Finding Sufficient Mutation Operators via Variable Reduction

Akbar Siami Namin and James H. Andrews
Department of Computer Science
University of Western Ontario
London, Ontario, CANADA N6A 5B7
Email: {asiamina,andrews} (at) csd.uwo.ca

***Abstract*— A set of mutation operators is "sufficient" if it can be used for most purposes to replace a larger set. We describe in detail an experimental procedure for determining a set of sufficient C language mutation operators. We also describe several statistical analyses that determine sufficient subsets with respect to several different criteria, based on standard techniques for variable reduction. We have begun to carry out our experimental procedure on seven standard C subject programs. We present preliminary results that indicate that the procedure and analyses are feasible and yield useful information.**

I. Introduction

When performing mutation testing or mutation analysis, we apply *mutation operators* to programs in order to create faulty versions of those programs. Typically, a mutation operator can be applied at many different locations in the source code or program structure graph, and may yield more than one mutant from each location. Hence, applying one mutation operator to one source program can result in more than one mutant, and may result in many mutants.

Over the years, many mutation operators have been proposed. Generating all mutants from all proposed operators would often result in an infeasibly large number of mutants. Researchers have therefore looked for a reduced set of mutation operators that will generate fewer mutants but still be "sufficient" for our purposes.

Offutt et al. [1] cast the sufficient mutation operators problem as follows: Given a set P of mutation operators, find a subset Q of P such that Q generates "many fewer" mutants than P, but such that a test suite that kills all the mutants from Q tends to kill "most" of the mutants from P. The phrases in quotation marks are necessarily imprecise, and how good the set Q is must be a matter of judgement.

In this paper, we generalize the question as follows: Given a set P of mutation operators, find a subset Q of P such that Q generates "many fewer" mutants than P, but such that from the behaviour of a test suite on the mutants from Q, we can obtain an "accurate" prediction of its behaviour on the mutants from P. We find the subset Q by using several *variable reduction* techniques based on the statistical literature, and combining the results of all the techniques. We use C programs as our experimental subjects rather than the Fortran programs of some previous studies. The purpose of this workshop paper is to describe our experimental technique and statistical analyses in detail, and to present some preliminary results based on one subject program.

A. Structure of Paper

The remainder of this paper is structured as follows. In Section II, we discuss previous work. In Section III, we present our procedure for obtaining the raw data on which we base our analyses. In Section IV, we describe the statistical analyses that we perform for obtaining our set of sufficient mutation operators. In Section V, we present our preliminary results and analyze how well the proposed set performs. In Section VI, we discuss the threats to validity and the implications of our results, and in Section VII we conclude.

II. Previous Work

A. Mutation Testing and Analysis

Mutation testing [2] needs no introduction to the participants of this workshop. We use the related phrase *mutation analysis* to refer to the practice of using mutants in experiments to analyze the effectiveness of software testing techniques. In a testing experiment, we typically want to see how effective and efficient different testing techniques are. Our measure of effectiveness is how many faults the technique finds. Mutation has been used by many researchers in order to generate faulty versions for such experiments, and recent work has shown that these mutants can behave very similarly to real faults [3].

Mutant generator systems that have been written and made available include Mothra [4] for Fortran programs, Proteum [5] for C programs, and MuJava [6] for Java programs. Mothra implements 22 mutation operators, while Proteum implements a wider set of 108 mutation operators. MuJava implements only the 5 conventional (non-OO) operators found by the Offutt et al. study [1], but also implements 24 class mutation operators. Since we are interested in revisiting the question of sufficient conventional mutation operators, Proteum is our best available choice.

B. Sufficient Mutation Operators

Previous empirical work on sufficient mutation operators is limited. Wong [7] compared a set of 22 mutation operators to a set of two mutation operators in two experiments, and found that the reduced set was significantly smaller but behaved similarly to the full set. Offutt et al. [1] compared a 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 created to kill all the mutants generated from that set of operators had high mutation adequacy ratios (average 99.51%) while requiring 77.56% fewer mutants on average.

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 an average of approximately 57.9 raw lines of code each (lines including comments and whitespace). The subjects in the Offutt et al. study were 10 Fortran programs having an average of 20.6 statements each. The Wong subjects have no `struct` definitions and very few pointers, and the descriptions of the Offutt et al. subjects suggest that they also have no or few structures or pointers. The significance of this is that pointer and structure field assignment statements (a) are affected by relatively few of the commonly-described mutation operators (in particular, they are affected by none of the operators in the sufficient sets identified by either Wong or Offutt et al), (b) are common in larger C programs, and (c) correspond to object field assignment statements, which are common in larger O-O programs.

The results of the previous experiments might not carry over to mutation adequacy for typical programs in C for a number of reasons. These reasons include the syntactic differences between Fortran and C, the differences between small and larger programs, and the differences between programs manipulating structures and pointers and those doing so very little or not at all. The only way to confirm whether the results do carry over is to do a new experiment.

The previous work on the subject also does not deal with the question of the all-mutants adequacy of test suites that achieve sufficient-mutant adequacy ratios of *less* than 100%. For instance, while we might know that test suites achieving 100% sufficient-mutant adequacy achieve an overall adequacy of 99.51%, this does not necessarily mean that test suites achieving 70% sufficient-mutant adequacy achieve an average all-mutants adequacy of 69.51%.

C. Variable Reduction

In our experiments, we apply a set of 108 mutation operators to programs, and measure the adequacy of test suites on the mutants resulting from each operator, and on all mutants combined. We therefore have a set of 109 experimental variables (the per-operator mutation adequacy ratios and the overall mutation adequacy ratio), and we want to choose a subset of the first 108 that allows us to predict the 109th. This problem is known in statistics as the *variable elimination, discarding, disregarding, selection* or *reduction* problem. We will refer to it by the term *variable reduction* in this paper.

Some multivariate techniques to decrease the number of variables have been proposed. Among them principal component analysis (PCA) [8] is the first one, in which we reduce the dimension space of variables into some new non-correlated variables. The scales of new variables are perpendicular to each other such that the first scale, as first component, shows a linear combination of the original correlated variables, whose contribution toward variance are the highest. PCA does not help us directly, since it still assumes that we collect all variables and then summarize them in a smaller set of variables. However, the variable reduction techniques developed for PCA are relevant.

Jolliffe in his papers [9] and [10] introduced some variable reduction methods to be used in PCA. He introduced eight different approaches in three groups, referred to as correlation analysis, principal component analysis, and cluster analysis [11]. It has been claimed that all eight different techniques produce the same set of operators, plus or minus one. However, it has been proven that the variable reduction problem has a non-unique solution [12].

[9] and [10] present some fundamental skeleton approaches for variable reduction techniques. These approaches must be tailored to a particular problem's criteria for preferring one variable over another, and this is what we did for the sufficient mutation operator problem. To our knowledge, this approach to the sufficient mutation operator problem is new.

III. Experimental Procedure

We take as our primary goal to discover a subset of mutation operators, generating a small number of mutants, such that the adequacy of a test suite on the subset of mutants can be used to accurately predict the adequacy of the test suite on all mutants. Our subsidiary goals are to consider as large and complex programs as are experimentally feasible, and to use as many approaches as possible to constructing the test suites. Here we describe the experimental procedure used to collect the raw data. We first describe the setup of the subject programs and the mutant generator. We then define precisely what sets of data we produce for each group of test suites, and what measurements we take from them. We describe the groups of test suites that we generate, and end by stating how we combine data from different subject programs[1].

A. Setup

Our experimental subject programs are the seven well-known Siemens programs, used first by Hutchins et al. [13] and developed further as research instruments by Rothermel, Harrold and others [14]. We obtained the Siemens programs from the Galileo Research Group Subject Infrastructure Repository (SIR) at the University of Nebraska - Lincoln. Each of the programs comes with a large number of test cases, referred to as the *test pool*. Each also comes with alternate versions containing faults, which were not needed for this study.

The Siemens programs have an average of approximately 327 NLOC (lines of code without comments or whitespace). This number of lines of code is bigger than that of previous studies. Furthermore, three of the Siemens programs use

[1]Rather than describing our finished, in-progress and projected work in a mixture of past, present and future tenses, we describe it all in the present tense, and then make clear in Section V what we have done so far.

structures (C `struct`s), and five use pointers in a non-trivial manner. However, the programs are still smaller than most C programs in use today, since the heavy processing requirements of our experiments constrain their size. We do plan to evaluate the sufficient mutation operator sets that we obtain on larger programs in the future.

We obtained the C mutant generator Proteum from José Carlos Maldonado at the University of São Paolo. Proteum defines $n = 108$ mutation operators. We refer to these mutation operators as μ_1 through μ_n. Examples include ORRN, the operator for replacing one relational operator by another, and SBRC, the operator for replacing a `break` statement with a `continue` statement. The operators overlap somewhat with the 76 operators defined for C by Agrawal et al. [15], and other operators are as described by Delamaro et al. [16] and Vincenzi et al. [17].

B. *Definitions of Sets and Measurements*

For each subject program P, we collect and calculate a number of measures and sets in our experiments. Here we define these measures and sets precisely.

Examples of measures and sets defined below include "$\text{NMG}_i(P)$" and "$\text{outcome}(M, C, P)$". Because the subject program P is a parameter of all these measures and sets, we simplify our notation by leaving out P as a parameter wherever possible. Thus, for instance, we speak of NMG_i and $\text{outcome}(M, C)$, leaving parameter P implicit.

Generally we use upper-case names (e.g. "NMG") for numbers, and lower-case names (e.g. "outcome") for non-numeric entities such as sets.

We generate mutants of P using each of the $n = 108$ Proteum mutation operators $\mu_1, \dots, \mu_n$.

- NMG_i is defined as the number of mutants of P generated using μ_i.
- $\text{class}(M)$, for a given mutant M of P, is defined as the i such that M is one of the mutants generated by applying mutation operator μ_i to P. In our experiments, $\text{class}(M)$ can be retrieved directly from the name assigned to the mutant.

A small number of the generated mutants do not compile, or do not link properly.

- NMC_i is defined as the number of mutants of P generated using μ_i that actually compiled and linked.

We run each compilable mutant M of P on each test case C in the test pool for P.

- $\text{outcome}(M, C) = t$ if the output of M on test case C is identical to the output of the gold version of P on test case C; otherwise $\text{outcome}(M, C) = f$.
- $\text{failures}(M)$ is the set of test cases C such that $\text{outcome}(M, C) = f$. We do not store $\text{outcome}(M, C)$ directly, but rather store the set $\text{failures}(M)$ in a file specific to that mutant M.
- We say that M is an *equivalent* mutant if $\text{failures}(M)$ is the empty set; otherwise we say that M is *non-equivalent*.
- NMNE_i is the number of mutants of P generated using μ_i that are non-equivalent.
- TNMNE is the total number of mutants that are non-equivalent; i.e. $\text{TNMNE} = \sum_{i=1}^{n} \text{NMNE}_i$.
- We say that test case C *kills* mutant M, or M is killed by C, if $\text{outcome}(M, C) = f$.
- $\text{kills}_i(C)$, for a test case C, is the set of mutants M generated using μ_i that are killed by C.
- $\text{kills}(C)$, for a test case C, is the set of all mutants killed by C; i.e. $\text{kills}(C) = \bigcup_{i=1}^{n} \text{kills}_i(C)$.

For each test suite $S = \{C_1, \dots, C_m\}$ that we generate, we calculate the most important sets and numbers of our experiments.

- $\text{kills}_i(S)$ is the set of mutants M generated using μ_i that are killed by some test case in S; i.e. $\text{kills}_i(S) = \bigcup_{j=1}^{m} \text{kills}_i(C_j)$.
- $\text{kills}(S)$ is the set of all mutants that are killed by some test case in S; i.e. $\text{kills}(S) = \bigcup_{j=1}^{m} \text{kills}(C_j)$, or equivalently $\text{kills}(S) = \bigcup_{i=1}^{n} \text{kills}_i(S)$.
- $\text{Am}_i(S)$, the Adequacy ratio for Mutants of operator μ_i for test suite S, is defined as $|\text{kills}_i(S)|/\text{NMNE}_i$.
- $\text{AM}(S)$, the Adequacy ratio for all Mutants, is defined as $|\text{kills}(S)|/\text{TNMNE}$.

Note that we define the adequacy ratio relative to the number of non-equivalent mutants. We do this because our primary interest is in the use of mutants in experiments, and in the context of experiments it is both feasible and desirable to accurately determine which mutants are equivalent.

C. *Test Suite Groups*

Ideally, for a given test suite S, the number of sufficient-set mutants that S kills would let us predict accurately the number of all mutants that S kills. However, there are many different ways to build test suites, and a set of sufficient operators for one of these ways might not suffice for another way. We therefore discover sufficient operators with respect to various ways of constructing test suites, and later compare the discovered sets of operators.

For each subject program, we generate nine groups of test suites: one group of singleton test suites, referred to as SING, four groups of randomly-selected test suites, referred to as RAND10, RAND20, RAND50, and RAND100, and four groups of coverage-based test suites, referred to as BLK, DEC, CUSE, and PUSE. We will now describe these groups in turn.

SING, for each subject program, is a group of 100 test suites consisting of one test case in each suite. That is, to construct SING, we choose 100 distinct test cases from the test pool, and put each in a separate test suite. We discover sufficient operators for this group in order to see if the set of sufficient operators for test suites can also be used to evaluate individual test cases.

The test suites in RAND10 are generated by randomly picking 10 test cases from the test pool for the subject program. We generate 100 test suites for each subject program in this way. These suites represent "arbitrary" test suites not constructed by following any coverage goals. The test suites in RAND20 (resp. RAND50, RAND100) are similarly generated by randomly picking 20 (resp. 50, 100) test cases from the test pool.

The test suites in BLK are generated to meet particular coverage goals. For each coverage percentage P from 50 to 99, we generate two test suites that achieve between P% and $P+1$% feasible block coverage, as measured by the ATAC coverage tool. DEC, CUSE, and PUSE are generated in the analogous way for decision, C-use and P-use coverage respectively. These suites represent test suites that may reasonably be constructed by trying to achieve high code coverage. We select a spread of many coverage percentages because we want to discover sufficient operators that work over a broad range of coverage goals, not just close to 100%. We start at 50% coverage because this seems to be the minimum goal it is likely anyone would want to achieve. We stop at 99% coverage because preliminary investigations have indicated that it would take an unreasonably long time to generate test suites that achieve 100% feasible coverage for the two hardest measures (C-use and P-use). Since we generate two test suites for each coverage percentage, the size of each group is again 100 test suites.

D. Combining Data from Subject Programs

Note that each group of test suites for each subject program contains 100 test suites. It would be infeasible to perform our statistical analyses for each of these test suite groups and each of the subject programs. We therefore combine the data for the seven subject programs. We calculate the values of Am_i and AM for each test suite, and then combine the data from the different subject programs but the same group into one set of data. Thus, for each of SING, RAND10, RAND20, RAND50, RAND100, BLK, DEC, CUSE, and PUSE, we obtain 700 observations, each observation consisting of values for each Am_i and for AM.

For each of the nine groups of test suites, we do three separate analyses on the data generated from the experiment. The goal of each analysis is to discover sufficient mutation operators, but the procedure differs between the analyses. These analyses are described in the next section. We thus end up with 27 sets of mutation operators, each judged as "sufficient" by a different analysis or with respect to a different group of test suites. We then compare these 27 sets of mutation operators in order to identify the operators that appear the most often.

IV. Statistical Analysis

We base our three variable reduction techniques on those introduced in [9] and [10], adapting them whenever it comes to the choice of rejecting or keeping a mutation operator. Here we call the techniques All-Subsets Regression analysis (SUB), Elimination-Based Correlation (EBC) analysis, and Cluster Analysis (CA).

A. All-Subsets Regression (SUB)

One valid view of the sufficient-operators problem is that it is merely a linear regression problem. We have 108 variables (the Am_i variables), and we are trying to find a small subset of those 108 that leads to the best linear model for the 109th (AM). This is known as the all-subsets regression problem. Thus, our first analysis translates the problem into an all-subsets regression problem.

We must choose a target number of operators for this analysis. Since Wong and Offutt et al. identified a subset of 9%-23% of the operators available to them, we set a target number of 20 operators (19%) out of the 108 operators available to us. We perform an all-subsets regression, locating the set of 20 or fewer operators that yields the best linear model of AM.

The open-source R statistical package [18] implements several algorithms for all-subsets regression. For our problem, it is infeasible to use the exhaustive method, so we use the "forward" method, which yields good models but may not yield all the best models.

B. Elimination-Based Correlation Technique (EBC)

A possible criticism of the all-subsets regression is that it may tend to unreasonably favour mutation operators that generate many mutants for a typical subject program. For instance, since ORRN replaces one relational operator by another, it typically generates more mutants than SBRC (replace `break` statement by `continue` statement), thus contributing more to AM for a typical suite. This may lead to ORRN being automatically chosen just because of this higher number of mutants generated.

A possible rejoinder to this criticism is that if ORRN generates many mutants, this may indicate that the corresponding faults are more likely to happen. From that point of view, the ORRN mutants would be better predictors of faulty behaviour in a real program, and so it is justified to weight them more heavily. However, for safety, and in order to reduce the number of mutants generated by the sufficient operator set, we also perform other analyses designed to address the possible criticism.

The second technique of correlation analysis (Elimination-Based or EBC) considers the correlation value between two mutation operators in the decision to reject or keep the operators. Our approach is slightly different from the one in [9] and [10] since, in that approach, the correlation values between a particular variable and *all* other variables will be computed and the variable with the highest correlation values with *other variables* will be rejected. However, in our approach, we consider the highest correlation value between *two* variables and reject one of them.

This technique makes a decision to reject an operator based on the number of generated mutation operators, rejecting the one whose number of generated mutants is higher. Although EBC is intuitive, there is no guarantee of constructing a subset of variables that will achieve the best solution. But as a general rule, it is essential to achieve the desired results in several different ways in order to draw a conclusion. EBC is given as Algorithm 1.

The algorithm starts by constructing the correlation matrix among mutation operators, by using their Am_i values. Then

Algorithm 1 Elimination-Based Correlation Analysis (EBC)

Input: Data from a group TS of test suites
Output: A set suf of sufficient mutation operators

```
1:  $M \leftarrow \{\mu_i | 1 \leq i \leq 108\}$          ▷ All mutation operators
2:  $suf \leftarrow \emptyset$
3:  Construct the correlation matrix **S**, for values of $Am_i$, between mutation operators $\mu_i \in M$
4:  while $\exists \mu_i, \mu_j \in M$ s.t. $|cor(Am_i, Am_j)| > 0.9$ do
5:      $(\mu_m, \mu_n) \leftarrow \{(\mu_m, \mu_n)| \; |cor(Am_m, Am_n)| = Max\{|cor(Am_m, Am_n)|\}$
6:      if $\sharp mutants_{\mu_m} \neq \sharp mutants_{\mu_n}$ then
7:          Remove from $M$ the one with more generated mutants, and place it in $suf$
8:      else
9:          if $|cor(Am_m, AM)| \neq |cor(Am_n, AM)|$ then
10:             Remove from $M$ the one which has less correlation with AM, and place it in $suf$
11:         else
12:             Remove one of $\mu_m, \mu_n$ randomly from $M$ and place it in $suf$
13:         end if
14:     end if
15: end while
16: Return $suf$
```

it proceeds while there are two operators whose correlation is greater than the *threshold* value of 0.9 (considered to be "very high" correlation by the standard Guilford scale [19]). The algorithm then rejects the operator whose number of generated mutants is greater.

As is shown in the algorithm, there might be some cases in which not only the correlation value between two operators is over the threshold value, but also the number of generated mutants of both operators are equal. In those cases, we see which one of those operators has more correlation with AM, the adequacy ratio for all mutants. We took into account the comparison with AM in order to have better model in the end.

C. Cluster Analysis (CA)

We also apply cluster analysis [11] to get another picture of our data pattern. The goal of cluster analysis is to develop a classification scheme that will partition the rows of a data matrix into k distinct groups or clusters [20]. Since we were interested in clustering mutation operators (variables) rather than test suites (observations), we computed the transpose matrix of the data matrix and treated the mutation operators as observations.

Cluster analysis represents the level of similarity of variables by a dendrogram, a binary tree with variables at its leaves. (R is also capable of generating such dendrograms.) Generally, two variables whose lowest common ancestor node is n levels up are more closely related than two whose lowest common ancestor node is $m > n$ levels up. The length of branches in the dendrogram represents how closely related two sibling nodes are. Figure 1 shows one such dendrogram, derived from the `tcas` program and the RAND100 group of test suites. It tells us, for instance, that the two rightmost operators (DirVarIncDec and OABN) are more closely related to each other than either is to any other mutation operator. However, it also tells us that they are less closely related to each other than the next two operators in (OLAN and VDTR) are to each other.

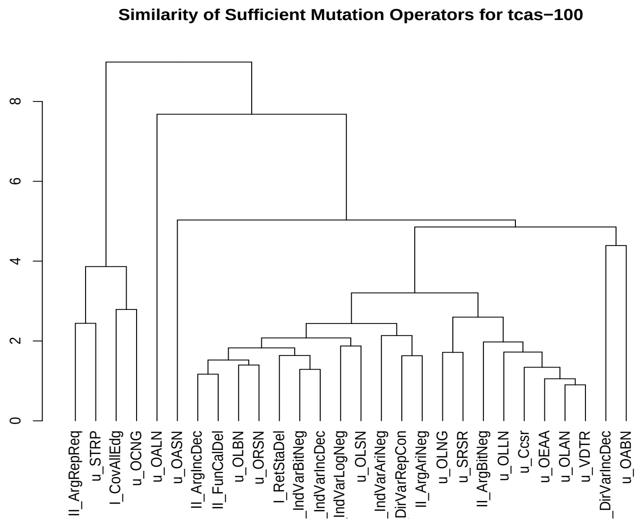

Fig. 1. Dendrogram for similarity of sufficient mutation operators for `tcas` with test suite size 100

By clustering two variables in a specific cluster with high similarity rate, one is able to reject one of the variables in favor of the other, therefore reducing the number of clusters and, consequently, the number of variables. While applying CA to our data, we faced two questions:

1) How deeply we can proceed in the elimination of variables (mutation operators) in a dendrogram?
2) Which operator is the best choice for elimination?

Addressing (1), it is not our goal to end up with a single cluster with just two operators. We need to set a condition between two operators in order to measure the relationship between them in a particular cluster. We again take the correlation coefficient between the variables into account in this. We start from a leaf cluster and measure the correlation value between its operators, again rejecting a variable if the correlation between them is higher than 0.9. The technique proceeds until the correlation value between the pair in each leaf cluster is over 0.9. Addressing (2), we again considered the operator with the highest number of generated mutants as the best candidate for rejection. CA is described in full as Algorithm 2.

V. Preliminary Results

We have applied our procedure so far only to `tcas`, the smallest and simplest of the seven Siemens programs, and generated only the test suites in SING, RAND10, RAND20, RAND50, and RAND100. To compensate for the lack of data

Algorithm 2 Cluster Analysis (CA)

```
Input: Data from a group $TS$ of test suites
Output: A set $suf$ of sufficient mutation operators
 1: $M \leftarrow \{\mu_i | 1 \leq i \leq 108\}$    ▷ All mutation operators
 2: $D_s \leftarrow$ the data matrix
 3: $D_s^t \leftarrow trans(D_s)$    ▷ the transpose of $D$
 4: repeat
 5:     $dend \leftarrow CA_{D_s^t}$    ▷ Cluster Analysis on $D_s^t$
 6:     $PAIRS \leftarrow \{c | c \in dend$ s.t. $|c| = 2\}$    ▷ the set of clusters of length 2 in dendrogram
 7:     for each cluster $c \in PAIRS$ do
 8:         Identify $\mu_1, \mu_2 \in c$    ▷ $|c| = 2$
 9:         if $|cor(Am_{\mu_1}, Am_{\mu_2})| \geq 0.9$ then
10:             if $\sharp mutants_{op_1} \neq \sharp mutants_{op_2}$ then
11:                 Reject the one with more generated mutants from $D_s^t$
12:             else
13:                 if $|cor(Am_{\mu_1}, AM)| \neq |cor(Am_{\mu_2}, AM)|$ then
14:                     Reject the one which has less correlation with AM from $D_s^t$
15:                 else
16:                     Reject one of the $\mu_1, \mu_2$ from $D_s^t$ randomly
17:                 end if
18:             end if
19:         end if
20:     end for
21: until $\exists \mu_1, \mu_2 \in M$ s.t. $|cor(Am_{\mu_1}, Am_{\mu_2})| \geq 0.9$
22: $suf \leftarrow \{\mu \in D_s^t\}$
23: Return $suf$
```

from the other six programs, we generated 300 rather than 100 test suites in each group. For this workshop paper, we report on the results of the analyses and our explorations of how we can combine the sets of mutation operators derived from them. The results support the conclusion that our procedure is experimentally feasible and yields informative data regarding sufficient mutation operators.

We began by generating mutants for `tcas`. 4937 mutants were generated by Proteum. For 49 of the operators, no mutants were generated, so we had for our analyses only 59 variables to deal with. 4935 of the mutants compiled, and none were equivalent to the original program (thus TNMNE = 4935).

A. SUB Analysis

For each group of test suites (SING, RAND10, RAND20, RAND50, and RAND100), running all-subsets regression in R yielded a list of linear models based on 20 or fewer operators. R also reported what correlation each model achieved with the data from the test suite group. Even though we used the incomplete "forward" method for all-subsets regression, there were over 20 very good models for each test suite group (correlation with AM of 0.995 or greater).

Operator / Description	♯mutants
I-DirVarAriNeg	44
Inserts Arithmetic Negation at Interface Variables	
I-DirVarRepReq	220
Replaces Interface Variables by Required Constants	
I-IndVarLogNeg	19
Inserts Logical Negation at Non Interface Variables	
I-IndVarRepReq	91
Replaces Non Interface Variables by Required Constants	
u-OABN	3
Arithmetic by Bitwise Operator	
u-OLSN	34
Logical Operator by Shift Operator	
u-VDTR	111
Domain Traps	
u-VGSR	794
Mutate Global Scalar References	
u-VTWD	74
Twiddle Mutations	
II-ArgLogNeg	3
Insert Logical Negation on Argument	
Total	1393

TABLE I

CORE SET FROM ALL-SUBSETS REGRESSION ANALYSIS (CORE-SUB)

For each test suite group, we identified the set of all operators that appeared in at least one of the best models generated (correlation of 0.995 or greater). There was no reason to believe that all five sets would be equal, and they were not. However, the slightly surprising result was that there was very little commonality among the operator sets. Only ten operators appeared in all five sets. We refer to the set of these ten operators as the *core-SUB* set of sufficient mutation operators. This set of operators, along with their descriptions found in the Proteum binaries, is shown in Table I. More detailed descriptions of these operators can be found in [15], [16], [17].

B. EBC Analysis

After doing the EBC analysis, we again ended up with a different set of operators for each test suite group. Seven operators appeared in all five sets. We refer to these operators as the *core-EBC* set, and they can be found in Table II. Another surprise is that only one of them (I-IndVarLogNeg) is shared with the core-SUB group.

C. Cluster Analysis

As with the other two analyses, we identified a different set of sufficient operators for each group of test suites, but here there was greater agreement, with 13 operators appearing in every group. This set of operators, the *core-CA* set, appears in Table III. One operator from the core-SUB set appear in this set, and also four operators from the core-EBC set; however, the operator that appears in both core-SUB and core-EBC (I-IndVarLogNeg) does *not* appear, leading to the conclusion that *no* operator appears in all three core sets.

D. Comparing Sufficient Operator Sets

Table IV shows a comparison of the savings achieved by the operator sets given by the three analyses. Core-EBC achieves

Operator / Description	♯mutants
I-IndVarAriNeg Inserts Arithmetic Negation at Non Interface Variables	19
I-IndVarLogNeg Inserts Logical Negation at Non Interface Variables	19
II-ArgRepReq Argument Replacement by Required Constants	5
u-OALN Arithmetic Operator by Logical Operator	2
u-OASN Arithmetic Operator by Shift Operator	2
u-OCNG Logical Context Negation	8
u-OLLN Logical Operator Mutation	17
Total	72

TABLE II

CORE SET FROM ELIMINATION-BASED CORRELATION ANALYSIS (CORE-EBC)

Operator / Description	♯mutants
I-CovAllEdg Coverage of Edges	12
I-DirVarRepCon Replaces Interface Variables by Used Constants	38
I-IndVarAriNeg Inserts Arithmetic Negation at Non Interface Variables	19
I-RetStaDel Deletes Return Statement	17
II-ArgIncDec Argument Increment and Decrement	54
u-OABN Arithmetic by Bitwise Operator	3
u-OALN Arithmetic Operator by Logical Operator	2
u-OASN Arithmetic Operator by Shift Operator	2
u-OCNG Logical Context Negation	8
u-OLNG Logical Negation	51
u-ORSN Relational Operator by Shift Operator	30
u-SRSR return Replacement	60
u-STRP Trap on Statement Execution	70
Total	366

TABLE III

CORE SET FROM CLUSTER ANALYSIS (CORE-CA)

	Core-SUB	Core-EBC	Core-CA
♯Mutants	1393	72	366
%Total	28.22%	1.45%	7.41%
%Saving	71.78%	98.55%	92.59%
♯Operators	10	7	13
%Total	9.25%	6.48%	12.03%
%Saving	90.75%	93.52%	87.97%

TABLE IV

COMPARISON OF CORE SETS

	Core Set	Multiple Regression Model
1	Core-SUB	$AM \simeq 0.012+$ $-0.049 * Am_{I-DirVarAriNeg}+$ $0.258 * Am_{I-DirVarRepReq}+$ $0.005 * Am_{I-IndVarLogNeg}+$ $0.120 * Am_{I-IndVarRepReq}+$ $-0.001 * Am_{u-OABN}+$ $0.080 * Am_{u-OLSN}+$ $0.062 * Am_{u-VDTR}+$ $0.376 * Am_{u-VGSR}+$ $0.102 * Am_{u-VTWD}+$ $0.036 * Am_{II-ArgLogNeg}$
2	Core-EBC	$AM \simeq -0.059+$ $0.363 * Am_{I-IndVarAriNeg}+$ $0.195 * Am_{I-IndVarLogNeg}+$ $0.078 * Am_{II-ArgRepReq}+$ $-0.037 * Am_{u-OALN}+$ $0.059 * Am_{u-OASN}+$ $0.161 * Am_{u-OCNG}+$ $0.180 * Am_{u-OLLN}$
3	Core-CA	$AM \simeq 0.029+$ $0.128 * Am_{I-CovAllEdg}+$ $0.182 * Am_{I-DirVarRepCon}+$ $0.167 * Am_{I-IndVarAriNeg}+$ $0.154 * Am_{I-RetStaDel}+$ $0.043 * Am_{II-ArgIncDec}+$ $-0.008 * Am_{u-OABN}+$ $-0.007 * Am_{u-OALN}+$ $0.023 * Am_{u-OASN}+$ $0.013 * Am_{u-OCNG}+$ $0.093 * Am_{u-OLNG}+$ $0.190 * Am_{u-ORSN}+$ $0.047 * Am_{u-SRSR}+$ $-0.054 * Am_{u-STRP}$

TABLE V

LINEAR MULTIPLE REGRESSION MODELS RESULTING FROM CORE SETS

the greatest savings, followed by Core-CA and Core-SUB. As expected, the SUB analysis, which does not take number of mutants into account, generates the largest number of mutants. However, all three analyses yield reasonable results, with even Core-SUB leading to an over 71% reduction in number of mutants generated (recall that Offutt et al.'s five operators yielded a 77.56% savings).

In order to compare the usefulness of the subsets for predicting AM (the overall mutation adequacy ratio), we performed multiple linear regressions to build a linear model for AM from the sets of operators given, using all data from all five test suite groups. These models are shown in Table V. Each model can be taken as a way of predicting what AM will be for a test suite, given only the Am$_i$ values from the sufficient set.

An interesting feature of these models is the presence of negative coefficients. For instance, Am_i for OALN (replace arithmetic operator by logical operator) has a negative coefficient in each of the core-EBC and core-CA models. This suggests that a test suite for `tcas` that kills more OALN mutants is predictably likely to kill *fewer* mutants overall. This observation is corroborated by the fact that OALN is on the longest branch leading directly from a leaf not in the dendrogram in Figure 1, suggesting that it is the operator with the “most different” behaviour.

Figure 2 shows a scatter plot of the value of AM predicted by the core-SUB linear regression model, against the actual value of AM. Each circle represents a test suite; the heavy diagonal line is the $x = y$ line of a hypothetical perfect model, and the thinner curve is a smoothing spline fitted to the data. Figures 3 and 4 show the same thing for core-EBC and core-CA respectively.

While all three models are good, core-SUB is the best, followed closely by core-CA, which uses far fewer mutants. Core-EBC does not fare as well, but it uses many fewer mutants even than core-CA. The goodness of fit may be a result of the better models using more mutants, or there may be other factors. Visual inspection of the graphs suggests that the cluster analysis (CA) is doing the best job at balancing number of mutants generated with goodness of fit.

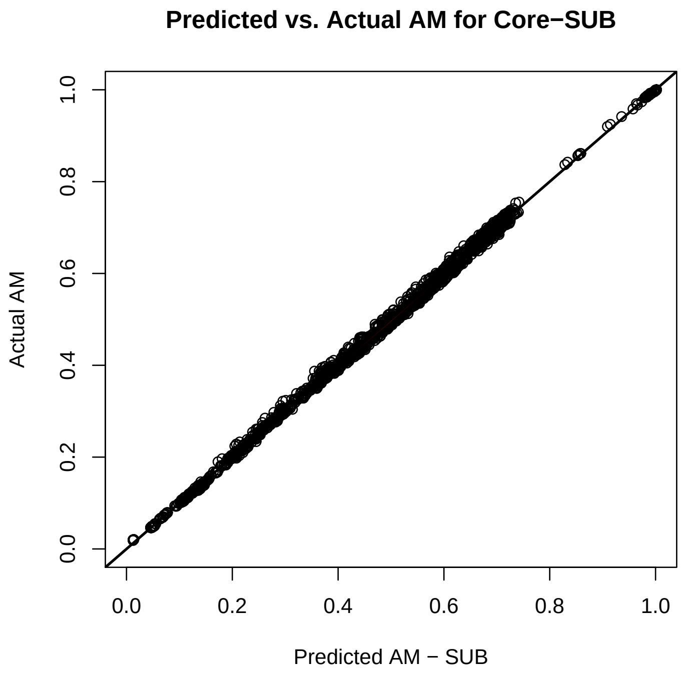

Fig. 2. Predicted vs. actual plot for Core-SUB linear regression model

Table VI shows another comparison between the predicted AM achieved by the three techniques and the actual AM. The high correlation values show that, indeed, they are all good predictors for AM. The results of t tests (paired and

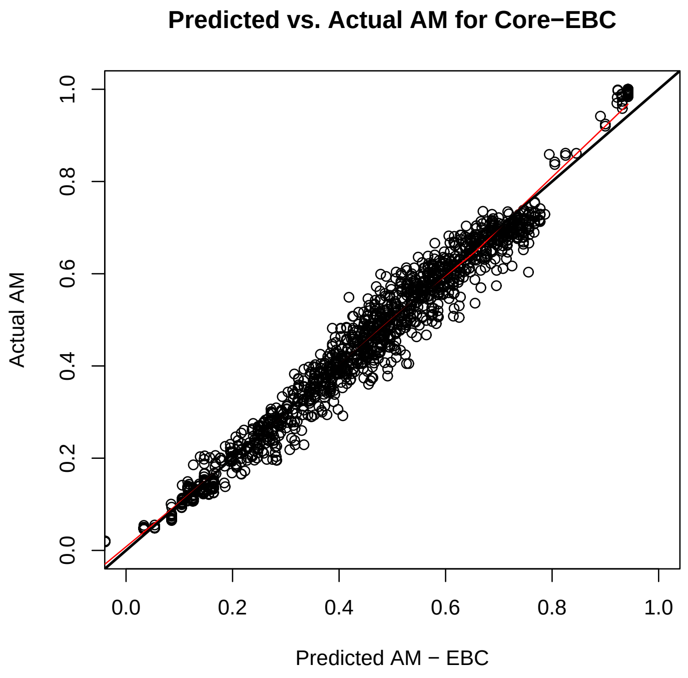

Fig. 3. Predicted vs. actual plot for Core-EBC linear regression model

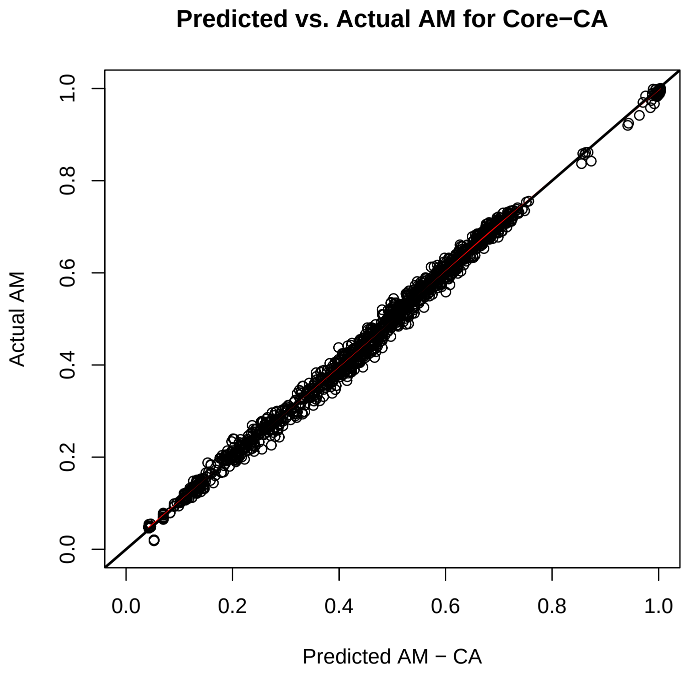

Fig. 4. Predicted vs. actual plot for Core-CA linear regression model

	Correlation with AM	*p-value paired t.test*	*p-value Welch two sample t.test*
$Am_{pred-SUB}$	0.9994673	0.9998	1
$Am_{pred-EBC}$	0.9883457	1	1
$Am_{pred-CA}$	0.9982409	0.9993	1

TABLE VI
ACTUAL VS. PREDICTED AM

Welch) are also shown, where the null hypothesis is that mean difference (resp. difference in means) between AM$_{Actual}$ and AM$_{Predicted}$ is 0. The p values are all much greater than the standard confidence level of 0.05, indicating that we cannot reject the null hypothesis. This suggests that the prediction is good in this respect as well.

VI. DISCUSSION

A. Threats to Validity

We do not expect that the preliminary results from `tcas` will necessarily carry over to the other programs in the Siemens subject program suite. The other programs are larger than `tcas`, and will result in more mutants; they also contain different and more complex control and data structures, which may result in Proteum generating more or fewer mutants for some operators. In particular, `tcas` is not one of the Siemens programs that contains C `struct`s or non-trivial pointers, so it has the same issues as the subject programs of earlier studies. We also expect the greater diversity of programs will result in regression models that are less accurate than those for the single program `tcas`.

Aside from the considerations arising from considering only one program so far, there are other threats to validity of our experiments. 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. The use of only non-equivalent mutants may be taken as a threat to construct validity, but as we noted in Section III-B, this is appropriate in the context of identifying a set of sufficient mutation operators for experiments.

Threats to external validity include the use of C programs that are still relatively small compared to commercial programs. Even the data collection and analysis done so far took some tens of hours of CPU time and much more time for subject preparation, so unfortunately we were not able to use larger programs. However, 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. We do note, however, that we have not attempted to handle object-oriented constructs. Mutant generators that implement class mutation operators, such as MuJava [6], are better suited to evaluation of sufficient mutation operator sets for object-oriented programs.

B. Data Combination and Analysis

In processing the available data, we are faced with the question of what order to perform the following steps in: (a) Combine data from subject programs; (b) Do statistical analyses; (c) Combine data from test suite groups; (d) As one evaluation, fit linear models. Clearly (a) must be done first, since the point of using different subject programs is to collect information not dependent on any one subject program. We have initially chosen to do the other steps in the order (b), (c) and (d). However, the unexpected lack of consensus between the different test suite groups may suggest that it is better to combine all data from all test suite groups before doing an analysis. This would result in only three sufficient operator sets to be compared to each other.

Alternatively, the lack of consensus between the different test suite groups may be nothing but an artifact of using only one subject program, which will disappear when we consider more than one. Although considering more than one subject may lead to less accurate linear models, all the linear models we arrived at for `tcas` were very good, and all reduced the number of mutants substantially.

Finally, it should be noted that the ability to predict AM is not necessarily the only measure of the goodness of a sufficient set of mutants for all purposes. The EBC and CA analyses derive sufficient sets essentially based on how differently the operators behave from one another. A practitioner performing mutation testing of a piece of software may take such information as suggesting that they should use all mutants in these sufficient sets, in order to make sure that their test suite kills as many diverse kinds of mutants as possible.

VII. CONCLUSIONS AND FUTURE WORK

We have interpreted the problem of identifying sufficient mutation operators as a variable reduction problem, and have described various approaches to the problem based on the literature. One of the analyses takes the overall mutation adequacy AM as the target, and the other two try to find a set of operators that are the most statistically distinct from each other as possible. We have described our experimental and analysis procedure in detail. Our preliminary results suggest that our procedure is feasible and does yield valuable information.

In the future, we will of course extend the processing and analysis to all test suite groups and all seven Siemens programs. This will involve much more computing. We will also study whether non-linear and other regression methods result in models that are a better fit to the data we have. Finally, we will study the resulting data with the goal of identifying one set of operators (or several sets of operators, each one for a different situation) that we can reasonably justify claiming as “sufficient”.

ACKNOWLEDGMENTS

This work is supported by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada

(NSERC). Akbar Siami Namin is further supported by an Ontario Graduate Scholarship. Thanks to Hyunsook Do and Greg Rothermel for their help in accessing the Siemens programs, to Jeff Offutt and Yu-Seung Ma for access to MuJava, 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 [18] was used for all statistical processing.

REFERENCES

[1] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutation operators," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, pp. 99–118, April 1996.

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

[3] 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, to appear.

[4] K. N. King and J. Offutt, "A Fortran language system for mutation-based software testing," *Software Practice and Experience*, vol. 21, no. 7, pp. 686–718, July 1991.

[5] 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)*, New Brunswick, NJ, July 1996, pp. 79–95.

[6] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava : An automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, June 2005.

[7] W. E. Wong, "On mutation and data flow," Ph.D. dissertation, Purdue University, December 1993.

[8] I. Jolliffe, *Principal Component Analysis*. Springer-Verlag, 1986.

[9] ——, "Disgarding variables in a principal component analysis. I: Artificial data," *Applied Statistics*, vol. 21, no. 2, pp. 160–173, 1972.

[10] ——, "Disgarding variables in a principal component analysis. II: Real data," *Applied Statistics*, vol. 22, no. 1, pp. 21–31, 1973.

[11] A. Rencher, *Methods of Multivariate Analysis*. Wiley Series in Probability and Statistics, 2002.

[12] G. McCabe, "Principal variables," *Technometrics*, vol. 26, no. 2, pp. 137–144, 1984.

[13] 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*, Sorrento, Italy, May 1994, pp. 191–200.

[14] 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)*, Washington, DC, USA, November 1998, pp. 34–43.

[15] 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," Department of Computer Science, Purdue University, Tech. Rep. SERC-TR-41-P, April 2006.

[16] M. E. Delamaro, J. C. Maldonado, A. Pasquini, and A. P. Mathur, "Interface mutation test adequacy criterion: An empirical evaluation," *Empirical Software Engineering*, vol. 6, pp. 111–142, 2001.

[17] A. M. R. Vincenzi, J. C. Maldonado, E. F. Barbosa, and M. E. Delamaro, "Unit and integration testing strategies for C programs using mutation," *Software Testing, Verification and Reliability*, vol. 11, pp. 249–268, 2001.

[18] W. N. Venables, D. M. Smith, and The R Development Core Team, "An introduction to R," R Development Core Team, Tech. Rep., June 2006.

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

[20] N. Timm, *Applied Multivariate Analysis*. Springer, 2002.