An Experimental Determination of Sufficient Mutant Operators

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

ACM Transactions on Software Engineering Methodology. 5(2):99–118, April 1996.

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. This paper quantifies the expense of mutation in terms of the number of mutants that are created, then proposes and evaluates a technique that reduces the number of mutants by an order of magnitude. Selective mutation reduces the cost of mutation testing by reducing the number of mutants. This paper reports experimental results that compare selective mutation testing with standard, or non-selective, mutation testing, and results that quantify the savings achieved by 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 the number of mutants.

---

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 create test data that causes a finite, well-specified set

---

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, 4A4, George Mason University, Fairfax, VA 22030, {ofut,alee}@isse.gmu.edu; G. Rothermel, Department of Computer Science, Clemson University, Clemson, SC 29634-1906, grother@cs.clemson.edu; R. H. Untch, Department of Computer Science, Box 48, Middle Tennessee State University, Murfreesboro, TN 37132, untch@knuth.mtsu.edu; C. Zapf, Siemens AG – Medical Group, BNES 13, Henkestrasse 127, 91052 Erlangen, Germany, zapf@erlh.siemens.de.

of faults to result in failure. The tester does this by finding test cases that cause faulty versions of the program to fail. These test cases will then either result in correct output from the test program (demonstrating its quality) or cause the test program to fail (detecting a fault). The technique thus serves two goals: it provides a test adequacy criterion, and leads to detection of faults in the program being tested.

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 on average saved 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, show savings of over 75% of the execution cost on our experimental programs. More importantly from a theoretical point of view, our analysis in Section 4 shows that selective mutation reduces the number of mutants generated by mutation systems from quadratic in the number of variable references to linear.

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, present data that quantify the savings of selective mutation, and discuss possible future directions.

1.1 Mutation Testing Overview

Mutation testing helps users create 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, the mutant 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 mutated statement represents a separate program. The most recent mutation system, Mothra [DGK+88, KO91], uses 22 mutant operators to test Fortran-77 programs. 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],

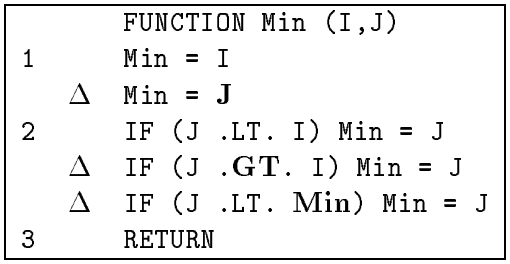

Figure 1: **Function Min.**

allows us to limit mutation operators to those that make simple changes. The coupling effect has been supported experimentally [Off92] in a study that compared test sets generated for mutants that involved changes in two places with test sets generated for single change mutants. Morell also analyzed the coupling effect from a theoretical perspective [Mor84]. The strongest support is by How Tai Wah, who proved probabilistically that the coupling effect holds under certain restrictions (for finite bijective functions) [Wah95]; his dissertation removes the restrictions.

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, the 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's output 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.* A test data set is said to be *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.

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 Mutant Operators.**

The syntactic modifications responsible for mutant programs are determined by a set of *mutant operators*. This set is determined by the language of the program being tested, and the mutation system used for testing. Mutant operators are created with one of two goals: to induce simple

syntax changes based on errors that programmers typically make (such as using the wrong variable name), or to force common testing goals (such as executing each branch). The Mothra mutant operators represent 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 mutant operators is represented by a three-letter acronym. For example, the "*a*rray reference for *a*rray reference *r*eplacement" (*AAR*) mutant operator causes each array reference in a program to be replaced by each other distinct array reference in the program. Budd's analysis of the number of mutants, discussed in Section 1.2, is based largely on the fact that the SVR mutant operator is the dominant operator.

Since mutant operators generate different numbers of mutant programs, Mathur [Mat91] proposed *selective mutation* [1] as mutation *without* the operators that create 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.

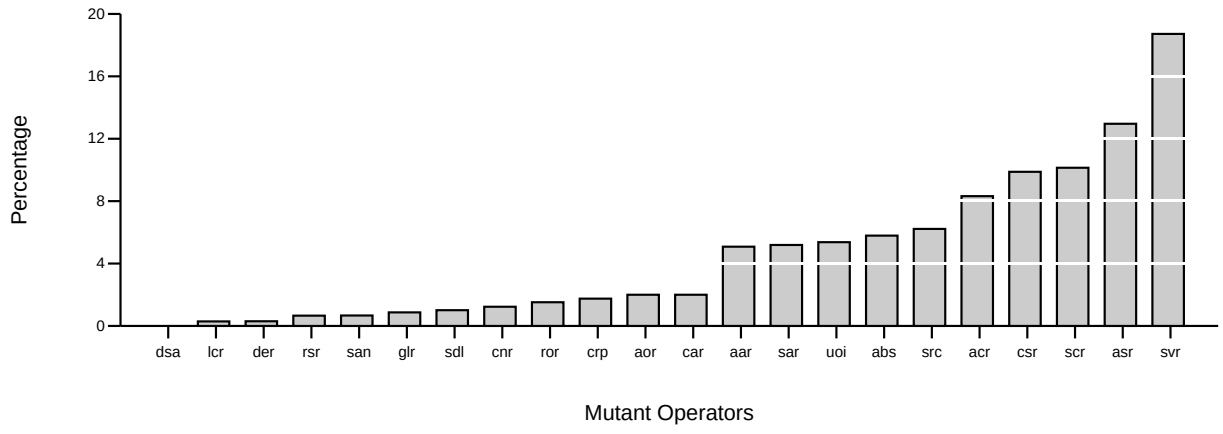

Figure 2: **Distribution of Mutants by Operator.**

Since the results from 6-selective mutation were so positive, we have extended selective mutation to try to experimentally determine whether further reductions in the number of operators can yield

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

effective mutation testing. The mutant operators used by Mothra can be divided into three general categories based on the syntactic elements that they modify. *Replacement of Operand* 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)* [2] to be mutation with only the expression and statement operators (not using the eleven operand replacement mutant operators), *replacement/statement-selective mutation (RS-selective)* to be mutation with only the replacement and statement mutant operators, and *replacement/expression-selective mutation (RE-selective)* to be mutation with only the replacement and expression mutant operators. Finally, in Section 3.3, we give results using *expression-selective mutation (E-selective)*, which is mutation with only the five expression modification mutant operators.

3 Experimentation with Selective Mutation

We evaluated selective mutation empirically. Two relationships that have been defined elsewhere are ProbBetter and ProbSubsumes. Weyuker, Weiss, and Hamlet [WWH91] 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 [MW94, Won93] 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 at least as 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

[2] In our previous paper [ORZ93], we defined selective in terms of the operators excluded; here we define selective in terms of the operators chosen.

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.

While it can easily be seen that non-selective mutation subsumes selective mutation, we hypothesize that selective mutation PROBSUBSUMES non-selective mutation. While it seems unlikely that selective mutation will consistently PROBSUBSUME non-selective, a fundamental aspect of software testing is that we are usually satisfied with partial results, and people who have experimented with mutation have observed that test sets with mutation scores exceeding 95% are difficult to create, and believe such test sets are effective [BDLS80, CM93, DO93, FWH96]. Thus, we refine our hypothesis to be: test sets that kill all mutants under selective mutation will have a high coverage level (high 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.

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. Godzilla attempts to generate a test case to kill a mutant by analyzing the properties a test case must have to kill the mutant, and representing these properties in the form of algebraic

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.**

systems of constraints. The constraint system for a mutant represents the paths that can be taken to reach the mutated statement, and what must be true at some execution of that statement for the mutant to cause an incorrect program state. Thus, the constraints are taken from the conditionals on the control flow graph of the program and the expression that is mutated. After creating these constraints, one for each mutant, Godzilla tries to satisfy each constraint system using a sophisticated heuristic procedure, which is fully described elsewhere [DO91, Off91]. The actual values chosen are within the space described by the constraint system, but are chosen arbitrarily from within that region.

Godzilla has very little user input; it primarily uses the test program and information about the mutants. The user only provides two inputs: a domain of values for the input parameters, and a limited set of assertions that can be inserted into the program. We used those assertions in six of the programs to further refine the input domains for certain variables. For example, the assertions encoded relationships among input variables, such as that an input length must be less than or equal to the declared length of the array, and stated requirements such as that certain input variables must be greater than 0. Godzilla does nothing except try to generate a test case that will kill each mutant – it has no other properties that are likely to bias the process. That is, based on a mutant, it creates a test requirement, formulates that as a constraint system, and tries to satisfy the constraint system to produce values.

To eliminate any possible bias that could be introduced by a specific test set, we generated five separate 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. The variances in both test set size and number of mutants killed for the 200 test sets were all quite low.

There were ten programs, and we generated five sets of test cases per program. We tried four

different variations of selective mutation test sets. Thus we had a total of 200 distinct sets of test data. However, Godzilla cannot always kill all mutants, so a few test cases had to be created by hand. These were created by the authors. The process was to look at each remaining live mutant and try to hand craft a test case to kill it. Although the actual number varied, the overwhelming number of test cases were generated by Godzilla. The 20 sets for `Banker` required the most hand generated test sets – between 1 and 5. `Banker` also required the most test cases, between 50 and 105. The average generated by hand was 1%, and in no case was more than 10% generated by hand.

Because of the overlap in requirements for killing mutants, many mutants are often killed by one test case, and many test cases do not kill any mutants. To be consistent with other empirical testing researchers, we define a test case to be *ineffective* if it does not kill any mutants, and eliminate ineffective test cases before running them on non-selective mutant sets.

Next, for each program, we created all *non-selective* mutants (using *all* mutant 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 an estimate for how close selective mutation comes to ProbSuming non-selective mutation.

3.2 Experimental Results for Selective Mutation

Mutation scores for each program, averaged over the 5 test sets, are shown in Tables 3 (ES-selective), 4 (RS-selective), and 5 (RE-selective). These tables 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) and above 99% for all programs under ES-Selective and RE-Selective mutation. The columns are the numbers of effective test cases.

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 and killed with selective mutation). Selective sets save from 6% to 72% of the mutants. Not surprisingly, the biggest gain is from ES-selective, which eliminates the operators that result in $O(Vals * Refs)$ mutants.

Program	Test Cases	Number of Live Mutants	Mutation Score
Banker	62.8	9.8	99.64
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
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
Banker	90.0	9.2	99.66
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
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
Banker	105.4	2.8	99.90
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
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
Banker	2765	629	77.25
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
Total	9587	2730	71.52

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

Program	Non-Selective Mutants	Selective Mutants	Percentage Saved
Banker	2765	2281	17.50
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
Total	9587	7436	22.44

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

Program	Non-Selective Mutants	Selective Mutants	Percentage Saved
Banker	2765	2620	5.24
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
Total	9587	9008	6.04

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

Program	Test Cases	Number of Live Mutants	Mutation Score
Banker	50.6	11.6	99.57
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
Average	13.3	2.9	99.51

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

Varying the number of test cases used on each program had little affect on results. The variances in size and number of mutants killed are very small. For example, the program with the largest variation in test set size for ES-selective mutation-adequate test sets was Find, which had from 10 to 17 test cases. The largest set had a (non-selective) mutation score that was only .11% larger than the smallest set – the scores ranged from 99.89 to 100. In fact, the largest test set killed only one more mutant (of the 1022 that existed) than the smallest set. This was typical of the programs we looked at. In other words, even the smallest selective mutation-adequate test sets killed almost as many mutants 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 wondered if the expression operators are the most useful operators. Thus, we extended our experimentation to measure the effectiveness of combining ES- and RE-selective mutation. *Expression-selective (E-selective)* mutation omits both the replacement and statement operators, leaving only five Mothra operators. Table 9 gives 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 sufficient to achieve almost full mutation coverage.

Table 10 gives 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 branches, and very little computation. Cal, on the other hand, contains very few branches, and is mostly a straight-line series of simple computations. Since the five operators in E-selective mutation modify logical and arithmetic expressions, programs with a large number of decisions and

Program	Non-Selective Mutants	Selective Mutants	Percentage Saved
Banker	2765	484	82.50
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
Total	9587	2151	77.56

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

arithmetic operations (such as Mid) can be expected to result in less savings than programs with few decisions and few arithmetic operations (such as Cal).

3.4 Comparing E-Selective Mutation with Randomly Generated Tests

As a control on our method for generating test data sets, we decided to evaluate E-selective mutation using randomly generated test data. This is an attempt to eliminate the possibility of a bias being introduced by our method of generating test data. We followed the same procedure as before, except that instead of using Godzilla, we developed random number generators for our programs (based on the Unix `random()` function). We generated inputs to the program in groups of 200, ran them against the E-selective mutants, and then eliminated all ineffective test cases. We repeated this process until all non-equivalent mutants were killed, or until we determined that further runs were not going to kill any more mutants.

The results from this control experiment are in Table 11. As can be seen, the results are almost identical to those in Table 9. Thus we see no evidence that the method of test data generation has an impact on our results.

The biggest difference between Table 9 and Table 11 is that random generation required many more test cases. While this is not a subject of this research, it points out a major problem of automatically generating test data using purely random techniques – it is expensive. In addition, the test case sizes reported in Tables 9 and 11 are of the effective test cases only. The number of test cases generated for mutation was bound by the number of mutants being targeted (this depends on which version of selective mutation is used, but Table 2 contains the total number of non-selective

Program	Test Cases	Number of Live Mutants	Mutation Score
Banker	61.4	10.4	99.62
Bub	4.0	0.0	100.00
Cal	19.2	9.2	99.67
Euclid	4.0	1.0	99.42
Find	15.8	0.8	99.92
Insert	8.4	0.4	99.90
Mid	26.2	0.0	100.00
Quad	6.0	0.0	100.00
Trityp	63.4	4.2	99.50
Warshall	3.2	1.2	98.56
Average	21.2	2.7	99.76

Table 11: **Non-Selective Mutation Scores of E-Selective Randomly Generated Adequate Sets.**

mutants). For all programs, we generated many more test cases using random generation than constraint-based generation – the lowest was 800 for Mid and the highest was 10,000 for for Trityp. Additionally, we had to generate significantly more test cases by hand when using the random number generators, greatly increasing the cost of test data generation.

3.5 Further Reductions in the Selective Set

The E-selective mutant operators form a coherent set of operators that result in far fewer mutants than the full set of mutants (in the next section, we show that E-selective mutation is linear in the number of variable references, rather than quadratic). But that does not mean that all of the E-selective operators are necessary for mutation testing. To determine whether further mutant operators can be eliminated, more data were gathered for each E-selective mutant operator.

For each program, test sets were generated that killed all the mutants generated by each of the five E-selective mutant operators. These test sets were then run against all mutants. For example, the ABS mutants were created, and test sets were generated to kill all of the ABS mutants. These test sets were generated using the same procedure as before; we generated five independent test sets, using Godzilla as far as possible, and when Godzilla did not ensure complete coverage, a small number of test cases were generated by hand.

The test sets generated were then run against all mutants, as before. This gave us, for each E-selective mutant operator, a list of all non-selective mutants that were killed by the test set based on that operator. We then asked the question, if only four of the five E-selective mutants are used, how many non-selective mutants will be killed? We computed this by merging four of the five

Program	With All Five	Without UOI	Without ROR	Without LCR	Without AOR	Without ABS
Banker	99.57	98.57	99.57	99.57	91.57	91.57
Bub	99.93	98.93	99.93	99.93	99.93	97.93
Cal	99.63	95.63	99.63	99.63	99.63	99.63
Euclid	99.00	97.30	99.30	99.30	99.30	99.30
Find	99.30	98.70	99.70	99.70	99.70	99.70
Insert	99.75	98.95	99.95	99.95	99.95	97.95
Mid	99.90	98.00	100.00	100.00	100.00	94.00
Quad	100.00	99.09	99.09	99.09	99.09	97.09
Trityp	99.36	98.36	98.36	99.36	99.36	98.36
Warshall	99.67	97.67	98.67	98.67	98.67	98.67

Table 12: **Non-Selective Mutation Scores using Four of Five E-Selective Mutation Adequate Sets.**

lists (eliminating duplicates). The results of this are shown in Table 12. We show the results of eliminating each mutant operator in turn.

In the table, the first column represents the non-selective mutation scores achieved by all five test sets, and the subsequent columns represent the non-selective mutation scores achieved by test sets generated without using that particular mutant operator. For example, the second column contains the mutation scores achieved by test data sets generated using the ABS, AOR, LCR, and ROR operators, but not UOI.

Unfortunately, the software testing literature offers no clear evidence that 100% coverage provides better testing than coverage at a lower level. Therefore we must draw conclusions from this data in a conservative way. Suppose we consider drawing the line at 99%. In Table 12, there are several scores below 99%. In the "Without UOI" column, the mutation score for `Cal` is 95.63%. Thus, we cannot conclude that UOI should be eliminated. Likewise, the score for Banker is 91.57% in both the "Without AOR" and "Without ABS" columns.

All the scores for LCR, however, are above 99%, except Warshall, for which the score is 98.67%. Thus we could conclude that the LCR operator contributed less to mutation testing of the subject programs than the other operators. If we follow our "99%" rule, we could round the score for Warshall up and eliminate the LCR operator. However, there is good reason not to take this step. The LCR operator replaces each logical connector (AND and OR) with other logical connectors, TRUE, and FALSE. These operators are somewhat uncommon, in particular, of the ten programs studied, eight had no LCR mutants. It would not be safe to draw strong conclusions from such a small number of mutants, therefore we do not recommend eliminating the LCR mutant.

The lowest score for ROR is 98.36%, which is only slightly below 99%, so an argument could also be made that it contributes less to mutation testing than the other operators. The ROR operator replaces each relational operator by other relational operators and TRUE and FALSE. Although it seems safer to eliminate this operator than any of the others, the ROR operator is precisely the operator that ensures branch coverage [OV96]; therefore, a strong argument can be made for keeping it. In the absence of further evidence, we conclude that all five E-selective mutant operators should be used.

4 Quantifying the Savings of Selective Mutation

This section tries to justify why an approximation method such as selective mutation is needed, and why selective mutation is valuable. A question we might like to ask is which operators generate mutants that are most useful from a testing perspective. Unfortunately, we do not presently have a context for answering, or even for precisely posing this question. This paper really asks the twin questions: A) Which mutant operators generate the most mutants?, and B) What happens if we do not consider these operators? The data in this section answer question A and give some justification that the answer to B is worth investigating.

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 on a sample of 96 Fortran-77 programs. The programs vary 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. The SAS statistical package [FL81] was used 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:

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

The parameters, or β_i values, represent the constants of the equations. Since we are only concerned with the limits of the equations (the Big-O forms), the β values have little significance. However, we do supply these values for the interested readers. 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 ρ 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 ($Refs$). First, we statistically test Acree's contention that the number of mutants is $O(Lines * Lines)$ for most programs, obtaining the following model:

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

The β values are: $\beta_0 = -37961$, $\beta_1 = 1521$, and $\beta_2 = 3$. For non-selective mutation, we find that there is only a .94% probability that the $Lines$ term does not contribute, and a 1.2% probability that the $Lines * Lines$ term does not contribute (the ρ values). Thus, we conclude that both terms are significant. The R^2 value is .488.

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:

$$E - SelectiveMutants \;=\; \beta_0 \;+\; \beta_1 * Lines. \tag{4}$$

The β values are: $\beta_0 = -320$ and $\beta_1 = 37$.

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 β values are: $\beta_0 = 23790$, $\beta_1 = -946$, and $\beta_2 = 1$. The R^2 value for this model was .988 for non-selective mutation. The ρ values were .0001 for each term.

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

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

The β values are: $\beta_0 = -25$, $\beta_1 = -24$, and $\beta_2 = 12$. The R^2 values for this model was .98, and the ρ 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. We also tried to use the number of language operators (as well as many other variables), but did not find a reasonable correlation.

Finally, we looked at the simple ratio of mutants to E-selective mutants (M/E) 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. Although we tried to relate this ratio to the programs, the closest we could find is that the ratio roughly goes up as the number of mutants goes up (as can be seen in Tables 6, 7, 8, and 10), but we do not feel comfortable drawing any general conclusions from this.

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

5 Conclusions and Future Work

This paper presents 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 mutant operators that replace all operands with all syntactically legal operands add very little to the effectiveness of mutation testing. Additionally, the mutant operators that modify entire statements add very little. Our data indicate that of the 22 mutant operators used by Mothra, 5 of these suffice to effectively implement mutation testing.

Section 4 shows that standard mutation takes time quadratic in the number of data references (actually, the number of data objects times the number of data references), whereas selective mutation can be effectively implemented in time linear in the number of data references. This is a major step forward in the practical application of mutation.

The 5 sufficient 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 [OV96], leading us to believe that extended branch coverage is in some sense a major part of mutation.

This research is unusual in that it takes a different direction from previous research in mutation testing. Previous researchers have tried to enhance the strength of mutation by adding more mutation operators, taking the "more is better" philosophy. On the contrary, we find that "less is more", or at least, that "less is nearly as good". Previous research focused on improving the strength of mutation testing, but without a clear metric for strength. We focus on efficiency and find that strength is (by our measure) only marginally compromised.

The results reported here, especially when combined with the techniques of weak mutation [OL94] and schema-based mutation [UOH93], lead us to hope that it may soon be possible to make mutation testing a practical reality in the form of tools that can used by software developers. Of course, we cannot make strong claims about the statistical significance of our results. We have no basis for claiming that our programs are a representative sample or that the test sets are representative. We tried to generate test cases in a realistic way – using tool support as far as possible, and hand generation thereafter. But the data certainly indicate that the idea is sound and the analysis shows that the technique has great potential.

Our experiment has two assumptions 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 indicate that larger programs have relatively more replacement operator mutants, thus the savings from selective mutation could be relatively greater.

These observations prompt one final question. If selective mutation involves the useful elimina-

tion of certain mutant operators, what is it about those operators that makes them easily killed by test cases that are adequate for other operators? Except for a very few special cases [OV96], there has been no analytical work on the subsumption of some mutant 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.

One possible reason for the E-selective adequate test sets to also kill R-selective and S-selective mutants is that most statements contain operators or references for which E-selective mutants are generated. That is, the E-selective mutant operators modify most statements. The five operators in the E-selective set modify every operator, every variable except variables on the left side of assignments, and all constants except `TRUE` and `FALSE`. In the programs studied in Section 3, we only found two kinds of statements that are not mutated by the E-selective set. Statements that assign a boolean constant (of which there is one, "`X = FALSE`"), and `GOTO` statements are not mutated. Therefore, the E-selective mutant operators require the tester to execute every statement, and test most statements in several ways.

5.1 Fault Size

Another possible explanation for the value of selective mutation can be found in the relationship between semantic and syntactic faults. Budd and Angluin [BA82] present a theoretical model of mutation in which mutants are considered to be in the "neighborhood" of the original program. Our perception of mutation may be subtly different depending on whether we think of the neighborhood of a program as being close in the syntactic sense or in the semantic sense.

In another paper [OH96], we define the *syntactic size* of a fault to be the fewest number of statements or tokens that need to be changed to get a correct program. Mutations have traditionally modified a single statement or token, therefore mutants are very small syntactically. We define the *semantic size* of a fault to be the relative size of the input domain for which the program is incorrect. Obviously, a mutant has both a syntactic and semantic aspect, but the sizes do not always correspond. A small syntactic change can result in a very large semantic fault – that is, it can affect almost every input. Also, a major syntactic rewrite of P can result in a very small semantic change.

Mutation systems create mutants by making syntactic changes. Some of these mutants are semantically small, even size zero (equivalent mutants), and some are "trivial", that is, the corresponding semantic fault is very large, and the mutant dies very quickly. We think it is likely that the most interesting mutants are those that are small semantically.

We suggest that the underlying goal of selective mutation is to try to only use operators that tend to produce mutants that have semantically small faults. If this model is correct, then selective mutants should contain a very high percentage of equivalent mutants. It turns out that the 5 operators in the selective set account for 57% of the equivalent mutants for the programs studied in Section 3.

6 Acknowledgments

We would like to thank Aditya Mathur for initially suggesting the notion of selective mutation. We would also like to thank one of the reviewers for suggesting that E-selective mutation works because the E-selective operators modify most statements in most programs.

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.

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

[BDLS80] T. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings of the 1980 ACM Principles of Programming Languages Symposium*, pages 220–233. ACM, 1980.

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

[CM93] B. Choi and A. P. Mathur. High performance mutation testing. *The Journal of Systems and Software*, 20(2):135–152, February 1993.

[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.

[DO93] R. A. DeMillo and A. J. Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering Methodology*, 2(2):109–127, April 1993.

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

[FWH96] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *The Journal of Systems and Software*, 1996. To appear.

[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.

[MW94] 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.

[Off91] A. J. Offutt. An integrated automatic test data generation system. *Journal of Systems Integration*, 1(3):391–409, November 1991.

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

[OH96] A. J. Offutt and J. H. Hayes. A semantic model of program faults. In *Proceedings of the 1996 International Symposium on Software Testing, and Analysis*, pages 195–200, San Diego, CA, January 1996. ACM Press.

[OL94] A. J. Offutt and S. D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, May 1994.

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

[OV96] A. J. Offutt and J. M. Voas. Subsumption of condition coverage techniques by mutation testing. Technical report ISSE-TR-96-100, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, 1996.

[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.

[Wah95] K. S. How Tai Wah. Fault coupling in finite bijective functions. *The Journal of Software Testing, Verification, and Reliability*, 5(1):3–47, March 1995.

[Won93] 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).

[WWH91] 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.