An Experimental Determination of Sufficient Mutation Operators

A. Jefferson Offutt
Ammei Lee
George Mason University
and
Gregg Rothermel
Roland Untch
Clemson University
and
Christian Zapf
Siemens Corporation

*** Draft – Submitted ***

Mutation testing is a technique for unit testing software that, although powerful, is computationally expensive. The principal expense of mutation is that many variants of the test program, called mutants, must be repeatedly executed. Selective mutation is a way to reduce the cost of mutation testing by reducing the number of mutants that must be executed. This paper reports experimental results that compare selective mutation testing to standard, or non-selective, mutation testing. The results support the hypothesis that selective mutation is almost as strong as non-selective mutation; in experimental trials selective mutation provides almost the same coverage as non-selective mutation, with a four-fold or more reduction in cost.

---

1 Introduction

Mutation testing is a technique, originally proposed by DeMillo et al. [DLS78] and Hamlet [Ham77], that requires a person testing a program to demonstrate that the program does not contain a finite, well-specified set of faults. The tester does this by finding test cases that cause faulty versions of the program to fail, and either getting correct output from the test program (demonstrating its quality) or causing the test program to fail (detecting a fault).

Offutt partially supported by the National Science Foundation under grant CCR-93-11967. Authors' addresses: A. J. Offutt and A. Lee, Department of Information and Software Systems Engineering, George Mason University, Fairfax, VA 22030, {ofut,alee}@isse.gmu.edu; G. Rothermel and R. Untch, Department of Computer Science, Clemson University, Clemson, SC 29634-1906, {grother,untch}@cs.clemson.edu; C. Zapf, Siemens AG – Medical Group, BNES 13, Henkestrasse 127, 91052 Erlangen, Germany, zapf@erlh.siemens.de.

Unit level testing techniques such as mutation, hold great promise for improving the quality of software. Unfortunately, most of these techniques are currently so expensive that we cannot afford to use them. In a previous paper [ORZ93], we gave a preliminary definition and evaluation of a mutation approximation method called *selective mutation*. There, we reported results that saved, on average, over 50% of the execution cost of mutation, with negligible loss of effectiveness by one measure. Since then, we have extended selective mutation and have gotten results that, on average, save over 75% of the execution cost on our experimental programs, and more on larger programs. More importantly from a theoretical point of view, our mechanism reduces the cost of mutation by a factor of the size of the program's data space.

In the following sections, we describe mutation testing, analyze its cost, and define the method of selective mutation. We then present experimental results to validate the effectiveness of selective mutation, and finally, discuss possible future directions.

1.1 Mutation Testing Overview

Mutation testing helps a user to create test data by interacting with the user to iteratively strengthen the quality of test data. During mutation testing, faults are introduced into a program by creating many versions of the program, each of which contains one fault. Test data are used to execute these faulty programs with the goal of causing each faulty program to fail. Hence we use the term mutation; faulty programs are *mutants* of the original, and a mutant is *killed* when a test case causes it to fail. When this happens, the mutant is considered *dead* and no longer needs to remain in the testing process since the faults represented by that mutant have been detected, and more importantly, it has satisfied its requirement of identifying a useful test case.

Figure 1 contains a small Fortran function with three mutated lines (preceded by the Δ symbol). Note that each of the mutated statements represents a separate program. The most recent mutation system, Mothra [DGK$^+$88, KO91], uses 22 mutation operators to test Fortran-77 programs. These operators have been developed and refined over 10 years through several mutation systems. The

```
        FUNCTION Min (I,J)
1       Min = I
    Δ   Min = J
2       IF (J .LT. I) Min = J
    Δ   IF (J .GT. I) Min = J
    Δ   IF (J .LT. Min) Min = J
3       RETURN
```

Figure 1: **Function Min.**

mutation operators are limited to simple changes on the basis of the *coupling effect*, which says 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 [DLS78]. The coupling effect has been supported experimentally [Off92] and theoretically [Mor84].

The mutation testing process begins with the construction of mutants of a test program. The user then adds test cases (generated manually or automatically) to the mutation system and checks the output of the program on each test case to see if it is correct. If the output is incorrect, a fault has been found and the program must be modified and the process restarted. If the output is correct, that test case is executed against each live mutant. If the output of a mutant differs from that of the original program on the same test case, the mutant is assumed to be incorrect and it is killed.

After each new test case has been executed against each live mutant, each remaining mutant falls into one of two categories. One, the mutant is functionally *equivalent* to the original program. An equivalent mutant always produces the same output as the original program, so no test case can kill it. Two, the mutant is killable, but the set of test cases is insufficient to kill it. In this case, new test cases need to be created, and the process iterates until the test set is strong enough to satisfy the tester.

The *mutation score* for a set of test data is *the percentage of non-equivalent mutants killed by that data*. We call a test data set *mutation-adequate* if its mutation score is 100%.

1.2 The Cost of Mutation Testing

The major computational cost of mutation testing is incurred when running the mutant programs against the test cases. Budd [Bud80] analyzed the number of mutants generated for a program and found it to be roughly proportional to the product of the number of data references times the number of data objects. Typically, this is a large number for even small program units. For example, 44 mutants are generated for the function `Min` shown in Figure 1. Since each mutant must be executed against at least one, and potentially many, test cases, mutation testing requires large amounts of computation. We explore this cost in more detail in Section 4.

2 Selective Mutation Testing

One way to reduce the cost of mutation testing is to reduce the number of mutant programs created, using an approximation approach originally suggested by Mathur [Mat91]. To understand his proposal, we must first discuss the method by which mutant programs are generated.

The syntactic modifications responsible for mutant programs are determined by a set of *mutation operators*. This set is determined by the language of the program being tested, and the mutation system used for testing. Mothra uses 22 mutation operators that are derived from studies of programmer errors and induce simple syntax changes based on errors that programmers typically make. This particular set of mutation operators represents more than ten years of refinement through several mutation systems. Table 1 lists the operators; their detailed descriptions are elsewhere [KO91]. Each of the 22 mutation operators is represented by a three-letter acronym. 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. Budd's results about the number of mutants are based largely on the fact that the SVR mutation operator is the dominant operator.

Since mutation operators generate mutant programs at different rates, Mathur [Mat91] proposed

Mutation Operator	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
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 1: **Mothra Mutation Operators.**

selective mutation [1] as being mutation *without* applying the mutation operators that result in the most mutants. We show the distribution of mutants by operator for a set of 28 programs in Figure 2. Mathur suggested omitting the SVR and ASR operators, which are the two operators that result in the most mutations. In our previous paper [ORZ93], we called this 2-selective mutation, and extended the theory to *N-selective mutation*, which omits the *N* most prevalent operators. We presented results based on 2-selective, 4-selective, and 6-selective mutation.

Since the results from 6-selective mutation were so positive, we have extended selective mutation to try to experimentally determine the fewest number of operators that are required to effectively use mutation testing. The mutation operators used by Mothra can be divided into three general

[1] In his paper, Mathur used the term "constrained mutation". With his agreement, we have chosen the new term "selective mutation".

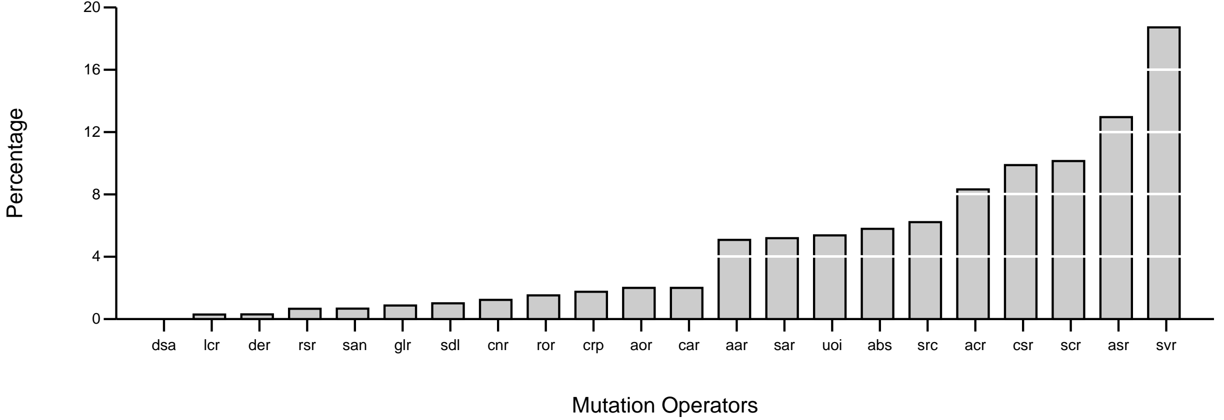

Figure 2: **Mutant Types Distribution for Mothra.**

categories based on the syntactic elements that they modify. *Operand replacement* operators replace each operand in a program with each other legal operand. Referring to Table 1, the operators AAR, ACR, ASR, CAR, CNR, CRP, CSR, SAR, SCR, SRC, and SVR perform operand replacement. *Expression modification* operators (ABS, AOR, LCR, ROR, UOI) modify expressions by replacing operators and inserting new operators. *Statement modification* operators (DER, DSA, GLR, RSR, SAN, SDL) modify entire statements. If the number of constants and variables in a program is $Vals$, the number of value references is $Refs$, and the number of source executable lines is $Lines$, then operand replacement operators result in $O(Vals * Refs)$ mutants, expression modification operators result in $O(Refs)$ mutants, and statement modification operators result in $O(Lines)$ mutants. We define *expression/statement-selective mutation (ES-selective)* to be mutation with only the expression and statement operators (not using the eleven operand replacement mutation operators), *replacement/statement-selective mutation (RS-selective)* to be mutation with only the replacement and statement mutation operators, and *replacement/expression-selective mutation (E-selective)* to be mutation with only the replacement and expression mutation operators. Finally, in Section 3.3, we define *expression-selective mutation (E-selective)* to be mutation with only the five expression modification mutation operators.

3 Experimentation with Selective Mutation

Our hypothesis is that selective mutation is almost as powerful as non-selective mutation, but significantly cheaper. Since test sets with mutation scores exceeding 95% are generally effective, we refine our hypothesis to be: test sets that kill all mutants under selective mutation will have over a 95% mutation score under non-selective mutation. Specifically, we hypothesize that ES-selective mutation, RS-selective mutation, and RE-selective mutation will lead to test sets that are over 95% mutation-adequate.

To evaluate our hypothesis, we compared selective mutation with non-selective mutation. To do this, we created test data sets that were selective mutation-adequate (achieving mutation scores of 100% when run against the selective mutants), and measured the mutation-adequacy of those test data sets. This section discusses the procedure used by, and the results of, those experiments.

3.1 Experimental Procedure

Ten Fortran-77 program units that cover a range of applications were chosen for the experiments. These programs range in size from 10 to 48 executable statements and had from 183 to 3010 mutants. The programs are described in Table 2.

Program	Description	Statements	Mutants	Equivalent
Banker	Deadlock avoidance algorithm	48	2765	43
Bub	Bubble sort on an integer array	11	338	35
Cal	Days between two dates	29	3010	236
Euclid	Greatest common divisor (Euclid's)	11	196	24
Find	Partitions an array	28	1022	75
Insert	Insertion sort on an integer array	14	460	46
Mid	Median of three integers	16	183	13
Quad	Real roots of quadratic equation	10	359	31
Trityp	Classifies triangle types	28	951	109
Warshall	Transitive closure of a matrix	11	305	35

Table 2: **Experimental Programs.**

We began by experimenting with ES-selective mutation. For each program, we first created the ES-selective mutants for the program (leaving out the AAR, ACR, ASR, CAR, CNR, CRP, CSR, SAR,

SCR, SRC, SVR operators). We then eliminated all equivalent mutants (previously determined by hand). Next, we used the automatic test data generator Godzilla [DO91] to generate test cases to kill as many non-equivalent mutants as possible, and generated additional test cases by hand when necessary. This process yielded test sets that were selective mutation-adequate. To eliminate any bias that could be introduced by one particular test set, we generated five sets of selective mutation-adequate test sets for each program and level of selective mutation. All our results are based on average scores for five test sets.

Next, for each program, we created all *non-selective* mutants (using *all* mutation operators) and eliminated all equivalent mutants. We then ran each set of selective mutation-adequate test cases on the non-selective mutants and computed the mutation score for these sets. This final score measures the effectiveness of selective mutation-adequate test sets on sets of non-selective mutants, and thus provides a measure of the relative effectiveness of selective mutation.

3.2 Experimental Results for Selective Mutation

Mutation scores for each of our programs, averaged over the 5 test sets, are shown in Tables 3 (ES-selective), 4 (RS-selective), and 5 (RE-selective). These data show that test sets that were 100% adequate for selective mutation were all almost 100% adequate for non-selective mutation. In fact, the mutation scores were above 98% for all but one program (Mid under RS-selective mutation).

Tables 6, 7, and 8 show the savings obtained by selective mutation in terms of the number of mutants. The "Percentage Saved" columns were computed by subtracting the number of selective mutants from the number of non-selective mutants and dividing the difference by the number of non-selective mutants (the percentage of mutants that did not have to be generated with selective mutation). Selective sets save from 6% to 72% of the mutants. Not surprisingly, the big gain is from ES-selective, which eliminates the operators that result in $O(Vals * Refs)$ mutants.

Varying the number of test cases used on each program had little affect on results. For example,

Program	Test Cases	Number of Live Mutants	Mutation Score
Bub	4.6	0.4	99.87
Cal	25.4	1.8	99.93
Euclid	3.0	1.2	99.30
Find	13.2	1.4	99.85
Insert	2.6	0.2	99.95
Mid	24.6	0.0	100.00
Quad	8.0	2.8	99.15
Trityp	40.8	6.2	99.26
Warshall	3.6	4.2	98.45
Banker	62.8	9.8	99.64
Average	13.9	2.0	99.54

Table 3: **Non-Selective Mutation Scores of ES-Selective Mutation Adequate Sets.**

Program	Test Cases	Number of Live Mutants	Mutation Score
Bub	3.0	6.0	98.02
Cal	38.6	6.0	99.78
Euclid	2.6	0.6	99.65
Find	13.4	8.8	99.07
Insert	1.8	5.6	98.65
Mid	6.0	32.8	80.71
Quad	7.0	1.8	99.45
Trityp	31.6	15.6	98.15
Warshall	4.2	0.0	100.00
Banker	90.0	9.2	99.66
Average	12.0	8.6	97.31

Table 4: **Non-Selective Mutation Scores of RS-Selective Mutation Adequate Sets.**

Program	Test Cases	Number of Live Mutants	Mutation Score
Bub	6.8	0.0	100.00
Cal	39.8	0.0	100.00
Euclid	3.6	0.0	100.00
Find	15.8	1.4	99.85
Insert	4.0	0.0	100.00
Mid	25.0	0.0	100.00
Quad	13.0	0.0	100.00
Trityp	47.0	0.0	100.00
Warshall	6.2	0.0	100.00
Banker	105.4	2.8	99.90
Average	17.91	0.2	99.97

Table 5: **Non-Selective Mutation Scores of RE-Selective Mutation Adequate Sets.**

Program	Non-Selective Mutants	Selective Mutants	Percentage Saved
Bub	338	147	56.51
Cal	3009	288	90.43
Euclid	195	115	41.03
Find	1022	422	58.71
Insert	460	197	57.17
Mid	183	133	27.32
Quad	359	190	47.08
Trityp	951	494	48.05
Warshall	305	115	62.30
Banker	2765	629	77.25
Total	9587	2730	71.52

Table 6: **Savings Obtained by ES-Selective Mutation.**

Program	Non-Selective Mutants	Selective Mutants	Percentage Saved
Bub	338	221	34.62
Cal	3009	2772	7.88
Euclid	195	106	45.64
Find	1022	765	25.15
Insert	460	316	31.30
Mid	183	68	62.84
Quad	359	185	48.47
Trityp	951	505	46.90
Warshall	305	217	28.85
Banker	2765	2281	17.50
Total	9587	7436	22.44

Table 7: **Savings Obtained by RS-Selective Mutation.**

Program	Non-Selective Mutants	Selective Mutants	Percentage Saved
Bub	338	308	8.88
Cal	3009	2958	1.69
Euclid	195	169	13.33
Find	1022	857	16.14
Insert	460	407	11.52
Mid	183	165	9.84
Quad	359	343	4.46
Trityp	951	903	5.05
Warshall	305	278	8.85
Banker	2765	2620	5.24
Total	9587	9008	6.04

Table 8: **Savings Obtained by RE-Selective Mutation.**

Program	Test Cases	Number of Live Mutants	Mutation Score
Bub	5.6	0.2	99.93
Cal	17.6	10.4	99.63
Euclid	3.0	1.2	99.30
Find	10.8	2.8	99.70
Insert	4.4	0.2	99.95
Mid	25.0	0.0	100.00
Quad	8.8	3.0	99.09
Trityp	42.0	5.4	99.36
Warshall	2.8	3.6	98.67
Banker	50.6	11.6	99.57
Average	13.3	2.9	99.51

Table 9: **Non-Selective Mutation Scores of E-Selective Mutation Adequate Sets.**

ES-selective mutation-adequate test sets for Find ranged from 10 to 17 test cases with the largest set increasing the (non-selective) mutation score by only .11% — the scores ranged from 99.89 to 100. In fact, the largest test set only killed 1 more mutant (of the 1022 that existed) than the smallest set. In other words, even the smallest selective mutation-adequate test sets were almost as effective as the largest such sets.

3.3 Combining ES- and RE-Selective Mutation

Since the mutation scores in Table 4 are lower than the scores in Tables 3 and 5, we assumed that the expression operators are the most powerful. Thus, we extended our experimentation to measure the effectiveness of combining ES- and RE-selective mutation. *Expression-selective (E-selective)* mutation is mutation omitting both the replacement and statement operators. This leaves only five Mothra operators. Table 9 give the mutation scores for E-selective mutation. All scores are above 98.5%, with an average over our 10 programs of 99.5%. This indicates that the five operators ABS, AOR, LCR, ROR, and UOI are the most important mutation operators, and probably the only operators necessary to ensure full mutation coverage.

Table 10 give the savings for E-selective mutation. These range from 37% to 92%, for an average of 78%. At the two extremes, Mid is a very small routine that has a relatively large number of

Program	Non-Selective Mutants	Selective Mutants	Percentage Saved
Bub	338	117	65.38
Cal	3009	237	92.12
Euclid	195	89	54.36
Find	1022	257	74.85
Insert	460	144	68.70
Mid	183	115	37.16
Quad	359	174	51.53
Trityp	951	446	53.10
Warshall	305	88	71.15
Banker	2765	484	82.50
Total	9587	2151	77.56

Table 10: **Savings Obtained by E-Selective Mutation.**

branches, and very little computation. Cal, on the other hand, contains very few branches, and is mostly a straight-line series of relatively simple computations. Since the five operators in E-selective mutation modify logical and arithmetic expressions, programs with a large number of decisions and arithmetic operations (such as Mid) will tend to result in less savings than programs with few decisions and few arithmetic operations (such as Cal).

4 Quantifying the Savings of Selective Mutation

It is clearly desirable to have an equation, or model, that relates the number of mutants generated for a given program to lexical characteristics of that program. Various models have been suggested for non-selective mutation. Budd [Bud80] claimed that the number of mutants for a program is $O(Vals * Refs)$, where $Vals$ is the number of data objects and $Refs$ is the number of data references. Acree et al. [ABD+79] claimed that the number of mutants is $O(Lines * Refs)$ – assuming that the number of data objects in a program is proportional to the number of lines. They went on to say that this is actually $O(Lines * Lines)$ for most programs.

We have checked these claims statistically. A sample of 96 Fortran-77 programs was used. The programs varied in size from 5 to 735 lines of code. Using Mothra, the mutants of each program were created. The number of mutants created for an individual program ranged from a low of 76

to a high of 3,911,460. For the entire sample, 11,180,104 mutants were generated. For each claim, the corresponding regression model was fitted to the data using the method of least squares. We used the SAS statistical package [FL81] for all our analysis.

Simple linear regression models with one explanatory, or independent, variable x_i can be written as:

$$Y_i = \beta_0 \ + \ \beta_1 x_i \ + \ \epsilon_i, \quad i = 1, 2, ..., n. \tag{1}$$

and multiple linear regression models with p explanatory variables x_{ij} can be expressed as:

$$Y_i \ = \ \beta_0 \ + \ \beta_1 x_{i1} \ + \ \beta_2 x_{i2} \ + \ ... \ + \ \beta_p x_{ip} \ + \ \epsilon_i, \quad i = 1, 2, ..., n. \tag{2}$$

The random variables ϵ_1, ϵ_2, ..., ϵ_n are errors that create scatter around the linear relationships.

For each model, the coefficient of determination was calculated. The *coefficient of determination*, or R^2 value, for a model is defined as the proportion of the variability in the predicted, or dependent, variable that can be accounted for by the explanatory variables of the model [Ott88]. The R^2 value provides a summary measure of how well the regression equation fits the data. For example, an R^2 value of 0.95 for a particular regression model means that the explanatory variables explain 95% of the variability in the Y values. Also, the *observed significance level*, or p value, of each β parameter was calculated; levels of .05 and less were deemed statistically significant.

We found the most accurate predictors of the number of mutants to be the number of lines ($Lines$), the number of variables and constant values ($Vals$), and the number of value references ($Refs$). Based on the lines of code, we obtain the following model:

$$Mutants \ = \ \beta_0 \ + \ \beta_1 * Lines \ + \ \beta_2 * Lines * Lines. \tag{3}$$

For non-selective mutation, we find that there is only a .06% probability that the $Lines$ term does not contribute, and a 1.22% probability that the $Lines * Lines$ term does not contribute. Thus, we conclude that both terms are significant.

For E-selective mutation, we find a .01% probability that the $Lines$ term does not contribute,

but a 96.68% probability that the $Lines * Lines$ term does not contribute. Thus, we conclude that a more appropriate model for E-selective mutation is:

$$Mutants \;=\; \beta_0 \;+\; \beta_1 * Lines. \tag{4}$$

On the other hand, the R^2 values for the two models are .488 and .508, respectively, meaning that only about half of the variation in the number of mutants is predictable from lines of code as an explanatory variable. Thus, we conclude that Acree et al. [ABD+79] were mistaken, and the number of lines in a program is not a valid predictor for the number of mutants.

Since Budd suggested that the number of mutants is based on the number of values times the number of value references, we obtained the following model:

$$Mutants \;=\; \beta_0 \;+\; \beta_1 * Lines \;+\; \beta_2 * Vals * Refs. \tag{5}$$

The R^2 value for this model was .988 for non-selective mutation. The p values were .0001 for each term.

For E-selective mutation, we obtained a different model:

$$Mutants \;=\; \beta_0 \;+\; \beta_1 * Lines \;+\; \beta_2 * Refs. \tag{6}$$

The R^2 values for this model was .979, and the p values were .0001 for each term. Thus, we conclude that E-selective mutation eliminates the $Vals$ factor from the number of mutants. This is as we expected, because the major category of mutants we are eliminating is the category that replaces variable references by all data values.

Finally, we looked at the simple ratio of mutants to E-selective mutants ($M/RS - S$) for our sample programs. On average, that ratio was approximately 17, which indicates that E-selective gives us an order of magnitude improvement. Unfortunately, the standard deviation is 21.1, indicating that this ratio is very program dependent. For example, in one program we had a ratio of only 1.59, and in another of 129.8.

Another way of reporting the improvement is to compare all mutants for the entire 96 programs. For standard mutation, we would have had to consider all 11,180,104 mutants. For E-selective, we would only have had to consider 231,972 mutants, a ratio of 48.2.

5 Conclusions and Future Work

This paper has presented results that indicate that previous mutation implementations are much more expensive than necessary, and support our hypothesis that selective mutation is effective and efficient. Specifically, our results indicate that the mutation operators that replace all operands with all syntactically legal operands add very little to the effectiveness of mutation testing. Additionally, the mutation operators that modify entire statements add very little. Our data indicates that of the 22 mutation operators used by Mothra, only 5 operators are sufficient to implement effective mutation testing. This result is a major step forward in the practical application of mutation, particularly since this allows mutation to be effectively implemented in time linear to the number of data references rather than references times the number of data objects.

The 5 operators are ABS, which forces each arithmetic expression to take on the value 0, a positive value, and a negative value, AOR, which replaces each arithmetic operator with every syntactically legal operator, LCR, which replaces each logical connector (AND and OR) with several kinds of logical connectors, ROR, which replaces relational operators with other relational operators, and UOI, which inserts unary operators in front of expressions. It is interesting to note that this set includes the operators that are required to satisfy branch and extended branch coverage [OV93], leading us to believe that extended branch coverage is in some sense a major part of mutation.

The results reported here, especially when combined with the techniques of weak mutation [OL] and schema-based mutation [UOH93], lead us to hope that it may soon be possible to make mutation testing a practical reality.

Our experiment has two assumption that warrant further investigation. One is our measurement

of selective effectiveness. We decided whether selective mutation was effective by computing the non-selective mutation scores of the test sets. Other measurements could be the relative abilities of the test sets to detect actual faults in the program, or other testing criteria. Another assumption is that our results will scale up to larger programs. The data in Section 2 indicates that on larger programs, there are relatively more replacement operator mutants, of mutants, thus the savings from selective mutation could be relatively greater.

A converse view, but one that may shed some light on mutation operators, could be based on the idea of operator strength. If we define the *operator strength* of mutation operator i to be the number of total mutants that are killed by test data that is generated to kill only the mutants of type i, then the operators with the greatest strength may be the most useful mutation operators.

These observations prompt one final question. If selective mutation involves the useful elimination of certain mutant operators, what is it about those operators that makes them easily killed by test cases that are sufficient for other operators? Except for a very few special cases [OV93], there has been no analytical work on the subsumption of some mutation operators by others. Investigation of this question might shed light on the coupling effect and on the nature of the sensitivity of errors to tests.

6 Acknowledgements

We would like to thank Aditya Mathur for initially suggesting the notion of selective mutation.

References

[ABD+79] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, September 1979.

[Bud80] T. A. Budd. *Mutation Analysis of Program Test Data.* PhD thesis, Yale University, New Haven CT, 1980.

[DGK+88] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff Alberta, July 1988. IEEE Computer Society Press.

[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.

[FL81] Rudolf J. Freund and Ramon C. Littell. *SAS for Linear Models: a guide to the ANOVA and GLM procedures*. SAS Institute Inc., Cary, NC, 1981.

[Ham77] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4), July 1977.

[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] A.P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference*, pages 604–605, Tokyo, Japan, September 1991.

[Mor84] L. J. Morell. *A Theory of Error-Based Testing*. PhD thesis, University of Maryland, College Park MD, 1984. Technical Report TR-1395.

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

[OL] A. J. Offutt and S. D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*. to appear.

[ORZ93] A. J. Offutt, Gregg Rothermel, and Christian Zapf. An experimental evaluation of selective mutation. In *Proceedings of the Fifteenth International Conference on Software Engineering*, pages 100–107, Baltimore, MD, May 1993. IEEE Computer Society Press.

[Ott88] Lyman Ott. *An Introduction to Statistical Methods and Data Analysis*. PWS-Kent Publishing Company, Boston, MA, third edition, 1988.

[OV93] A. J. Offutt and J. M. Voas. Subsumption of condition coverage techniques by mutation testing. In *Submitted for publication*, 1993.

[UOH93] R. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using program schemata. In *Proceedings of the 1993 International Symposium on Software Testing, and Analysis*, pages 139–148, Cambridge MA, June 1993.