Empirical Comparisons of Data Flow and Mutation Testing

A. Jefferson Offutt

Department of ISSE
George Mason University
Fairfax, VA 22030
phone: 703-993-1654
email: ofut@gmuvax2.gmu.edu

Kanupriya Tewary

Department of Computer Science
Clemson University
Clemson, SC 29643-1906
kanu@cs.clemson.edu

December 1992

Abstract

Data flow and mutation testing are two powerful white box testing techniques for unit-level software testing. Unfortunately, they cannot be completely compared on an analytical basis, for example, mutation is incomparable on an inclusion basis with most data flow criteria. This paper shows that mutation includes the All-defs data flow criterion, but is incomparable with other data flow criteria, and presents results from two empirical comparisons of mutation with the All-uses data flow criterion. For the first comparison, we define a *coverage* measure that is used for the comparison. The coverage of data flow by mutation is between 99% and 100%, which means that test data that satisfied the mutation criterion also satisfied the data flow criterion in almost all cases. The coverage of mutation by data flow was between 80 and 90%. For the second comparison, we use test data that satisfy the two criteria to detect faults, and compare the criteria on the basis of the faults found. The empirical evidence strongly indicates that while data flow-adequate test sets are close to being mutation-adequate, mutation-adequate test sets are almost invariably data flow-adequate.

1 INTRODUCTION

Mutation testing and data flow testing are two powerful unit testing techniques that have not been successfully compared on an analytical basis. Extensive research has been done to develop both techniques, and although substantial technical problems remain to be solved for them to be used in practical situations, both techniques offer substantial potential for improving the testing process, resulting in higher quality software. Both techniques are white box in nature [Whi87] and require substantial computational resources. 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 in three steps. First we show that mutation *includes* one version of data flow testing (All-defs). Unfortunately, the All-defs criterion is rather low on the data flow inclusion hierarchy as given by Rapps and Weyuker [RW85],

so this relationship is only of marginal value. Second, we compare mutation and the All-uses data flow criterion on an experimental basis, to see whether either method covers the other in the sense of how close test data that satisfies one technique comes to satisfying the other. Third, we compare mutation and All-uses to see whether one demonstrates more fault detection ability than the other by executing faulty versions of programs and comparing how many faults are found by test data that satisfies each technique.

Our results lead us to believe that while mutation offers more stringent testing that data flow, 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 an overview of mutation and data flow testing and reviews related research. Subsequent sections present our analytical results, and discuss our experimental procedures and results.

1.1 Adequacy Criteria

There are two aspects of any testing method. The first is test data generation, which may be manual, automated, or a combination of both. The second aspect of a testing method is the stopping rule, or *adequacy* of the generated test data. The original definition of adequacy stated that 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 [DLS78, BA82]. Budd and Angluin defined adequacy with respect to a given criterion [BA82], and Frankl and Weyuker [FW88] extended this to define an *adequacy criterion* to be a predicate that is used to determine when the program has been tested enough [FW88]. If a set of test data is adequate with respect to a certain testing criterion and the program executes correctly on that test data, we are confident that our testing is complete.

1.2 Data Flow Testing

Rapps and Weyuker [RW82, RW85] define a family of data flow path selection criteria and examine the relationships between them. Frankl and Weyuker [FW88] extend these definitions to apply to a large subset of Pascal, and modify the criteria so that they satisfy the *applicability property*. 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 the 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 2 types: a *computation use* (*c-use*) 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 does not contain a definition of X.

Frankl and Weyuker define seven data flow criteria. *All-defs* requires that for each definition of a variable X in P, the set of paths Π executed by the test set T contains a definition-clear subpath from the definition to at least one c-use or one p-use of X. *All-c-uses* requires that for each definition of X in P, and each c-use of X reachable from the definition, Π contains a definition-clear subpath from the definition to all reachable c-uses of X. *All-p-uses* requires that for each definition of X in P, and each p-use of X reachable from the definition, Π contains a definition-clear subpath from the definition to all reachable p-uses of X. *All-c-uses/some-p-uses* requires that for each definition of X in P, if there exists at least one c-use of X reachable from the definition, Π contains a definition-clear subpath from the definition to at all reachable c-uses of X, otherwise, Π contains a definition-clear subpath from the definition to a reachable p-use of X. *All-p-uses/some-c-uses* requires that for each definition of X in P, if there exists at least one p-use of X reachable from the definition, Π contains a definition-clear subpath from the definition to all reachable p-uses of X, otherwise, Π contains a definition-clear subpath from the definition to a reachable c-use of X. *All-uses* requires that for each definition of X in P, Π contains a definition-clear subpath from the definition to all reachable c-uses p-uses of X. *All-du-paths* requires that for each definition of X in P, Π contains all definition-clear subpaths from the definition to all reachable c-uses of X and all reachable successors of a p-use of X, such that each subpath contains no loops, or is one complete loop. *All-paths* requires that all paths through the program be executed.

These testing criteria are all comparable based on the notion of *inclusion*, which has been defined by Rapps and Weyuker as a measure of relative strength [RW85]. A criterion C_1 *includes* another criterion C_2 iff for every program, any test set T that satisfies C_1 also satisfies C_2 [FW88]. In this hierarchy, All-paths includes All-du-paths, All-du-paths includes All-uses, which in turn includes all other techniques. In this paper, we show a relationship between mutation and All-defs, and present experimental results from a comparison of mutation with All-uses. For this experimentation, we follow Frankl and Weiss [FW91] and use the All-uses criterion, which provides a data flow criterion that is intermediate in the inclusion hierarchy.

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 [FW88] 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 some test set that is adequate for the criteria and the program [Wey86]. An applicable criteria 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.

1.3 Mutation Testing

Mutation testing is a fault based testing technique introduced by DeMillo et al. [DLS78]. 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 [DLS78, Off92] 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* produced by a mutation operator produces a mutant program or simply, a *mutant*. A mutant is *killed* by a test case that forces 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 mutant(s) 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.

1.4 Review of Related Work

Although there has been much informal discussion on the relative strengths of mutation and data flow testing, we know of only two attempts to compare the two techniques. Budd compared mutation with data flow testing on an intuitive basis [Bud81]. He suggested that mutation is a stronger testing technique because it makes erroneous data flow possibilities emerge as non-equivalent mutants. An attempt to kill these mutants forces the data flow criterion to be satisfied. No theoretical or experimental evidence was provided to support these arguments.

A later study was done by Mathur [Mat91]. He conducted an experimental comparison of All-du-pairs

with mutation testing, by having students in a class generate test data by hand to satisfy both criteria and compare the scores. They used one set of test cases per program and did not record equivalent mutants and unexecutable subpaths. These experiments indicated that mutation-adequate test data was closer to being data flow-adequate than data flow-adequate test data was to being mutation-adequate.

2 INCLUSION RELATIONSHIP

By Frankl and Weyuker's definition, a criterion C_1 *includes* another criterion C_2 iff for every program, any test set T that satisfies C_1 also satisfies C_2 [FW88]. This is similar to the definition of subsumption given by Clarke et al.: A criterion C_1 *subsumes* a criterion C_2 iff every set of execution paths P that satisfies C_1 also satisfies C_2 [CPRZ85]. Here, we will use the later definition of inclusion.

The All-defs data flow criterion requires that each definition of a variable reach at least one use. Although mutation testing does not explicitly have this requirement, the requirement is met implicitly through the *svr* mutation operator. The *sdl* operator (statement deletion) deletes each statement in the program. To show inclusion, we restrict our attention to only statements that contain variable definitions. Assume that M_1 is an *sdl* mutation that deletes statement S_i with a definition of a variable X. To kill M_1, a test case t must 1) cause the mutated statement to be reached (reachability), 2) cause the execution state of the program after execution of S_i to be incorrect (necessity), and 3) cause the final output of the program to be incorrect (sufficiency) [DO91]. For this mutant, the necessity condition is trivial, since by deleting the statement, X is no longer assigned the value. For the final output of the mutant to be incorrect, there are two cases. First, if X is an output variable, t must have caused an execution of a subpath from the deleted definition of X to the output without an intervening definition. Since the output is considered a use, this satisfies the criterion. Second, if X is not an output variable, then the nondefinition of X at S_i must result in an incorrect output state. This is only possible if X is used at some later point during execution without being redefined. Thus, t will satisfy the All-defs criterion for the definition of X at S_i, and the *sdl* mutation operator ensures that mutation subsumes All-defs.

3 COMPARISON MEASURES

The method of comparison used in this experiment was to generate test data that satisfied one criterion and then measure how close it came to satisfying the other criterion. We define *coverage* as the amount by which test data that is adequate with respect to criterion A satisfies criterion B. Thus coverage of criterion A by

criterion B is 100% if and only if test data that is adequate for criterion A is also adequate for criterion B. More formally, let A and B be two adequacy criteria, and $F_A(T)$ and $F_B(T)$ be the functions that measure whether a test set T 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 $F_A(T_B)$ and the coverage of criterion B by criterion A is $F_B(T_A)$. Since a criterion covers itself, $F_A(T_A) = 100\%$ and $F_B(T_B) = 100\%$.

In our experiment, we compare the adequacy criteria for mutation and data flow testing. As indicated in the previous section, a test set is mutation adequate if it succeeds in killing all non-equivalent mutants. Our coverage measure for mutation is the *mutation score*, which is defined as follows. Let M_t be the total number of mutants generated for a program, M_k be the number of mutants killed by a set of test cases T, and M_q be the number of equivalent mutants for the program being tested. Then the mutation score $MS(P,T)$ of test set T for program P is:

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

We define a similar measure for the data flow adequacy of a test set. A test set is data flow adequate for the All-uses criterion if it executes all DU-pairs. We define the *data flow score* of a test set as follows. Let D_t be the total number of DU-pairs for the program being tested, D_s be the number of DU-pairs that have been satisfied by the test set and let D_i be the number of DU-pairs that can never be satisfied due to the presence of unexecutable subpaths in the program. Then the data flow score $DFS(P,T)$ of test set T for program P can be computed as follows:

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

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 $F_M(T_D)$ and $F_D(T_M)$ for each program in our sample set over several test case sets.

4 EXPERIMENTAL PROCEDURE

This experiment was conducted on five subroutines. Since both mutation and data flow testing are primarily unit testing techniques, we felt that this experiment should (at least initially) be performed at the unit level. This study also involved a significant amount of hand-analysis (determining equivalent mutants and unexecutable subpaths), which would be difficult for larger, integrated subsystems. Four of these programs were square root calculation programs. `bisect` calculates the square root of a number using the method of interval bisection, `newton` uses Newton's method, `secant` uses the modified linear interpolation method and `regula` uses the regula falsi method [GW85]. The square root programs were chosen because their C-language versions were available and could therefore be used with the data flow testing tool. The fifth was the classic `trityp`, which inputs three integers that represent the relative lengths of the sides of a triangle and classifies the triangle as equilateral, isosceles, scalene or illegal.

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. Mothra uses twenty-two different mutation operators [KO91] (listed in Appendix A). All first order mutants were enabled 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 [DO91]. For the data flow analysis part of the experiment we used Combat, a data flow tool for C programs developed at Clemson University [HK92]. Combat is a compiler based tester for unit testing of C procedures on Sun-4 machines. The data flow criteria supported are All-Nodes, All-Edges, All-Defs and All-Uses. There is no test data generation tool associated with Combat and all test data must be generated manually.

The experiment was carried out in the following steps:

Step 1 Preparation of sample programs
Step 2 Determination of equivalent mutants
Step 3 Determination of unexecutable subpaths
Step 4 Generation of mutation adequate test sets
Step 5 Generation of data flow adequate test sets
Step 6 Computation of mutation score for each data flow adequate test set
Step 7 Computation of data flow score for each mutation adequate test set
Step 8 Analysis of results

4.1 Program Preparation

Since Combat tests C programs and Mothra tests Fortran-77 programs, we needed two versions of each test program. The first step in program preparation was to translate each of the square root programs (which were in C) to Fortran. The translations had to be as direct as possible, and whenever structural modifications had to be made to the program for the translation, we first modified the C version before translating. For example, some of the C programs used the trinary operator (`?`), which does not exist in Fortran, so the C versions were rewritten to eliminate use of this operator.

4.2 Determination of Equivalent Mutants and Unexecutable Subpaths

The determination of equivalent mutants was by far the most time consuming step in the experiment. The four square root programs used real numbered variables. and each program required two inputs, the number whose square root was to be determined and the precision. We used Godzilla to generate test cases to kill as many mutants as possible for each program (Godzilla was able to kill between 80.8% and 86.8% of the mutants), then used hand analysis to either determine that the rest were equivalent, or created additional test cases that would kill them.

The determination of unexecutable subpaths in the C versions of the test programs was carried out using Combat. Combat provides a control flow graph of the program being tested and this aids in the manual generation of test cases as well as the inspection of the program to identify infeasible or unexecutable paths. In addition to the control flow graph provided, complete coverage information of a test case can be obtained after each run. None of the square root programs had any unexecutable subpaths. The number of executable statements, the total and equivalent number of mutants, and the total and unexecutable number of DU-Pairs for these programs are shown in Table 1.

PROGRAM	STATEMENTS	DU-PAIRS	UNEXECUTABLE DU-PAIRS	MUTANTS	EQUIVALENT MUTANTS
bisect	22	83	0	511	62
newton	14	62	0	394	48
secant	17	78	0	694	84
regula	19	105	0	661	52
trityp	28	178	24	951	109

Table 1: Experimental Programs

4.3 Generation of Mutation and Data Flow Adequate Test Sets

To avoid any bias that could be introduced by a particular set of test cases, we generated 5 separate test sets for each program and both criteria. The mutation-adequate test sets were generated automatically using Godzilla and when necessary, augmented by hand, and the data flow-adequate test sets were generated by hand. We consider a *minimum* test case set for a criterion to contain the smallest number of tests 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 until we had minimal test sets, but did not attempt to create minimum sized test sets.

The `regula` program required a huge amount of computation time for one particular test case that eventually killed a large number of mutants. The test case took a little over 240 clock hours to execute due to the nature of the iterative loop in `regula` and the fact that the input error margin (epsilon value) for this test case was extremely small. Due to time constraints we could not repeat the application of this particular test case in all five trials. For this reason we generated only one 100% mutation adequate test set for `regula`. For the other four test sets we first generated as many test cases as possible using Godzilla (the same procedure as for the other experimental programs) and then for the adequacy calculations added the "strong" test case mentioned above, that killed the rest of the mutants.

Table 2 gives the average 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 does data flow. Weyuker [Wey90] 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 less important during initial testing, but may become still be important during regression testing.

Program	Mutation Adequate	Data Flow Adequate
bisect	26.6	5.8
newton	26.0	5.4
secant	25.5	4.4
regula	23.0	23.8
trityp	48.2	23.2

Table 2: Average Number of Test Cases Per Set

Program	Test Set 1	Test Set 2	Test Set 3	Test Set 4	Test Set 5	Average
bisect	92.20	92.20	92.20	87.08	91.09	90.95
newton	75.72	81.79	81.79	89.88	81.50	82.14
secant	79.34	78.50	80.66	80.00	83.27	80.35
regula	87.52	86.86	88.01	88.01	86.04	87.29
trityp	90.86	88.24	91.33	88.24	86.10	88.95

Table 3: $F_M(T_D)$:Mutation Scores for Data Flow Adequate Test Sets

4.4 Coverage Measurements

We computed the coverage measurements by calculating the mutation scores of each of the 5 data flow-adequate test sets and the data flow scores of each of the 5 mutation-adequate test sets. 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 3. The scores for each of the five test sets are shown, as well as the average mutation scores.

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 data flow scores. The mutation adequate test sets were data flow adequate for all except the `regula` program. For three out of five trial runs for this program, the same DU-pair was not satisfied.

Program	Test Set 1	Test Set 2	Test Set 3	Test Set 4	Test Set 5	Average
bisect	100.00	100.00	100.00	100.00	100.00	100.00
newton	100.00	100.00	100.00	100.00	100.00	100.00
secant	100.00	100.00	100.00	100.00	100.00	100.00
regula	100.00	99.04	99.04	99.04	100.00	99.42
trityp	100.00	100.00	100.00	100.00	100.00	100.00

Table 4: $F_D(T_M)$:Data Flow Scores for Mutation Adequate Test Sets

In addition to the information provided by the number of mutants killed by the data flow adequate test sets, it is interesting to note the mutants left alive. Table 5 shows the percentage of live mutants of each type after the application of data flow adequate test sets. The mutation operators used by Mothra are listed in Appendix A and are described in detail by Offutt and King [KO91]. 12 of the 22 operators used by Mothra are represented in Table 5; 9 of the other 10 operators generated no mutants for these programs, and the other operator generated only 12 mutants for `trityp` that were easily killed.

Unfortunately, the mutants left alive in Table 5 do not allow any generalizations about the type of mutants that will not be handled by data flow adequate test sets. They are quite evenly distributed over

	aor	abs	crp	csr	lcr	ror	san	scr	sdl	src	svr	uoi	SUM
bisect	4.7	10.8	31.3	8.0	28.6	17.6	0.0	2.4	8.3	9.1	10.9	21.8	153.5
newton	12.3	7.8	48.0	6.1	22.8	27.1	6.6	11.0	17.5	13.3	24.4	38.6	235.5
secant	9.0	30.5	63.6	15.1	38.1	25.7	0.0	14.7	2.0	66.0	12.8	23.7	301.2
regula	22.0	33.8	0.0	0.0	7.1	11.4	0.0	5.6	0.0	0.0	10.2	19.3	109.4
trityp	10.0	2.3	12.8	13.8	14.2	11.2	0.0	0.0	0.0	0.0	7.1	11.5	82.9
AVG	11.6	17.0	31.1	8.6	22.2	18.6	1.3	6.7	5.6	17.7	13.1	23.0	176.5

Table 5: Profile of Live Mutants for Data Flow Adequate Test Sets

all operator classes, indicating that the extra coverage afforded by mutation is evenly distributed over the kinds of faults detected by each mutation operator.

5 A FAULT DETECTION EXPERIMENT

To further asses the relative merits of the testing techniques, we inserted several faults into each of the programs, and evaluated the test sets from section 4 based on the number of faults detected by the test sets. So as to avoid any bias, we introduced faults according to the following two considerations:

1. faults must not be equivalent to mutants; otherwise the mutation-adequate test data would by definition detect them,
2. 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 transpositions of variables,
2. modify more than one arithmetic or relational operator,
3. change precedence of operation (i.e., by changing parenthesis),
4. delete a conditional or iterative clause.

The changes were only applied when a change did not violate one of the considerations above.

To gather the results, we inserted the faults 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 6. The Mutation column gives the number of faults detected by the mutation-adequate test cases, averaged over the 5 sets of data for each program, and The Data Flow column gives the number of faults detected by the data flow-adequate test cases, averaged over the 5 sets of data for each program. All the mutation sets detected all the faults. Although the data flow-adequate test sets did not detect all faults, they did, on average, detect 15.8 of 17, or 93% of the total faults.

Program	Faults	Mutation	Data Flow
bisect	5	5.0	5.0
newton	3	3.0	3.0
secant	2	2.0	1.6
regula	3	3.0	3.0
trityp	4	4.0	3.2
TOTALS	17	17.0	15.8

Table 6: Number of Faults Found by Data Flow-Adequate and Mutation-Adequate Test Data

6 CONCLUSIONS

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 85.96%. 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.

The mutation adequate test data however, comes very close to covering the data flow criterion. The average coverage of data flow by mutation is 99.88% for our five programs. We can infer that a program that has been completely tested with mutation analysis methods will usually be very close to have been tested to the All-uses data flow criterion.

These conclusions are supported by the faults that the test sets detected. Although the mutation-adequate test sets detected more faults than did the data flow-adequate test sets, the difference was small.

The only program for which the mutation adequate test sets were not data flow adequate was `regula`. Table 2 shows that the average number of test cases required for the data flow adequate test set for `regula` is 23.8, which is slightly more than the average number of test cases required by the mutation adequate test sets. This is not true for any other program in the sample set and may be an explanation for why the mutation-adequate test cases were not data flow-adequate for `regula`. Indeed, this may indicate that a principal reason why the data flow coverage for mutation test sets is higher than the inverse is simply because mutation requires more test cases. Further evidence for this would have been given if the coverage of data flow by mutation for `regula` was lower than for the other programs, but that value is 87.29, which is in the middle for our five programs.

Although the ability to automatically generate test data (such as with Godzilla for mutation, or a tool using similar techniques for data flow) means that requiring larger numbers of test cases is not as expensive as if generated manually, 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 must be considered when choosing a test methodology.

7 ACKNOWLEDGMENTS

Thanks to P. Kolte and Dr. Mary Jean Harrold for the use of Combat and the Kolte-set of square root programs.

A APPENDIX

The complete set of mutation operators used by the Mothra mutation system is shown in Table 7. The mutation operators used by Mothra were derived from studies of programmer errors and either correspond to simple errors that programmers typically make or enforce common testing heuristics (such as execute every statement). This particular set of mutation operators represents more than ten years of refinement through several mutation systems. The operators in this set not only require that the test data meet statement and branch coverage criteria, extremal values criteria, and domain perturbation, but also directly model many types of errors. Each of the 22 mutation operators is represented to by the three-letter acronym given on the left. For example, the "*a*rray reference for *a*rray reference *r*eplacement" (*aar*) mutation operator causes each array reference in a program to be replaced by each other distinct array reference in the program.

Type	Description
aar	array reference for array reference replacement
abs	absolute value insertion
acr	array reference for constant replacement
aor	arithmetic operator replacement
asr	array reference for scalar variable replacement
car	constant for array reference replacement
cnr	comparable array name replacement
crp	constant replacement
csr	constant for scalar variable replacement
der	DO statement end replacement
dsa	DATA statement alterations
glr	GOTO label replacement
lcr	logical connector replacement
ror	relational operator replacement
rsr	RETURN statement replacement
san	statement analysis (replacement by TRAP)
sar	scalar variable for array reference replacement
scr	scalar for constant replacement
sdl	statement deletion
src	source constant replacement
svr	scalar variable replacement
uoi	unary operator insertion

Table 7:*Mothra Mutation Operators*

References

[BA82] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, November 1982.

[Bud81] T. A. Budd. Mutation analysis: Ideas, examples, problems, and prospects in computer program testing. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 129–148. North-Holland, 1981.

[CPRZ85] 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.

[DLS78] 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.

[DO91] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[FW88] 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.

[FW91] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 154–164, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.

[GW85] C. F. Gerald and P. O. Wheatley. *Applied Numerical Analysis*. Addison-Wesley Publishing Company Inc., 3rd edition, 1985.

[HK92] M. J. Harrold and P. Kolte. Combat: A compiler based data flow testing system. In *Proceedings of the Tenth Annual Pacific Northwest Software Quality Conference*, pages 311–323, Portland OR, October 1992. Lawrence and Craig.

[KO91] 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.

[Mat91] Aditya P. Mathur. On the relative strengths of data flow and mutation based test adequacy criteria. Technical report SERC-TR-94-P, Software Engineering Research Center, Purdue University, West Lafayette IN, March 1991.

[Off92] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.

[RW82] 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.

[RW85] 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.

[Wey86] E. J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, 12:1128–1138, December 1986.

[Wey90] E. J. Weyuker. The cost of data flow testing: An empirical study. *IEEE Transactions on Software Engineering*, 16(2):121–128, February 1990.

[Whi87] L. J. White. Software testing and verification. In Marshall C. Yovits, editor, *Advances in Computers*, volume 26, pages 335–390. Academic Press, Inc, 1987.