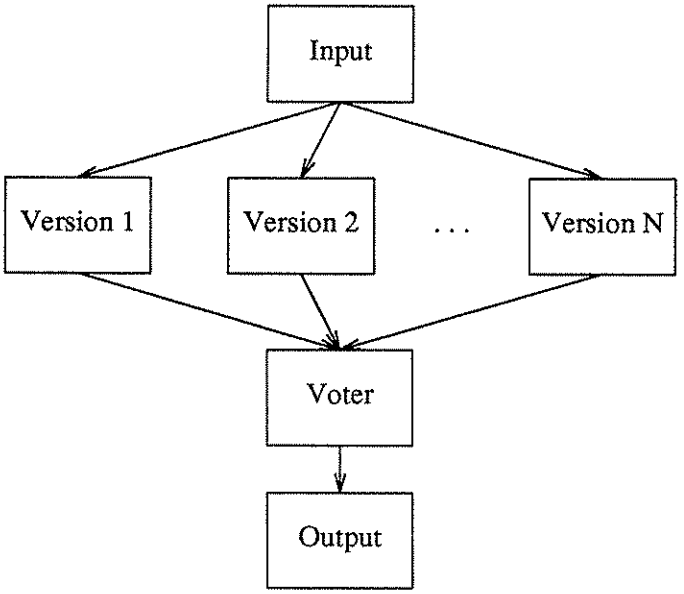

Figure 1.2. Multi-Version Programming to Achieve Fault Tolerance

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.

Comparison Testing

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.

$p_{55} = 1$

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:

$$q_B = \bar{q}_A = (1-r)p.$$

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.

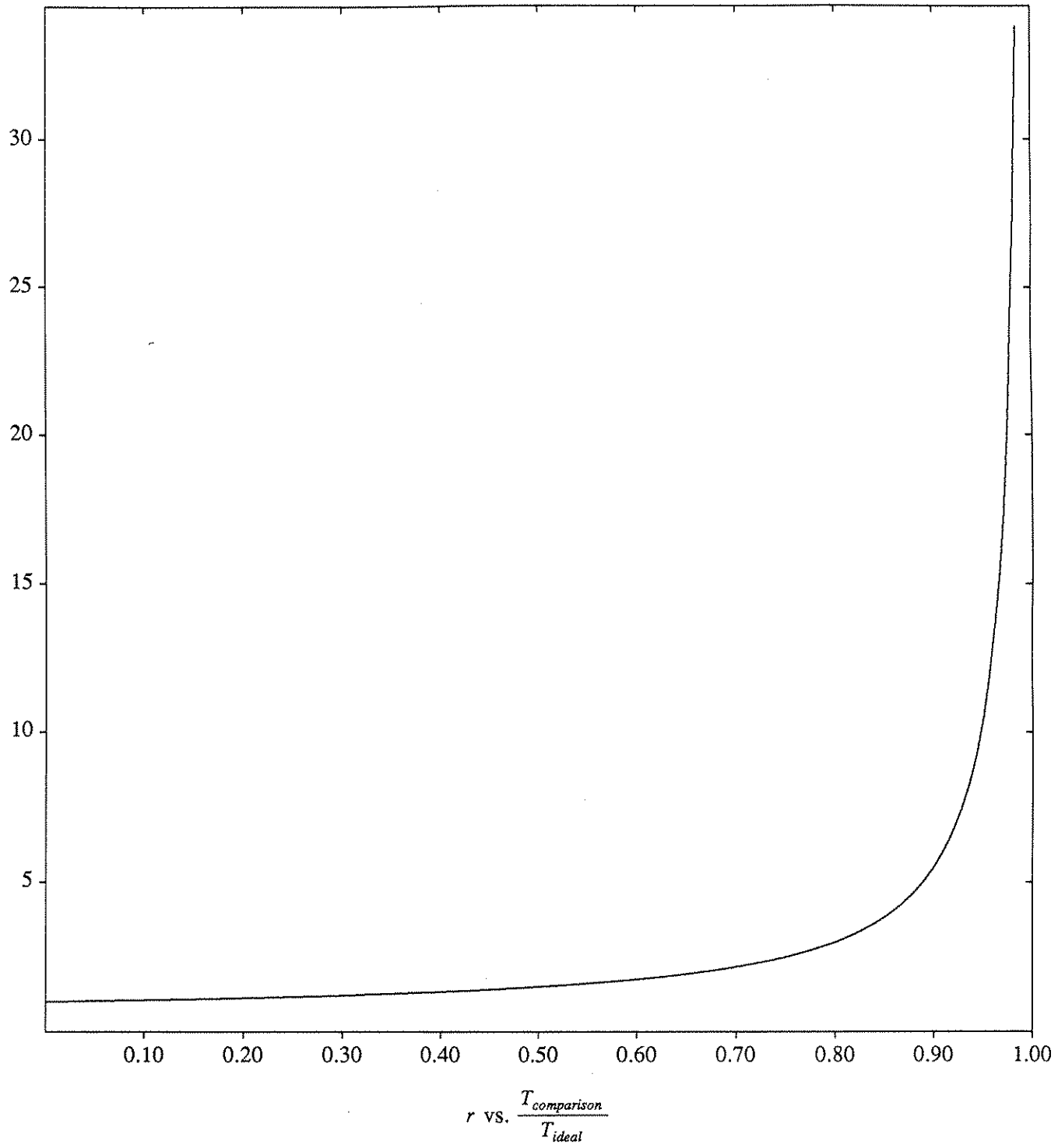

Figure 5.1. Effect of *r* on Fault Observation Time

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:

Comparison Testing

(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. $q_B = \tilde{q}_A$. 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.

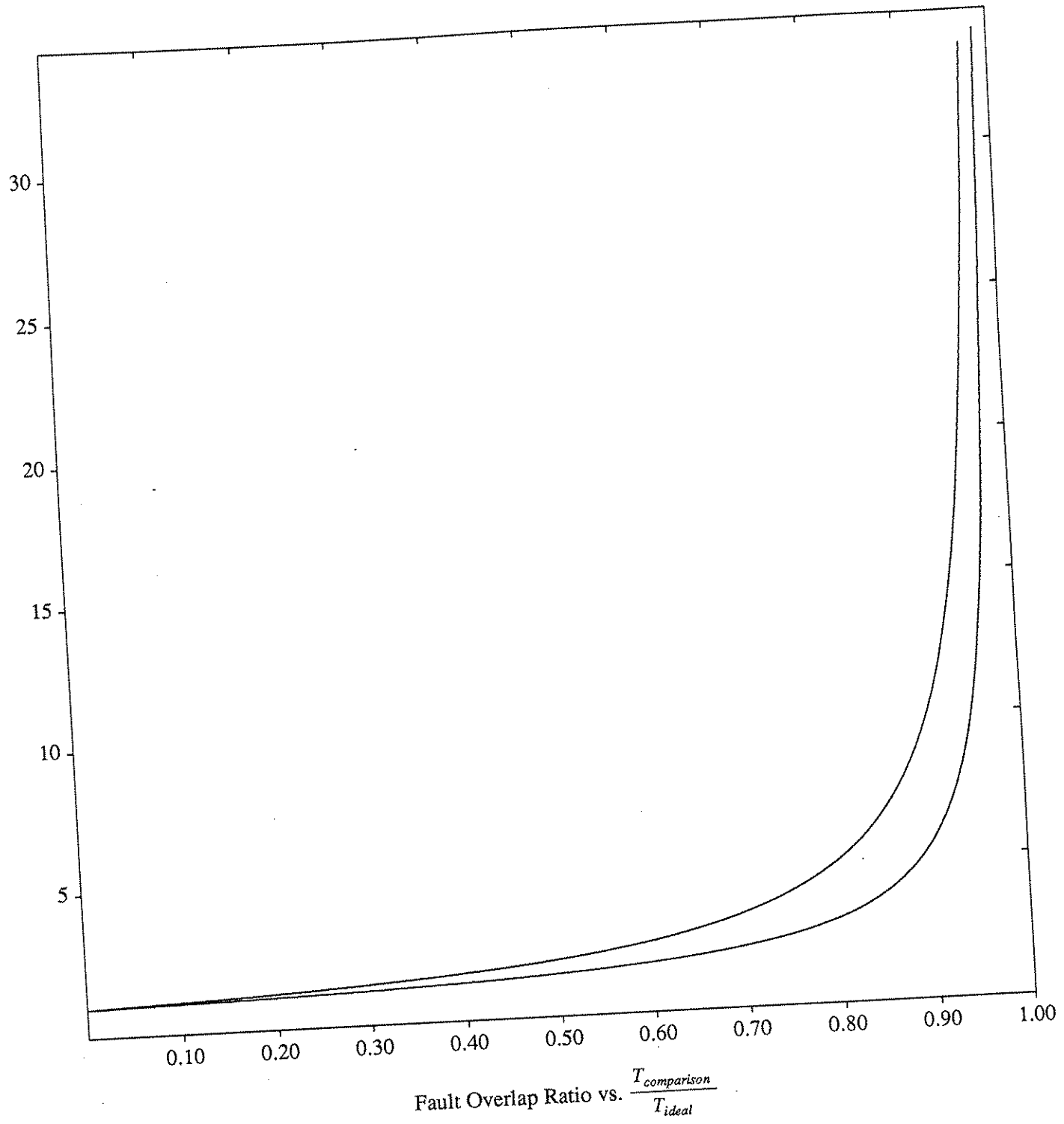

Figure 6.1. Comparison of Worst and "Average" Case

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:

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

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

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

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

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

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.