Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria

Research Paper

Monica Hutchins, Herb Foster, Tarak Goradia, Thomas Ostrand
Siemens Corporate Research, Inc.
755 College Road East, Princeton, NJ 08540 U.S.A.
Email: {mhutchins,hfoster,tgoradia,tostrand}@scr.siemens.com

Abstract

This paper reports an experimental study investigating the effectiveness of two code-based test adequacy criteria for identifying sets of test cases that detect faults. The all-edges and all-DUs (modified all-uses) coverage criteria were applied to 130 faulty program versions derived from seven moderate size base programs by seeding realistic faults. We generated several thousand test sets for each faulty program and examined the relationship between fault detection and coverage. Within the limited domain of our experiments, test sets achieving coverage levels over 90% usually showed significantly better fault detection than randomly chosen test sets of the same size. In addition, significant improvements in the effectiveness of coverage-based tests usually occurred as coverage increased from 90% to 100%. However, the results also indicate that 100% code coverage alone is not a reliable indicator of the effectiveness of a test set. We also found that tests based respectively on controlflow and dataflow criteria are frequently complementary in their effectiveness.

1 Introduction

Controlflow-based code coverage criteria have been available to monitor the thoroughness of software tests at least since the 1960's [11, 12, 24]. More recently, dataflow-based methods have been defined and implemented in several tools [7, 10, 15, 20]. Various comparisons have been made of the theoretical relations between coverage methods [4]. However, the questions of real concern to researchers and potential users of these adequacy criteria deal with their actual effectiveness in detecting the presence of faults in programs. Test managers and developers would like to know whether the investment in systems to monitor code coverage is worthwhile, and whether the effort to look for additional tests that increase coverage is well-spent. They would like to know the additional cost of achieving adequate coverage, the payback for that cost, and in particular, whether fault detection increases significantly if test sets are adequate or close to adequate according to the criteria.

In an effort to answer these questions, we have performed experiments comparing dataflow coverage and controlflow coverage using the dataflow coverage system Tactic developed at Siemens Corporate Research [20]. To make our results as relevant as possible to professional software developers and testers, we searched available public archives for specifications and C programs that would be suitable for the study. We ended up with seven moderate-size C programs, into which we seeded 130 different faults.

Section 2 of the paper describes the test adequacy criteria that are monitored by Tactic. Section 3 briefly describes some previous work relating to evaluation of adequacy criteria. Section 4 presents the goals of our study, discusses assumptions and the design of the experiments, and describes the programs used in the study. Section 5 explains some of the data analysis. In Section 6 we describe our observations. Section 7 contains conclusions.

2 The Test Adequacy Criteria

2.1 Dataflow Coverage

Dataflow-based adequacy criteria stipulate that a test set must exercise certain *def-use associations* that exist in the code. A *def* of a memory location is an operation that writes a value to the location. A *use* of a location is an operation that reads the location's current value. A *def-use association* (DU) *for a given location* is a pair consisting of a def and a use of the location, such that there is a controlflow path in the code from the def to the use on which there is no intermediate redefinition or undefinition of the location. A *test case exercises* a particular def-use association if the test case causes execution to arrive at the site of the def operation and execute the def, and subsequently arrive at the site of the use operation and execute the use, without having executed any other def or undefinition of the memory location. A *test set exercises* a DU if at least one test case in the set exercises the DU.

Note that a DU is defined in terms of static properties of the code, i.e., in terms of the existence of a path in the code's controlflow graph, while exercising a DU is defined in terms of dynamic execution. To satisfy the *all-*

0270-5257/94 $3.00

DUs criterion, a test set must exercise every executable DU in the program.

Test adequacy criteria based on dataflow were proposed by Rapps and Weyuker [22, 23], Ntafos [18], and Laski and Korel [16] as alternatives to the controlflow-based measures of test adequacy.

The first dataflow adequacy tool was implemented by Frankl, Weiss, and Weyuker [7,8], who built the ASSET system that operated on Pascal code in accordance with the definitions of Rapps and Weyuker. The inputs to ASSET are a Pascal program and a set of test data; its output is a determination of whether or not the test data are adequate with respect to one of the Rapps-Weyuker dataflow criteria.

The Tactic system for C programs evaluates test sets for all-DUs adequacy, as well as for all-edges adequacy. Since our ultimate goal is to produce a test adequacy tool that is usable in production environments, Tactic handles almost all of standard C, including structures, arrays, pointer references, and interprocedural dataflow. A key feature of Tactic is its ability to perform accurate analysis of code for def-use associations where one or both of the def and use are indirect references [20, 21]. Although this capability greatly enhances the system's ability to track dataflow accurately, it also greatly increases the number of def-use associations detected for a program, as well as creating the problem of determining whether or not there exists a runtime def-use association that corresponds to a statically found def-use association involving a pointer reference or a structure element.

Our dataflow criterion *all-DUs* differs from the *all-uses* criterion originally defined by Rapps and Weyuker and used as the basis of ASSET. Because of the following three major differences, test sets satisfying *all-DUs* are in general not comparable to sets satisfying *all-uses*. First, we do not distinguish between *c-uses* and *p-uses*; a *use* is any occurrence in the program where a value is accessed from memory. In the Rapps-Weyuker theory, *p-uses* were defined so that satisfaction of all-uses would imply satisfaction of branch coverage, since a *p-use* of every variable that appears in the predicate of a decision statement is placed on every branch that leads away from the predicate. In our system, *all-DUs* does not subsume branch coverage, since our definition does not provide a way to force execution of any particular branch leaving a predicate node. We do not combine DU and branch coverage in a single criterion since each measure has its separate benefits. In addition, for these experiments, we were interested in studying the effects of the individual test requirements induced by each type of coverage.

Second, our definition of *use* is not restricted to named variables, since in C it is possible to reference a memory location through a pointer without having a variable associated with the location. Thus, a def occurs when dynamically allocated memory is assigned a value with a statement such as *p = 15; a corresponding use occurs in a statement such as x = *q + 5, if there have been no intervening definitions of the location, and q points to the location assigned to through p. We also capture interprocedural def-use associations, where the def occurs in a calling procedure, and the use in the called procedure is through a dereferenced pointer or a global variable reference.

Third, our method of checking test execution for satisfaction of the dataflow criteria differs from that used in ASSET, where a DU is considered exercised if a def-clear path from a def node to a use node is executed. In the presence of pointer dereferences, it is not sufficient to monitor the execution of paths from def sites to use sites, since this does not guarantee that the def and use are of the same memory location, or even that a def or use have been executed. Hence Tactic considers a DU to be exercised only when an actual write of a memory location (the def) is followed by an actual fetch from that location (the use).

2.2 Controlflow Coverage

Edge coverage by Tactic extends traditional branch coverage by considering not only edges based on explicit controlflow statements in the code, but also edges based on implicit controlflow in Boolean expressions. For example, Tactic considers the C statement

```
if (a && b && c) x=5;
else x=10;
```

to have 6 edges, not 2; four sets of values for a, b, and c are required to exercise all the edges. Beizer [3] refers to this level of coverage as *predicate coverage.*

3 Other Experimental Work

Frankl and Weiss conducted a study [6] which compared the all-edges criterion to the all-uses criterion for nine Pascal programs. For some of the subject programs, they concluded that test sets satisfying the all-uses criterion were more effective at detecting faults than sets satisfying all-edges. We discuss the Frankl-Weiss study in greater detail in Section 4.2.3.

A study by Basili and Selby [2] attempted to compare three techniques: code reading by stepwise abstraction, functional testing using equivalence partitioning and boundary value analysis, and structural testing using statement coverage. Seventy-four programmers applied the three techniques to four unit-sized programs containing a total of 36 faults in a fractional factorial experiment, giving observations from 222 testing sessions on the effectiveness of the testing methods. They also did a cost analysis and a characterization of the faults detected.

A study by Thevenod-Fosse, Waeselynck, and Crouzet [25] used mutation scores to compare the effectiveness of deterministic structural testing techniques to their own method of test generation (structural statistical testing). The mutations were automatically created from four small C programs, creating a total of 2914 mutants; equivalent mutants were eliminated by hand. The test sets for the deterministic part of the testing were created by hand for each criterion under consideration; for each criterion and for each program, at most 10 test sets were designed, with at most 19 members in each test set. The resulting

mutation scores of these test sets were used to determine the relative effectiveness of the methods. The study concluded that structural statistical testing was more effective.

Foreman and Zweben's [5] study examined the effectiveness of several variants of controlflow and dataflow criteria in detecting thirty of the real faults that were documented by Knuth [14] during development of the TeX program. A testing criterion was considered effective at detecting a fault only when *all* test sets satisfying the criterion revealed the fault. Application of the all-uses criterion guaranteed detection of thirteen of the thirty faults. Eleven of these thirteen were also guaranteed to be detected by tests satisfying branch coverage.

4 Goals and Design of the Experimental Study

4.1 Goals

Broadly stated, our goal for these experiments was to obtain meaningful information about the effectiveness of controlflow and dataflow coverage methods for fault detection. We hoped to document the different capabilities of the two coverage methods. We also tried to answer questions concerning the use of code-based coverage criteria, e.g., whether it is necessary to achieve 100% coverage to benefit from using a criterion.

An important goal was to carry out the experiment on realistic programs, using test generation techniques that are similar to actual practice.

4.2 Experimental Design

While designing an experiment for comparing the fault detection ability of two testing strategies, the choice of subject *faulty programs* is clearly very important. Ideally, they should represent both the *program space* and the *fault space*. Our sampling of the program space was severely restricted, partly due to the availability of resources for carrying out the experiments and partly due to the limitations of our prototype. Our sampling of the fault space was also restricted. Ideally, the most desirable types of faults to study would be real faults that have been recorded in the course of development of production software. However, since there is only scant information available to us about production faults, we decided to manually create and seed faults into the subject programs. For the experiment, we created 130 faulty program versions from seven moderate size base programs by seeding realistic faults. Characteristics of the base programs and the seeded faults are described in Section 4.3.

The results of this study should be interpreted keeping in mind this limited representation of the program space and the fault space.

Adequacy criteria such as all-DUs or all-edges define a stopping rule, but do not specify how the test cases are to be generated, analyzed, and validated. There are several possible models for a testing strategy using a coverage-based test adequacy criterion. In this section, we first describe one such model and discuss the compromises made in order to implement this model in our experiments. Then we describe the experimental procedure based on the compromised model.

4.2.1 Model of Coverage-based Testing

As shown in Figure 1, our experimental design is based on the following model of a coverage-based testing strategy. First, the tester generates an initial set of test cases and runs the coverage analysis on it. If the coverage is inadequate, the tester generates additional test cases until adequate coverage is attained. The resulting intermediate test set may be too big or may contain test cases that are redundant with respect to the goal of achieving adequate coverage. Hence the tester may employ a strategy to prune the intermediate test set and obtain a test set with a smaller number of test cases having the same coverage. The purpose of this pruning may be to reduce the effort for validating the test cases and/or the effort for maintenance of the test set.

Ideally, satisfaction of an adequacy criterion by a test set would assure a specific level of fault detection regardless of the methods used in generating the test cases and in pruning the test sets. However, our experience in this study indicates that for the all-DUs and all-edges criteria, this is not true: two test sets satisfying the same coverage criterion may differ widely in their fault detection ability. The quality of the final test set produced may be affected by the methods used for initial test generation, additional test generation, and test set pruning. Usually the initial test generation method is independent of the coverage criterion, while the additional test generation and test set pruning methods use the coverage information produced by the test cases previously generated. Below, we discuss the practical problems encountered in specifying these methods and

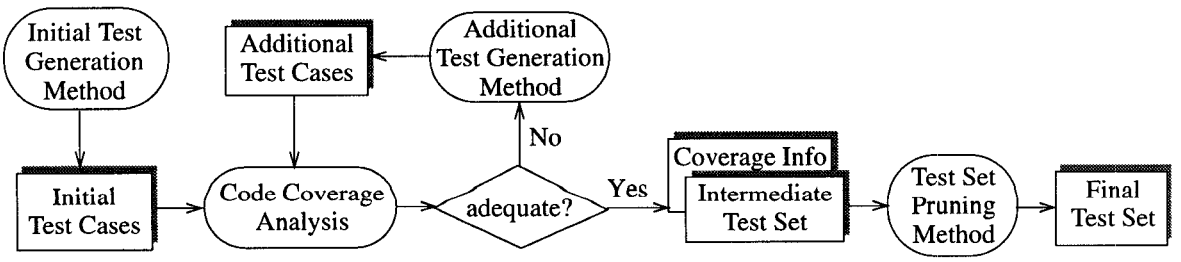

Figure 1: Model of Coverage-based Testing

describe the compromises made to work around the problems.

Test Generation

Testers use their own expertise and domain knowledge in designing initial and additional test cases. We found that adequate test sets produced by different testers varied in their effectiveness at detecting faults. Interestingly, adequate test sets produced by the same tester also varied in their effectiveness. Therefore, in order to compare two testing strategies for a given program, it was necessary to carry out a statistical comparison of the fault detection abilities of a large number of independent test sets that were produced by using each of the testing strategies. Towards this goal, we introduce the notion of a *test pool* and describe the *test set generation method* used for choosing test sets from the test pool. Ideally, the test pool for a specific (program, specification, tester) combination should be the set of all test cases that are *accessible* to the tester while testing the program against the specification. By *accessible* test cases, we mean those from which the tester is likely to choose, given his/her expertise and knowledge of the program domain. In reality, the test cases accessible while following one testing strategy may be very different from those accessible while following another strategy. *For simplicity, we assume that the accessible test cases do not vary significantly between the testing strategies based on the all-DUs and all-edges criteria.* Also, the likelihood of choosing a test case may differ across the test pool, hence we would also need a probability distribution function to accompany the test pool. *For lack of a better option, we use a uniform distribution.* To minimize the bias introduced by a specific tester, we used two to three testers for generating the test pool. Given the test pool, the test set generation method was straightforward: test cases are randomly selected from the test pool until a desired size test set is obtained. We hope that the test sets thus produced would represent the variety of test cases that the testers are likely to produce. The actual procedure used to generate the test pools for the subject programs is described later.

Test Pruning

As mentioned earlier, the purpose of test set pruning could be to reduce the effort for validating the test cases and/or the effort for maintenance of the test set. Since our goal was to examine the effectiveness of test adequacy criteria, test cases that did not improve coverage were of little value. If an intermediate test set consisted of n test cases, t_1 through t_n, generated in that order, the test set pruning method eliminated a test case t_i if it did not improve the cumulative coverage obtained by test cases t_1 through t_{i-1}.

Summary

In our experiment, we consider a fixed set of faulty programs and a fixed group of testers. In order to simulate the behavior of the testers for selecting test cases several times using each of two coverage criteria, we introduce the notion of a test pool that approximates the set of test cases accessible to the testers while testing a specific (program, specification) pair. Random selection from this test pool is used as the test generation method. The test pruning method is dependent on the coverage criteria: following the order of test generation, test cases that do not improve coverage are eliminated.

Below, we elaborate the actual procedure used for generating the test pool and test sets, and for collecting the experimental data.

4.2.2 Actual Experimental Procedure

Test Pool Generation

We produced the test pool for each program in two stages that correspond to the way a tester might use an adequacy criterion in practice. The first stage consisted of creating a set of test cases according to good testing practices, based on the tester's understanding of the program's functionality and knowledge of special values and boundary points that are easily observable in the code. To create this *initial test pool* (*ITP*), the tester applied the category-partition method to write test specification scripts for the Siemens TSL[1] tool [1, 19] to produce test cases. The tester examined the coverage achieved by the TSL-produced tests, and modified the test specification to improve coverage. At some point, the tester decided that it was time to move on to the second stage where he/she individually examined the unexercised coverage units in the code and attempted to write test cases to exercise them. In fact, the goal was to insure that each exercisable coverage unit was covered by at least 30 *different* test cases, where two test cases are considered different if the simple control paths that they exercise differ[2]. Since the test pools as described so far were constructed from the base versions of the programs, we also examined each faulty version, and added necessary test cases to the pool to insure that each exercisable coverage unit in the faulty versions was covered by at least 30 cases. The cases in this *additional test pool* (*ATP*) were mostly hand generated, although sometimes new TSL scripts were written to specify the test cases. The final set of test cases produced by augmenting the initial pool is called the *test pool* (*TP*). Table 2 gives the sizes of the test pools for each base program and indicates the relative size of the corresponding initial and additional test pools. For every test case in a test pool, we ran each of the faulty versions of the program and recorded in a table the outcome (correct = no fault detection, incorrect = fault detection) and the list of edges and DUs exercised by the test case.

[1] TSL is a compiler whose input includes a specification of the functional characteristics of the software to be tested, together with a description of the runtime testing environment. The tool's output is an executable test script of the test cases.

[2] The reason for requiring 30 different test cases for each coverage unit is to ensure that the results are not biased by the ability of a small number of test cases to detect faults. The sample size of 30 ensures that any observed correlation between a coverage unit and fault detection has reasonable statistical significance.

Generating Test Sets

To measure the fault detecting ability of test sets at different levels of Edge and DU coverage, we generated for each criterion approximately 5000 test sets for each faulty program from the program's test pool. We realized that it is generally not possible to generate test sets that exactly achieve a specified coverage level. Therefore, we used the following strategy to indirectly obtain test sets with a variety of coverage levels.

To generate a test set of desired size N for a given program, we applied the test generation and test pruning methods in parallel.[3] The test cases were randomly selected out of the test pool. If a selected test case increased the coverage achieved by the previously selected tests on the program, it was added to the test set. The set was considered complete as soon as either its size reached N or its coverage reached 100% (since no test case can increase coverage after that). In some cases, therefore, the resulting test set size was smaller than the desired test set size. In order to generate test sets with a wide variety of coverage levels, the desired sizes were chosen randomly from the integers 1,2,...,R, where R was determined for each program by trial and error as a number slightly larger than the size of the largest test set reaching 100% coverage.

4.2.3 Comparison with the Experimental Design of Frankl and Weiss

The design of our experiment is somewhat similar to that of the Frankl and Weiss experiment [6] comparing dataflow- and controlflow-based adequacy criteria. Below, we briefly describe their experiment and compare it with ours.

Frankl and Weiss used the ASSET system to study nine Pascal programs, each with a single existing error. For each program, they first generated a large set of test cases called the universe. Each test case was executed, its output was checked for correctness, and the program path it exercised was recorded by ASSET. In the evaluation phase of the study, the recorded information was used to determine each test set's edge coverage and def-use coverage.

Test sets of a chosen size S were built by randomly selecting S test cases from the universe. Each test set's all-uses coverage percentage was calculated, and it was recorded whether or not one or more test cases in the set detected the fault. The size S was chosen such that significant numbers of both all-uses adequate and all-edges adequate test sets would be chosen. On average, the all-uses adequate test sets were larger in size than the all-edges adequate test sets. The largest of the programs considered had 74 executable edges and 106 executable uses.

[3] The resulting test sets are the same regardless of whether the test generation and test pruning methods are applied in sequence or in parallel.

Thus, the Frankl and Weiss study differed from our study in the following aspects:

- small Pascal programs vs. moderate-size C programs
  ASSET system vs. Tactic system
  existing faults vs. seeded faults
  very few faults vs. relatively large number of faults,
- no test set pruning vs. removal of test cases that do not improve coverage,
- all-uses vs. all-DUs, and
- different methods used for generating the test pool (universe).

4.3 Subject Faulty Programs

Base Programs

The base programs were chosen to meet special criteria. To allow creation of a reasonable test pool, they must have an understandable specification. Because each program must be understood by several people (to seed faults, to create tests, and to examine the code for infeasible def-use associations and edges), they must not be overly complex. But they also have to be large and complex enough to be considered realistic, and to permit the seeding of many hard-to-find errors. Each program must be compilable and executable as a stand-alone unit. The programs used for the experiment are C programs obtained from various sources, ranging in length from 141 to 512 lines of code. Table 1 shows the number of lines of code (excluding blanks and comments) in each of the base programs, the number of executable edges and DUs, and a brief description of each program.

Table 1: Base Programs

Program	LOC	Executable		Description
		Edges	DUs	
replace	512	191	664	pattern replace
tcas	141	46	57	altitude separation
usl.123	472	97	268	lexical analyzer
usl.128	399	159	240	lexical analyzer
schedule1	292	62	294	priority scheduler
schedule2	301	80	217	priority scheduler
tot_info	440	83	292	information measure

Seeding Faults

We created faulty versions of each base program by seeding individual faults into the code. The faults are mostly changes to single lines of code, but a few involve multiple changes. Many of the faults take the form of simple mutations or missing code. Creating N faulty versions from the same base program has significant benefits: the understanding gained from studying the code applies to all N versions, and the work involved in generating the test pools applies to all the versions. Perhaps most significant, the existence of the (presumed

Table 2: Faulty Versions and Test Pools

Base Program	Number of faulty versions	Test Pool (TP)			Range of failure ratios in the test pool
		Initial Tests (ITP)	Additional Tests (ATP)	Final Size (ITP + ATP)	
replace	32	79%	21%	5548	.0005-.056
tcas	39	65%	35%	1562	.0006-.084
usl.123	7	99%	1%	4092	.0007-.056
usl.128	10	99%	1%	4076	.0079-.086
schedule1	9	90%	10%	2637	.0027-.100
schedule2	10	77%	23%	2666	.0008-.024
tot_info	23	64%	36%	1067	.0019-.159

correct) base version supplies us with an oracle to check the results of test cases executed on the faulty versions.

To produce meaningful results, we had to place certain requirements on the seeded faults. The faults had to be neither too easy nor too hard to detect. If they were too hard, then all fault detection ratios would have been essentially zero, and no visible differences among the techniques would have been observable. If they were too easy, then almost any test set would have detected them, and there would have been no measurable effect of the coverage to observe. We set a lower bound of 3 detecting test cases, and an upper bound of 350 for each faulty program. The program **usl.128**, for instance, with a test pool size of 4076 cases, had 10 faulty versions; the hardest fault was detected by 32 test cases (detection ratio .0079), and the easiest by 350 (detection ratio .086). More than 55 of the originally seeded faults were not included in the study because of very low detection and more than 113 were not included because of very high detection. The 130 faults included in the study were created by 10 different people, mostly without knowledge of each other's work; their goal was to be as realistic as possible, by introducing faults that reflected their experience with real programs. For each base program, Table 2 gives the number of faulty versions, the composition and sizes of the test pool, and the range of failure ratios of the test pool over the faulty versions of the program.

5 Data Analysis

The basic data collected for the experiments was the fault detecting ability of the test sets generated for each faulty program. Since we wanted to see how fault detection varied as coverage levels increased towards 100%, the test generation procedure was designed to produce a wide range both of test set sizes and coverage percentages, specifically to produce at least 30 test sets for each 2% coverage interval for each program.

The resulting data allowed us to examine the relationships among the coverage level, size, and fault detection attributes of the test sets produced by applying each of the testing strategies to each faulty program. Figure 2 shows an example of the graphs we used to study these relationships for each individual faulty program. The coverage graph shows the relationship between fault detection and the coverage levels of test sets, and the size graph shows the relationship between fault detection and the sizes of test sets.

The horizontal axis of the coverage graph is divided into 2% wide intervals (e.g., 91-93%) ending at 99%, and the rightmost interval which is 1% wide (99-100%). Each plotted point represents the fault detection ratio of all the test sets whose coverage is within an interval. The fault detection ratio for a given interval is *m/n*, where *n* is the total number of test sets whose coverage percent is in the interval, and *m* is the number of these sets that contain a fault-detecting test case. The fault detection ratio for each interval is plotted against the midpoint of the interval. The graph shows two plots, one for DU coverage and one for Edge coverage. In the example, .75 of the test sets with a 97-99% DU coverage level and .43 of the test sets with a 97-99% Edge coverage level detected the fault.

The horizontal axis of the size graph is divided into size intervals of width 2. For each interval and for each coverage type, the fault detection ratio is defined as *m/n*, where *n* is the total number of test sets of that coverage type in the size interval, and *m* is the number of these sets that contain at least one test case that detects the fault in the program. The fault detection ratios for each interval are plotted against the midpoint of the interval. We also analytically computed the fault detection ratio for test sets of size *s* that are randomly chosen from the program's test pool. This function is referred to as F_{random}, and is shown on the size graph together with the two plots for the coverage-based strategies. This makes it possible to investigate the role of test set size in determining fault detection. In the example of Figure 2, test sets for the interval 22-23 had the following fault detection ratios:

random sets (of size 23):	.42
Edge-based sets:	.48
DU-based sets:	.70

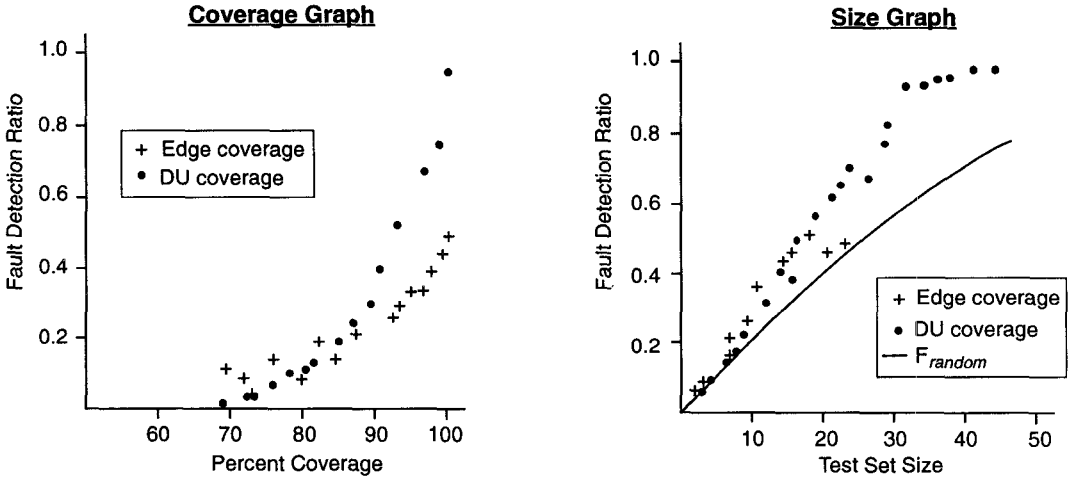

Figure 2: Fault Detection Ratios for One Faulty Program

In the example of Figure 2, DU-based test sets at coverage levels from 90-100% clearly outperform Edge-based test sets, and in addition DU-based test sets of size 10 or greater outperform random test sets. In general, however, the performance of both types of coverage varied widely. Therefore, we attempted to classify each faulty program according to the method that seemed most effective in detecting its fault. To do this, we first defined six relations *DU>Edge*, *Edge>DU*, *DU>Random*, *Edge>Random*, *Random>DU* and *Random>Edge* as described below.

For each faulty program, we fitted second order, least squares curves to the coverage and size plots for each strategy. The curves fitted to the coverage plots are FC_{DU} and FC_{edge}, and the curves fitted to the size plots are FS_{DU} and FS_{edge}.

For each fault, we say that ***DU>Edge*** if $FC_{DU}(100\%)$ is greater than $FC_{edge}(100\%)$, and the difference is larger than the standard deviation of the difference between the measured fault detection ratios and their least squares approximations [9]. A similar definition applies for ***Edge>DU***, with "Edge" and "DU" interchanged.

For a given test set size s, the notation $F_{random}(s)$ is the probability that a randomly chosen set of s test cases from the test pool contains at least one fault-detecting test case, i.e., $F_{random}(s)$ is the expected fault detection ratio of random test sets of size s. To avoid bias in favor of the coverage strategies, F_{random} is always calculated from the test pool with the higher failure ratio, either TP or ITP. Let d be the size of the largest test set generated for DU coverage, and let Max_{DU} be the maximum value of $FS_{DU}(s)$ for the sizes (1,...,d). Similarly, let e be the size of the largest test set generated for Edge coverage, and let Max_{edge} be the maximum value of $FS_{edge}(s)$ for the sizes (1,...,e). We say that ***DU>Random*** if $Max_{DU}>F_{random}(d)$, and ***Edge>Random*** if $Max_{edge}>F_{random}(e)$, and these differences satisfy a standard deviation property similar to that for *DU>Edge* and *Edge>DU*. Similarly, we say that ***DU<Random*** if $Max_{DU}<F_{random}(d)$ and ***Edge<Random*** if $Max_{edge}<F_{random}(e)$.

For 24 faults, all detection ratios were so low that it did not make sense to compute the above relations, hence we refer to them as *low detection faults*. Specifically, a fault is a low detection fault if its detection ratios were lower

Table 3: Classification of Faults

Class	Characteristics	Number of faults	Fault Detection Ratio at 100% coverage min, avg, max
DU	DU > Edge and DU > Random	31	.19, .67, 1.0
Edge	Edge > DU and Edge > Random	25	.17, .57, .99
DU-&-Edge	DU > Random and Edge > Random and not (DU > Edge or Edge > DU)	32	.14, .59, 1.0
Coverage Total	DU > Random or Edge > Random	88	-
Non-Coverage	DU < Random and Edge < Random	9	-
Other	cannot classify	9	-

than .125 at the highest achieved coverage with both coverage methods, as well as with random test sets as large as the largest coverage-based sets. The other 106 faults are called *high detection faults*. Table 3 gives a classification of all the high detection faults based on the truth values of the above relations. The table also gives the variation in fault detection at 100% coverage level in cases for which either of the coverage criterion is better than random. Nine faults could not be placed into any of the first five categories because they did not satisfy any of the needed combinations of relations. For example, DU and Edge might have had approximately equal fault detection ratios at 100%, while neither was significantly better than Random. These nine faults are put in the "Other" category.

6 Observations

Fault Classification

Since the faults in our study are not necessarily representative of the kinds of faults found in practice, we resist the temptation to make any general inferences based on the relative frequencies of the different fault classes. Clearly the majority of faults in our study lend themselves better to detection by test cases based on one or the other coverage-based method, although both coverage methods performed noticeably worse than random test selection for some faults. We were not able to discern any characteristics of the faults, either syntactic or semantic, that seem to correlate with higher detection by either method.

Limitation of Coverage as an Adequacy Criterion

The fault detection ratio at 100% coverage varied significantly across different faults in the same fault class. For example, the fault detection ratio of test sets with 100% DU coverage varied from .19 to 1.0 with an average of .67 for the 31 faults in the DU class. These and similar numbers for Edge coverage show that high coverage levels alone do not guarantee fault detection. We conclude that by itself, 100% coverage, either edge or def-use based, is *not* an indication that testing has been adequate. Rather, code coverage seems to be a good indicator of *test inadequacy*. If apparently thorough tests yield only a low coverage level, there is good reason to continue testing and try to raise the coverage level. The value of doing this can be seen by examining the detection ratios of test sets as their coverage levels approach 100%.

Detection Behavior in the 90-100% Coverage Range

For most faults, the detection ratio of test sets increases markedly as their coverage increases. This is especially noticeable as the coverage increases from 90% to 100%. Figure 2 shows an example of the increase in effectiveness when the last 10% of coverage is achieved. At 90% DU coverage, the detection ratio is 0.4, while at 100% coverage, the ratio has increased to 0.95. For Edge coverage, the increase is less dramatic, but the ratio still rises from 0.33 to 0.48. Tables 4 and 5 show the average fault detection ratios over the high detection faults for the five intervals in the 91-100% range.

Table 4: DU Coverage vs. Random Selection

% DU Coverage	91-93%	93-95%	95-97%	97-99%	99-100%
average size of DU coverage test sets	7.9	9.1	11.3	14.2	17.4
average fault detection ratio of DU coverage test sets	.20	.25	.33	.42	.51
average % superiority in fault detection of DU coverage test sets over same size random test sets	1%	14%	33%	52%	68%
average % increase in the size of random test sets required to yield the same fault detection as the DU coverage test sets	*	21%	46%	79%	160%

* The observed difference is not statistically significant (less than 95% confidence).

Table 5: Edge Coverage vs. Random Selection

% Edge Coverage	91-93%	93-95%	95-97%	97-99%	99-100%
average size of Edge coverage test sets	7.6	8.5	9.7	11.2	12.6
average fault detection ratio of Edge coverage test sets	.28	.31	.35	.41	.46
average % superiority in fault detection of Edge coverage test sets over same size random test sets	40%	48%	50%	68%	75%
average % increase in the size of random test sets required to yield the same fault detection as the Edge coverage test sets	51%	64%	77%	112%	163%

Table 6: DU Coverage vs. Edge Coverage

% Coverage	95-97%	97-99%	99-100%
average % difference in size of DU coverage test sets over Edge coverage test sets	1%	9%	21%
average % difference in fault detection of DU coverage test sets over Edge coverage test sets	*	*	38%

*The observed difference is not statistically significant (less than 95% confidence).

The Size Factor

Higher coverage test sets typically are larger than lower coverage sets. To investigate the influence of test set size on fault detection, we asked the following questions:

- For a specific coverage interval, what is the percentage superiority in fault detection of coverage-based test sets over that of same size random test sets?
- For a specific coverage interval, what percentage increase in the size of a random test set is needed to produce the same fault detection as the coverage-based test sets?

Table 4 addresses the above questions for test sets based on DU coverage in the 91-100% range. The figures clearly indicate the benefit of selecting test sets from the test pool using DU coverage rather than random selection. Table 5 presents similar results for test sets based on Edge coverage in the 91-100% range. The averages in the tables are computed over the 106 high detection faults.

DU vs. Edge Coverage

Achieving a given DU coverage level generally requires larger test sets than the same Edge coverage level. To examine whether the DU coverage test sets had any significant benefit over Edge coverage sets, we computed the average percentage difference in size and fault detection between the DU coverage test sets and the Edge coverage test sets for the high detection faults. Table 6 shows these numbers for the three coverage intervals in the 95-100% range. In this range, the sizes of DU coverage sets were greater than the Edge coverage sets with >99% confidence. Correspondingly, the DU coverage sets had better fault detection, although with lower confidence. We would have liked to analyze the data further to investigate whether or not the better fault detection of the DU coverage sets was primarily due to larger test sets. However, the results of such an analysis would not be meaningful since the differences in fault detection are not statistically significant. From these results we conclude that there is no clear winner between the two coverage criteria.

Fault Detection of the Test Pool Parts

Our two-phase test pool generation procedure allows us to compare the fault detection ability of the initial test pools against that of the coverage-enhanced test pools. Recall that the additional test cases were added to the initial test pools to increase coverage and ensure that each executable coverage unit is exercised by at least 30 different test cases. For more than half the faulty programs, these additional test cases were much more successful at detecting faults than the test cases of the initial test pool. Expressing the percentage of detecting tests in the ATP as a multiple of the percentage of detecting tests in the ITP, we found the factor to vary from 0 to 480. For 73 of the 130 faults, the ATP was more than twice as successful in fault detection as the ITP. Table 7 shows the number of faults with this factor in several ranges.

Table 7: Relative Fault Detection by Initial and Additional tests

det (*ATP*) / det (*ITP*)	Number of faults
0 - 1.0	22
1.01 - 2.0	17
2.01 - 10.0	36
10.1 - 480	37
det (*ITP*) = 0	18

The numbers demonstrate the value of using the coverage criteria to motivate the creation of additional test cases.

7 Conclusions

We have carried out an experimental study investigating the effectiveness of dataflow- and controlflow-based test adequacy criteria. The all-edges and all-DUs coverage criteria were applied to 130 faulty program versions derived from seven moderate size base programs by seeding realistic faults. For each faulty program, several thousand test sets were generated and the relationship between fault detection and coverage was examined.

Our results show that both controlflow and dataflow testing are useful supplements to traditional specification-based and informal code-based methods. The frequently higher detection rates achieved by the coverage-based tests that were added to the initial test pool show that the criteria can be very useful at instigating the generation of high-yield test cases that may be omitted otherwise.

On the other hand, achieving 100% coverage is not necessarily a good indication that the testing is adequate, as shown by the large number of low detection faults, as well as by the wide variation in the fault detection ratios at 100% coverage.

The analysis of detection ratios as coverage increases shows that 100% coverage, although not a guarantee of fault detection, is much more valuable than 90 or 95%.

Finally, we saw an approximately equal split between faults that were detected at higher ratios by Edge coverage and by DU coverage, leading us to conclude that the two methods, according to the way they are measured by Tactic, are frequently complementary in their effectiveness.

Acknowledgements

Surya Kasu, Yu Lu, and Evelyn Duesterwald performed exhaustive (and sometimes exhausting) analysis of the subject programs using Tactic. Maryam Shahraray gave us advice with the statistical analysis of our experimental data. Elaine Weyuker provided insightful advice and comments on the experimental design and interpretation of the results.

We are grateful to Arun Lakohtia of the University of Southern Louisiana, for contributing two of our subject programs.

References

1 M. J. Balcer, W. Hasling, and T.J. Ostrand, "Automatic Generation of Test Scripts from Formal Test Specifications", Proc. of the Third Symposium on Testing, Analysis, and Verification, ACM Press, New York, (1989), 210-218.

2 V. Basili and R. Selby, "Comparing the Effectiveness of Software Testing Strategies", IEEE Trans. Softw. Eng. SE-13, (December 1987).

3 B. Beizer, Software Testing Techniques, 2nd ed., Van Nostrand Reinhold, New York, 1990.

4 L. Clarke, A. Podgurski, D. Richardson, and S. Zeil, "A Formal Evaluation of Data Flow Path Selection Criteria", IEEE Trans. Softw. Eng. SE-15, (November 1989).

5 L. Foreman and S. Zweben, "A Study of the Effectiveness of Control and Data Flow Testing Strategies", J. Systems Software 21 (1993), 215-228.

6 P.G. Frankl and S.N. Weiss, "An Experimental Comparison of the Effectiveness of the All-uses and All-edges Adequacy Criteria", Proc. of the Fourth Symposium on Testing, Analysis, and Verification, ACM Press, New York, (1991) 154-164.

7 P.G. Frankl, S.N. Weiss, and E.J. Weyuker, "ASSET: A System to Select and Evaluate Tests", Proc. IEEE Conf. Software Tools, New York, (April 1985).

8 P.G. Frankl and E.J. Weyuker, "A Dataflow Testing Tool", Proc. IEEE Softfair II, San Francisco, (December 1985).

9 C. R. Hicks, Fundamental Concepts in the Design of Experiments, Saunders College Publishing, Fort Worth, 1982.

10 J.R. Horgan and S. London, "Data Flow Coverage and the C Language", Proc. of the Fourth Symposium on Testing, Analysis, and Verification, ACM Press, New York, (1991) 87-97.

11 W.E. Howden, "Reliability of the Path Analysis Testing Strategy", IEEE Trans. Softw. Eng. SE-2, (July 1976).

12 J.C. Huang, "An Approach to Program Testing", ACM Comp. Surveys 7 (Sept. 1975), 113-128.

13 B. W. Kernighan and P. J. Plauger, Software Tools in Pascal, Addison-Wesley, Reading, MA, 1981.

14 D. Knuth, "The Errors of TeX", Software: Practice and Experience 19, (1989), 607-685.

15 J. Laski, "Data Flow Testing in STAD", J. Systems Software 12 (1990), 3-14.

16 J.W. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy", IEEE Trans. Softw. Eng. SE-9, no 3, (May 1983), 347-354.

17 N. B. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements Specification for Process-Control Systems", University of California at Irvine, CA, Information and Computer Science Technical Report, 1992.

18 S. Ntafos, "An Evaluation of Required Element Testing Strategies", Proc. Seventh Int. Conf. Software Engineering, (March 1984), 250-256.

19 T. J. Ostrand and M. J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests", Communications of the ACM 31, 6 (June 1988), 676-686.

20 T. J. Ostrand and E. J. Weyuker, "Data Flow-Based Adequacy Analysis for Languages with Pointers", Proc. of the Fourth Symposium on Testing, Analysis, and Verification, ACM Press, New York, (1991) 74-86.

21 H. D. Pande, B. G. Ryder, and W. A. Landi, "Interprocedural Def-Use Associations in C Programs", Proc. of the Fourth Symposium on Testing, Analysis, and Verification, ACM Press, New York, (1991), 139-153.

22 S. Rapps and E.J. Weyuker, "Data Flow Analysis Techniques for Test Data Selection", Proc. Sixth Int. Conf. Software Engineering, Tokyo, (September 1982).

23 S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information", IEEE Trans. Softw. Eng. SE-11, (April 1985).

24 H. Schiller, "Using MEMMAP to measure the extent of program testing", Report TR 1836, IBM Systems Development Division, Poughkeepsie, NY, (Feb. 1969).

25 P. Thevenod-Fosse, H. Waeselynck, Y. Crouzet, "An Experimental Study on Software Structural Testing: Deterministic Versus Random Input Generation", Proc. Twenty-First International Symposium on Fault-Tolerant Computing, Montreal, Canada, (June 1991), 410- 417.