

An Experimental Evaluation of Data Flow and Mutation Testing

A. JEFFERSON OFFUTT[†]

Department of ISSE - 4A4, George Mason University, Fairfax, VA 22030 U.S.A., ofut@isse.gmu.edu

AND

JIE PAN[†]

PRC, 12005 Sunrise Valley Drive, Reston, VA 22091 U.S.A., pan_jenny@prc.com

AND

Kanupriya Tewary

*Oracle Corporation, Box 659311, 500 Oracle Parkway, Redwood Shores, CA 94065 U.S.A.,
ktewary@us.oracle.com*

AND

Tong Zhang

SRA Corporation, 2000 15th Street North, Arlington, VA 22201 U.S.A.

SUMMARY

This paper presents two experimental comparisons of data flow and mutation testing. These two techniques are widely considered to be effective for unit-level software testing, but can only be analytically compared to a limited extent. We compare the techniques by evaluating the effectiveness of test data developed for each. For a number of programs, we develop ten independent sets of test data; five to satisfy the mutation criterion, and five to satisfy the all-uses data flow criterion. These test sets are developed using automated tools, in a manner consistent with the way a test engineer might be expected to generate test data in practice. We use these test sets in two separate experiments. First we measure the effectiveness of the test data that was developed for one technique in terms of the other technique. Second, we investigate the ability of the test sets to find faults. We place a number of faults into each of our subject programs, and measure the number of faults that are detected by the test sets. Our results indicate that while both techniques are effective, mutation-adequate test sets are closer to satisfying the data flow criterion, and detect more faults.

KEY WORDS Software testing Data flow Mutation Experimentation

[†]Partially supported by the National Science Foundation under grant CCR-93-11967.

INTRODUCTION

Mutation testing and data flow testing are two unit testing techniques that have recently matured enough to be used by industrial software developers. Both techniques are thought to provide a higher level of testing than older techniques such as statement and branch coverage (for example, mutation and data flow subsume statement and branch coverage), but are also more costly to apply, and require automation. Many engineering advances have been made in the past few years to support mutation and data flow, and commercial tools are now available (PiSCES for mutation and ATAC for data flow), which are currently being used in practical situations. Unfortunately, the relative merits of these techniques are still not well understood. Test engineers and test managers need objective, factual studies such as this to make well-informed decisions about testing. In this paper, we try to establish some idea of the practical cost to benefit tradeoffs between mutation and data flow testing, based on experience with the techniques.

Both techniques are white box in nature [1] and require large amounts of computational and human resources (although recent engineering advances are reducing both types of cost). Although experience has led us to believe there is significant overlap between the two techniques, they have not been successfully compared on either an analytical or experimental basis. We attempt the comparison using two experiments. First, we compare mutation and the all-uses data flow criterion to see whether either method covers the other in the sense of how close test data sets that satisfy one technique come to satisfying the other. Second, we compare mutation and all-uses by executing faulty versions of programs and comparing how many faults are found by test data sets that satisfy each technique.

Our results lead us to believe that while mutation offers more stringent testing than data flow does, both techniques provide benefits the other lacks. Our eventual goal is to find a way to test software that provides the advantages of both techniques, either by combining the two techniques or by deriving a new technique that offers the power of both mutation and data flow testing.

The remainder of this section includes a short discussion on the notion of test adequacy criteria, provides overviews of mutation and data flow testing and reviews related research. Subsequent sections present our analytical results, and discuss our experimental procedures and results. Details about the programs and faults we used can be found in a technical report [2].

Adequacy Criteria

There are two aspects of any testing process. The first is test data generation, which may be manual, automated, or a combination of both. The second aspect of a testing process is the stopping rule, or *adequacy* of the generated test data. Early researchers in mutation defined adequacy as follows: a test set is *adequate* if, for every fault in the program being tested, there is a test case in the test set that detects that fault [3, 4].

Budd and Angluin defined adequacy with respect to a given criterion [4], and Weyuker [5] extended this to define an *adequacy criterion* to be a predicate that is used to determine when the program has been tested enough.

By Frankl and Weyuker’s definition, a criterion C_1 *includes* another criterion C_2 if and only if for every program, any test set T that satisfies C_1 also satisfies C_2 [6]. This is similar to the definition of subsumption given by Clarke et al.: A criterion C_1 *subsumes* a criterion C_2 if and only if every set of execution paths P that satisfies C_1 also satisfies C_2 [7]. A more recent definition for analytically comparing two criteria is that of *properly covers* [8]. In this paper, we use Frankl and Weyuker’s definition of inclusion, but the more familiar and accepted term of subsumption.

Data Flow Testing

Rapps and Weyuker [9, 10] define a family of data flow path selection criteria and examine the relationships among them. A program unit P is considered to be an individual subprogram (main program, procedure, or function). A subprogram is decomposed into a set of *basic blocks*, which are maximal sequences of simple statements with one entry point such that if the first statement is executed, all statements in the block will be executed. The subprogram is represented by a *control flow graph (CFG)*, in which the nodes are basic blocks and the edges correspond to possible flow of control between basic blocks.

A *data definition* of a variable is a location where a value is stored into memory (assignment, input, etc.), and a *data use* is a location where the value of the variable is accessed. Uses are subdivided into two types: a *computation use (c-use)* directly affects a computation or is an output, and a *predicate use (p-use)* directly affects the flow of control. C-uses are considered to be on the nodes in the CFG and p-uses are on the edges. A *definition-clear subpath* for a variable X through the CFG is a sequence of nodes that do not contain a definition of X .

Frankl and Weyuker define seven data flow criteria. In this paper, we only consider all-uses, because empirical evidence shows that the all-uses criterion is effective and low in cost (in terms of the number of test cases), compared to the other data flow criteria [6, 11, 12]. *All-uses* requires that for each definition of a variable X in P , the set of paths executed by the test set T contains a definition-clear subpath from the definition to all reachable c-uses and p-uses of X . A *DU-pair* is a definition and a use of the same variable such that the use is reachable from the definition; we say that we satisfy all-uses by covering all DU-pairs.

One difficulty with applying data flow techniques is that of unexecutable subpaths. The definition-clear subpaths that are used in data flow testing are based on the CFG, which is a static representation of the program, and it may not be possible to execute all of the subpaths. Frankl and Weyuker [6] suggest modifications to the data flow criteria so that they satisfy the *applicability property*. An adequacy criterion C is *applicable* if and only if for every program P there exists at least one test set that is adequate for the criterion

and the program [5]. An applicable criterion excludes subpaths that cannot be executed. Unfortunately, it is undecidable whether a particular set of subpaths is executable, so recognition of unexecutable subpaths is typically done by hand.

Mutation Testing

Mutation is a fault-based testing technique introduced by DeMillo et al. [3] and Hamlet [13]. Mutation testing is based on the assumption that a program will be well tested if all simple faults are detected and removed. The coupling effect [3, 14] states that complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect most complex faults.

Simple faults are introduced into the program by *mutation operators*. Each change or *mutation* created by a mutation operator is encoded in a *mutant program*. A mutant is *killed* by a test case that causes it to produce incorrect output. A test case that kills a mutant is considered to be effective at finding faults in the program, and the mutants it kills are not executed against later test cases. *Equivalent mutants* are mutant programs that are functionally equivalent to the original program and therefore cannot be killed by any test case. Like unexecutable subpaths, determination of equivalent mutants is usually done by hand. The goal of mutation is to find test cases that kill all non-equivalent mutants; a test set that does so is adequate relative to mutation.

COMPARISON OF TESTING CRITERIA

Although there has been much informal discussion on the relative strengths of mutation and data flow testing, we know of only a few attempts to compare the two techniques. The first problem is how to make the comparison. Analytical comparisons show theoretical relationships between techniques, and are most satisfying because they allow claims that are true in all situations. Empirical comparisons show relations that are based on specific studies. Although it is difficult to show that empirical results hold in all situations, analytical comparisons cannot always be made, but empirical comparisons can.

Offutt and Voas [15] have analytically compared mutation with all-defs data flow testing. *All-defs* requires that each definition of a variable in the program reaches at least one use. They show that mutation subsumes all-defs, that is, any set of test cases that satisfies mutation will also satisfy all-defs. Unfortunately, there is no such analytical relationship between mutation and all-uses, thus comparisons have been made empirically.

We consider two relationships that have been defined elsewhere. Weyuker, Weiss, and Hamlet [16] suggest a relationship called `PROBBETTER`:

A testing criterion C_1 is **PROBBETTER** than C_2 for a program P if a randomly selected test set T that satisfies C_1 is more “likely” to detect a failure than a randomly selected test set that satisfies C_2 .

Mathur and Wong [17, 18] suggest a different relationship called **PROBSUBSUMES**:

A testing criterion C_1 **PROBSUBSUMES** C_2 for a program P if a test set T that is adequate with respect to C_1 is “likely” to be adequate with respect to C_2 . If C_1 **PROBSUBSUMES** C_2 , C_1 is said to be more “difficult” to satisfy than C_2 .

The **PROBBETTER** relation is defined with respect to the fault detection capability of test sets, whereas the **PROBSUBSUMES** relation is defined with respect to the difficulty of satisfying one criterion in terms of another. Both are probabilistic relations between two testing criteria and are defined in terms of specific programs. Although this means that it is difficult to draw general conclusions from any one study, as the number and variety of programs studied increases, our confidence in the validity of a **PROBSUBSUMES** or a **PROBBETTER** relationship with a larger set of programs also increases.

Mathur and Wong [17, 18] used the **PROBSUBSUMES** relation in experimental comparisons of all-uses data flow testing with mutation testing, by manually generating test data to satisfy both criteria and comparing the scores. They used 4 programs and 30 sets of test cases per program and detected equivalent mutants and unexecutable subpaths by hand. This study indicated that mutation-adequate test sets were closer to being data flow-adequate than data flow-adequate test sets were to being mutation-adequate. Their study was quite similar to ours, particularly insofar as we both compared mutation with all-uses data flow. We used different sets of programs and test cases, and different methods of generating the test cases. While we use more programs (10 rather than 4), Mathur and Wong generated more test sets per program (30 rather than 5). The major difference is that we have also measured the fault detection ability of the test case sets. Although no experiment can conclusively show that one test technique is better than another, more and different programs and studies can increase our confidence in the validity of such a conclusion.

Mathur and Wong’s study developed from a previous study by Mathur [19] that used human testers to compare the difficulty of satisfying another data flow criterion (all-DU paths [6]) and mutation. This study found that the mutation criterion was more difficult to satisfy than all-DU paths, but the study had several problems, as detailed by Mathur and Wong [17].

Recently, Tewary [20] devised algorithms for inserting faults into programs using the program dependence graph. These algorithms were demonstrated by inserting faults into programs and comparing the fault detection ability of mutation and data flow testing. She found that when the faults involved changes to the control dependence relations in the program dependence graph, the mutation-adequate and data flow-adequate test sets were almost equally effective in detecting the faults.

EXPERIMENTAL HYPOTHESES AND CONDUCT

We compare mutation and data flow in two different ways. First, it seems reasonable to suppose that if test sets created for one criterion also satisfy another, then the first criterion can in some sense be considered to be “better” than the second. This is the essence of the `PROBSUBSUMES` relationship. Thus, we have tried to determine if mutation-adequate test sets always cover data flow, and vice versa. Second, an independent and perhaps more practically useful question is whether tests sets created for a testing technique will actually find faults in programs. This is the essence of the `PROBBETTER` relationship.

For our comparison, we have formulated the following hypotheses:

- Hypothesis 1:** Mutation testing `PROBSUBSUMES` all-uses data flow.
- Hypothesis 2:** All-uses data flow testing `PROBSUBSUMES` mutation.
- Hypothesis 3:** All-uses data flow is testing `PROBBETTER` than mutation.
- Hypothesis 4:** Mutation testing is `PROBBETTER` than all-uses data flow.

We chose 10 program units that cover a range of applications. These programs range in size from 10 to 29 executable statements, have from 183 to 3010 mutants, and have from 10 to 101 DU-pairs. The programs are described in Table 1. For each program, we give a short description and the number of executable Fortran statements. We also give the number of DU-pairs (both predicate and computation) and the number of infeasible DU-pairs, and the number of mutants and equivalent mutants. Because of the nature of the two techniques, programs typically have many more mutants than DU-pairs. There also tends to be a lot of overlap in the test cases in the sense that one test case will usually kill many mutants, and often cover several DU-pairs. The programs are listed in a technical report [2].

Program	Description	Statements	DU-pairs	Infeasible	Mutants	Equivalent
Bub	Bubble sort on an integer array	11	29	1	338	35
Cal	Days between two dates	29	28	0	3010	236
Euclid	Greatest common divisor (Euclid’s)	11	10	1	196	24
Find	Partitions an array	28	100	13	1022	75
Insert	Insertion sort on an integer array	14	29	1	460	46
Mid	Median of three integers	16	30	0	183	13
Pat	Pattern matching	17	55	3	513	61
Quad	Real roots of quadratic equation	10	15	0	359	31
Trityp	Classifies triangle types	28	101	14	951	109
Warshall	Transitive closure of a matrix	11	44	0	305	35

Table 1: **Experimental Programs.**

We used three tools for our experimentation. The Mothra mutation system automates the process of mutation testing by creating and executing mutants, managing test cases, and computing the mutation score. We used all twenty-two Mothra mutation operators [21] for this experiment. To generate test data to satisfy mutation, we used Godzilla, an automated constraint-based test case generator that is integrated with Mothra [22]. For the data flow analysis part of the experiment we used ATAC, a data flow tool for C programs developed by Bellcore [23]. ATAC implements all-uses by having the requirement that, if a

predicate uses the same variable in more than one condition, each condition must be evaluated separately. There is no test data generation tool associated with ATAC, thus we generated test data to satisfy all-uses by using a special-purpose random test data generator of our own devising. This tool repetitively generated test cases, keeping test cases that covered new DU-pairs, and throwing away test cases that did not. We feel that these methods of generating test data are realistic in the sense that if a coverage-based criterion is used, they are reasonable ways that test engineers might be expected to generate test data in practice.

Since Mothra tests Fortran-77 programs and ATAC tests C programs, we had to translate each program into both languages. We started with Fortran versions of the programs, and first made sure that the programs did not use any features of Fortran-77 that would not translate directly into C. Then we hand-translated the programs into C, taking care to use as direct a translation as possible so as not to introduce any bias into our results by using different programs. The control flow graphs, executable statements, and DU-pairs were exactly the same for all pairs of programs. We tested our translations by running both versions on every test case that we generated, and comparing the outputs of the two versions. As described below, this amounted to a total of 10 different test sets per program.

We define *test requirements* to be things that must be satisfied or covered; for example, reaching statements are test requirements for statement coverage, killing mutants are test requirements for mutation, and executing DU-pairs are test requirements for data flow testing. Both mutation and data flow have problems with unrealizable test requirements. Mutation systems create equivalent mutants, which cannot be killed, and data flow systems ask for infeasible DU-pairs to be covered. For each program, as part of our preparation, we identified all equivalent mutants and infeasible DU-pairs by hand.

For each program, we generated test sets that were mutation-adequate and test sets that were data flow-adequate. To avoid any bias that could be introduced by any particular test set, we generated five independent test sets for each criteria. Thus, for each program, we had ten test sets; five mutation-adequate test sets, and five data flow-adequate test sets, for a total of 100 test sets for our ten programs. We consider a *minimum* test case set for a criterion to contain the smallest number of cases necessary to satisfy the criterion, and a *minimal* test case set to be a satisfying set such that if any test case was removed, the set would no longer satisfy the criterion. We eliminated redundant test cases (by incrementally adding test cases, and only keeping those that contributed to satisfying the criteria) until we had minimal test sets, but did not attempt to create minimum test sets. The minimal test case sets are in the technical report [2].

EXPERIMENTS AND ANALYSIS

We compare the two techniques of all-uses and mutation testing on three bases, a coverage measurement (using the PROBSUBSUMES relationship), fault detection capability (using the PROBBETTER relationship),

and test set size. We discuss each of these three comparisons in the following subsections.

Coverage Measurement Experimentation

The method of comparison used in our first experiment was to generate test data sets that satisfied one criterion and then measure how close they came to satisfying the other criterion. We define *coverage* as the amount by which a test set that is adequate for a program with respect to criterion A satisfies criterion B . Thus coverage of criterion A by criterion B is 100% if and only if a test set that is adequate for criterion A is also adequate for criterion B . More formally, let A and B be two adequacy criteria, and $F_A(T, P)$ and $F_B(T, P)$ be the functions that measure whether a test set T for a program P is adequate for the criteria. Let T_A be a set of test data that is adequate with respect to criterion A and T_B be a set of test data that is adequate with respect to criterion B . Then the coverage of criterion A by criterion B is $F_A(T_B, P)$ and the coverage of criterion B by criterion A is $F_B(T_A, P)$. Since a criterion covers itself, $F_A(T_A, P) = 100\%$ and $F_B(T_B, P) = 100\%$.

Our coverage measure for mutation is the *mutation score*. If M_t is the total number of mutants generated for a program, M_k is the number of mutants killed by a set of test cases T , and M_q is the number of equivalent mutants for the program being tested, then the mutation score is:

$$MS(P, T) = \frac{M_k}{M_t - M_q}. \quad (1)$$

We define the *data flow score* of a test set as follows. If D_t is the total number of DU-pairs for the program being tested, D_s is the number of DU-pairs that have been satisfied by the test set and D_i is the number of DU-pairs that can never be satisfied because of unexecutable subpaths, then the data flow score is:

$$DFS(P, T) = \frac{D_s}{D_t - D_i}. \quad (2)$$

The mutation score of a test set that is data flow adequate will give us the coverage of mutation by data flow. Similarly, the data flow score of a test set that is mutation adequate will give us the coverage of data flow by mutation. For our experiment, if the mutation criterion is denoted by M and the data flow criterion is denoted by D , then F_M is a function that computes the mutation score for a set of test data using Equation 1 and F_D is a function that computes the data flow score for a set of test data using Equation 2. We compute values of $F_M(T_D, P)$ and $F_D(T_M, P)$ for each program in our sample set over several test case sets.

Coverage scores

We computed the coverage measurements by calculating the mutation scores of each of the five data

Program	Test Set 1	Test Set 2	Test Set 3	Test Set 4	Test Set 5	Mean
Bub	94.06	95.38	98.02	97.03	98.02	96.50
Cal	72.89	80.54	72.96	67.91	70.00	72.86
Euclid	94.15	95.32	97.66	97.08	95.32	95.51
Find	94.40	94.72	90.18	94.30	94.83	93.69
Insert	98.55	98.55	98.55	98.31	99.52	93.69
Mid	81.76	81.18	79.41	82.35	80.59	81.06
Pat	66.37	81.64	89.38	62.17	92.70	78.45
Quad	89.94	89.33	90.24	89.94	89.63	89.82
Trityp	84.44	85.63	83.73	83.97	83.49	84.25
Warshall	99.26	94.81	99.26	88.89	94.81	95.41

Table 2: $F_M(T_D)$:Mutation Scores of Data Flow-Adequate Test Sets

flow-adequate test sets and the data flow scores of each of the five mutation-adequate test sets as described above. The programs were run on a Sun 4 SPARC workstation running SunOS version 4.1.1. The mutation scores of the data flow-adequate test sets are shown in Table 2, and the data flow scores of the mutation-adequate test sets are shown in Table 3. The scores for each of the five test sets are shown, as well as the average (mean) mutation and data flow scores.

Program	Test Set 1	Test Set 2	Test Set 3	Test Set 4	Test Set 5	Mean
Bub	100.00	100.00	100.00	100.00	100.00	100.00
Cal	100.00	100.00	100.00	100.00	100.00	100.00
Euclid	100.00	100.00	100.00	100.00	100.00	100.00
Find	100.00	98.85	98.85	98.85	98.85	99.08
Insert	96.43	96.43	96.43	96.43	92.86	95.71
Mid	100.00	100.00	100.00	100.00	100.00	100.00
Pat	100.00	98.08	100.00	100.00	100.00	99.62
Quad	100.00	100.00	100.00	100.00	100.00	100.00
Trityp	100.00	100.00	100.00	100.00	100.00	100.00
Warshall	95.45	95.45	95.45	95.45	95.45	95.45

Table 3: $F_D(T_M)$:Data Flow Scores for Mutation-Adequate Test Sets

All the data flow scores are very high, and most of the mutation scores are very high. In neither case, however, can we conclude that the test sets are adequate for the other criterion. Thus, we cannot say that either hypothesis 1 or 2 is correct, but we do find more evidence to support hypothesis 1 than hypothesis 2. The mean mutation scores of the data flow-adequate test sets is 88.66, and the mean data flow scores of the mutation-adequate test sets is 98.99. The mutation-adequate tests cover all but one or two DU-pairs for all ten programs. Thus, it does appear that by satisfying mutation, we have in some sense come close to satisfying data flow, but we do not know the benefits of “almost” satisfying a testing criterion. We also could not find a pattern among the mutants not killed by the data flow-adequate test sets, so we see no way to make a general statement about what might be missing in such test sets.

Fault Detection Experimentation

To further assess the relative merits of the testing techniques, we inserted several faults into each of the programs, and evaluated the test sets based on the number of faults detected by them. So as to avoid any bias, we introduced faults according to the following considerations:

1. faults must not be equivalent to mutants; otherwise the mutation-adequate test data would by definition detect them,
2. faults should not be N-order mutants (else the coupling effect would indicate that mutation-based test cases should find the fault, biasing our results in favor of mutation),
3. the faults should not have a high failure rate, or the detection becomes trivial.

A general outline of our fault creation procedure is that for each program statement, we attempted to:

1. create multiple related transpositions of variables (e.g., substituting one variable for another throughout, or exchanging the use of two variables),
2. modify multiple, related, arithmetic or relational operators,
3. change precedence of operation (i.e., by changing parenthesis),
4. delete a conditional or iterative clause,
5. change conditional expressions by adding extra conditions,
6. change the initial values and stop conditions of iteration variables.

The changes were only applied when a change did not violate one of our considerations. For the most part, these resulted in faults that appear to be realistic in the sense that they look like mistakes that programmers typically make. None of the faults were found by all test cases. Additionally, neither criterion seemed biased towards any of our fault types in the sense that the criterion always found faults of that type. The actual faults are listed in the technical report [2].

To gather the results, we inserted each fault separately, creating N incorrect versions of each program. This allowed us to always know which fault a test case detected when the faulty program failed. The data are shown in Table 4. The Mutation column gives the percentage of faults detected by the mutation-adequate test cases, averaged over the five sets of data for each program, and the Data Flow column gives the percentage of faults detected by the data flow-adequate test cases, averaged over the five sets of data for each program. The mutation sets detected all the faults for six of our ten programs, and the least percentage of faults detected was 67% for **Find**. The data flow sets detected all the faults for two programs, and as few as 15% for one program (**Insert**). On average, the mutation sets detected 92% of the faults, versus 76% of the faults for the data flow sets. Thus, we conclude that our data support hypothesis 3, but not hypothesis 4.

Program	Faults	Mutation %	Data Flow %
Bub	5	100	92
Cal	10	98	56
Euclid	6	83	83
Find	6	67	47
Insert	4	75	15
Mid	5	100	100
Pat	6	100	87
Quad	6	100	100
Trityp	7	100	86
Warshall	5	100	92
TOTALS/MEAN	60	92	76

Table 4: Percentage of Faults Found by Mutation-Adequate and Data Flow-Adequate Test Data

Test Set Size

Table 5 gives the mean number of test cases for the mutation-adequate test sets and the data flow-adequate test sets for each program. The most obvious observation is that in most cases, mutation requires many more test cases than data flow does. Weyuker [24] discusses comparing the costs of testing criteria based on the number of test cases. With the ability to automatically generate test data, this cost is somewhat less important during initial testing, although the cost of examining the outputs still makes the number of test cases a factor. Additionally, the number of test cases is still important during regression testing.

Program	Mutation Adequate	Data Flow Adequate
Bub	6.6	1.4
Cal	36.0	6.2
Euclid	4.0	1.0
Find	14.0	6.2
Insert	3.8	3.0
Mid	24.6	6.0
Pat	26.4	5.8
Quad	13.4	2.0
Trityp	51.4	14.0
Warshall	4.8	3.2

Table 5: Mean Number of Test Cases Per Set

CONCLUSIONS

In this paper, we have presented results from three empirical studies. First, we compared mutation with all-uses data flow on the basis of a “cross scoring”, where tests generated for each criterion are measured against the other. Second, we measured the fault detection of test data generated for each criterion, and

compared the results. Third, we compared the two techniques on the basis of the number of test cases generated to satisfy them, in a rough attempt to compare their relative costs.

For our programs, the mutation scores for the data flow-adequate test sets are reasonably high, with an average coverage of mutation by data flow of 88.66%. While this implies that a program tested with the all-uses data flow criterion has been tested to a level close to mutation-adequate, it may still have to be tested further to obtain the testing strength afforded by mutation.

However, the mutation-adequate test sets come very close to covering the data flow criterion. The average coverage of data flow by mutation is 98.99% for our ten programs. We can infer that a program that has been completely tested with mutation analysis will usually be very close to having been tested to the all-uses data flow criterion – within one or two DU-pairs of being complete. This provides some evidence that mutation **PROBSUBSUMES** all-uses data flow. On the other hand, mutation required more test cases in almost every case than data flow testing did, providing a cost to benefit tradeoff between the two techniques.

These conclusions are supported by the faults that the test sets detected. Although both mutation-adequate and data flow-adequate test sets detected significant percentages of the faults, the mutation-adequate test sets detected an average of 16% more faults than the data flow-adequate test sets. The difference was as high as 60% for one program. This provides some evidence that mutation is **PROBBETTER** than all-uses data flow.

Of course, these experiments have limitations that are difficult to avoid in this area. To do certain kinds of statistical analyses of data, we must be able to assume that the data is based on “representative samples” from the population being studied. Unfortunately, we have no way to choose a representative sample of software, test cases, or faults, so we are limited in our ability to use statistical analysis tools to make claims of significance.

The programs we studied are also relatively small, which leaves the question of how the conclusions might scale up to large software systems. Although we cannot know without further study, it is worth observing that both mutation and data flow testing are expected to be used for unit testing, not integration or system testing. That is, the techniques are typically applied to subroutines and functions individually, and largely independently of other subroutines and functions. Although there is ongoing research in applying data flow intra-procedurally, this research has not yet been put into practical use.

A positive aspect is that our results are in general agreement with those of Mathur and Wong [17]. Both studies found that mutation offers more coverage than data flow, but at a higher cost. Although their study did not include any fault detection, we also found that mutation-adequate test sets detected more faults, which is in general agreement with both of our other results. The fact that both studies, performed at about the same time by different researchers using different programs and test cases, got similar results, greatly strengthens both conclusions.

Although the ability to automatically generate test data means that requiring large numbers of test cases is not as expensive as if generated manually (although the cost of the verifying the outputs and recognizing equivalent mutants or infeasible DU-pairs will still be present), smaller test sets will still save effort during regression testing. If our results can be considered to be applicable to all programs, as well as the programs we investigated, then it seems that mutation offers more coverage, but at a higher cost, a tradeoff that practical testers must consider when choosing a test methodology.

ACKNOWLEDGMENTS

It is a pleasure to acknowledge Bellcore, and specifically Bob Horgan and Saul London, for the use of the testing tool ATAC and for support using it. We also thank the anonymous referees for many helpful suggestions.

References

- [1] L. J. White. Software testing and verification. In Marshall C. Yovits, editor, *Advances in Computers*, volume 26, pages 335–390. Academic Press, Inc, 1987.
- [2] A. Jefferson Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang. Experiments with data flow and mutation testing. Technical report ISSE-TR-94-105, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, 1994.
- [3] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.
- [4] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, November 1982.
- [5] E. J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, 12:1128–1138, December 1986.
- [6] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.
- [7] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A comparison of data flow path selection criteria. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 244–251, London UK, August 1985. IEEE Computer Society.
- [8] P. G. Frankl and E. J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19(3):202–213, March 1993.
- [9] S. Rapps and E. J. Weyuker. Data flow analysis techniques for test data selection. In *Software Engineering 6th International Conference*. IEEE Computer Society Press, 1982.
- [10] S. Rapps and W. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

- [11] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.
- [12] E. J. Weyuker. How good is data flow testing? Technical report, NYU, 1989.
- [13] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4), July 1977.
- [14] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.
- [15] A. J. Offutt and J. M. Voas. Subsumption of condition coverage techniques by mutation testing. Submitted for publication, 1994.
- [16] E. J. Weyuker, S. N. Weiss, and R. G. Hamlet. Comparison of program testing strategies. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 1–10, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.
- [17] A. P. Mathur and W. E. Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *The Journal of Software Testing, Verification, and Reliability*, 4(1):9–31, March 1994.
- [18] Weichen Eric Wong. *On Mutation and Data Flow*. PhD thesis, Purdue University, December 1993. (also Technical Report SERC-TR-149-P, Software Engineering Research Center, Purdue University, West Lafayette, IN).
- [19] A. P. Mathur. On the relative strengths of data flow and mutation based test adequacy criteria. In *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*, Portland OR, 1991. Lawrence and Craig.
- [20] K. Tewary and M. J. Harrold. Fault modeling using the program dependence graph. In *Proceedings of the Fifth International Symposium on Software Reliability Engineering*, pages 126–135, Monterey CA, November 1994.
- [21] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software—Practice and Experience*, 21(7):685–718, July 1991.
- [22] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.
- [23] J. R. Horgan and S. London. ATAC: A data flow coverage testing tool for C. In *Proceedings of the Symposium of Quality Software Development Tools*, pages 2–10, New Orleans LA, May 1992.
- [24] E. J. Weyuker. The cost of data flow testing: An empirical study. *IEEE Transactions on Software Engineering*, 16(2):121–128, February 1990.