Sufficient Mutation Operators for Measuring Test Effectiveness

Akbar Siami Namin
Dept. of Computer Science
Univ. of Western Ontario
London, Ont., Canada
N6A 5B7
asiamina@csd.uwo.ca

James H. Andrews
Dept. of Computer Science
Univ. of Western Ontario
London, Ont., Canada
N6A 5B7
andrews@csd.uwo.ca

Duncan J. Murdoch
Dept. of Statistics and
Actuarial Sciences
Univ. of Western Ontario
London, Ont., Canada
N6A 5B7
murdoch@stats.uwo.ca

ABSTRACT

Mutants are automatically-generated, possibly faulty variants of programs. The mutation adequacy ratio of a test suite is the ratio of non-equivalent mutants it is able to identify to the total number of non-equivalent mutants. This ratio can be used as a measure of test effectiveness. However, it can be expensive to calculate, due to the large number of different mutation operators that have been proposed for generating the mutants.

In this paper, we address the problem of finding a small set of mutation operators which is still sufficient for measuring test effectiveness. We do this by defining a statistical analysis procedure that allows us to identify such a set, together with an associated linear model that predicts mutation adequacy with high accuracy. We confirm the validity of our procedure through cross-validation and the application of other, alternative statistical analyses.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

General Terms

Experimentation, Measurement

Keywords

Mutation analysis, testing effectiveness

1. INTRODUCTION

Measurement of the cost and effectiveness of a testing technique is an important component of research on software testing. With information from researchers about the cost and effectiveness of various techniques on various kinds of subject software, a practitioner can make an informed choice of which technique to use. However, while there are several clear criteria for measuring the cost of a testing technique (number of test cases, running time, effort required to develop), clear criteria for measuring effectiveness are elusive.

ICSE'08, May 10–18, 2008, Leipzig, Germany.

Recently researchers have used *mutation analysis* to measure testing effectiveness [6, 5, 19, 9, 24]. To perform mutation analysis on a subject program, we apply operators called *mutation operators* to the program source code, generating *mutants* of the code. Each mutant represents a possibly faulty variant of the original program. We then measure the effectiveness of a test suite or a testing technique by running it on all mutants, and computing its *mutation adequacy score*, referred to here as AM, that is the ratio of mutants detected to total number of non-equivalent mutants.

It has been shown that AM can be a good predictor for the effectiveness of a test suite on real faults [2, 3]. It can therefore be used for experimental subject software for which real faults are not known. However, a problem with this technique is that mutation analysis can take an infeasibly long time due to the large number of mutants generated. For instance, we have found that the 108 operators of the Proteum mutant generation system [7] applied to a program of 137 net lines of code produces a total of 4,937 mutants. Measuring the effectiveness of a testing technique using such large numbers of mutants can simply take too long.

We would therefore prefer to measure mutation adequacy scores with respect to a smaller set of operators. The question is, what set of operators will allow us to obtain a sufficiently accurate measurement of overall mutation adequacy? Such a set of mutation operators is referred to as a *sufficent set* in the mutation literature [20]. The sufficient set question has been studied by several researchers [26, 20, 4], but only from the perspective of the related practice of mutation testing, not mutation analysis.

The sufficient set question can be seen as an instance of the problem variously known as the *variable reduction*, *variable subset selection* or *model selection* problem, in which we try to find a set of predictor variables that can accurately predict a response variable. In this paper, which completes and extends an analysis presented in [23], we adapt existing statistical techniques in order to identify a set of mutation operators that generates a small number of mutants, but that still allows us to predict accurately the mutation adequacy of a testing technique on the full set of mutants.

The key research contributions of the paper are:

1. We identify a statistical procedure that can be used to find a sufficient set of mutation operators, while considering all possible subsets rather than a small number of specific subsets.

2. We apply the procedure in order to identify a sufficient set that can be used to accurately measure mutation adequacy for a large, comprehensive set of mutation operators.

3. We confirm the validity of this sufficient set by cross-validation and by applying other, alternative statistical analyses.

This research therefore moves testing researchers closer to the goals of being able to quickly and accurately measure and compare how effective different testing techniques are at finding faults, and of being able to communicate this information to testing practitioners.

1.1 Definitions

We assume a set $\{\mu_1, \ldots, \mu_z\}$ of *mutation operators*. Each mutation operator is a function from a program P to a set $\mu_i(P)$ of *mutants* of P. For this paper, we use the 108 mutation operators of Proteum [7], i.e. $z = 108$. We say that a test case *kills* a mutant if it exposes the failing behaviour of that mutant. As is conventional, we assume that the original program is a "gold version" without important faults, and we assume that a test case kills a mutant if the mutant gives different output from the gold version. If a mutant is such that no test case can kill it, we say that the mutant is *equivalent* (to the gold version). We say that a test suite S kills a mutant if any test case in S kills the mutant.

For a given P, we define:

- $Nm_i(P)$ is the number of non-equivalent (NE) mutants generated by operator μ_i on P.
- $NM(P)$ is the total number of NE mutants generated by all operators, i.e.

$$NM(P) = \sum_{i=1}^{z} Nm_i(P)$$

For a given P and test suite S, we define:

- $Km_i(P, S)$ is the number of NE mutants generated by operator μ_i on P, that are killed by S.
- $KM(P, S)$ is the total number of NE mutants generated by all operators on P, that are killed by S, i.e. $KM(P, S) = \sum_{i=1}^{z} Km_i(P, S)$.
- $Am_i(P, S)$ is mutation adequacy ratio of S with respect to the NE mutants generated by μ_i, i.e. $Am_i(P, S) = Km_i(P, S)/Nm_i(P)$.
- $AM(P, S)$ is mutation adequacy ratio of S with respect to the full set of NE mutants, i.e.

$$AM(P, S) = KM(P, S)/NM(P)$$

When it is clear from context, we drop P and/or S, writing only Am_i, AM, etc.

We interpret the sufficient subset problem as the search for a subset of mutation operators and a linear model that will allow us to predict AM accurately from the Am_i measures, while generating only a small number of mutants. That is, we wish to find a set $\{s_1, s_2, \ldots, s_j\} \subset \{1, 2, ..., z\}$, an intercept k and a set of coefficients $\{c_1, c_2, \ldots, c_j\}$ such that

$$AM \cong k + c_1 Am_{s_1} + c_1 Am_{s_1} + \cdots + c_j Am_{s_j}$$

We refer to the set, intercept and coefficients as a *model* of AM. In searching for the set of operators, we balance the accuracy of our prediction against the number of mutants generated by $\mu_{s_1} \ldots \mu_{s_j}$. We prefer a small number of mutants, but not so small a number that we cannot find an accurate model of AM.

1.2 Paper Organization

In Section 2, we review related work. In Section 3, we describe the experimental procedure that we followed to collect the raw data on which the analyses are based. In Section 4, we describe the statistical analysis of the data, the results of the analysis, and the cross-validation procedures and their outcomes. In Section 5, we offer discussion of the experiment and its results. In Section 6, we conclude and describe potential future work.

2. RELATED WORK

The previous research most closely related to ours has to do with mutation testing, the sufficient subset problem, and mutation analysis.

2.1 Mutation Testing

Mutation testing, first proposed by Hamlet [14] and DeMillo et al. [8], is a process by which mutation is used to enhance a test suite. In mutation testing, the tester generates mutants of the SUT, and runs all current test cases on all mutants. The tester then classifies any mutants that are not killed by the test suite. If the tester judges the mutant to be equivalent to the original (an undecidable problem in general, but one that can sometimes be judged by the tester with high confidence), the mutant is eliminated from consideration. Otherwise, the tester adds to the suite a test case that kills the mutant. Ideally, the process ends when the test suite has achieved an AM of 1.0, i.e. when it kills all non-equivalent mutants.

In mutation testing, each mutant is intended to represent a possible fault that a program could have, so generating (and killing) more mutants means greater confidence in the thoroughness of a test suite. However, so many mutants can be generated that it becomes infeasible to perform full mutation testing. Various ways of speeding up mutation testing have been proposed, including compiling many mutants in one executable, partially automatic detection of equivalent mutants, and execution of mutants just long enough to determine with reasonable accuracy whether the mutant has been killed [21].

Since the first research on source code mutation [8], many mutation operators have been proposed. Agrawal et al. [1] list and describe 76 operators. The Proteum mutant generation system [7], the most comprehensive system that we are aware of, implements 108 operators, including most of the ones described by Agrawal et al. More mutation operators means more mutants and theoretically more thorough test suites, but it is not clear whether the many proposed mutation operators are actually necessary. For instance, since the mutation of changing a logical operator often has a similar effect to negating a decision, it is reasonable to ask whether both operators are really necessary. This has motivated the search for a *sufficient set* of mutation operators, defined, in the context of mutation testing, as a set such that a test suite achieving an AM of 1.0 on mutants generated from the set is likely to achieve a high AM on the full set of mutants.

2.2 Sufficient Operators

Given a subset σ of the mutation operators $\{\mu_1, \ldots, \mu_z\}$ and a test suite S, we define $AMS_\sigma(S)$ as the fraction of the non-equivalent mutants generated by σ that are killed by S. The mutation testing sufficient set problem is that of finding a set σ such that $AMS_\sigma(S) = 1.0$ implies a high $AM(S)$ with high probability.

Previous empirical work on a sufficient set of operators for mutation testing is limited. Wong [26] compared a set of 22 mutation operators to a set w of two mutation operators; he found that test suites with $AMR_w = 1.0$ had an average AM of 0.972, while w

generated 82.54% fewer mutants on average than the full set of 22 mutation operators did. Offutt et al. [20] compared the same set of 22 mutation operators with four fixed subsets of those mutation operators. They judged one subset, called E (a subset with five mutation operators) to be best, based on the fact that test suites with $AMS_E = 1.0$ had high AM (average 0.995) while E generated 77.56% fewer mutants on average.

Finally, Barbosa et al. [4] defined a sequence of six guidelines for selecting a sufficient set, such as "Consider one operator of each mutation class" and "Evaluate the empirical inclusion among the operators". They carried out the guidelines on the 71 operators that were implemented in Proteum as of 2001. They thus obtained a set $SS27$ of 10 operators such that test suites with $AMS_{SS27} = 1.0$ had AM of 0.996 on average, while $SS27$ generated 65.02% fewer mutants than the full set. They showed that on their subject programs, the final sufficient set determined a higher AM than the Offutt et al. or Wong sets.

The experimental subjects across the two experiments of the Wong study were eight Fortran programs, which were translated into C so that coverage could be measured; the C programs had a total of 463 raw lines of code (lines including comments and whitespace), or an average of approximately 57.9 raw lines of code each. The subjects in the Offutt et al. study were 10 Fortran programs with a total of 206 statements, or an average of 20.6 statements each, and the subjects in the Barbosa et al. study were 27 C programs with a total of 618 LOC, or an average of 22.89 LOC each. The Wong subjects have no `struct` definitions, and the descriptions of the Offutt et al. and Barbosa et al. subjects suggest that they also have few complex data structures, making it unclear whether the results of their experiments would carry over to complex C programs and object-oriented programs.

There are three main differences between this previous work and our work. First, we use somewhat larger subject software (see Section 3.1) that has a wide range of control and data structures, including `structs`. Second, while the previous studies are focused on mutation *testing*, ours is focused on mutation *analysis*. We are therefore interested in finding a sufficient set that will allow us to predict AM accurately across a wide range of values, not just close to the 1.0 level. Third, we treat the problem as a standard variable reduction problem. We thus take into account all possible subsets of operators, including those not considered by the operator selection procedures used by previous researchers.

2.3 Mutation Analysis

In the early 1990s, researchers began to measure and compare accurately the effectiveness of testing strategies by measuring how many faults the testing strategies find. Frankl and Weiss [12] and Hutchins et al. [15] manually seeded faults in subject software, and measured how many faults were found by different test procedures. Later, Frankl and Iakounenko [11] performed similar experiments, but on a subject for which real faults identified during testing and operational use were known; the faults were re-seeded into the software to create the faulty versions.

Other researchers, such as Briand et al. [6, 5], Mayer and Schneckenburger [19], Do and Rothermel [9], and Tuya et al. [24], later seeded faults in subject software using mutation operators from the mutation testing literature. The advantage of this practice over hand-seeding is greater replicability, and the advantages of using mutants over both hand-seeded faults and real faults include decreased effort and greater numbers of potentially faulty versions, leading to greater statistical significance of results.

Andrews et al. [2, 3] showed that, despite the relative simplicity of the faults introduced by mutation operators, mutants behaved very similarly to real faults for the purposes of measuring test effectiveness. Taking AF to stand for the fraction of hand-seeded or real faults eliminated by a test suite, they showed that average $AM(S)$ was similar to average $AF(S)$ for the real faults of the Frankl and Iakounenko software [12] but higher than average $AF(S)$ for the hand-seeded faults of the Hutchins et al. software [15]. This in turn suggests that automatically generating mutants and computing AM is a reasonable method for evaluating test effectiveness.

Program	NLOC	NTC	NMG	NM
`printtokens`	343	4130	11741	1551
`printtokens2`	355	4115	10266	1942
`replace`	513	5542	23847	1969
`schedule`	296	2650	4130	1760
`schedule2`	263	2710	6552	1497
`tcas`	137	1608	4935	4935
`totinfo`	281	1052	8767	1740

Table 1: Description of subject programs. NLOC: Net lines of code. NTC: Number of test cases. NMG: Number of mutants generated by all mutation operators. NM: Number of selected mutants that were non-equivalent.

In a preliminary study of the sufficient set problem for mutation analysis, Siami Namin and Andrews [23] treated the problem as a variable reduction problem, and used three analysis procedures to identify a reduced set of operators. In this paper, we complete and extend this work, replacing the forward-selection procedure used there for all-subsets regression by a more accurate and appropriate procedure (cost-based LARS, to be described below). We also extend it by performing cross-validation of the results, and analyzing the resulting sufficient set and model.

In [23] and this paper, we measure test suite effectiveness relative to non-equivalent mutants only, not relative to all (equivalent or non-equivalent) mutants. We do this in order to focus on finding a good set of mutation operators for researchers who are doing experiments in which testing techniques are compared. By definition, equivalent mutants are those that cannot be killed by any testing technique, so in such experiments, researchers are typically uninterested in the (identically poor) effectiveness of testing techniques on those mutants.

However, it should be noted that ignoring equivalent mutants produces larger differences in measured effectiveness between testing techniques, since all kill ratios are scaled up. Similar research to ours could be done with test effectiveness measured relative to all mutants, and this might change the results, since some mutation operators might generate higher numbers of equivalent mutants than others.

3. EXPERIMENTAL PROCEDURE

In this section, we describe the subject software used in the experiment, our method for collecting the raw data on which the statistical analysis was based, and our method for calculating the costs of the mutation operators.

3.1 Subject Software

Our subject software is the seven Siemens programs [15], available via the Subject Infrastructure Repository (SIR) at the University of Nebraska-Lincoln. This is a diverse collection of C programs that include code involving integer and floating-point arithmetic, `structs`, pointers, memory allocation, loops, `switch` statements and complex conditional expressions. The programs total

2188 NLOC (net lines of code, or lines of code without comments or whitespace), and thus have an average of 312.57 NLOC each. One advantage of the Siemens programs is that each comes with a large pool of diverse test cases, first written by Hutchins et al. and augmented by Rothermel et al. [22]. These pools allow a wide range of different test suites to be constructed following different algorithms or coverage criteria, each of which may represent an arbitrary test suite chosen by a software engineer. Detailed descriptive statistics of the Siemens programs are found in Table 1.

There are practical constraints on the size of programs we were able to use. To do all the computations that went into the experiments reported here took about 10 weeks of continuous CPU time, not counting false starts. Most of this CPU time consisted of running all test cases on a large, representative sample from the generated mutants. Increasing the size of the subject programs would have meant quadratically increasing the amount of CPU time needed, since a larger program means both more mutants and more test cases. Larger programs might contain more complex paths and more complex cause-effect chains from faults to failures, so it is possible that such programs might behave differently than those we studied. However, the Siemens programs contain most of the control and data structures used in even large-scale C programs, mitigating this threat to external validity.

3.2 Data Collection

For each subject program, we generated all the mutants from all the operators defined by Proteum; see column NMG in Table 1 for the total numbers. Exploratory work with the smallest program (`tcas`) indicated that using all mutants in our experiment would take too much processing to be feasible, so for each of the other programs we selected 2000 mutants randomly. In order to spread the selected mutants evenly over all mutation operators, we computed the ratio $2000/NM(P)$ and selected that ratio of the mutants from each mutation operator.

We then ran all test cases in the entire test pool on all selected mutants, and recorded which test cases killed which mutants. Mutants that were not killed by any test cases were considered equivalent, for the purposes of the experiment. The remaining number of non-equivalent mutants is listed as column NM in Table 1. Note that the number for `tcas` is higher because we did not start by selecting 2000 mutants.

For each program, we generated 100 test suites, consisting of two of each size from one test case to 50 test cases. We chose these sizes because exploratory work indicated that this provided test suites with a broad range of AM values, from 0.0 to 1.0. The test cases that went into each suite were randomly chosen. We then tabulated which test suites killed which mutants, and computed the value of $Am_i(S)$ for each mutation operator μ_i and each test suite S. Finally, we computed the actual AM value for all test suites. Note that the notion of equivalence used in the calculation of AM is based on all test cases in the test pool, not just on the selected test suites; thus the calculation takes into account both "easy" mutants (those killed by many test cases) and "hard" mutants (those killed by few test cases).

The data collection therefore yielded, for each of the seven programs, and for each of the 100 subject test suites, values for AM and for Am_i for each $i, 1 \leq i \leq 108$. These are the values that we used in the subsequent analysis.

3.3 Cost Calculation

Since we are interested in comparing the numbers of mutants generated by different sets of mutation operators, we also computed the "cost" associated with each operator, such that an operator μ_i has a lower cost than an operator μ_j if μ_i generates fewer mutants than μ_j, when averaged across all subject programs.

Thus we define $cost(i, P)$, the cost of μ_i for program P, as the number of non-equivalent mutants generated by μ_i, divided by the total number of non-equivalent mutants of P generated by all operators. We then computed the overall cost $cost(i)$ of μ_i as the average of $cost(i, P)$ across all subject programs P. This computation avoids biasing the measurement of cost toward programs which have more mutants overall.

4. ANALYSIS AND CROSS-VALIDATION

In this section, the main section of the paper, we first (Section 4.1) describe alternative approaches to the model selection problem, including the method (cost-based LARS) that we adopted. In Section 4.2, we describe the results of the model selection procedure on the data from all seven subject programs.

In our application of linear models, an important concern is the risk of overfitting, i.e. the risk of finding a model of the data that fits the available data (including the noise in that data) so well that it would fail to fit other data. To check our procedure for overfitting, we performed a sevenfold cross-validation. The procedure and results of this cross-validation are contained in Section 4.3.

As a further check on our results, we applied two other subset selection procedures to see whether they yielded substantially different results. The outcome of this study is contained in Section 4.4.

4.1 Approaches to Subset Selection

As pointed out in Section 1.1, we seek a subset of the mutation operators, and a linear model, that will allow us to predict AM accurately from the Am_i measures associated with the subset. This can be viewed as a classic *variable reduction* or *subset selection* problem, where we try to find a subset of *predictor variables* and a linear model that predicts accurately the value of a *response variable*. In our study, the predictor variables are the Am_i measures, and the response variable is AM.

An obvious, and inefficient, approach to the subset selection problem is to construct all possible subsets, perform a linear regression on each of them, and then derive a sequence $M_1, M_2, \ldots, M_i$ of models in which each M_i is the best model with the same cost or lower cost. Since for us this would mean constructing 2^{108} models, this is not feasible.

Other traditional subset selection algorithms include the greedy algorithm known as *forward selection* [10]. In forward selection, we first select the predictor variable with the highest correlation to the response variable, and then iteratively select the predictor variable with the highest correlation to the residual resulting from the application of the previous predictors. These algorithms are effective when the number of variables is small, but can fail to find the best model when the number of variables increases.

Least Angle Regression (LARS) is a generalization of the forward selection and other algorithms, employing a more effective formula and therefore using less computer time [10]. Like forward selection, the LARS procedure starts with a model with all coefficients equal to zero. The procedure finds the predictor variable, x_{j_1}, which is the most correlated predictor with the response variable. The procedure takes a step in the direction of this predictor until some other predictor, x_{j_2}, has the most correlation with the current residual. Unlike forward selection, however, instead of continuing along x_{j_1}, LARS proceeds in a direction equiangular between the two predictors until a third variable x_{j_3} shows up. LARS then proceeds in a direction equiangular between x_{j_1}, x_{j_2} and x_{j_3}, and so on. LARS is preferred over the traditional methods,

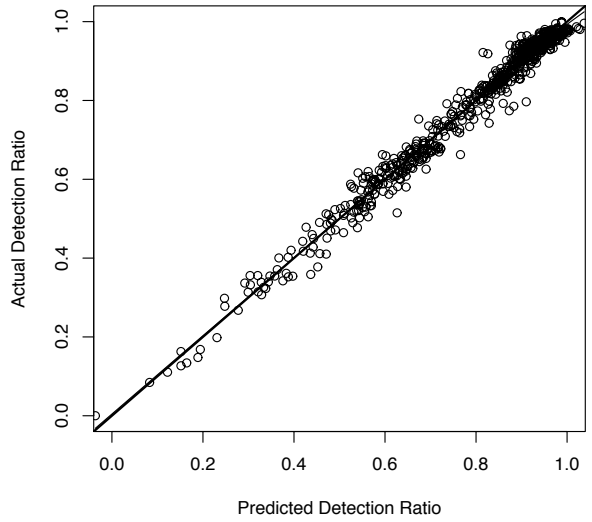

Figure 1: Actual vs. predicted value of AM for the chosen model, with $x = y$ line.

because it is computationally less expensive and yields models that predict the response variable more accurately.

For our application, we needed a slight modification to LARS. Given two sets of operators that yield models of identical accuracy, we prefer not necessarily the set that is smaller, but the set that yields fewer mutants, i.e. that has the lowest total cost (see Section 3.3). The third author therefore adapted the implementation of LARS in the R statistical package [25] so that it optimized *total cost* of predictor variables, rather than merely the *number* of predictor variables. The resulting algorithm, which we refer to as Cost-Based LARS (CBLARS), was applied to the data from the seven subject programs.

4.2 CBLARS Subset Selection

CBLARS yielded a sequence of models with increasing total cost of mutation operators and increasingly good prediction of AM, as measured by the adjusted R^2 value[1]. We chose the first model in the sequence with an adjusted R^2 value of 0.98 or more. This threshold is subjective, and was based on our subjective balancing of the cost and accuracy of the selected models; however, we note that our cross-validation (see below) indicated that overfitting did not occur at this level. Inspection of leverage and Cook distance measures[2] provided by R indicated that the model was not being unduly skewed by any of our individual observations. As a test for goodness of fit, we checked the residuals of the model using a quantile-quantile plot[3]. This showed that the residuals have normal distributions, as desired.

Summary statistics of the resulting model are shown in Table 2. The model itself is shown in Table 3, with the operators in the model listed by abbreviation, description, and coefficient of Am_i; the intercept of the model is also given. The descriptions come from the literature on C mutation operators [1] and the Proteum tool [7]. A plot of the actual value of AM (y-axis) vs. the value predicted by the model (x-axis) is found in Figure 1.

Model number in sequence	42
Number of operators	28
Number of mutants	1139
Percentage of mutants	7.40
Adjusted R^2	0.98
Df	671
Residual standard error	0.03
Variance of predicted values	0.04
MSE	< 0.01

Table 2: Summary statistics of the model selected by the CBLARS procedure.

The model identifies a sufficient set of 28 mutation operators that generate only 1139 mutants across all seven programs. This represents 7.40% of the mutants generated by all 108 Proteum operators, or a saving of approximately 92.6%. Thus, although a relatively large number of mutation operators are included in the sufficient set, they tend to all generate a relatively small number of mutants. We defer further discussion of the particular model selected by the CBLARS procedure until Section 5.

4.3 Cross-Validation

To check our procedure for overfitting, we performed a seven-fold cross-validation. In cross-validation, we use a subsample of the available data as a *training set* to develop a model, apply the model to the rest of the data (the *test set*), and measure how well the model applies to the test set. In K-fold cross-validation, we partition the data into K subsamples, and perform K cross-validations, using each subsample as a test set. For our application, the most appropriate selection of the data for each subsample is the data from one particular program, since we want to see how accurately the model resulting from our subset selection procedure would be for other programs.

For each program P, we therefore set aside the data for P as the test set, and performed the CBLARS procedure on the training set formed by the data from the other six programs. Because we wanted to achieve a reduction in number of mutants generated of at least 90%, we considered only the models from the sequence that generated 10% or fewer of the mutants of the program. We then applied each model in the sequence to program P, measuring how well it fit by measuring the mean squared error (MSE).

We found that within the sequence of models observed, the MSE initially fluctuated but then went down to a reasonable level. An example of this phenomenon is illustrated in Figure 2, where the x-axis represents the ordinal number of the model generated by CBLARS, the y-axis represents the MSE of the model on the test set, and each point represents one model from the sequence.

We noted that for all programs, the first model that achieved an adjusted R^2 of 0.98 or more had close to minimal MSE. We therefore selected 0.98 as our threshold for adjusted R^2, and selected the first model that achieved that level of adjusted R^2 or higher. To measure the goodness of fit of the selected model, we computed the square of the Pearson correlation (r^2), which measures how much of the variability of actual AM could be explained by variation in predicted AM. Since an important application of mutation analysis

[1] R^2 is a measure of goodness of fit, but tends to automatically reward models with a large number of predictors. Adjusted R^2 is a related measure that avoids this problem by imposing a small penalty for each additional predictor.

[2] The leverage of a data point is a measure of how large an effect the point has on a regression model; the Cook distance of a data point is a measure of how much a model would change if the point were deleted.

[3] A quantile-quantile plot is a method for visualizing whether two data sets have similar distributions. It can be used to detect whether the residuals (the differences between predicted and actual values) have the desired normal distribution.

Name	Description	Coefficient	NumMut
IndVarAriNeg	Inserts Arithmetic Negation at Non Interface Variables	0.162469	126
IndVarBitNeg	Inserts Bit Negation at Non Interface Variables	0.168583	121
RetStaDel	Deletes return Statement	0.051984	67
ArgDel	Argument Deletion	-0.016532	12
ArgLogNeg	Insert Logical Negation on Argument	0.131566	30
OAAN	Arithmetic Operator Mutation	0.041376	71
OABN	Arithmetic by Bitwise Operator	-0.02075	27
OAEA	Arithmetic Assignment by Plain Assignment	-0.194022	2
OALN	Arithmetic Operator by Logical Operator	0.022149	40
OBBN	Bitwise Operator Mutation	-0.023452	3
OBNG	Bitwise Negation	-0.00275	6
OBSN	Bitwise Operator by Shift Operator	-0.035337	3
OCNG	Logical Context Negation	0.097971	57
OCOR	Cast Operator by Cast Operator	0.024727	9
Oido	Increment/Decrement Mutation	0.069952	14
OLAN	Logical Operator by Arithmetic Operator	0.027124	119
OLBN	Logical Operator by Bitwise Operator	-0.00362	67
OLLN	Logical Operator Mutation	0.041048	25
OLNG	Logical Negation	0.06966	78
OLSN	Logical Operator by Shift Operator	-0.003438	45
ORSN	Relational Operator by Shift Operator	0.160376	91
SGLR	goto Label Replacement	0.07605	1
SMTC	n-trip continue	0.031519	9
SMVB	Move Brace Up and Down	-0.062404	2
SSWM	switch Statement Mutation	-0.020412	15
STRI	Trap on if Condition	0.094324	85
SWDD	while Replacement by do-while	0.032363	1
VGPR	Mutate Global Pointer References	-0.091281	13
(Intercept)		-0.036398	

Table 3: Sufficient set and linear model resulting from cost-based LARS analysis of the mutation operators.

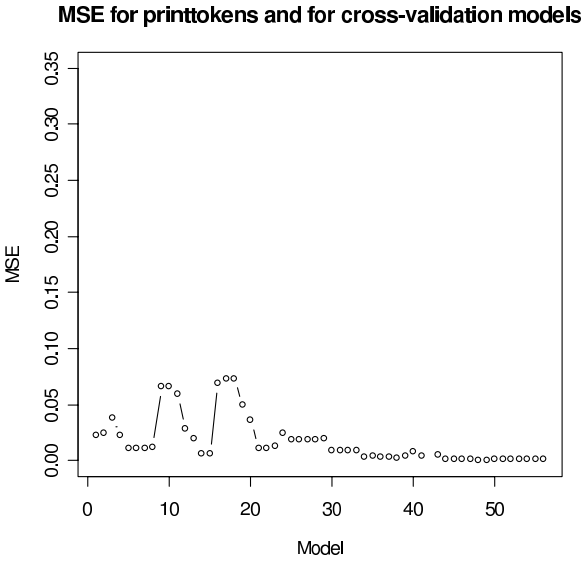

Figure 2: Plot of model number (x-axis) vs. MSE (y-axis) for the sequence of cross-validation models generated by the CBLARS procedure on the `printtokens` subject program.

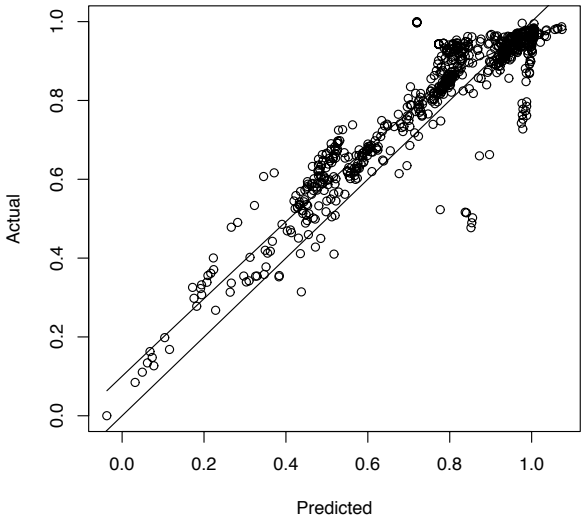

Figure 3: Values for AM predicted by cross-validation models vs. actual values for AM, with $x = y$ line and smoothing spline trend line. The graph shows all data points from all subject programs.

Program	MSE	r^2	Spearman	Kendall
`printtokens`	0.008	0.930	0.769	0.601
`printtokens2`	0.003	0.924	0.967	0.849
`replace`	0.008	0.963	0.973	0.876
`schedule`	0.003	0.747	0.873	0.725
`schedule2`	< 0.001	0.982	0.911	0.755
`tcas`	0.022	0.936	0.965	0.866
`totinfo`	0.015	0.841	0.900	0.765

Table 4: Goodness-of-fit measures on cross-validation models. Each row represents the model trained on the other programs and applied to the indicated program.

is in comparing testing techniques, we also computed the Spearman and Kendall rank correlations, as measures of the degree to which the predicted AM would rank the effectiveness of two given testing techniques correctly.

The results, for each of the seven cross-validation models, are in Table 4. The r^2 values are never lower than 0.747, and the Spearman and Kendall correlations are low only in the case of the `printtokens` subject program, for which most predicted and actual values of AM are clustered around 0.8, and for which MSE was low and r^2 high. These values indicate that overfitting was not a major problem when applying the stated procedure to the data from any six of the programs. This in turn lends support to the belief that the model based on the data from all seven programs is not overfitted, and that it would be a reasonable model to use on other C programs.

Figure 3 offers a visualization of this data, by combining the data from all seven cross-validation models into one figure. Each point represents the predicted and actual values of AM for some test suite of some subject program.

4.4 Other Subset Selection Procedures

To further validate our sufficient set selection procedure, we performed two other statistical procedures from the literature [16, 17] for the variable reduction problem. These procedures took cost into account, but did not attempt to fit an optimal linear model. They therefore provide a fresh perspective on the problem.

In Elimination-Based Correlation Analysis (EBCA), we repeatedly identify which pair of predictor variables is most highly correlated with each other, and we eliminate the one that has a higher cost. We continue the procedure until no pair of variables has a correlation higher than 0.90, the "very high" level from the standard Guilford scale [13].

Cluster Analysis (CA) represents the level of similarity of variables by a dendrogram, a binary tree with variables at its leaves such that more similar variables are grouped closer together. (R is capable of generating such dendrograms.) Generally, two variables whose lowest common ancestor node in a CA dendrogram is n levels up are more closely related than two whose lowest common ancestor node is $m > n$ levels up. When applying CA to the variable elimination problem, we identify sibling pairs of variables in the dendrogram that have correlation greater than 0.90 and eliminate the one that has a higher cost. This may result in a somewhat different set of variables from that given by EBCA, since the elimination of a variable may make two variables new siblings.

Performing EBCA and CA on our data yielded one sufficient set for each procedure. Both sets were strict supersets of the set identified by CBLARS, indicating that CBLARS was indeed identifying a core useful set of operators.

As with the CBLARS procedure (see Section 4.3), we performed sevenfold cross-validation on the EBCA and CA procedures, to see how cost-effectively they produced a sufficient set compared to CBLARS. For each training set, we generated the EBCA and CA sufficient sets, and then performed a linear regression to get the best model resulting from the sets. We then measured the accuracy of the model using the Mean Squared Error (MSE), and the cost of the models using the percentage of mutants generated.

Figure 4 shows the results. CBLARS had a high MSE for one of the subject programs (`tcas`), but on average its MSE was not much greater than that of the other two methods. However, the CBLARS sufficient set generated over 30% fewer mutants on average than either of the other techniques. This result lends support to the belief that our CBLARS subset selection procedure made a good tradeoff between cost and accuracy of the selected subset.

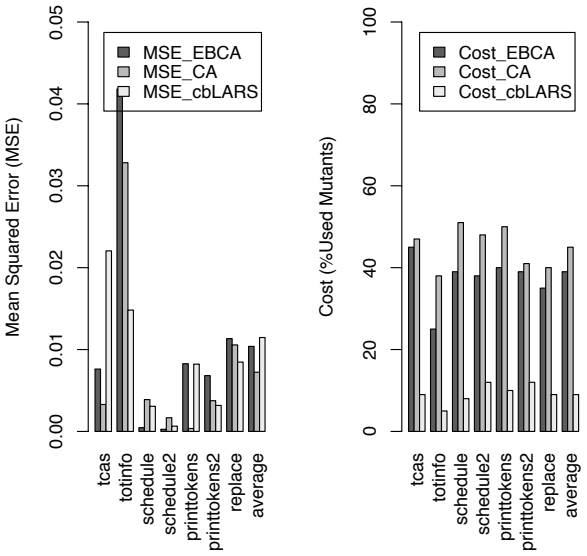

Figure 4: Comparison of mean squared error (left) and cost (right) of cross-validation models produced by CBLARS, EBCA and CA.

4.5 Summary

Our cross-validation indicates that the procedure for applying CBLARS described in Section 4.2, when applied to a set of data from six of our subject programs, yields an accurate prediction of AM scores for test suites of the seventh subject program. This in turn implies that our procedure for applying CBLARS, when applied to the data from all seven subject programs, would yield an accurate prediction of AM scores for test suites of some other, arbitrary eighth program.

We therefore believe that researchers faced with the problem of efficiently evaluating the mutation adequacy ratio of a test suite S of a program P can get an accurate estimate by carrying out the following procedure:

- Generate the mutants of P from the 28 operators listed in Table 3.
- Run S on all the resulting mutants.
- Measure the Am_i ratios of S for each class of mutants.
- Apply the linear model in Table 3 to get an estimate of AM.

5. DISCUSSION

In this section, we compare our results to those of previous researchers, give observations on the set of 28 operators selected by our procedure, and discuss threats to the validity of our empirical results.

5.1 Comparison to Previous Results

It is difficult to compare our results directly to the previous work on the sufficient set problem, since we had somewhat different goals. However, some general principles can be observed.

In the work of Wong [26], Offutt et al. [20], and Barbosa et al. [4], the measure of goodness of a sufficient set σ was the average AM of a test suite that achieved $AMS_\sigma = 1.0$. In our work, the goodness of a sufficient set is measured instead by measures such as the adjusted R^2 of the associated model. The results are therefore not comparable directly.

However, we did observe that in the cross-validation, even without applying the linear model – i.e. computing AMS on the sufficient set – when the value of AMS was 1.0, the value of AM was usually very high. This suggests that by generating the mutants from the 28 identified operators and adding enough test cases to kill all of them, a tester is likely to obtain a test suite that would kill a high percentage of the mutants that would be generated by all operators.

In terms of cost saving, we report a reduction of approximately 92.6% in the number of mutants generated by only the sufficient set of operators. Although this is a higher saving than the highest reported so far (82.54% by Wong), it applies to a much larger set of mutation operators (108 as opposed to the 22 of Wong's study). However, the procedure followed in this paper considers all possible subsets of operators, so it is likely that the results would be competitive with previous results even if we had started with a smaller initial set of operators.

5.2 Mutation Operators Selected

Here we make some qualitative observations about the mutation operators selected by the statistical procedure we followed.

We note that in all the selection procedures that we followed, whether LARS, EBCA or CA, no mutation operator that mutated constants was ever selected. For instance, the operator Cccr, which replaces one integer or floating-point constant by another from the program of the same type, was never selected in any CBLARS model, and was eliminated by EBCA and CA. This contradicts conventional wisdom on the sufficient set problem, which suggests that one operator from each operator category (i.e., constant, statement, variable and operator) should be chosen [4].

The immediate cause of the lack of constant mutation operators is that the Am_i of each of the constant mutation operators was highly correlated with the Am_j of some other operator, but the constant mutation operators generated more mutants. The deeper cause may be that the constant mutation operators produce many mutants, each of which has a very similar behaviour to some mutant resulting from mutating some source code relational operator or arithmetic operator.

We also note the high proportion of negation-like mutation operators in the sufficient set, such as ArgLogNeg ("insert logical negation on argument"). Of the 108 Proteum operators, only 28 were selected for the sufficient set, or 26%. However, of the 12 Proteum operators that mention negation in their description, six were selected, or 50%. This may be because such an operator generates only one mutant per location, as opposed to, for instance, a mutation that replaces an arithmetic operator with one out of a set of arithmetic operators.

5.3 Threats to Validity

Threats to external validity include the use of C programs that are still relatively small compared to commercial programs. This threat is mitigated by the facts that the C programs are large and complex enough to include a broad range of control and data structures, and that the three dominant languages in programming today (C, C++ and Java) all use very similar syntax in their non-OO constructs. However, larger programs may have more complex control and data flow patterns, which may lead to different results. We also note that we have not attempted to handle object-oriented constructs. Mutant generators that implement class mutation operators, such as MuJava [18], are better suited to evaluation of sufficient mutation operator sets for object-oriented programs.

A threat to construct validity is the selection of 2000 mutants from each program other than `tcas`. This was necessary to make

the study feasible, but could have resulted in decreased accuracy of measurement. We mitigated this risk by ensuring that the same proportion of mutants from each operator was chosen, and by running all test cases on the selected mutants in order to accurately calculate the AM and Am_i measures for all generated test suites relative to the selected mutants.

Threats to internal validity include the correctness of the mutant generation, compilation, running and data collection processes. We rely on Proteum for mutant generation, and minimize the other threats to internal validity by reviewing our data-collection shell scripts and doing sanity checks on our results.

6. CONCLUSIONS AND FUTURE WORK

Using statistical methods, we have identified a subset of the comprehensive Proteum mutation operators that generates less than 8% of the mutants generated by the full set, but that allows us to accurately predict the effectiveness of a test suite on the mutants generated from the full set of operators. Cross-validation indicated that the procedure that we used to identify the set was reasonable and that our results would extend to other programs. Using the model, researchers can therefore accurately estimate the effectiveness of a test suite on a program without having to generate all of the mutants of the program.

This research is part of a long-term goal of supplying researchers and practitioners with tools to allow them to accurately measure the effectiveness of testing techniques. By decreasing the cost of mutation analysis, our work makes this method of effectiveness measurement open to more researchers and may ultimately lead to standard ways of measuring testing technique effectiveness. The qualitative observations we made about the sufficient set we found – for instance, the absence of constant mutation operators and the presence of operators that tend to produce few mutants at each location – may also help in designing a set of mutation operators suitable for practitioners to use in mutation testing.

Future work includes the application of the same procedure to test suites constructed to achieve given coverage measures, in order to ensure that the sufficient set identified is not specific to randomly-selected suites. We also plan further repetitions and validations of the proposed subset selection procedures, and extension of the experiments to mutation operators specific to object-oriented programs.

7. ACKNOWLEDGEMENTS

Thanks for useful comments and suggestions to the faculty advisors at the Doctoral Symposium at ICSE 2007, in particular to David Notkin, Mauro Pezze, David Rosenblum, Barbara Ryder, and Mary Shaw. Thanks to Hyunsook Do and Gregg Rothermel for their help in accessing the Siemens programs, and to José Carlos Maldonado and Auri Vincenzi for access to Proteum. Thanks also to Aditya Mathur for bibliographic references and useful discussion. The R statistical package [25] was used for all statistical processing. This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and an Ontario Graduate Scholarship.

8. REFERENCES

[1] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Department of Computer Science, Purdue University, Lafayette, Indiana, April 2006.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, St. Louis, Missouri, May 2005. 402-411.

[3] J. H. Andrews, L. C. Briand, Y. Labiche, and A. Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, August 2006.

[4] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for C. *Software Testing, Verification and Reliability*, 11:113–136, 2001.

[5] L. C. Briand, Y. Labiche, and M. M. Sówka. Automated, contract-based user testing of commercial-off-the-shelf components. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, pages 92–101, Shanghai, China, 2006.

[6] L. C. Briand, Y. Labiche, and Y. Wang. Using simulation to empirically investigate test coverage criteria based on statechart. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 86–95, Edinburgh, UK, May 2004.

[7] M. E. Delamaro and J. C. Maldonado. Proteum – a tool for the assessment of test adequacy for C programs. In *Proceedings of the Conference on Performability in Computing Systems (PCS 96)*, pages 79–95, New Brunswick, NJ, July 1996.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11:34–41, April 1978.

[9] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, 2006.

[10] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. Least angle regression. *Annals of Statistics*, 32(2):407–499, 2004.

[11] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 153–162, Lake Buena Vista, FL, USA, 1998.

[12] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

[13] J. P. Guilford. *Fundamental Statistics in Psychology and Education*. McGraw-Hill, New York, 1956.

[14] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[15] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering (ICSE 1994)*, pages 191–200, Sorrento, Italy, May 1994.

[16] I. Jolliffe. Disgarding variables in a principal component analysis. I: Artificial data. *Applied Statistics*, 21(2):160–173, 1972.

[17] I. Jolliffe. Disgarding variables in a principal component analysis. II: Real data. *Applied Statistics*, 22(1):21–31, 1973.

[18] Y.-S. Ma, J. Offutt, and Y. R. Kwon. MuJava : An automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, June 2005.

[19] J. Mayer and C. Schneckenburger. An empirical analysis and comparison of random testing techniques. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 105–114, Rio de Janeiro, Brazil, 2006.

[20] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, April 1996.

[21] A. J. Offutt and R. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, San Jose, CA, October 2000.

[22] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of the International Conference on Software Maintenance (ICSM '98)*, pages 34–43, Washington, DC, USA, November 1998.

[23] A. Siami Namin and J. H. Andrews. Finding sufficient mutation operators via variable reduction. In *Mutation 2006*, Raleigh, NC, USA, November 2006.

[24] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva. Mutating database queries. *Information and Software Technology*, 49(4):398–417, 2007.

[25] W. N. Venables, D. M. Smith, and The R Development Core Team. An introduction to R. Technical report, R Development Core Team, June 2006.

[26] W. E. Wong. *On Mutation and Data Flow*. PhD thesis, Purdue University, December 1993.