An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing

Phyllis G. Frankl * Stewart N. Weiss†

Abstract

An experiment comparing the effectiveness of the all-uses and all-edges test data adequacy criteria was performed. The experiment was designed so as to overcome some of the deficiencies of previous software testing experiments. A large number of test sets was randomly generated for each of nine subject programs with subtle errors. For each test set, the percentages of executable edges and definition-use associations covered were measured and it was determined whether the test set exposed an error. Hypothesis testing was used to investigate whether all-uses adequate test sets are more likely to expose errors than are all-edges adequate test sets. All-uses was significantly more effective than all-edges for five of the subjects, and appeared guaranteed to detect the error in four of them. Further analysis showed that in four of these subjects, all-uses-adequate test sets were more effective than all-edges-adequate test sets of similar size. Logistic regression analysis was used to investigate whether the probability that a test set exposes an error increases as the percentage of definition-use associations or edges covered by it increases. The evidence did not strongly support this conjecture. Error exposing ability was shown to be strongly positively correlated to percentage of covered definition-use associations in only four of the nine subjects. Error exposing ability was also shown to be positively correlated to the percentage of covered edges in four (different) subjects, but the relationship was weaker.

*Author's address: Computer Science Dept., Polytechnic University, 6 Metrotech Center, Brooklyn, N.Y. 11201. E-mail: pfrankl@poly.edu. Supported in part by NSF Grants CCR-8810287 and CCR-9206910 and by the New York State Science and Technology Foundation Center for Advanced Technology program.

†Author's address: Computer Science Dept., Hunter College, 695 Park Ave., New York, NY 10021. E-mail: weiss@roz.hunter.cuny.edu. Supported in part by NSF Grant CCR-9008394 and by PSC-CUNY Award 661341.

1 Introduction

Considerable effort in software testing research has focussed on the development of software test data adequacy criteria, that is, criteria that are used to determine when software has been tested "enough", and can be released. Numerous test data adequacy criteria have been proposed, including those based on control flow analysis [25, 26], data flow analysis [23, 29, 31, 34] and program mutation [9]. Tools based on several of these criteria have been built [8, 14, 28] and many theoretical studies of their formal properties and of certain aspects of their relations to one another have been done [6, 12, 15, 34]. But surprisingly, relatively little work has focussed on the crucial question: how good at exposing errors are the test sets that are deemed adequate according to these criteria?

In this paper, we describe an experiment addressing this question. One factor that makes it difficult to answer this question is that for a given program P and adequacy criterion C, there is typically a very large number of adequate test sets. If P is incorrect, then usually some of these test sets expose an error while others do not. Most previous experiments have failed to sample this very large space of test sets in a statistically sound way, and thus have given potentially misleading results. The goals of this research were twofold: 1) to develop an experiment design that allows the error-detecting ability of adequacy criteria to be compared in a meaningful way, and 2) to use that design to measure and compare several adequacy criteria for a variety of subject programs.

In Section 2, below, we define a notion of the effectiveness of an adequacy criterion that, for a given erroneous program and specification, measures the likelihood that an adequate set will expose an error. The higher the effectiveness of criterion C, the more confidence we can have that a program that has been tested on a C-adequate test set without exposure of an error is indeed correct. Our experiment measured effectiveness by sampling the space of C-adequate test sets. For each of nine subject programs, we generated a large number of test sets, determined the extent to which each test set satisfied certain adequacy criteria, and determined whether or not each test set exposed an error. The data were used to measure and compare effectiveness of several adequacy criteria and to address several related questions.

We limited our attention to three adequacy criteria. The *all-edges* criterion, also known as branch testing, is a well-known technique which is more widely used in practice than other, more sophisticated adequacy criteria. It requires the test data to cause the execution of each edge in the subject program's flow graph. The *all-uses* data flow testing criterion requires the test data to cause the execution of paths going from points at which variables are assigned values to points at which those values are used. It has received considerable attention in the research community and is considered promising by many testing researchers. For comparison, we also consider the *null criterion*, which deems any test set to be adequate.

The design of the experiment allowed us to address three types of related questions. Given a subject program and a pair of criteria, C1 and C2, we investigated,

1. **Overall comparison of criteria:** Are those test sets that satisfy criterion C1 more likely to detect an error than those that satisfy C2? More generally, are those test sets that satisfy $X\%$ of the requirements induced by C1 more likely to detect an error than those that satisfy $Y\%$ of the requirements induced by C2?

2. **Comparison of criteria for fixed test set size:** For a given test set size, n, are those test sets of size n that satisfy criterion C1 more likely to detect an error than those that satisfy C2?

3. **Relationship between coverage and effectiveness:** How does the likelihood that the test set detects an error depend on the extent to which a test set satisfies a criterion and the size of the test set?

The overall comparisons of the criteria give insight into which criterion should be selected when cost is not a factor. If C1 is more effective than C2, but C1 typically demands larger test sets than C2, one may ask whether the increased effectiveness arises from differences in test set sizes, or from other, more intrinsic characteristics of the criteria. The comparison of criteria for fixed test set size addresses this issue by factoring out differences in test set size. Lastly, investigation of the relationship between coverage and effectiveness is useful because in practice it is not unusual to demand only partial satisfaction of a criterion.

We believe that the results reported here should be of interest both to testing researchers and to testing practitioners. While practitioners may be primarily interested in the experiment's results and their implications for choosing an adequacy criterion, researchers may also find the novel design of the experiment interesting.

This paper is organized as follows. Section 2 of this paper defines effectiveness and reviews the definitions of the relevant adequacy criteria. Section 3 describes the design of the experiment, Section 4 describes the statistical analysis techniques used, and Section 5 describes the subject programs on which the experiment was performed. The experiment's results are presented in Section 6 and discussed further in Section 7. The experiment design is compared to related work in Section 8, and conclusions are presented in Section 9.

2 Background

2.1 Effectiveness of an adequacy criterion

The goal of testing is to detect errors in programs. We will say that a test case t *exposes an error* in program P, if on input t, P's output is different than the specified output. A test set T *exposes an error*, or is *exposing*, if at least one test case t in T exposes an error.

Consider the following model of the testing process:

- A test set is generated using some test data generation technique.
- The program is executed on the test set, the outputs are checked, and the adequacy of the test set is checked.
- If at least one test case exposes an error, the program is debugged and regression tested; if no errors are exposed but the test set is inadequate, additional test cases are generated.
- This process continues until the program has been executed on an adequate test set that fails to expose an error.

At this point, the program is released. Although the program is not guaranteed to be correct, the "better" the adequacy criterion, the more confidence one can have that it is correct.

Note that we have explicitly distinguished between two aspects of testing: test generation and application of a test data adequacy criterion. A test generation technique is an algorithm which generates test cases, whereas an adequacy criterion is a predicate which determines whether the testing process is finished. Test generation algorithms and adequacy criteria that are based on the structure of the program being tested are called *program-based* or *white-box*; those that are not based on the structure of the program are called *black-box*. Black-box techniques are typically *specification-based*, although some, such as random testing, are not. It is possible in principle to use white box techniques as the basis for test generation. For example, one could examine the program text and devise test cases which cause the execution of particular branches. However, it is usually much more difficult to generate a test case that causes execution of a particular branch than to simply check whether a branch has been executed by a given test case. Thus, in practice, systematic approaches to test generation are usually black-box, while systematic approaches to checking adequacy are often white-box. Black box test generation techniques may involve devising test cases intended to exercise particular aspects of the specification or may randomly sample the specification's domain. The testing techniques investigated in this paper combine black box test generation techniques and white-box adequacy criteria. In particular, we explore whether one white box adequacy criterion is more likely than another to detect a bug when a particular (random) black box testing strategy is used.

We now define a measure of the "goodness" of an adequacy criterion that captures this intuition. Let P be an incorrect program whose specification is S, and let C be an adequacy criterion. Consider all the test sets T that satisfy C for P and S. It may be the case that some of these test sets expose an error, while others do not. If a large percentage of the C-adequate test sets expose an error, then C is an effective criterion for this program.

More formally, consider a given probability distribution on the space of all C-adequate test sets for program P and specification S. We define the *effectiveness* of C to be the probability that a test set selected randomly according to this distribution will expose an error. In practice, test sets are generated using a particular test generation strategy G, such as random testing with a given input distribution, some other form of black-box testing, or a systematic white-box strategy. This induces a distribution on the space of C-adequate test sets. We define the *effectiveness of criterion C for P and S relative to test generation strategy G* to be the probability that a C-adequate test set generated by G will expose an error in P. In this paper, we will be concerned with effectiveness of criteria relative to various random test generation strategies.

To see that this notion of effectiveness captures the intuition of the "goodness" of an adequacy criterion, let $p_C(P)$ denote the effectiveness of criterion C for program P. The probability that a randomly selected C-adequate test set T will *not expose an error*, i.e., that we will release P, treating it as if it were correct, is $1 - p_C(P)$. In particular, if P is incorrect, this is the probability that after testing with a C-adequate test set we will *mistakenly* believe that P is correct. Now suppose $p_{C1}(P) \geq p_{C2}(P)$ for all programs P in some class $\mathcal{P}$. Then, since $1 - p_{C1}(P) \leq 1 - p_{C2}(P)$, the probability of our mistakenly believing P to be correct after using C2 as an adequacy criterion is at least as great as if we had used C1 as the adequacy criterion. Thus after testing P without exposing an error, we can have at least as much confidence that P is correct if we used criterion C1 as if we used criterion C2. Weiss has defined a more general notion of the effectiveness of an adequacy criterion and discussed its relation to confidence in program correctness [37].

Most previous comparisons of adequacy criteria have been based on investigating whether one criterion subsumes another. Criterion C1 *subsumes* criterion C2 if, for every program P and specification S, every test set that satisfies C1 also satisfies C2. It might seem, at first glance, that if C1 subsumes C2, then C1 is guaranteed to be more effective than C2 for every program. This is not the case. It may happen that for some program P, specification S, and test generation strategy G, test sets that only satisfy C2 may be better at exposing errors than those that satisfy C1. Hamlet has discussed related issues [21]. Weyuker, Weiss, and Hamlet [40] and Frankl and Weyuker [17, 16] have further examined the relationship between subsumption and error-detecting ability.

2.2 Definitions of the adequacy criteria

This study compares the effectiveness of three adequacy criteria: the all-edges criterion, the all-uses criterion, and the null criterion. Two of these criteria, all-edges and all-uses, are members of a family of criteria, sometimes called structured testing criteria, that require the test data to cause execution of representatives of certain sets of paths through the flow graph of the subject program. The null criterion considers any test set to be adequate; thus application of the null criterion is the same as not using any adequacy criterion at all. We have included the null criterion in this study in order to allow comparison of all-edges and all-uses adequate sets to arbitrary sets.

The all-edges criterion, also known as branch testing, demands that every edge in the program's flow graph be executed by at least one test case. All-edges is known to be a relatively weak criterion, in the sense that it is often easy to devise a test set that covers all of the edges in a buggy program without exposing the bug. A much more demanding, but completely impractical, criterion is *path testing*, which requires the execution of every path through the program's flow graph.

In an effort to bridge the gap between branch-testing and path-testing, Rapps and Weyuker [34] defined a family of adequacy criteria based on data flow analysis similar to that done by an optimizing compiler. The all-uses criterion belongs to this family[1]. Other data flow testing criteria have also been defined [23, 29, 31]. Roughly speaking, these criteria demand that the test data exercise paths from points at which variables are defined to points at which their values are subsequently used. Occurrences of variables in the subject program are classified as being either *definitions*, in which values are stored, or *uses*, in which values are fetched. For example, a variable occurrence on the left-hand side of an assignment statement is a definition; a variable occurrence on the right hand side of an assignment statement or in a Boolean expression in a conditional statement is typically a use. A *definition-use association* (dua) is a triple (d,u,v) such that d is a node in the program's flow graph in which variable v is defined, u is a node or edge in which v is used, and there is a definition-clear path with respect to v from d to u. A test case t *covers* dua (d,u,v) if t causes the execution a path that goes from d to u without passing through any intermediate node in which v is redefined. The all-uses criterion demands that the test data cover every dua in the subject program.

We had previously designed and implemented a tool, ASSET [11, 12, 14], that checks the extent to which a given test set for a given Pascal program satisfies all-uses and various other data flow testing criteria. ASSET analyzes the subject program to determine all of the definition-use associations in a particular program unit and builds a modified program whose functionality is identical to the original subject program except that it also outputs a trace of the path followed when a test case is executed. After executing the modified program on the given test set, ASSET analyzes the traces to determine the extent to which the adequacy criterion has been satisfied and outputs a list of those definition-use associations that still need to be covered. For this experiment, we modified ASSET so that it could also check whether a test set satisfies all-edges and replaced ASSET's interactive user interface by a batch interface.

One problem with the all-edges and all-uses criteria, as originally defined, is that for some programs, no adequate test set exists. This problem arises from infeasible paths through the program, i.e., paths that can never be executed. The problem is

[1]The original criteria were defined for programs written in a simple language. The definitions were subsequently extended by Frankl and Weyuker [15] to programs written in Pascal. We adopt their conventions and notation here.

particularly serious for the all-uses criterion because for many commonplace programs, no adequate test set exists. For example, the problem occurs with any program having a **for** loop in which the lower and upper bounds are non-equal constants. Frankl and Weyuker defined a new family of criteria, the *feasible data flow testing criteria*, which circumvents this problem by eliminating unexecutable edges or definition-use associations from consideration [12, 15], and showed that under reasonable restrictions on the subject program, the feasible version of all-uses subsumes the feasible version of all-edges.

It is important to note that the original (infeasible) criteria are not really used in practice; they do not apply to those programs that have infeasible edges or duas, and they are the same as the feasible versions for other programs. Ideally, testers should examine the program to eliminate infeasible edges and duas from consideration. In reality, they often stop testing when some arbitrary percentage of the edges or duas has been covered, without investigating whether the remaining edges/duas are infeasible, or whether they indicate deficiencies in the test set. For this reason, we felt that it was also important to examine the relationship between the percentage of the duas covered by a test set and its likelihood of exposing an error. In the remainder of this paper, we will, by abuse of notation, use the terms all-edges and all-uses to refer to the feasible versions of these criteria.

3 Experiment Design

The goal of this experiment was to measure and compare the effectiveness of various adequacy criteria relative to random test generation strategies for several different subject programs. To measure the effectiveness of criterion C, we can

- generate a large number of C-adequate test sets,
- execute the subject program on each test set,
- check the outputs and consider a test set *exposing* if and only if the program gives the wrong output on at least one element of the test set, and
- calculate the proportion of C-adequate test sets that are exposing.

If the proportion of C1-adequate test sets that expose an error is significantly higher than the proportion of C2-adequate test sets that expose an error, we can conclude that criterion C1 is more effective than C2 for the given program and test generation strategy.

In the present experiment, we generated test sets randomly and compared the all-edges, all-uses, and null criteria. The programs on which we experimented, and the exact notion of "randomly generated test sets" used for each are described in Section 5 below.

We collected the data in such a way as to allow for comparison of a family of variants of all-edges and all-uses. Rather than just checking whether or not each test set satisfied all-edges (all-uses), we recorded the number of executable edges (duas) covered by the test set. This allowed us to use the collected data to measure not only the effectiveness of all-edges and all-uses, but also of such criteria as X% edge coverage and Y% dua coverage. In addition it allowed us to investigate the correlation between percentage of edges (duas) covered and error exposing ability.

For each subject program, we first identified the unexecutable edges and duas and eliminated them from consideration. We then generated a large set of test cases called the universe, executed each test case in the universe, checked its output, recorded whether it was correct, and saved a trace of the path executed by that test case.[2] We sampled the space of adequate test sets as follows: we selected various test set sizes, n, then for each n, generated many test sets by randomly selecting n elements from the universe, using a uniform distribution. Note that we did not use a uniform distribution on the space of test *sets*, but rather, used a distribution that arises from a practical test generation strategy. We then determined whether or not each test set was exposing and used ASSET to check how many executable edges and duas were not covered by any of the paths corresponding to the test set.

Some care was necessary in choosing appropriate test set sizes. If the generated test sets are too small, then relatively few of them will cover all edges (all duas), so the results will not be statistically significant. On the other hand, if the test sets are too large, then almost all of them will expose errors, making it difficult to distinguish between the effectiveness of the two criteria. To overcome this problem, we generated our test sets in "batches", where each batch contained sets of a fixed size. After observing which sizes were too large or too small, we generated additional batches in the appropriate size range, if necessary. This "stratification" of test sets by size also allowed us to investigate whether all-uses is more effective than all-edges for test sets of a given size.

The design of the experiment imposed several constraints on the subject programs:

- The input domain of the program had to have a structure for which there was some reasonable way to generate the universe of test cases. For example, while there are several reasonable ways to randomly generate matrices, it is less clear how to randomly generate inputs to an operating system.

- Because of the large number of test cases on which each program was executed, it was necessary to have some means of automatically checking the correctness of the outputs.

[2] In two of the subjects (`matinv1` and `determinant`), there were a few executable duas that were not covered by any element of the universe. We dealt with this by considering any dua that wasn't covered by the universe to be unexecutable.

- The failure rate of the program had to be low; i.e., errors had to be exposed by relatively few inputs in the universe. Otherwise, almost any test set that was big enough to satisfy all-edges would be very likely to expose an error. We were surprised to discover that many of the programs we considered as candidates for this experiment, including many that had been used in previous software quality studies, had to be rejected because of their high failure rates.

The available tool support (ASSET) imposed an additional constraint – the subject programs had to be either written in Pascal or short enough to translate manually. When translation was necessary, program structure was changed as little as possible.

Note that our experiment considered the effectiveness of all-edges adequate sets in general, *not* the effectiveness of those all-edges adequate sets that fail to satisfy all-uses. This models the situation in which the tester releases the program when it has passed an all-edges adequate test set without caring whether or not the test set also satisfies all-uses. In an alternative model, the tester would classify a test set as all-edges adequate only if it satisfied all-edges and did *not* satisfy all-uses. If all-uses is shown to be more effective than all-edges using our model, then the difference would be even greater if we were to use the alternative model.

Also note that our design introduces a bias in favor of all-edges. We used test sets that were big enough to insure the selection of a statistically significant number of all-uses adequate sets, *not* just a significant number of all-edges adequate sets. This resulted in the selection of many all-edges adequate sets that were bigger, thus more likely to expose an error, than the all-edges adequate sets that would be selected by a practitioner using our model of the testing process.

4 Data Analysis Techniques

Recall that we are interested in comparing the effectiveness of all-uses to all-edges and to the null criterion for *each* of a variety of subject programs. We treat each subject program's data as that of a separate experiment. Throughout this section, when we refer to the effectiveness of a criterion we mean its effectiveness for a particular program and test generation strategy. For clarity, we describe the techniques used to compare all-uses with all-edges. The techniques for comparing all-uses and all-edges to the null criterion and for comparing coverage of $X\%$ of the duas to coverage of $Y\%$ of the edges are identical.

If we have randomly chosen N C-adequate test sets, and X is the number of these that exposed at least one error, then $\hat{p_C} = X/N$, the sample proportion, is a good estimator of p_C, the effectiveness of C. In fact, if the probability that a C-adequate test set exposes an error is governed by a binomial distribution, then $\hat{p_C}$ is a minimum variance unbiased estimator of the effectiveness of C [3], i.e., it is good statistic for estimating effectiveness.

4.1 Overall comparison of criteria

The first question posed was whether or not all-uses adequate test sets are significantly more effective than all-edges adequate test sets. Let $\hat{p_u}$ be the proportion of all-uses adequate sets that exposed an error and let $\hat{p_e}$ be the proportion of all-edges adequate sets that exposed an error. If $\hat{p_u}$ is significantly higher than $\hat{p_e}$ then there is strong statistical evidence that all-uses is more effective than all-edges. If not, the data do not support this hypothesis.

This observation suggests that hypothesis testing techniques are suitable for answering this question. In hypothesis testing, a research, or alternative, hypothesis is pitted against a null hypothesis, and the data are used to determine whether one hypothesis is more likely to be true than the other. Our research hypothesis, that all-uses is more effective than all-edges, is expressed by the assertion $p_e < p_u$. The null hypothesis is that the two criteria are equally effective, expressed by $p_e = p_u$. Note that we chose a one-sided test because we wanted to know whether all-uses is *more* effective than all-edges, not just whether they are different. It is important to realize that the goal in hypothesis testing is quite conservative; we uphold the null hypothesis as true unless the data is strong testimony against it, in which case we reject the null hypothesis in favor of the alternative.

Since we are using the sample proportions as estimators of the effectiveness of the criteria, our decision to accept or reject the null hypothesis reduces to a decision as to whether or not the difference between the sample proportions is significantly large. In particular, we should reject $p_e = p_u$ if and only if $\hat{p}_u - \hat{p}_e$ is greater than some prespecified *critical value*.

Using a standard statistical technique for establishing the critical values [5], we call a sample *sufficiently large* if there are at least five exposing and five unexposing test sets in the sample. For sufficiently large samples, the difference $\hat{p}_u - \hat{p}_e$ is approximately normally distributed, with mean $p_u - p_e$ and variance $\sigma_u^2 + \sigma_e^2$. If we assume that the parent populations are binomial then $\sigma_C^2 = p_C(1 - p_C)/N_C$, where N_C is the population size. This enables us to calculate critical values, significance probabilities and confidence intervals for $p_u - p_e$. The significance probabilities indicate the strength of the evidence for rejection of hypotheses, and the confidence intervals give an indication of how much better one criterion is than the other, if at all. To be conservative in our interpretation of the data, we chose a significance level of $\alpha = 0.01$, meaning that if the null hypothesis is rejected, the probability that all-edges is *actually* as effective as all-uses is at most 1/100.

In several of our subjects, every all-uses adequate test set exposed an error, so that the normal approximation could not be used. In these cases, we calculated confidence intervals separately for p_u and p_e. Inspection of the data showed that all-uses was clearly more effective than all-edges for these subjects, making further analysis unnecessary.

4.2 Comparison of criteria for fixed size test sets

The second question we asked dealt with the effect of test set size on the previous results. The all-uses adequacy criterion in general requires larger test sets than does the all-edges criterion. Since the probability that a test set exposes an error increases as its size increases, for some subjects all-uses may be more effective than all-edges simply because it demands larger test sets. On the other hand, the increased effectiveness of all-uses may result from the way the criterion subdivides the input domain [39].

To determine whether differences in the effectiveness of the criteria were primarily due to differences in the sizes of adequate test sets, we analyzed the data on a "by-size" basis. In Tables 6, 7, 8, we display the sample data for each of the subject programs by size, arranging close sizes into groups. The intent of this table is to give descriptive evidence of the relationship between all-uses and all-edges for fixed size test sets. Where there was enough data, we also did hypothesis testing on the individual size groups and reported the results in the right hand columns.

4.3 Relationship between coverage and effectiveness

The third question to be answered is whether there is a relationship between the extent to which a test set satisfies the all-uses (or all-edges) criterion and the probability that the test set will expose an error. This is the most difficult of the questions, and the technique we employed to answer it is logistic regression.

A regression model gives the mean of a response variable in a particular group of variables as a function of the numerical characteristics of the group. If Y is the response variable and $X_1, X_2, \ldots X_n$ are the predictors, we denote the mean of Y, given fixed values $\bar{x} = x_1, x_2, \ldots x_n$, by $\mu_{Y|\bar{x}}$. Ordinary linear (or higher order) regression models are not suitable for data in which the response variable takes on yes-no type values such as "exposing" or "not exposing", in part because regression equations such as

$$\mu_{Y|\bar{x}} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots \beta_n x_n \tag{1}$$

put no constraints on the value of $\mu_{Y|\bar{x}}$. The right hand side can take on any real value whereas the left hand side must lie between 0 and 1. The right hand side is also assumed to follow a normal distribution whereas the left hand side generally does not. There are other serious problems that make linear regression a poor choice for modeling proportions [1].

Logistic regression overcomes these problems and provides many important advantages as well. In logistic regression the left hand side of Equation 1 is replaced by the *logit* of the response variable,

$$\log\left(\frac{\mu_{Y|\bar{x}}}{1-\mu_{Y|\bar{x}}}\right)$$

and the right hand side can be any real-valued function of the predictors. The expression

$$\frac{\mu_{Y|\bar{x}}}{1 - \mu_{Y|\bar{x}}}$$

is frequently called the *odds ratio* of the response variable. Because $\mu_{Y|\bar{x}}$ lies between 0 and 1, the odds ratio can assume any positive real value, and so its logarithm, the logit, can assume any real value. Thus, in logistic regression, Equation 1 becomes one in which neither left nor right hand side is constrained. Algebraic manipulation shows that if

$$\log\left(\frac{\mu_{Y|\bar{x}}}{1 - \mu_{Y|\bar{x}}}\right) = f(\bar{x})$$

then

$$\mu_{Y|\bar{x}} = \frac{\exp f(\bar{x})}{1 + \exp f(\bar{x})}$$

This equation is the regression equation, and the goal is to find the "simplest" function $f(\bar{x})$ that explains the data well.

In our analysis, we treated test set size and fraction of coverage of definition-use associations (or edges) as the predictor variables, and used logistic regression to determine the extent, if any, to which the probability of exposing an error was dependent upon these variables. We used the CATMOD module of SAS to assist with the regression analysis, using maximum likelihood estimation, and measuring goodness of fit with χ^2 tests of the model parameter estimates and of the likelihood ratio.

5 Subject Programs

Our nine subjects were obtained from seven programs, all of which had naturally occurring errors. Three of our programs were drawn from the Duran and Ntafos study [10]; high failure rates made the rest of the Duran-Ntafos subjects unsuitable for our experiment. The programs `buggyfind`, `textfmt`, and `transpose`, described below, were used by Duran and Ntafos; the remainder came from other sources. We obtained two subjects from `buggyfind` by using two different input distributions. Recall that ASSET monitors coverage of edges or duas in a single program unit. We obtained two subjects from a matrix inversion program by instrumenting two different procedures.

In this section, we describe the programs, the procedures for selecting random test data for them, and the method used to check outputs. Table 1 gives the numbers of lines of code, edges, duas, executable edges, executable duas in the instrumented procedures, and proportions of failure causing test cases in each universe.

subject	LOC	edges	duas	exec edges	exec duas	failure rate
detm	60	78	298	74	103	0.001
find1	33	34	114	34	93	0.066
find2	33	34	114	34	93	0.018
matinv1	60	78	298	74	106	0.001
matinv2	28	30	81	30	62	0.001
strmtch1	22	13	49	13	49	0.032
strmtch2	23	14	56	14	54	0.062
textfmt	26	21	50	21	42	0.052
transpose	78	44	97	42	88	0.023

Table 1: Subject Programs

5.1 Buggyfind

Hoare's `find` program [24] takes as input an array `a` and an index `f` and permutes the elements of `a` so that all elements to the left of position `f` are less than or equal to `a[f]` and all elements to the right of position `f` are greater than or equal to `a[f]`. Boyer, Elspas, and Levitt [4] analyzed an erroneous variant of this program, dubbed `Buggyfind`, which represented a failed attempt to translate Hoare's program into LISP.

For our experiment, we translated the LISP version into Pascal, and tested it using two different distributions of random inputs. For `find1`, the test universe consisted of 1000 randomly generated arrays, where the array sizes were randomly selected between zero and 10, the elements were randomly selected in the range 1 to 100, and the values of `f` were randomly selected between 1 and the array size. For `find2` the universe contained one test case for each array with elements selected from $\{0, 1, 2, 3, 4\}$ and range $[0 \ldots n]$, for each n from 0 to 5. This distribution more closely approximates uniform random selection from all test cases with array sizes from 0 to 5.

In both `find1` and `find2` we checked the output by checking whether all elements to the left of position `f` were less than or equal to `a[f]` and all elements to the right of position `f` were greater than or equal to `a[f]`.

5.2 TextFormat

Goodenough and Gerhart [19] analyzed an erroneous text formatting program. They identified seven problems with the program and traced them to five faults. Four of these faults either were too blatant to be useful for this experiment, or could not be replicated in Pascal versions of the program. Our `textfmt` program is a Pascal version of Goodenough and Gerhart's corrected `textfmt` program, in which we have re-inserted the remaining fault. This fault, which corresponds to Goodenough and Gerhart's problems N5 and N6, causes leading blanks and adjacent blanks/line-breaks to be handled improperly. We would have liked to re-insert the other faults to produce additional subject programs, but either they could not be replicated in Pascal, or they led to failure rates that were too high.

Each test case was a piece of text, 15 characters long, generated by repeated uniform random selection of a character from the set consisting of uppercase letters, lowercase letters, and blank and newline characters. Outputs were checked by comparing the output text to a correct version, using the UNIX `diff` command.

5.3 Transpose

The next subject program was a transpose routine from a sparse matrix package, Algorithm 408 of the Collected Algorithms of the ACM [30], in which two faults had subsequently been identified [20]. We translated the corrected FORTRAN program into Pascal, and re-introduced one of the faults. Our universe could not expose the other alleged fault. The failure occurs when the first row of the matrix consists entirely of zeros.

Since the sparse matrix package was designed to reduce memory storage for matrices whose densities do not exceed 66%, we chose the test cases randomly from among the set of all R by C matrices with densities between 0 and 66%, where $0 \leq C \leq R \leq 50$. The matrix transpose package required that C be at most R. Positions of the zeros in the matrices were chosen randomly, and the non-zero entries were filled with their row-major ordinal values. To check the outputs we simply compared elements M[i,j] and M[j,i] for all i and j.

5.4 String-matching programs

Two of our subject programs were brute-force string-matching programs. They input some text and a pattern and are supposed to output the location of the first occurrence of the pattern in the text, if it occurs, and zero if the pattern never occurs. The first subject, `strmtch1`, resulted from a flawed attempt to modify a string-matching program from a Pascal textbook [7] so that it could handle variable length texts and patterns. The error occurs when a pattern of length zero is entered; in this case the program returns the value two. Note that there are several reasonable specifications of what the program should do in this case, but returning the value two is not among them. We had previously observed that the all-uses criterion is guaranteed to expose this error, because one of the duas can only be executed when the pattern length is zero. We did not know the effectiveness of all-edges or the null criterion for this program.

Our second erroneous string match program, `strmtch2`, reflects a different error that also occurred naturally. In the implementation, the maximum length of a pattern is shorter than it should be, so the program is sometimes working with a truncated version of the pattern, hence sometimes erroneously reports that it has found a match.

For each of the string-matching programs, the universe consisted of all (text, pattern) pairs on a two letter alphabet with text length and pattern length ranging from zero to four. Outputs were checked by comparing them to those produced by a correct program.

5.5 Matrix Manipulation

Three of the subject programs were derived from a mathematical software package written as a group project in a graduate software engineering course. One of the programs in this package was a matrix inversion program, which used LU decomposition [33]. The error in this program was not just an implementation error, but rather was a case of choosing a known algorithm that did not quite meet the specification. The problem arose because the LU decomposition algorithm detects some, but not all, singular matrices. Thus, given some singular matrices, the program returns an error message, while given others, it returns a matrix that is purported to be the inverse of the input matrix. It is interesting to note that several well-known numerical methods textbooks [33] describe the LU decomposition algorithm with at most a brief mention of the singularity problem; it is thus very easy to imagine a professional programmer misusing the algorithm.

The algorithm has two steps, called decomposition and backsolving. The decomposition step, implemented in procedure `ludcmp`, returns the LU decomposition of a row-wise permutation of the input matrix. In the backsolving step, achieved by repeated calls to the procedure `lubksb`, the triangular matrices are used to compute the inverse.

In subject program `matinv1`, procedure `ludcmp` was instrumented, while in `matinv2`, `lubskb` was instrumented. In both cases, test sets were drawn from the same universe, which consisted of square matrices with sizes uniformly selected between 0 and 5 and with integer entries selected uniformly between 0 and 24. Outputs were checked by multiplying the output matrix by the input matrix, and comparing to the identity matrix.

Subject program `determinant` used the LU decomposition to compute the determinant of a square matrix. The program operated by calling the `ludcmp` procedure, then multiplying the diagonal elements of the resulting lower-triangular matrix. Like the matrix inversion program, `determinant` produces an error message on some singular matrices, but computes the determinant incorrectly on others. While the errors in these programs are related to one another, it is worth noting that the sets of inputs on which the two programs fail are not identical.

We instrumented the `ludcmp` procedure, and generated another universe in the same way as the universe used for the matrix inversion subjects. To check the outputs, we compared them to results obtained using an inefficient but correct program based on calculating minors and cofactors.

6 Results

The results of the experiment are presented below, organized into three subsections corresponding to each of the three types of questions we asked initially:

1. Are those test sets that satisfy criterion C1 more likely to detect an error than those that satisfy C2?

2. For a given test set size, are those test sets that satisfy criterion C1 more likely to detect an error than those that satisfy C2?

3. How does the likelihood that the test set detects an error depend on the extent to which a test set satisfies the criterion?

In each of these subsections we present and describe tables that summarize the data and its statistical analysis, and we interpret this analysis.

6.1 Overall comparison of criteria

Tables 2, 3, and 4 summarize the results of the comparisons of effectiveness of all-uses to all-edges, all-uses to null, and all-edges to null. The columns labeled N_e, N_u, and N_n give the total numbers of adequate test sets for criteria all-edges, all-uses, and null, respectively, and the columns labeled $\hat{p_e}$, $\hat{p_u}$, and $\hat{p_n}$ give the proportions of these that expose errors. The sixth column of each table gives the significance probability, where applicable; an entry of **** indicates that hypothesis testing could not be applied. A "yes" in the column labeled "$p_e < p_u$?" indicates that all-uses is significantly more effective than all-edges. The columns labeled "$p_n < p_u$?" and "$p_n < p_e$?" answer analogous questions. Where the answer to this question is "yes", confidence intervals are shown in the last column. In those cases where the normality assumption held, a confidence interval for the *difference* in effectiveness between the two criteria (e.g., $p_u - p_e$) is given, while in the other cases, confidence intervals around the effectiveness of *each* criterion are shown. For example, the first row of Table 2 indicates that for `determinant` we are 99% confident that the effectiveness of all-edges lies between 0 and 0.08, whereas that of all-uses lies between 0.52 and 1.0. The second row indicates that for `find1` we are 99% confident that the effectiveness of all-uses is between 0.06 and 0.16 *greater* than that of all-edges.

Examination of these tables shows that for five of the nine subjects, all-uses was more effective than all-edges at 99% confidence; for six subjects, all-uses was more effective than the null criterion; and for five subjects all-edges was more effective than null. Note that for `strmtch2` all-uses would be considered more effective than all-edges at 98% confidence. Further interpretation of these results is given in Section 7.

Subj.	N_e	$\hat{p_e}$	N_u	$\hat{p_u}$	Sig.	$p_e < p_u$?	Confidence
detm	169	0.041	7	1.000	****	yes	[0.00,0.08] vs. [0.52,1.00]
find1	1678	0.557	775	0.667	0.000	yes	[0.06,0.16]
find2	3182	0.252	43	0.256	0.476	no	
matinv1	3410	0.023	76	1.000	****	yes	[0.02,0.03] vs. [0.94,1.00]
matinv2	4789	0.001	4406	0.001	0.500	no	
strmtch1	1584	0.361	238	1.000	****	yes	[0.33,0.39] vs. [0.98,1.00]
strmtch2	1669	0.535	169	0.615	0.015	no	
textfmt	1125	0.520	12	1.000	****	yes	[0.48,0.56] vs. [0.68,1.00]
transpose	1294	0.447	13	0.462	0.456	no	

Table 2: All-edges vs. All-uses

Subj.	N_n	$\hat{p_n}$	N_u	$\hat{p_u}$	Sig.	$p_n < p_u$?	Confidence
detm	6400	0.032	7	1.000	****	yes	[0.03,0.04] vs. [0.52,1.00]
find1	2000	0.484	775	0.667	0.000	yes	[0.13,0.24]
find2	3500	0.234	43	0.256	0.366	no	
matinv1	4000	0.020	76	1.000	****	yes	[0.01,0.03] vs. [0.94,1.00]
matinv2	5000	0.001	4406	0.001	0.500	no	
strmtch1	2000	0.288	238	1.000	****	yes	[0.26,0.31] vs. [0.98,1.00]
strmtch2	2000	0.456	169	0.615	0.000	yes	[0.06,0.26]
textfmt	2000	0.391	12	1.000	****	yes	[0.36,0.42] vs. [0.68,1.00]
transpose	3000	0.407	13	0.462	0.336	no	

Table 3: Null Criterion vs. All-uses

Subj.	N_n	$\hat{p_n}$	N_e	$\hat{p_e}$	Sig.	$p_n < p_e$?	Confidence
detm	6400	0.032	169	0.041	0.255	no	
find1	2000	0.484	1678	0.557	0.000	yes	[0.03,0.12]
find2	3500	0.234	3182	0.252	0.040	no	
matinv1	4000	0.020	3410	0.023	0.179	no	
matinv2	5000	0.001	4789	0.001	****	no	
strmtch1	2000	0.288	1584	0.361	0.000	yes	[0.03,0.11]
strmtch2	2000	0.456	1669	0.535	0.000	yes	[0.04,0.12]
textfmt	2000	0.391	1125	0.520	0.000	yes	[0.08,0.18]
transpose	3000	0.407	1294	0.447	0.006	yes	[0.00,0.08]

Table 4: Null Criterion vs. All-edges

Subj.	$N_{e'}$	$\hat{p_{e'}}$	$N_{u'}$	$\hat{p_{u'}}$	Sig.	$p_{e'} < p_{u'}$?	Confidence
detm	6304	0.033	168	0.042	0.259	no	
find1	1838	0.527	1246	0.634	0.000	yes	[0.06,0.15]
find2	3341	0.244	2425	0.274	0.004	yes	[0.00,0.06]
matinv1	3882	0.020	150	0.513	0.000	yes	[0.39,0.60]
matinv2	4789	0.001	4406	0.001	****	no	
strmtch1	1960	0.293	1424	0.384	0.000	yes	[0.05,0.13]
strmtch2	1950	0.465	795	0.564	0.000	yes	[0.05,0.15]
textfmt	1772	0.429	123	1.000	****	yes	[0.40,0.46] vs. [0.96,1.00]
transpose	2913	0.411	100	0.420	0.428	no	

Table 5: All-but-two duas vs. All-but-two edges

We next compare test sets that cover $X\%$ of the duas to test sets that cover $Y\%$ of the edges. In particular, Table 5 compares test sets that cover all but two duas to those that cover all but two edges. For example, since `determinant` has 103 executable duas and 74 executable edges, the table compares 98% dua coverage to 97% edge coverage for that program. Note that although there was only one program, `strmtch2`, for which the result of hypothesis testing changed from "yes" to "no" in going from 100% coverage to "all-but-two" coverage, the effectiveness of all-uses fell dramatically for several subjects. On the other hand, in one subject, `find2`, "all-but-two" dua coverage was actually slightly more effective than 100% dua coverage. Additional comparisons of $X\%$ dua coverage to $Y\%$ edge coverage can be made by examining the raw data [38].

6.2 Comparison of criteria for fixed size test sets

In Tables 6, 7, and 8, the test sets are grouped according to their sizes. In four of the nine subjects, all-uses adequate test sets are more effective than all-edges adequate sets and null-adequate sets of similar size. Thus it appears that in four of the five subjects for which all-uses was more effective than all-edges and in four of the six for which all-uses was more effective than the null criterion, the improved effectiveness can be attributed to the inherent properties of all-uses, not just to the fact that all-uses adequate test sets are larger on the average than all-edges and null adequate test sets. In contrast, all-edges adequate sets were not more effective than null-adequate sets of the same size for *any* of the subjects. This indicates that, in those cases where all-edges was more effective than null, the increased effectiveness was primarily due to the fact that all-edges demanded larger test sets than the null criterion.

Subj.	size	N_e	$\hat{p_e}$	N_u	$\hat{p_u}$	Sig.	$p_e < p_u$?
detm	1-6	5	0.000	0	-	-	
	7-12	5	0.000	0	-	-	
	13-18	10	0.000	0	-	-	
	19-24	25	0.040	1	1.000	****	
	25-30	73	0.000	0	-	-	
	>30	51	0.118	6	1.000	****	yes
find1	1-5	211	0.299	1	0.000	****	
	6-10	470	0.440	97	0.474	0.270	no
	11-15	497	0.616	285	0.653	0.151	no
	16-20	500	0.718	392	0.727	0.383	no
find2	1-5	205	0.088	0	-	-	
	6-10	482	0.168	0	-	-	
	11-15	496	0.228	4	0.000	****	
	16-20	1999	0.296	39	0.282	0.575	no
matinv1	1-6	205	0.000	0	-	-	
	7-12	484	0.015	6	1.000	****	yes
	13-18	550	0.013	7	1.000	****	yes
	19-24	579	0.021	12	1.000	****	yes
	25-30	595	0.024	14	1.000	****	yes
	31-35	498	0.032	16	1.000	****	yes
	36-40	499	0.042	21	1.000	****	yes
matinv2	1-5	4090	0.001	3714	0.001	0.500	no
	6-10	699	0.001	692	0.001	****	
strmtch1	1-5	194	0.155	0	-	-	
	6-10	416	0.298	77	1.000	****	yes
	11-15	484	0.388	161	1.000	****	yes
	16-20	490	0.469	0	-	-	
strmtch2	1-5	238	0.366	1	1.000	****	
	6-10	447	0.438	16	0.438	0.500	no
	11-15	486	0.591	49	0.592	0.495	no
	16-20	498	0.649	103	0.650	0.492	no
textfmt	1-5	99	0.354	0	-	-	
	6-10	258	0.399	0	-	-	
	11-15	348	0.511	2	1.000	****	
	16-20	420	0.640	10	1.000	****	yes
transpose	10-20	359	0.306	1	0.000	****	
	22-32	935	0.501	12	0.500	0.503	no

Table 6: All-edges vs. All-uses By Size

Subj.	size	N_n	$\hat{p_n}$	N_u	$\hat{p_u}$	Sig.	$p_n < p_u$?
detm	1-6	6100	0.034	0	-	-	
	7-12	1700	0.048	0	-	-	
	13-18	600	0.015	0	-	-	
	19-24	1100	0.026	1	1.000	****	
	25-30	3000	0.033	0	-	-	
	>30	500	0.122	6	1.000	****	yes
find1	1-5	1600	0.500	1	0.000	****	
	6-10	900	0.550	97	0.474	0.9085	no
	11-15	500	0.616	285	0.653	0.1450	no
	16-20	500	0.718	392	0.727	0.3824	no
find2	1-5	3100	0.243	0	-	-	
	6-10	1900	0.268	0	-	-	
	11-15	500	0.230	4	0.000	****	
	16-20	2000	0.295	39	0.282	0.5694	no
matinv1	1-6	3700	0.021	0	-	-	
	7-12	1500	0.022	6	1.000	****	yes
	13-18	600	0.012	7	1.000	****	yes
	19-24	600	0.020	12	1.000	****	yes
	25-30	600	0.023	14	1.000	****	yes
	31-35	500	0.032	16	1.000	****	yes
	36-40	500	0.042	21	1.000	****	yes
matinv2	1-5	4400	0.001	3714	0.001	0.5000	no
	6-10	700	0.001	692	0.001	****	no
strmtch1	1-5	1600	0.299	0	-	-	
	6-10	900	0.337	77	1.000	****	yes
	11-15	500	0.376	161	1.000	****	yes
	16-20	500	0.460	0	-	-	
strmtch2	1-5	1600	0.473	1	1.000	****	
	6-10	900	0.507	16	0.438	0.6960	no
	11-15	500	0.574	49	0.592	0.4021	no
	16-20	500	0.646	103	0.650	0.4690	no
textfmt	1-5	1600	0.412	0	-	-	
	6-10	900	0.446	0	-	-	
	11-15	500	0.480	2	1.000	****	
	16-20	500	0.618	10	1.000	****	yes
transpose	10-20	1200	0.298	1	-	-	
	22-32	1800	0.479	12	0.500	0.4411	no

Table 7: Null vs. All-uses By Size

Subj.	size	N_n	$\hat{p_n}$	N_e	$\hat{p_e}$	Sig.	$p_e < p_u$?
detm	1-6	6100	0.034	5	0.000	****	
	7-12	1700	0.048	8	0.000	****	
	13-18	600	0.015	10	0.000	****	
	19-24	1100	0.026	25	0.040	****	
	25-30	3000	0.033	73	0.000	****	
	>30	500	0.122	51	0.118	0.533	no
find1	1-5	1600	0.500	211	0.299	1.000	no
	6-10	900	0.550	470	0.440	1.000	no
	11-15	500	0.616	497	0.616	0.500	no
	16-20	500	0.718	500	0.718	0.500	no
find2	1-5	3100	0.243	205	0.088	1.000	no
	6-10	1900	0.268	482	0.168	1.000	no
	11-15	500	0.230	496	0.228	0.530	no
	16-20	2000	0.295	1999	0.296	0.472	no
matinv1	1-6	3700	0.021	205	0.000	****	
	7-12	1500	0.022	545	0.013	0.889	no
	13-18	600	0.012	550	0.013	0.438	no
	19-24	600	0.020	579	0.021	0.451	no
	25-30	600	0.023	595	0.024	0.454	no
	31-35	500	0.032	498	0.032	0.500	no
	36-40	500	0.042	499	0.042	0.500	no
matinv2	1-5	4400	0.001	4090	0.001	****	
	6-10	700	0.001	699	0.001	****	
strmtch1	1-5	1600	0.299	194	0.155	1.000	no
	6-10	900	0.337	416	0.298	0.914	no
	11-15	500	0.376	484	0.388	0.348	no
	16-20	500	0.460	490	0.469	0.388	no
strmtch2	1-5	1600	0.473	238	0.366	0.998	no
	6-10	900	0.507	447	0.438	0.989	no
	11-15	500	0.574	486	0.591	0.293	no
	16-20	500	0.646	498	0.649	0.460	no
textfmt	1-5	1600	0.412	99	0.354	0.862	no
	6-10	900	0.446	258	0.399	0.903	no
	11-15	500	0.480	348	0.511	0.183	no
	16-20	500	0.618	420	0.640	0.244	no
transpose	10-20	1200	0.298	359	0.306	0.385	no
	22-32	1800	0.479	935	0.501	0.134	no

Table 8: Null vs. All-edges By Size

Subject	$f(s,c)$
detm	****
find1	$-7.234 + 6.866c - 0.657s + 0.732cs$
find2	$-3.826 - 0.009s^2 - 0.707c^2 s + 1.028cs$
matinv1	$17.940 - 584.6c + 572.0c^2 + 0.005cs^2 - 0.265cs$
matinv2	$-29.513 + 24.146cs - 7.225cs \ln s - 24.182c^2 \ln s$
strmtch1	$-1.579 - 19.093c^2 + 139.3c^{16} - 282.1c^{26} + 201.5c^{39}$ $+16.816 \ln s - 17.346c^4 \ln s + 16.598 \ln c \ln^2 s$
strmtch2	****
textfmt	$2600.2 + 1125.0 \ln c - 5203.1c + 3668.2c^2 - 1043.3c^3 - 0.116c^3 \ln s$
transpose	$-6.830 - 52.359 \ln c - 213.4 \ln^2 c + 1.932c^6 \ln s$

Table 9: Logistic Regression Results for dua Coverage

Subject	$f(s,c)$
detm	$-5.4699 - 0.000563s^2 + 0.0930cs$
find1	$-17.6406 + 0.1213s + 16.4491c$
find2	$-9.3524 + 0.1211s + 6.6455c - (2.58e - 11)s^8$
matinv1	$-6.6803 + 2.5532c^{32} + (4.6e - 7)s^4$
matinv2	*****
strmtch1	$-1.7563 - 7.0121s + 7.1065cs$
strmtch2	$1.8530 - 0.4064s + 0.4962sc^{18} + 0.9350c^{18}$
textfmt	*****
transpose	$-1.7747 + 0.0625cs$

Table 10: Logistic Regression Results for edge Coverage

6.3 Relationship between coverage and effectiveness

The results of logistic regression are shown in Tables 9 and 10. As was discussed in Section 4.3, each regression equation is of the form

$$Prob(exposing) = \frac{\exp f(s,c)}{1 + \exp f(s,c)}$$

where $f(s,c)$ is a function of the predictor variables, s, the test set size, and c, the fraction of duas or edges covered by the test set. The table gives the functions $f(s,c)$ for each subject program for which we were able to find a good-fitting model. The asterisks in some table entries indicate that the data are so scattered that any function that gives a good fit is too complex to offer much insight into the relationship, if any exists, between coverage and effectiveness. The regression equations give us information about the way in which the effectiveness of a test set depends, if at all, upon coverage of the criterion and test set size. Because the functions $f(s,c)$ have several terms, it is difficult to understand very much about the dependence relationship by inspection of the table alone. However, for some of the subjects, $f(s, c)$ is simple enough that one can restrict one's attention to the coefficient of the term containing the highest power of c. If this coefficient is positive and has a large magnitude, then effectiveness depends strongly and positively on coverage. This is the case, for example, for the `find1` subject for both dua and edge coverage. For `find1`, all terms involving c are positive for both types of coverage, so we can safely conclude that effectiveness is strongly correlated to dua and edge coverage.

For some of the subject programs, we have included graphs of Prob(exposing) versus proportion of duas (edges) covered at selected test set sizes so that the reader can see the relationship more clearly. In the figures, the all-uses and all-edges graphs are superimposed; to distinguish them, we use dashed lines for the all-edges graph and dot-dashed lines for the all-uses graphs. In Figure 1, one can see that for `find1` the probability of exposing an error increases monotonically as coverage increases, for test sets with 15 test inputs. In fact, for any test set size, this would be true; we picked $s = 15$ for illustrative purposes only.

For some of the other subjects, the relationship is harder to determine. Careful analysis of Table 9, however, shows that for `find1, matinv1,` and `strmtch1`, effectiveness depends strongly and positively on coverage of duas. Similarly, analysis of Table 10 shows that for `find1, matinv1, strmtch1` and `strmtch2`, effectiveness depends positively on coverage of edges. The graphs in most of these cases tend to look very much like the graph for `strmtch1` illustrated in Figure 2. In these cases, the probability of exposing an error is negligible unless a sufficiently large percentage of duas or edges is covered. Then as more duas or edges are covered, the probability rises sharply to a plateau on or near 1.0. We offer a possible explanation for this in the next section.

In summary, there is a clear dependence of effectiveness on extent of coverage of the all-uses criterion in only three of the nine subjects; in a fourth subject, `transpose`, the probability of exposing an error increases as the percentage of duas covered increases, except that it drops slightly when the percentage gets close to 100%. In four of the subjects such a dependence exists for the all-edges criterion.

7 Discussion

Close examination of the data led us to several interesting observations. While all-uses was not always more effective than all-edges and the null criterion, in most of those cases where it was more effective, it was *much* more effective. In contrast, in those cases in which all-edges was more effective than the null criterion, it was usually only a little bit more effective.

For `buggyfind`, all-uses performed significantly better than all-edges when the `find1` universe was used, apparently because all-uses required larger test sets. However, when the `find2` universe was used there was little difference between the criteria. Also, the

Figure 1: Coverage vs. Prob(exposing) for `find1`

Figure 2: Coverage vs. Prob(exposing) for `strmtch1`

Figure 3: Coverage vs. Prob(exposing) for `find2`

Figure 4: Coverage vs. Prob(exposing) for `textfmt`

effectiveness of each criterion is dramatically better for `find1` than for `find2`. This shows that even relatively minor changes in the test generation strategy can profoundly influence the effectiveness of an adequacy criterion and that blanket statements about "random testing" without reference to the particular input distribution used can be misleading.

For `matinv1`, all-uses appears to be guaranteed to detect the error, while for `matinv2`, in which a different procedure is instrumented, all-uses performs poorly. This is in part due to the fact that it is very easy to satisfy all-uses in `matinv2`, and partly due to the possibility that the procedure instrumented in `matinv2` has nothing to do with the bug.

Both `matinv1` and `detm` involve instrumenting the `ludcmp` procedure. While 100% dua coverage appears to guarantee detection of the error in both of these cases, 98% dua coverage is much more effective for `detm` than for `matinv1` (0.556 vs 0.026). This is interesting because it shows that an adequacy criterion can be more or less effective for a given procedure depending upon the program in which that procedure is used.

In four of the subjects, `determinant`, `matinv1`, `textfmt`, and `strmtch1`, coverage of all duas appears to guarantee detection of the error. We knew this prior to the experiment for `strmtch1`, but were surprised to see it for the other three programs.

Figure 3 contains the graphs for `find2`. In the figure, two graphs each for all-uses and all-edges, for test set sizes of 10 and 20, are superimposed. For both test set sizes, the all-edges graph shows that the probability of exposing an error monotonically increases as the number of covered edges increases. The all-uses graphs have upward slope until roughly 70% of the duas have been covered, after which they take a downturn. This phenomenon might be the result of insufficient data above 70% coverage combined with good a fit of the regression curve to the data.

The graphs shown in Figures 2 and 4, and the data from `determinant` and `matinv1` found elsewhere [38], exhibit an interesting phenomenon. At high values of coverage, the probability P of error detection is 1.0; then, as coverage decreases, a point is reached at which P decreases rapidly. The raw data [38] corroborate this:

- For `matinv1`, the effectiveness goes from 1.0 at 100% coverage to 0.03 at 98% coverage, i.e., for test sets that covered *at least* 98% of the executable duas.
- For `strmtch1`, the effectiveness is 1.0 at 100% coverage, 1.0 at 98% coverage (all-but-one dua) and about 0.35 at 95% coverage (all but 2 duas).
- For `textfmt`, the effectiveness of dua coverage from 100% down to 83% is 1.0, then the effectiveness of 80% dua coverage falls to 0.58. But strangely enough, for values of c between about 0.4 and 0.6, it is 1.0 again. This arises from the fact that all test sets with coverage $c = 0.524$ were exposing, and the curve is fitted closely to the data. It is likely that the only way to achieve coverage of exactly 0.524 is for a particular set of paths to have been executed by the test set and that executing this set of paths *guarantees* exposing the error. At the same time, test sets that

execute a different set of paths that covers more duas are not guaranteed to expose the error.

- For `determinant`, the effectiveness goes from 1.0 at 100% coverage to 0.556 at 98% coverage.

Thus it appears that in each of these cases, not only was there one or more duas whose coverage guaranteed error detection, but that the remaining duas were largely irrelevant.

This phenomenon has profound consequences for testing practitioners, who might deal with the unexecutable dua problem by testing until some arbitrary predetermined percentage of the duas have been covered. If it so happens that the test set fails to cover any of the "crucial" duas, the test set may be much less likely to detect the error than if it had covered 100% of the executable duas.

For example, in `matinv1`, there are a total of 298 duas, only 206 (69%) of which are executable. Suppose it has been decided that test sets that cover 200 of the duas will be deemed adequate. Then it will be quite possible to test without hitting any of the crucial duas, so the chance of exposing the error will be much less than it would have been with coverage of 100% of the executable duas.

Consequently, we recommend that practitioners using data flow testing put in the effort required to weed out unexecutable def-use associations, and only accept test sets that achieve 100% coverage of the executable duas. A heuristic for doing this is presented by Frankl [13] and the issue of how this affects the cost of using a criterion is discussed by Weiss [37].

We more closely examined the programs in which all-uses was guaranteed to expose the error, to gain insight into situations in which all-uses seems to perform well. In each of these cases, the fault could be classified as a "missing path error", i.e., a situation in which the programmer forgot to take special action under certain circumstances.[3] This is particularly interesting, because structured testing criteria are usually considered to be poor at exposing missing path errors, since test cases that execute the "missing path" are not explicitly demanded. However, in these cases, it turns out that all of the test cases that cover some particular dua happen to be test cases that *would have* taken the missing path, had it been present. Consequently, all-uses guaranteed that these errors were detected.

[3] In `strmtch1` the missing path would return 0 or write an error message if the null pattern was input; in `textfmt` the missing path would skip certain statements if `bufpos = 0`, and in `determinant` and `matinv1` the missing path would return an error message when the input was singular. In the matrix manipulation program, an explicit check for singularity was presumably omitted for efficiency reasons.

8 Related Work

Most previous work comparing testing techniques falls into two categories: theoretical comparisons of adequacy criteria and empirical studies of test generation techniques. There have been many theoretical comparisons of adequacy criteria, but only a few of these have addressed error detecting ability. In this section, we summarize simulations, experiments, and analytical studies that have addressed the fault detecting ability of various testing techniques.

Several papers have investigated the fault detecting ability of a generalization of path testing called "partition testing". Duran and Ntafos [10] performed simulations comparing random testing to partition testing, in which, using hypothetical input domains, partitions, and distributions of errors, they compared the probabilities that each technique would detect an error. Their conclusion was that, although random testing did not always compare favorably to path testing, it did well enough to be a cost effective alternative. Considering those results counterintuitive, Hamlet and Taylor [22] did more extensive simulations, and arrived at more precise statements about the relationship between partition probabilities, failure rates, and effectiveness, which corroborated the Duran-Ntafos results. Jeng and Weyuker [39] attacked the same problem analytically and showed that the effectiveness of partition testing depends greatly on how failure causing inputs are distributed among the "subdomains" of the partition. Frankl and Weyuker [17, 16] investigated the conditions under which one criterion is guaranteed to be more likely to detect a fault than another, and found that the fact that C1 subsumes C2 does not guarantee that C1 is better at detecting faults than C2. Stronger conditions with more bearing on fault-detecting ability were also described. It should be noted that these studies used a different model of test set selection than we used. For a program with m subdomains, they considered test sets of size mk arising from an idealized test generation scheme, namely, independent random selection of k elements from each subdomain, using a uniform distribution on each subdomain. In contrast, for various values of n, we generated test sets by independent random selection of n elements from the universe, then considered only those sets that were adequate.

Several empirical studies have counted the number of errors detected by a particular technique on programs with either natural or seeded errors. Duran and Ntafos [10] executed roughly 50 randomly generated test cases for each of several programs and calculated the percentages of these that exposed errors. Girgis and Woodward [18] seeded five textbook FORTRAN programs with errors and subjected them to various test methods. As Hamlet pointed out [21], experiments of this nature may give misleading results because they typically rely on a small number of test sets to represent each testing technique. Furthermore, some of these studies [18] employ a very liberal notion of when a test case detects an error.

An experiment by Basili and Selby [2], comparing statement testing, partitioning with boundary value analysis and code-reading by stepwise abstraction, differed somewhat from the others in this category, in that it used human subjects to generate tests and evaluated the extent to which their expertise influenced the results. The use of human subjects in this type of experiment is a laudable goal; after all, as long as testing is under human control, human factors will influence results. A problem with this approach is that one cannot necessarily extrapolate to the population one is trying to model, since the human sample may not be representative of that population.

A third category of studies involved measuring the extent to which test sets generated using some particular technique satisfied various adequacy criteria, or the extent to which test sets that satisfy one adequacy criterion also satisfy another. For example, Duran and Ntafos [10] measured the extent to which randomly generated test sets with roughly 20 to 120 test cases satisfied the LCSAJ, all-edges, required pairs, and TER_n criteria. Several studies of this nature have been performed on mutation testing; for example, DeMillo, Lipton, and Sayward measured the extent to which randomly generated test sets satisfied mutation testing on the `buggyfind` program [9], and Offutt measured the extent to which test sets that kill first-order mutants also kill second-order mutants [32]. Note that this type of study does not address the question of error-detecting ability.

While the above cited studies each contributed in some way toward better understanding of software testing, there are several noteworthy differences between each of them and the experiment described in this paper: Our experiment

- compared the *error detecting ability* of *adequacy criteria*, as opposed to error detecting ability of test generation techniques, and as opposed to other characteristics of adequacy criteria;
- was designed to allow the application of rigorous statistical techniques;
- investigated real adequacy criteria (as opposed to hypothetical partitions of the input domain) on real programs with naturally occurring bugs.

None of the above mentioned papers has all three of these attributes.

Finally, we note that there have also been many experimental studies that did use rigorous statistical techniques to investigate other software quality issues [27, 35, 36]. However since none of these were aimed at evaluating the effectiveness of adequacy criteria, they are not directly relevant here.

9 Conclusions

We have described an experiment comparing the effectiveness of randomly generated test sets that are adequate according to the all-edges, all-uses, and null test data adequacy criteria. Unlike previous experiments, this experiment was designed to allow for statistically meaningful comparison of the error-detecting ability *adequacy criteria*. It involved generating large numbers of test sets for each subject program, determining which test sets were adequate according to each criterion, and determining the proportion of adequate test sets that exposed at least one error. The data was analyzed rigorously, using well established statistical techniques.

The first group of questions we posed was whether C1 is more effective than C2 for each subject and for each pair of criteria. For five of the nine subjects, all-uses was significantly more effective than all-edges; for six subjects, all-uses was significantly more effective than the null criterion; for five subjects all-edges was more effective than null. Closer examination of the data showed that in several of the cases in which all-uses did well, it actually did very well, appearing to guarantee error detection. We also compared test sets that partially satisfied all-uses to those that partially satisfied all-edges. In six subjects, test sets that covered all but two definition-use associations were more effective than test sets that covered all but two edges. Thus, test sets that cover all (or almost all) duas are sometimes, but not always more likely to expose an error than those that cover all (almost all) edges.

The second group of questions limited attention to test sets of the same or similar size. All-uses adequate test sets appeared to be more effective than all-edges adequate sets of similar size in four of the nine subjects. For the same four subjects, all-uses adequate test sets appeared to be more effective than null adequate sets of similar size. In contrast, all-edges adequate sets were not more effective than null adequate sets of similar size for any of the subjects. This indicates that in those cases where all-edges was more effective than the null criterion, the increased effectiveness was due primarily to the fact that all-edges test sets were typically larger than null-adequate test sets. In most of the cases where all-uses was more effective than all-edges or than the null criterion, the increased effectiveness appears to be due to other factors, such as the way the criterion concentrates failure-causing-inputs into subdomains.

The third group of questions investigated whether the probability that a test set exposes an error depends on the size of the test set and the proportion of definition-uses associations or edges covered. This is an important question because it is not uncommon for testers and testing researchers to assume implicitly that confidence in the correctness of a program should be proportional to the extent to which an adequacy criterion is satisfied. Logistic regression showed that in four of the nine subject programs the error-exposing ability of test sets tended to increase as these test sets covered more definition-use associations. It also showed that in a different set of four subject programs, there was a weaker, but still positive correlation between the error-exposing ability of test sets and the percentage of edges covered by these sets. However, even in those subjects where the probability that a test set exposes an error depended on the proportion of definition-use associations or edges covered, that dependence was usually highly non-linear. This indicates that one's confidence in the correctness of a program should not in general be proportional to the percentage of edges or duas covered.

In summary, our results show that all-uses can be *extremely* effective, appearing to guarantee error detection in several of the subjects. It did not always perform significantly better than all-edges or the null criterion, but in most of our subjects it did. On the other hand, all-edges was not very effective for most of our subjects; in fact, in none of the subjects did all-edges or almost-all-edges adequate test sets perform significantly better than randomly selected null-adequate test sets of the same size.

We make no claim that our collection of subject programs is representative of all software, and therefore we do not believe it is sensible to extrapolate from our results to software in general. The primary contribution of this research is the methodology we used for the experiment; we believe that our results are both sound and interesting and should motivate further research. In addition, even this relatively small scale experiment allowed us to observe the existence of several interesting phenomena, noted in Section 7. The foremost direction for future research is to perform similar experiments on a much larger collection of subjects, including large programs. Our design could also be used to compare other adequacy criteria. Experiments comparing the effectiveness of various adequacy criteria when non-random test generation strategies are used would also be useful. We hope that other researchers will join us in performing such experiments in the future.

Acknowledgments: The authors would like to thank Prof. Al Baranchik of the Hunter College Mathematics Department for advice on statistical methods, Mohammed Ghriga and Roong-Ko Doong for helping prepare the subject programs, Zhang Ming for helping with the data analysis, Tarak Goradia for useful comments on an earlier version of the paper, and the Hunter College Geology and Geography Department for use of their statistical analysis software. An anonymous referee made several useful suggestions on the presentation of the material.

References

[1] A. Agresti. *Analysis of Ordinal Categorical Data.* John Wiley & Sons, Inc., New York, 1984.

[2] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, SE-13(12):1278–1296, Dec. 1987.

[3] G. K. Bhattacharyya and R. A. Johnson. *Statistical Concepts and Methods.* John Wiley & Sons, Inc., New York, 1977.

[4] R. Boyer, B. Elspas, and K. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Notices*, 10(6):234–245, June 1975. International Conference on Reliable Software.

[5] D. R. Byrkit. *Elements of Statistics.* D. van Nostrand Company, New York, 1980.

[6] L. Clarke, A. Podgurski, D. Richardson, and S. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, SE-15(11):244–251, Nov. 1989.

[7] D. Cooper and M. Clancy. *Oh! Pascal!* W.W. Norton & Company, New York, 1982.

[8] R. A. DeMillo, D. S. Guindi, W. McCracken, A. Offutt, and K. King. An extended overview of the Mothra software testing environment. In *Proceedings Second Workshop on Software Testing Verification and Analysis*, pages 142–151. IEEE Computer Society Press, July 1988.

[9] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978.

[10] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10(7):438–444, July 1984.

[11] P. G. Frankl. ASSET user's manual. Technical Report 318, Computer Science Department, Courant Institute of Mathematical Sciences, New York University, Sept. 1987.

[12] P. G. Frankl. *The Use of Data Flow Information for the Selection and Evaluation of Software Test Data.* PhD thesis, Courant Institute of Mathematical Sciences, New York University, Oct. 1987.

[13] P. G. Frankl. Partial symbolic evaluation of path expressions (version 2). Technical Report PUCS-105-90, Polytechnic University Computer Science Dept., July 1990.

[14] P. G. Frankl, S. N. Weiss, and E. J. Weyuker. ASSET – a system to select and evaluate tests. In *Proceedings of the IEEE Conference on Software Tools*, pages 72–79, Apr. 1985.

[15] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, SE-14(10):1483–1498, Oct. 1988.

[16] P. G. Frankl and E. J. Weyuker. Assessing the fault-detecting ability of testing methods. In *ACM SIGSOFT '91 Conference on Software for Critical Systems*, pages 77 –91. ACM Press, Dec. 1991.

[17] P. G. Frankl and E. J. Weyuker. A formal analysis of the fault detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 1993. To appear.

[18] M. Girgis and M. Woodward. An experimental comparison of the error exposing ability of program testing criteria. In *Proceedings IEEE Workshop on Software Testing*, pages 64–73. IEEE Computer Society Press, July 1986.

[19] J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975.

[20] F. Gustavson. Remark on algorithm 408. *ACM Transactions on Mathematical Software*, 4:295, 1978.

[21] D. Hamlet. Theoretical comparison of testing methods. In *Proceedings ACM SIGSOFT Third Symposium on Software Testing, Analysis, and Verification*, pages 28–37. ACM Press, Dec. 1989.

[22] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):206–215, Dec. 1990.

[23] P. Herman. A data flow analysis approach to program testing. *The Australian Computer Journal*, 8(3):92–96, Nov. 1976.

[24] C. Hoare. Proof of a program: Find. *Commun. ACM*, 14(1):39, 1971.

[25] W. E. Howden. A survey of dynamic analysis methods. In *Tutorial: Software Testing and Validation Techniques*, pages 209–231. IEEE Computer Society Press, 1978.

[26] J. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3):113–128, Sept. 1975.

[27] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1), Jan. 1986.

[28] B. Korel and J. W. Laski. STAD – a system for testing and debugging: user perspective. In *Proceedings Second Workshop on Software Testing Verification and Analysis*, pages 13–20. IEEE Computer Society Press, July 1988.

[29] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–354, May 1983.

[30] J. M. McNamee. Algorithm 408: A sparse matrix package (part I) [F4]. *Commun. ACM*, 14(4), Apr. 1971.

[31] S. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, Nov. 1984.

[32] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):5–20, Jan. 1992.

[33] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, Cambridge, UK, 1986.

[34] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-14(4):367–375, Apr. 1985.

[35] J. H. Rowland and Y. Zuyuan. Experimental comparison of three system test strategies: preliminary report. In *Proceedings ACM SIGSOFT Third Symposium on Software Testing, Analysis, and Verification*, pages 141–145. ACM Press, Dec. 1989.

[36] M. Vouk, D. McAllister, and K. Tai. An experimental evaluation of the effectiveness of random testing of fault-tolerant software. In *Proceedings IEEE Workshop on Software Testing*, pages 74–81. IEEE Computer Society Press, July 1986.

[37] S. N. Weiss. Methods of comparing test data adequacy criteria. In *COMPSAC 90*, pages 1–6, Oct. 1990.

[38] S. N. Weiss and P. G. Frankl. Comparison of all-uses and all-edges: Design, data, and analysis. Technical Report CS-91-03, Hunter College Computer Science Dept., Mar. 1991.

[39] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[40] E. J. Weyuker, S. N. Weiss, and D. Hamlet. Comparison of program testing strategies. In *Proceedings Fourth Symposium on Software Testing, Analysis, and Verification*, pages 1–10. ACM Press, Oct. 1991.

List of Figures

List of Tables