An Experimental Evaluation of Selective Mutation

A. Jefferson Offutt

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

Gregg Rothermel
Christian Zapf

Department of Computer Science
Clemson University
Clemson, SC 29634

Abstract

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 approximate mutation testing that saves execution 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 significant reductions in cost.

`Fifteenth International Conference on Software Engineering, pages 100--107, Baltimore, Maryland, May 1993.`

1 Introduction

Mutation testing is a technique, originally proposed in 1978 [DLS78], that asks the tester to demonstrate that the test 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 get correct output from the test program (demonstrating its quality) or also cause the test program to fail (detecting a fault).

Unit level testing techniques, of which mutation is one, hold great promise for improving the quality of software. Unfortunately, the application of these techniques is currently so expensive that we cannot afford to use them. This paper presents extensions to, and experimental work on an approximation method for mutation that has the potential to significantly reduce the computation cost of mutation.

1.1 Mutation Testing Overview

Mutation testing helps a user 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 is used to execute these faulty programs with the goal of causing each faulty program to fail. Hence 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.

Figure 1 contains a simple 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 types of mutation operators to test Fortran-77 programs. These operators have been developed and refined over 10 years through several mutation systems. The 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 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

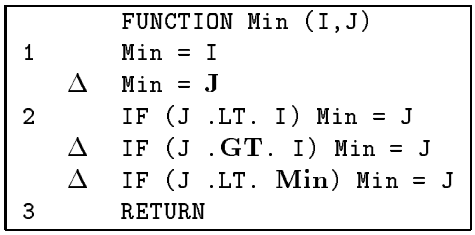

Figure 1: **Function Min.**

mutant. If the output of a mutant differs from that of the original program on the same test case, it is assumed to be incorrect and the mutant is killed.

After all of the test cases have been executed against all of the live mutants, each of the remaining mutants 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] has analyzed the number of mutants generated for a program and found them to be roughly proportional to the product of the number of data references times the size of the set of data objects. Typically, this is a large number. 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.

2 Selective Mutation Testing

One way to reduce the cost of mutation testing is to reduce the number of mutant programs we create, using an approximation approach 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 Mutation Operators.**

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 correspond to simple errors that programmers typically make. This particular set of mutation operators [KO91] represents more than ten years of refinement through several mutation systems. Each of the 22 mutation operators is represented to by a three-letter acronym (see Table 1). 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.

Since these operators generate mutant programs at different rates, Mathur [Mat91] proposed *selective mu-*

tation [1] as non-selective mutation *without* applying the mutation operators that result in the most mutants. Mathur theorizes that selective mutation has complexity *linear* in program size, rather than quadratic in the number of variable references, and yet retains much of the effectiveness of non-selective mutation. Mathur suggests omitting only the SVR and ASR operators, which we call 2-selective mutation. We also extend his theory by omitting the next most prevalent operators, letting the term *N-selective mutation* refer to mutation omitting the N most plentiful operators. To apply selective mutation, we first empirically investigate how many mutants are created by Mothra, and determine which operators create the most mutants.

2.1 Observed Number of Mutations

The following results are based on 28 Fortran-77 programs. The size of the programs varies from 8 to 164 lines of code, and the number of mutants created for each program ranges from 43 to 27331, for a total of 81159.

First we tried to empirically validate Budd's estimate of the complexity of mutation by trying to establish relationships between the number of generated mutants and the number of lines of code, the number of variables, the number of variable references, the number of branches, and McCabe's metrics [Som92]. We used simple and multiple linear regression models to establish relationships between these data. Simple linear regression models with one explanatory 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:

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

The random variables ϵ_1, ϵ_2, ..., ϵ_n are errors that create scatter around the linear relationships. When using linear regression models, the *coefficient of determination* provides a summary statistic that measures how well the regression equation fits the data. It was used to decide whether a relationship between some data existed. For example, a coefficient of determination of 0.95 for a particular regression model means that the explanatory variable explains 95% of the variability in the Y values [HL87].

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

We found the most accurate predictor of the number of mutants to be the number of variables ($Vars$) and the number of variable references ($Varrefs$). Using the SAS statistical package, the following formula has a coefficient of determination of 0.889:

$$\begin{aligned} Mutants &= \beta_0 + \beta_1 Vars + \beta_2 Varrefs + \\ &\quad \beta_3 VarsVarrefs. \end{aligned} \tag{3}$$

Formula 3 is therefore an approximation model that establishes a relationship between the number of mutants generated by Mothra and the number of variables and variable references in a Fortran program.

When we considered only single unit programs (one subroutine or function, or a main program without any subroutines or functions), we found results that were significantly more accurate than with complete programs. With single units, our coefficient of determination for Formula 3 was 0.96. With only single units, we were also able to derive the following formulas, with coefficients of determination of 0.96 and 0.95, respectively:

$$Mutants = \beta_0 + \beta_1 Lines + \beta_2 LinesLines, \tag{4}$$

$$Mutants = \beta_0 + \beta_1 Vars + \beta_2 VarsVars. \tag{5}$$

These formulae agree with the suggestion by Acree et al. [ABD+79] that the number of mutants is on the order of the square of the number of lines in the program. This is probably because the number of variable references in most programs tend to be within a constant of the number of lines.

Since the number of program units seemed to be important, we attempted to incorporate $Units$ into our formula, and developed the following formula, with a coefficient of determination of 0.91:

$$\begin{aligned} Mutants &= \beta_0 + \beta_1 Vars + \beta_2 Varrefs + \\ &\quad \beta_3 Units + \beta_4 VarsVarrefs. \end{aligned} \tag{6}$$

Unfortunately, similar functions based on Formulas 4 and 5 resulted in coefficients of determination that were too low for accurate representation.

We were also able to compute accurate coefficients of determination for formulas 3, 4, 5, and 6 for 2-selective, 4-selective, and 6-selective mutation. Table 2 shows the coefficients of determination for N-selective mutation and the corresponding formulas. Although Mathur [Mat91] theorized that selective mutation would result in a number of mutants that is linear in terms of the number of variable references in the program, our data indicate that this is not true. The number of mutants is still quadratic, even with

	Formula 3	Formula 4	Formula 5	Formula 6
2-selective	0.92	0.94	0.91	0.92
4-selective	0.93	0.94	0.93	0.90
6-selective	0.97	0.97	0.97	0.96

Table 2: **Coefficients of Determination for N-selective.**

6-selective mutation. This is unfortunate, but the savings are still substantial.

To determine which mutation operators are the most plentiful, we counted all the mutants of each type for our 28 programs. The distribution of these mutants is shown in Figure 2. The two mutant operators that occur the most are SVR (scalar variable replacement), and ASR (array for scalar replacement) — the same operators singled out by Mathur for elimination. The SVR operator replaces every scalar variable in a program unit by each scalar variable of compatible type visible in the unit. The ASR operator replaces every scalar variable in the unit by every array reference of compatible type in the unit. It is not surprising that these operators are the most plentiful, since they replace all variable references by all available variables, precisely the term that dominates in formula 3.

3 Experimentation With Selective Mutation

To investigate Mathur's hypothesis that selective mutation is almost as powerful as non-selective mutation, 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 a set of 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 programs that cover a range of applications were selected 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 3.

We began by experimenting with 2-selective mutation. For each program, we first created the selective mutation mutants for the program (leaving out the ASR and SVR operators). We then eliminated all equivalent mutants. 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 a few 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. All our results are based on average scores for these test sets.

Program	Test Cases	Number Live Mutants	Mutation Score
Banker	57.4	0.0	100.00
Bub	7.2	0.0	100.00
Cal	50.0	0.0	100.00
Euclid	3.8	0.0	100.00
Find	14.6	0.6	99.94
Insert	3.8	0.0	100.00
Mid	25.2	0.0	100.00
Quad	12.2	0.0	100.00
Trityp	44.6	0.2	99.98
Warshall	7.2	0.0	100.00
Average	22.6	0.1	99.99

Table 4: **Non-Selective Mutation Scores Achieved by 2-Selective Mutation Adequate 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 2-Selective Mutation

The mutation scores for each of our programs, averaged over the 5 test sets, are shown in Table 4. This data indicates that test sets that were 100% adequate for 2-selective mutation were all almost 100% adequate for non-selective mutation. In fact, on 8 out of 10 programs, selective sets *were* 100% adequate for non-selective mutation.

Table 5 shows the savings obtained by selective mutation in terms of the number of mutants. The "Per-

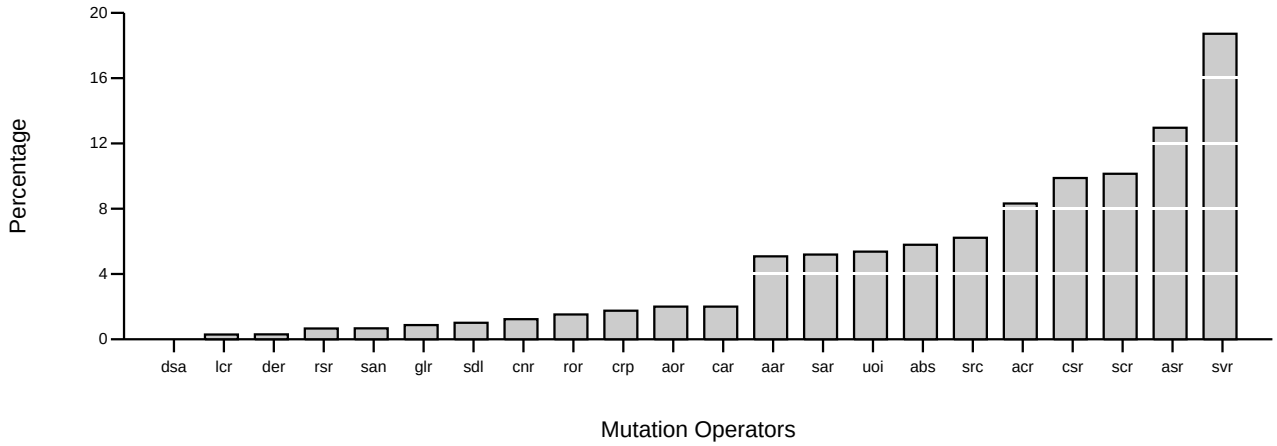

Figure 2: **Mutation Operator Distribution for Mothra.**

Program	Non-Selective Mutants	Selective Mutants	Percentage Saved
Banker	2765	1944	29.69
Bub	338	277	18.05
Cal	3010	2472	17.87
Euclid	196	142	27.56
Find	1022	622	39.15
Insert	460	352	23.48
Mid	183	133	27.32
Quad	359	279	22.28
Trityp	951	810	14.83
Warshall	305	259	15.08
Total	9589	7290	23.98

Table 5: **Savings Obtained by 2-Selective Mutation.**

centage Saved" column was 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 14.83% to 39.15% over non-selective sets on the experimental programs, for an average of 23.98%.

Varying the number of test cases used on each program had little affect on results. For example, selective mutation-adequate test sets for FIND ranged from 12 to 19 test cases with the larger set increasing the (non-selective) mutation score by only 0.21% — the scores ranged from 99.79 to 100.00. In fact, the larger test set only killed two more mutants (of the 1022 that existed) than the smaller set. Also, selective mutation-adequate sets for BANKER ranged in size from 45 test cases to 71, but all test sets were 100% mutation-adequate. In other words, even the smallest selective mutation-adequate test sets were almost as effective as the largest such sets.

The results lend credence to the hypothesis that selective mutation testing is a cost-effective alternative to non-selective mutation testing. Selective mutation provided results almost equivalent to non-selective mutation on the given set of programs, at a 23.98% savings over non-selective mutation (in terms of total mutant programs generated). These results appear stable over a range of sizes of test sets.

3.3 Extending Selective Mutation

If selective mutation as defined thus far is useful, it is because the SVR and ASR operators generate a large percentage of a program's mutants, and are easily killed by tests that kill other types of mutants. It seems reasonable to wonder if *other* operand replacement operators might as usefully be omitted in selective mutation. Figure 2 shows that the two operators CSR (constant-for-scalar replacement) and SCR

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 3: **Experimental Programs.**

Program	Test Cases	Number Live Mutants	Mutation Score
Banker	38.4	0.2	99.99
Bub	6.6	0.0	100.00
Cal	31.4	0.8	99.97
Euclid	2.4	2.2	98.72
Find	13.2	0.0	100.00
Insert	5.0	0.0	100.00
Mid	24.8	0.0	100.00
Quad	10.2	0.0	100.00
Trityp	44.8	2.0	99.76
Warshall	7.4	0.0	100.00
Average	18.4	0.5	99.84

Table 6: **Non-Selective Mutation Scores Achieved by 4-Selective Mutation Adequate Sets.**

Program	Non-Selective Mutants	Selective Mutants	Percentage Saved
Banker	2765	1546	44.09
Bub	338	224	33.73
Cal	3010	1804	33.72
Euclid	196	119	40.07
Find	1022	525	39.29
Insert	460	276	48.63
Mid	183	133	27.32
Quad	359	212	40.95
Trityp	951	579	39.12
Warshall	305	205	32.79
Total	9589	5623	41.36

Table 7: **Savings Obtained by 4-Selective Mutation.**

(scalar-for-constant replacement) are the next most prevalent; they may also be good candidates for exclusion during selective mutation.

To test this hypothesis, we repeated the experiment described in Section 3.1, with 4-selective mutation — defined as non-selective mutation without the SVR, ASR, CSR, and SCR mutants. We generated 5 new sets of test data per program to kill all non-equivalent 4-selective mutants, then computed the non-selective mutation score (all operators).

As Table 6 shows, even without these four mutation operators, test sets had an average non-selective mutation score of 99.84%, and the lowest mutation score was 98.72%. This score is not significantly different from that for 2-selective mutation, but is accompanied by significant increases in savings, averaging over 40% (see Table 7).

Finally, taking the hypothesis one step further, we considered 6-selective mutation – in which ASR, CSR, SCR, SVR, CAR, and ACR mutants are omitted. Tables 8 and 9 show the results of these experiments. With 6-selective mutation, the non-selective mutation scores are still very high. Since the savings for 6-selective mutation exceed 60%, this appears to be a very effective way to reduce the cost of mutation by more than half.

4 Conclusions and Future Work

This paper has defined and extended the mutation approximation approach of selective mutation, and presented empirical results that support the hypothesis that selective mutation is an effective alternative to non-selective mutation. For small programs, selective mutation-adequate test sets are effectively equivalent to non-selective test sets in their power to kill mutants, and offer reasonably good reductions in the cost of testing by reducing the number of mutants generated.

Program	Test Cases	Number Live Mutants	Mutation Score
Banker	28.4	1.2	99.96
Bub	6.6	0.0	100.00
Cal	26.0	5.2	99.81
Euclid	2.8	1.8	98.95
Find	13.4	2.2	99.77
Insert	4.0	0.0	100.00
Mid	24.8	0.0	100.00
Quad	10.0	2.0	99.39
Trityp	39.8	0.0	100.00
Warshall	4.2	2.0	99.26
Average	16.0	1.4	99.71

Table 8: **Non-Selective Mutation Scores Achieved by 6-Selective Mutation Adequate Sets.**

Program	Non-Selective Mutants	Selective Mutants	Percentage Saved
Banker	2765	1186	57.11
Bub	338	197	41.71
Cal	3010	738	75.48
Euclid	196	119	40.07
Find	1022	505	50.59
Insert	460	206	55.22
Mid	183	133	27.32
Quad	359	200	44.29
Trityp	951	533	43.95
Warshall	305	162	46.89
Total	9589	3782	60.56

Table 9: **Savings Obtained by 6-Selective Mutation.**

Additional work on more substantial programs is ongoing. On more substantial programs, SVR and ASR mutants account for a far greater percentage of mutant programs than they did in this study. For example, a 78-line Fortran program that calculates the roots of an arbitrary polynomial over a given interval yields 7435 mutants, of which 51.14% are SVR and ASR mutants; substantially more than in the programs used in this study. It seems possible that as the percentage of SVR and ASR mutants increases, the potential for benefiting from selective mutation also increases. We are currently exploring this.

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. It would also be interesting to attempt to characterize the types of mutant programs not killed by selective mutation-adequate test sets.

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? Investigation of this question might shed light on the coupling effect and on the nature of the sensitivity of errors to tests.

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.

[HL87] R. V. Hogg and J. Ledolter. *Engineering Statistics*. Macmillan Publishing Company, 1987.

[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.

[Som92] I. Sommerville. *Software Engineering*. Addison-Wesley Publishing Company Inc., 4th edition, 1992.