Efficiency of mutation operators and selective mutation strategies: An empirical study

Elfurjani S. Mresa       Leonardo Bottaci∗
Department of Computer Science, University of Hull, Hull HU6 7RX, U.K.

December 1999

Abstract

This paper investigates the mutation scores achieved by individual operators of the Mothra mutation system and their associated costs in order to determine the most efficient operators. The cost of mutation analysis includes both test set generation and equivalent mutant detection. The score and cost information is then used as a heuristic for choosing a subset of the operators for use in efficient selective mutation testing. Experiments were performed using a sample of 11 programs and a number of test sets for each program. The results show that the use of efficient operators can provide significant efficiency gains for selective mutation if the acceptable mutation score is not very close to one. When mutation scores very close to one are required, a randomly selected proportion of the mutants provides a more efficient strategy than a subset of efficient operators.

KEY WORDS: software testing, selective mutation testing, mutation operators, efficiency

1 Introduction

Mutation analysis, originally proposed by DeMillo et al. (1978) and Hamlet (1977), is a fault-based technique for testing software. The basic idea is to find a set of test cases that will reveal the faults that might be expected to be present in a given program under the assumption that a fault is manifest as a small modification to the correct program code. To do this, many copies of the test program are each modified by introducing a fault in the form of a small syntactic modification. A copy of the program that contains such a seeded fault is called a mutant and a mutant is said to be killed by a test case if for that test the output of the mutant differs from that of the original program under test. The tester’s task is to produce a set of test cases that can kill all the mutants of a given program. In this way, mutation analysis provides a test adequacy criterion (DeMillo et al. 1988) in the form of the mutation score, the proportion of killable mutants killed by a test set. There are a number of detailed descriptions of mutation testing in the literature (Acree et al. 1979; Budd 1980; DeMillo et al. 1988; Lipton and Sayward 1978; Mresa 1997).

With little prior knowledge of the faults that may exist in any given program, mutant generation must necessarily be comprehensive with the result that even small programs generate large numbers of mutants. As a result, mutation testing is generally regarded as too expensive to use in practice (Choi et al. 1989; Weiss and Fleysihakker 1993; Duncan 1993). This is in spite of the fact that mutation analysis can provide high levels of testing confidence.

Reducing the cost of mutation analysis without significantly reducing its effectiveness has been a primary concern of many researchers, and although progress has been made, the problem is still far from being solved. One simple approach uses a small randomly selected subset of all the generated mutants (Acree 1980; Mathur and Wong 1993). In this approach, termed “randomly selected x% mutation” (Mathur and Wong 1993), all mutants are first generated as in full mutation. Secondly, x% of the generated mutants are selected randomly for mutation analysis and the remaining mutants are ignored. In evaluating x% selective mutation for the Fortran mutation system Mothra (DeMillo and Spafford 1986; DeMillo et al. 1988; King and Offutt 1991), test sets that were

∗Contact author for correspondence.
adequate (mutation score of 100%) for 10% selective mutation were found to be nearly adequate in a full mutation analysis.

Researchers have also investigated how redundant mutants might be eliminated by discarding the mutants of specific mutation operators. A mutation operator is a rule for creating mutants of a certain kind. Mathur (?) analysed the 22 mutation operators used by Mothra (see Table 1) and observed that just a few operators result in the generation of a large proportion of a program’s mutants. Mathur suggested the exclusion of the two most prodigious operators: the scalar variable replacement (svr) and the array reference for scalar variable replacement (asr).

This proposal was renamed “selective mutation” and extended by Offutt et al. (1993) to exclude the next most prodigious operators using the term “N-selective mutation” to refer to mutation excluding the N most prodigious mutation operators. In this study, 2-selective mutation adequate test sets achieved a mean mutation score of 99.99% over ten subject programs with a 24% saving in the number of mutants generated. 4-selective mutation adequate test sets achieved a mean mutation score of 99.84% with a 41% saving in the number of mutants generated and 6-selective adequate test sets achieved a mean score of 99.71% with a 60% saving in the number of mutants generated.

<table>
<thead>
<tr>
<th>Mut. Op.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aar</td>
<td>array reference for array reference replacement</td>
</tr>
<tr>
<td>abs</td>
<td>absolute value insertion</td>
</tr>
<tr>
<td>aor</td>
<td>array reference for constant replacement</td>
</tr>
<tr>
<td>asr</td>
<td>arithmetic operator replacement</td>
</tr>
<tr>
<td>car</td>
<td>array reference for scalar variable replacement</td>
</tr>
<tr>
<td>cnr</td>
<td>constant for array reference replacement</td>
</tr>
<tr>
<td>crp</td>
<td>comparable array name replacement</td>
</tr>
<tr>
<td>csr</td>
<td>constant for scalar variable replacement</td>
</tr>
<tr>
<td>der</td>
<td>do statement end replacement</td>
</tr>
<tr>
<td>dsa</td>
<td>data statement alterations</td>
</tr>
<tr>
<td>glr</td>
<td>goto label replacement</td>
</tr>
<tr>
<td>lcr</td>
<td>logical connector replacement</td>
</tr>
<tr>
<td>ror</td>
<td>relational operator replacement</td>
</tr>
<tr>
<td>rsr</td>
<td>return statement replacement</td>
</tr>
<tr>
<td>san</td>
<td>statement analysis</td>
</tr>
<tr>
<td>sar</td>
<td>scalar variable for array reference replacement</td>
</tr>
<tr>
<td>scr</td>
<td>scalar for constant replacement</td>
</tr>
<tr>
<td>sdl</td>
<td>statement deletion</td>
</tr>
<tr>
<td>src</td>
<td>source constant replacement</td>
</tr>
<tr>
<td>svr</td>
<td>scalar variable replacement</td>
</tr>
<tr>
<td>uoi</td>
<td>unary operator insertion</td>
</tr>
</tbody>
</table>

Table 1: Mothra mutation operators

Extending this approach further, Offutt et al. (1996) divided the Mothra mutation operators into three categories according to the syntactic elements that they modify, namely: statements, operands of expressions and operators of expressions. When the mutation operators of each category in turn were excluded, the lowest of the three mutation scores obtained using the operators of the remaining two categories was in excess of 97%. In a further experiment, the mutation operators for both operands of expressions and statements were excluded to leave only the five mutation operators for the operators within expressions, i.e. abs, uoi, lcr, aor and ror. The test sets that were adequate with respect to the mutants produced by these five operators achieved a mutation score of 99.5%.

In the same experiment, Offutt also estimated the cost of selective mutation by the number of mutants generated. For a given program, the number of mutants generated by full mutation was found to be of the order of the product of the number of values (constants and variables) with the number of value references. In contrast, the number of mutants generated by operators that mutate the operators of expressions was found to be of the order of the number of value references.
The relatively high mutation scores obtained by the various selective mutation methods clearly indicate considerable redundancy in the mutants generated by the mutation operators of the Mothra mutation system. The prospect of exploiting this redundancy is clearly attractive and all the selective mutation approaches have been able to show substantial reductions in the number of mutants considered. In each case, however, test set adequacy is sacrificed. This places the tester in the difficult position of having to decide if the loss of adequacy is more than compensated for by the reduction in the number of mutants.

Although the reduction in the number of mutants considered may be in excess of 50% when the loss of test set adequacy is less than 1%, figures such as these must be assessed cautiously. It cannot be assumed that a test set that kills 99% of the mutants killed by an adequate test set is able to detect 99% of the real faults that are detected by the adequate test set. Such a conclusion requires the assumption that all mutants are equally effective in forcing the construction of real fault detecting tests. This is unlikely since many mutants are killed by almost any test case and other mutants are very difficult to kill. Moreover, a 50% reduction in the number of mutants does not imply a 50% reduction in the cost of the test. If one considers the number of tests generated, for example, the savings from selective mutation are less dramatic. The issue of the cost of mutation testing is considered in more detail in Section 2.2.

Although one might argue about the magnitude, it is clear that reducing the number of mutants leads to a lower test cost. Testing costs, however, form only a part of the cost of software. It is important to remember that cost savings in unit testing may well be “paid for” later in the software development process as faults that are undetected at unit testing are found at integration testing or once the application is deployed. The selective mutation tradeoff is therefore worth taking only if the test cost saving more than compensates for the increased risk that a fault remains undetected by the less than adequate test set. This increased risk is likely to be very difficult to measure since it is dependent on the nature of the software and the application area in which the software is deployed. For this reason, it is by no means clear that selective mutation is, in overall terms, cost effective. In summary, the work to date on selective mutation tells the tester that there is a tradeoff between test cost and test adequacy but does not tell the tester how to exploit the tradeoff.

In certain situations, however, it is reasonable to ignore the cost of undetected faults when assessing the benefits of selective mutation. For example, when comparing two strategies for selective mutation that achieve equal mutation scores, it is reasonable to assume that the costs for undetected faults are equal in each strategy. In such a situation, a testing cost saving is indeed an overall cost saving and the tester may rationally select the lower cost strategy. By factoring out the effect of unknown costs in this way, meaningful cost comparisons can be made. This method of comparing testing strategies is considered in more detail in Section 4.

Those researchers that have looked at reducing mutant redundancy by removing specific operators have considered operators in terms of the operators’ mutant generation rate (Offutt et al. 1993) or on the researchers’ judgement of the relative usefulness of different mutation operators (Offutt et al. 1996). Offutt et al. (1993) speculated that an important property of an operator (they called it “strength”) is the number of total mutants killed by a test set that is adequate for the mutants of that operator alone. The strength is in effect the mutation score property applied to a mutation operator. Offutt et al. pointed out that the operators with high strengths or mutation scores might be the most useful for selective mutation. This suggestion is adopted in this paper as a heuristic for choosing suitable operators for selective mutation.

The remainder of this paper begins with a section in which measures of mutation operator score and cost are defined and is followed by a report of an experiment to determine the score and cost of each individual mutation operator. Following that, the relation “more efficient than” between two mutation operators is defined and extended directly to compare mutation strategies. There then follows a report of a second experiment in which the efficiency of some specific selective mutation strategies are investigated, leading to a discussion and finally, a conclusion.

## 2 Definition of score and cost of individual mutation operators

In the context of selective mutation, an efficient operator can be expected to generate mutants that require the construction of test cases that are able to kill not only the mutants of the given operator but also most of the mutants of other operators and to do this at a low cost. The following notation will be useful in making this idea precise.

Let \( p \) denote a program under test. The set of mutants generated by applying all the mutation operators from a given set under consideration to the test program \( p \) is denoted by \( M^p \). A mutant is called an equivalent mutant...
if it is functionally equivalent to the original program, otherwise, it is non-equivalent. Let $M_{p}^{o}$ and $M^{p}$ denote all the equivalent and non-equivalent mutants in $M_{p}^{o}$ respectively, i.e. $M_{p}^{o} \cup M^{p} = M_{p}^{o}$. Those mutants in a set of mutants $M$ killed by at least one test case in a set of test cases $t$ is denoted by $kill(M,t)$. Clearly, for every test set $t$, $kill(M^{p},t) \subseteq M^{p}$ and $kill(M_{p}^{o},t) = \emptyset$.

A set of test cases that can kill all mutants in $M^{p}$ is denoted by $p'$, i.e. $kill(M^{p},p') = M^{p}$, and $p'$ is called a mutation-based adequate test data set with respect to the program $p$. Let $M_{p}^{oe}$ represent the set of all mutants produced by applying the mutation operator $o$ to the program $p$. The corresponding sets of equivalent and non-equivalent mutants in $M_{p}^{oe}$, are denoted by $M_{p}^{oe}$ and $M_{p}^{o}$ respectively. A set of test cases that kills all the non-equivalent mutants $M_{p}^{o}$ is denoted by $t_{p}^{o}$, i.e. $kill(M_{p}^{o},t_{p}^{o}) = M_{p}^{o}$, and $t_{p}^{o}$ is called an $o$-adequate test set.

### 2.1 Operator mutation score

The mutation score property of a given mutation operator seeks to measure, in general terms, the extent to which tests constructed to kill the mutants of that operator are also able to kill the mutants of all other operators. The mutation score of a test set $t$, with respect to the set of non-equivalent mutants of a program $p$, $M^{p}$, is the proportion of mutants in $M^{p}$ killed by tests in $t$, i.e.

$$score(t,p) = \frac{\#kill(M^{p},t)}{\#M^{p}}$$

where $\#$ denotes the size or cardinality of a set. The mutation score of an $o$-adequate test set $t_{p}^{o}$ is therefore

$$score(t_{p}^{o},p) = \frac{\#kill(M^{p},t_{p}^{o})}{\#M^{p}}$$

For a given program $p$, the extent to which $o$-adequate test sets in general satisfy the full mutation adequacy criterion may be estimated by calculating the mean mutation score of a sample of $o$-adequate test sets, $T_{p}^{o}$, i.e.

$$score(T_{p}^{o},p) = \frac{\sum_{t_{p}^{o} \in T_{p}^{o}} score(t_{p}^{o},p)}{\#T_{p}^{o}}$$

#### 2.1.1 Sampling adequate test sets

Clearly, for a given program $p$ and set of mutants $M$ any set that contains an $o$-adequate test set is also $o$-adequate and as a result there is a very large set of $o$-adequate test sets from which to draw a sample. Most of these test sets will have a mutation score of 1 and hence the mean of a random sample from the set of all $o$-adequate test sets will be strongly biased towards a mutation score of 1. Moreover, most of these test sets will be irrelevant to practical testing where the cost of generating an adequate test set should be as low as is feasible.

In the previously mentioned work on selective mutation, the usual approach towards producing sample test sets is to employ an incremental procedure for generating tests which terminates when the given mutation adequacy criterion is attained. Offutt et al. (1996) sampled effective sequences of tests generated by Godzilla (?), an automatic test data generation tool. Informally, each test in an effective sequence is non-redundant with respect to the tests that precede it. The definition of non-redundancy for a set of test cases is given first followed by the definition of an effective sequence of tests.

Let $t$ be a set of test cases and let $c$ be a test such that $c \in t$ then $c$ is non-redundant with respect to $t$ and set of mutants $M$ if $c$ kills a mutant in $M$ not killed by any other test in $t$, i.e. $kill(M,t \setminus \{c\}) \subset kill(M,t)$. A set of test cases $t$ is non-redundant with respect to a set of mutants $M$ if every test in $t$ is non-redundant with respect to $t$. It follows that all the subsets of a non-redundant set are also non-redundant. Note that a non-redundant test set is not necessarily minimal with respect to the mutants killed by $t$, i.e. a non-redundant test set $t$ may not be the smallest set that kills all the mutants killed by $t$. 

4
Consider now a sequence of test cases \( t = \langle c_1, c_2, \ldots, c_n \rangle \) in which no test is repeated. \( t \) is an effective sequence of tests with respect to a set of mutants \( M \) if for every prefix\(^1\) of \( t \), \( \langle c_1, c_2, \ldots, c_i \rangle \), the last test in the prefix \( c_i \) is non-redundant with respect to the tests in the prefix. For example, suppose that test case \( c_1 \) is able to kill only mutant \( m_1 \), \( c_2 \) is able to kill only mutant \( m_2 \) and that \( c_3 \) is able to kill all three mutants \( m_1, m_2 \) and \( m_3 \). If tests \( c_1 \) and \( c_2 \) were to be applied before \( c_3 \) they would be effective and hence the following are effective sequences: \( \langle c_1, c_2, c_3 \rangle \), \( \langle c_2, c_1, c_3 \rangle \), \( \langle c_1, c_3 \rangle \) and \( \langle c_3 \rangle \) but \( \langle c_1, c_3, c_2 \rangle \) is not. It follows that the last test in an effective sequence \( t \) is non-redundant with respect to the set \( t \) and that any sequence (non-repeating) of tests from a non-redundant test is an effective sequence.

Effective sequences tend to be larger than necessary for achieving the required criterion, i.e. they tend to contain redundant tests. Consider an effective sequence of tests that satisfies a selective mutation adequacy criterion and also contains redundant tests. These tests may not be redundant with respect to full mutation adequacy with the result that a redundant effective sequence is likely to achieve a higher full mutation score than any of its non-redundant subsets.

Research in selective mutation testing has shown that test sets that achieve various selective mutation adequacy criteria are in general able to achieve high full mutation scores. This research has all been done using effective sequences rather than non-redundant test sets. This choice is understandable given that the researchers involved were concerned with measuring the actual cost saving that might be achieved in practical mutation testing.

Since this paper is concerned with measuring efficiency it is more important to uncover differences rather than actual cost savings. In experiments where the full mutation scores achieved tend to be high, i.e. close to the limit of 100%, there is a danger that the differences between scores will be small and thus it will be difficult to decide if these small differences are significant or due to sampling variation. Selective mutation adequate test sets that are non-redundant have the advantage that the full mutation scores that they produce are not as close to the limit of 100% and so differences should be more readily observed. In addition, one would expect less variation in the scores of non-redundant selective mutation adequate test sets since these sets form a proper subset of the selective mutation adequate effective sequences. In order to investigate the potential advantages of non-redundant test sets for comparing different selective mutation strategies, sample test sets were drawn from both effective sequences and non-redundant test sets.

Although the score of a mutation operator might be expected to vary little across different test sets for a single program, it is not obvious that this should be the case across a range of different programs. The operator based selective mutation approaches, of which \( N \)-selective mutation is an example, are predicated, however, on the assumption that the mutation scores of operators do not vary greatly between programs. Without prejudging this issue, the score of an operator is defined by averaging over a sample of programs \( p \in P \) as well as a sample of \( o \)-adequate test sets \( T^p_o \). The mutation score of an operator \( o \) with respect to a sample of \( o \)-adequate test sets \( T^p_o \) and programs \( P \) is thus defined as

\[
\text{score}(o, T^p_o, P) = \frac{\sum_{p \in P} \text{score}(T^p_o, p)}{\#P}
\]  

(2)

In the experiments reported later in this paper, a sample of 11 programs was used.

### 2.2 Cost of a mutation operator

The relative cost of a mutation operator, with respect to some given set of operators, seeks to measure the cost of performing a selective mutation analysis using that single operator relative to the cost of a full mutation analysis employing all the given operators. The major cost of achieving a given mutation criterion arises from the time required to (a) generate test cases, and (b) examine mutants to identify equivalent mutants. These two cost components are considered in turn.

#### 2.2.1 Cost of test data generation

In much of the related work on selective mutation testing the cost of achieving the selective mutation criterion is estimated by the number of mutants generated by the criterion or the number of tests required for adequacy. But

\(^1\)A prefix of \( t \) is a sequence that when concatenated to the front of another sequence, possibly empty, forms \( t \).
non-equivalent mutants are costly largely because they must be executed and a large number of mutants may have a relatively low execution cost if they are all killed by a few tests and conversely a smaller number of mutants may have a relatively high execution cost if many tests are required to kill them.

The cost of test data generation will obviously depend on the method used. Hand generated test cases are typically the most expensive. In contrast, randomly generated tests with a uniform distribution across the input domain are relatively inexpensive but tend to be much less effective in that each test is likely to kill fewer mutants and so many more tests are required. An automatic test data generation tool such as Godzilla lies between these two extremes.

In general, an automatic test data generation tool, such as Godzilla, uses a two stage procedure to produce a test set. In the first stage, a set of candidate tests is generated. In principle, this could be done without regard to the structure of the program or a more targeted approach may be used. In the case of Godzilla, a set of constraints is generated from a given mutant to describe the conditions that the input values must satisfy in order to either kill the mutant or to have a reasonably good chance of doing so. It is these constraints that Godzilla solves to produce candidate test cases.

A candidate test case that satisfies a constraint set may not actually kill the corresponding mutant but since constraint satisfaction is a relatively inexpensive procedure (Offutt 1991), a constraint set may be solved many times, with different solutions selected randomly, in order to increase the probability of generating an effective test case. In practice, however, it is more likely that a single test case will kill several mutants and even though Godzilla is able to perform a number of optimisations during the generation of candidate tests (in particular, equivalent constraints may be removed and test cases sought that satisfy multiple constraints) Godzilla nonetheless generates many candidate test cases that are redundant.

Given that the set of test cases must be executed on the set of mutants to obtain a mutation score and that the output of the test program must be validated for each test case, it is usually cost effective to incorporate a second stage to the test set generation method that attempts to remove redundant tests from the set of candidate tests. Redundant test cases may be removed entirely by executing each one on all mutants and removing any that do not kill any mutant that is not killed by some other test. This is likely to be a costly procedure. Less costly are the effective sequences produced by Godzilla. Initially, a candidate test case is selected and applied to all the mutants. If this test case kills any mutant it is retained as an effective test case and another test is applied to the remaining live mutants, if any. This test case is similarly retained if it kills any of the mutants left alive after execution of the previous test case and so on. Test cases that kill none of the remaining live mutants are redundant and hence discarded.

Even allowing for the various optimisations that may be performed during the candidate generation stage, it is the cost of repeatedly executing candidate test cases against many mutants that dominates the time taken by Godzilla. In terms of the notation introduced earlier, the first test case \( c_1 \) is executed against every mutant in a given set \( M \). The second test case is executed against those mutants left alive after the first test case, i.e. \( M \setminus \text{kill}(M, \{c_1\}) \), and so on. For a given set of mutants \( M \), the execution cost of a sequence of candidate test cases \( t = \langle c_1, c_2, \ldots, c_n \rangle \), is measured as the total number of mutant executions.

\[
\text{execCnt}(t, M) = \#M + \\
\#M - \#\text{kill}(M, \{c_1\}) + \\
\#M - \#\text{kill}(M, \{c_1, c_2\}) + \ldots + \\
\#M - \#\text{kill}(M, \{c_1, c_2, \ldots, c_{n-1}\})
\]  

(3)

For a given set of mutants \( M \), the number of mutant executions is determined by the number and effectiveness of the tests. The lower bound is \( M \), which occurs when the first candidate test case kills all the mutants. In the work of Offutt et al. (1996), the mutation cost was measured simply as the number of mutants generated. This is in fact the best case cost. The average cost is likely to be higher since several tests are required to kill the mutants of all but the most trivial programs. With small programs, however, the average cost may not be too far away from the lower bound. For one of the subject programs used in this study, four tests were adequate to kill all its mutants, and one test killed nearly all of them.

The upper bound for the execution cost is the product of the number of mutants and the number of candidate test cases, i.e. \( \#M \times \#t \), which occurs when no test, except possibly the last, is effective. A significant number
of any program’s mutants are usually killed by just one or two of the initial test cases hence this upper bound is unrealistic and the typical cost is likely to be much lower.

In principle, however, the execution cost is unbounded since only practical considerations limit the size of $t$. If the candidate tests are generated randomly then any limit on the size of $t$ would be self-imposed. In the case of Godzilla, a constraint set may be satisfied for each mutant and hence the maximum cost occurs when the number of candidate test cases is equal to the number of mutants. Given the enormous variety of possible methods by which candidate test sets may be produced, it seems inappropriate to attempt to estimate the efficiency of these methods in general and hence an ideal candidate test generation method was assumed in which no candidate test is discarded and thus the test set generation cost in this study is measured by formula 3 where $t$ is the final adequate test set.

The mean generation cost of a sample of $o$-adequate test sets, $T_o$, with respect to the mutants of $o$, is simply the mean of the individual costs, i.e.

$$\text{genCost}(T_o, M_{oa}) = \frac{\sum_{e \in T_o} \text{execCnt}(e, M_{oa})}{\#T_o}$$

In order to compare meaningfully the cost of the same selective mutation strategy applied to programs that generate different numbers of mutants, the relative cost is measured. The relative mean generation cost of a sample of $o$-adequate test sets, $T_o$, for a given program $p$, is the ratio of the cost of generating a selective mutation adequate test set to the cost of generating a full mutation adequate test set. This was measured as

$$\text{genCostRel}(T_o, M_{oa}, p) = \frac{\text{genCost}(T_o, M_{oa})}{\text{execCnt}(p, M_{oa})}$$

### 2.2.2 Cost of equivalent mutant identification

The previous research on selective mutation, as covered in Section 1, has largely ignored the cost of identifying equivalent mutants. This cost is particularly difficult to quantify since it is an activity that requires human intervention although Offutt et al. (1997) have demonstrated that some equivalent mutants can be detected automatically. In the experience of the authors, however, identifying equivalent mutants is often the activity that consumes the most time and as a result cost measures that ignore this cost risk being substantially distorted.

The time consumed identifying equivalent mutants will, of course, depend on the tester’s skill and familiarity with the program and cost estimates will necessarily be crude. A simple approach is to count the number of mutants that must be examined. This includes all the equivalent mutants and any non-equivalent mutants that have not been killed by the automatically generated test set. A simple count of the live mutants seems an unsatisfactory cost measure, however, since some mutants are clearly far easier to identify as equivalent than others. This led the authors to consider ways to estimate the difficulty of identifying a mutant as equivalent and to consideration of the reasoning a tester must perform to investigate mutant equivalence.

As an illustration, consider the example subprogram that is shown in Figure 1, together with four of its mutations, labelled M1, M2.1 M2.2 M3. The function $F$ inputs a real number $x$ and computes $x^n$ where $n$ is the second integer input. If $x^n$ is greater than 0.5 the output is 1 and 0 otherwise. The statement $S1$ has been mutated by the src operator to produce M1. There are two mutations of $S2$, M2.1 in which $n$ is replaced by its absolute value and M2.2 in which the mutant is killed if $n$ takes the value 0. In the mutation M3 of the statement $S3$, $x$ is replaced by its absolute value.

As an example of a mutant that is easily confirmed as equivalent, consider mutant M1. Equivalence is relatively easy to investigate in this case because it is not necessary to consider the program state either before or after the execution of the mutated statement. The tester needs only to reason that any integer value for $n$ is less than 1 will also be less than 0.5. Equivalent mutants of this kind the authors called expression equivalent mutants.

A second kind of mutant is illustrated by the mutated statements M2.1 and M2.2. To investigate the equivalence of either of these two mutations requires consideration of the program state immediately prior to the execution of the mutated statements. In particular, it is necessary to reason that $n$ cannot be less than 1. If this
REAL FUNCTION F(x, n)
REAL x, y
INTEGER n, i
S3 y = x
M3 y = ABS(x)
S1 IF (n .LT. 1) THEN
M1 IF (n .LT. 0.5) THEN
   n = 1
ENDIF
S2 DO 10 i = 1, n
M2.1 DO 10 i = 1, ABS(n)
M2.2 DO 10 i = 1, ZPUSH(n)
   y = y * y
10 CONTINUE
IF (y .GT. 0.5) THEN
   F = 1
ELSE
   F = 0
ENDIF
RETURN
END

Figure 1: Examples of three different kinds of equivalent mutation

pre-condition is satisfied then the mutant is effectively an expression equivalent mutant. This kind of mutant was called a pre-condition equivalent mutant.

A third kind of equivalent mutant is illustrated by the example mutant M3. When \( x \) is negative, the value of \( y \) in the mutant differs from that in the original program leading to distinct states in the original and the mutant programs. To investigate equivalence for this kind of mutant it is necessary to trace the execution of the program beyond the mutated statement and establish if the state difference is propagated to the output. The difference in program state after execution of the mutated statement is the weak mutation distinguishing condition, and so this kind of mutant was called a weak equivalent mutant.

There is a sense in which a tester investigating a pre-condition equivalent mutant requires access to more information than is necessary to investigate an expression equivalent mutant and thus the investigation of the pre-condition mutant ought to be more costly. Similarly, the investigation of a weak equivalent mutant ought to be more costly than a pre-condition equivalent mutant since this typically requires investigating the state immediately prior to the execution of the mutated statement and, in addition, it is necessary to trace the execution of the program beyond the mutated statement to establish the final program state.

For each of the 11 programs considered in this study, the authors analysed the mutants left alive by the automatically generated test data and counted the numbers of expression, pre-condition and weak equivalent mutants and found that the vast majority were pre-condition mutants. Expression equivalent mutants were rare and weak equivalent mutants were typically just a few percent with BANKER having the highest proportion at about 10%. The reason for this is that BANKER is a function program unit that outputs one of the two logic values and programs with small output domains propagate little of their internal state information to the output. This agrees with the results of Voas and others on program sensitivity analysis (Voas et al. 1991; Voas 1992; Voas and Miller 1992; Voas and Miller 1993). In Voas’ terminology, programs and functions that have high domain to range ratios typically have a higher information loss than those of lower ratios and are therefore more difficult to test.

The more important finding, however, was that the authors’ experience did not, as a rule, confirm their hypothesis about the time taken to investigate the different kinds of equivalent mutant and where it did, the difference was often not large with perhaps as much variation between mutants of different statements as between different kinds of equivalent mutant. What was clear, however, was that different mutations of the same statement were significantly easier to investigate once the first mutation had been investigated. This was because different mutations of the same statement could usually be confirmed equivalent by reference to the same set of variables that occurred in the original statement. In the example shown in Figure 1, mutants M2.1 and M2.2 may both be confirmed as
equivalent by reference to the program state that exists immediately prior to the execution of these statements, i.e.
the value of \( n \). The necessary program state need be established just once for a given statement but may be reused,
with little additional work, for each mutation of that statement.

This suggests that the cost of equivalent mutant investigation should be based not only on the number of mutants
examined but also on the number of program states that must be investigated. In the experience of the authors in
performing this study, a reasonable approximation to the number of program states examined is the number of
statements with at least one equivalent mutant. This takes into account that the automatically generated test data
usually killed most of the non-equivalent mutants and expression equivalent mutants are rare.

Following the notation for mutants, let \( S^p_e \) be those statements in \( p \) that have equivalent mutants and let \( S^p_{oe} \)
be those statements that have equivalent mutants of the operator \( o \) then the relative statement cost of examining
equivalent mutants for operator \( o \) was defined as the proportion of statements for which there is at least one
equivalent mutant of the operator \( o \) out of the total number of statements with at least one equivalent mutant.

\[
\frac{\#S^p_{oe}}{\#S^p_e}
\]

The proportion of equivalent mutants for the operator \( o \) was also counted as a measure of the number of mutants
examined.

\[
\frac{\#M^p_{oe}}{\#M^p_e}
\]

Combining the two cost components, the relative equivalence cost of an operator \( o \) was measured as

\[
equivCost(o, p) = \frac{\#S^p_{oe}}{2\#S^p_e} + \frac{\#M^p_{oe}}{2\#M^p_e}
\]

The two components are given equal weighting in the absence of any objective evidence for which is the most
important. On the same grounds, the relative test generation cost and the relative equivalence detection cost are
given equal weighting in a combined cost measure.

\[
cost(o, T^p_o, p) = \frac{\text{genCostRel}(T^p_o, M^p_{oa}, p) + \text{equivCost}(o, p)}{2} \tag{4}
\]

An interesting question is whether the cost of a mutation operator is significantly influenced by the specific
choice of program. Without prejudging this question, the cost of an operator \( o \) with respect to a sample of \( o \-
adequate test sets \( T^p_o \) and programs \( P \) is defined as the mean cost of \( T^p_o \) across the programs \( p \in P \).

\[
cost(o, T^p_o, P) = \frac{\sum_{p \in P} \text{cost}(o, T^p_o, p)}{\#P} \tag{5}
\]

3 Experimental determination of score and cost of individual mutation operators

The objective of the first experiment reported in this paper is to measure the score and cost of 21 of the mutation
operators of the Mothra mutation system. Table 1 lists the complete set of operators; detailed descriptions can
be found in the work of King and Offutt (1991). Only the data statement alterations operator (dsa) was excluded
from this study. This operator mutates non-executable statements that initialise data structures and is particular to
Fortran. All data structure initialisation in the subject programs was done using executable statements.
<table>
<thead>
<tr>
<th>Program</th>
<th>Source Stmts</th>
<th>operators excluded</th>
<th>Total mutants</th>
<th>Non-equiv mutants</th>
<th>Adq test set size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number</td>
<td>percent</td>
</tr>
<tr>
<td>AREASG</td>
<td>51</td>
<td>der</td>
<td>5728</td>
<td>5442</td>
<td>95.01</td>
</tr>
<tr>
<td>BANKER</td>
<td>42</td>
<td>2972</td>
<td>2826</td>
<td>95.09</td>
<td>63</td>
</tr>
<tr>
<td>CALVAL</td>
<td>40</td>
<td>cnr glr</td>
<td>5660</td>
<td>5514</td>
<td>97.42</td>
</tr>
<tr>
<td>FINDCNT</td>
<td>38</td>
<td>1879</td>
<td>1781</td>
<td>94.78</td>
<td>13</td>
</tr>
<tr>
<td>MINV</td>
<td>44</td>
<td>4092</td>
<td>3783</td>
<td>92.45</td>
<td>17</td>
</tr>
<tr>
<td>PAT0</td>
<td>24</td>
<td>780</td>
<td>715</td>
<td>91.67</td>
<td>18</td>
</tr>
<tr>
<td>RPCALC</td>
<td>53</td>
<td>der</td>
<td>5001</td>
<td>4786</td>
<td>95.70</td>
</tr>
<tr>
<td>SEQSTR</td>
<td>71</td>
<td>2903</td>
<td>2769</td>
<td>95.38</td>
<td>22</td>
</tr>
<tr>
<td>SORT</td>
<td>19</td>
<td>879</td>
<td>787</td>
<td>89.53</td>
<td>5</td>
</tr>
<tr>
<td>STREQL</td>
<td>44</td>
<td>glr</td>
<td>2837</td>
<td>2681</td>
<td>94.50</td>
</tr>
<tr>
<td>TRETRV</td>
<td>55</td>
<td>2592</td>
<td>2486</td>
<td>95.91</td>
<td>35</td>
</tr>
<tr>
<td>mean</td>
<td>43.7</td>
<td>3211</td>
<td>3045</td>
<td>94.83</td>
<td>30.0</td>
</tr>
</tbody>
</table>

Table 2: The number of source statements, mutants, non-equivalent mutants and test set size for each program used in this study

3.1 Subject programs

The experiments were performed using a sample of 11 program units, the descriptions of which are given in Appendix A. The aim in selecting the programs used in this study was to avoid any obvious source of bias by selecting programs that cover a range of different applications and are sufficiently large so that each generates the mutants of a wide range of operators. In only four of the programs, were no mutants generated for one or two of the 21 operators studied; these operators are listed in Table 2 which also provides descriptive statistics for each of the programs used. The table shows the number of statements, mutants and the number of tests required to kill all non-equivalent mutants.

On average, non-redundant test sets were about 50% smaller than the corresponding effective test sequences from which they were generated. The number of test cases required to achieve adequacy for each program is influenced by the number of mutants but also the nature of the program under test. For example, the results show that the BANKER program requires relatively more test cases. As mentioned in the previous section, BANKER is a function program unit that outputs one of the two logic values and thus has a high domain to range ratio.

3.2 Experiment method

For each of the 11 programs \( p, p=1, \ldots, 11 \), used in this study, the following steps were performed:

1. Each mutation operator \( o, o=1, \ldots, 21 \), as listed in order in Table 1 but excluding dsa, was applied to \( p \) to construct the sets of mutants \( M^p_o \) and hence \( M^p_a \).
2. An adequate effective test sequence was generated for \( M^p \) as described in Section 3.2.1. From this sequence, an adequate and non-redundant test set was constructed as also described in Section 3.2.1. The equivalent mutants \( M^p_e \) were identified by inspection of mutants not killed by the automatically generated test data.
3. For each non-empty set of non-equivalent mutants, \( M^p_o, o=1, \ldots, 21 \),
   a. 10 \( o \)-adequate effective sequences and corresponding non-redundant test sets were generated as described in Section 3.2.1. Let \( T^p_o \) denote the set of 10 \( o \)-adequate effective sequences and also the corresponding 10 \( o \)-adequate non-redundant test sets.
   b. The mean score of the 10 \( o \)-adequate test sets, \( \text{score}(T^p_o, p) \), was calculated according to formula 1 for both the effective sequences and the non-redundant test sets.
   c. The mean cost of the 10 \( o \)-adequate test sets, \( \text{cost}(o, T^p_o, p) \) was calculated according to formula 4 for both the effective sequences and the non-redundant test sets.
The score \( score(o, T^o, P) \) (according to formula 2) and cost \( cost(o, T^o, P) \) (according to formula 5) of each mutation operator was then computed by averaging over all 11 programs. This was done for both the effective sequences and the non-redundant test sets.

### 3.2.1 Adequate test set generation

This study differs from all previous related studies in that two forms of adequate test set were generated. In previous studies (Mathur and Wong 1994; Offutt 1992; Offutt and Lee 1994; Offutt et al. 1996; Offutt et al. 1993) only effective test sequences were used whereas in this study, both effective sequences and non-redundant test sets are used.

The adequate effective test sequences used in this study were constructed as follows. Firstly, the automatic test data generator tool, Godzilla, was used to generate a set of test cases. For a few operators and programs, Godzilla was able to generate an \( o \)-adequate test set. For a few operators and programs, Godzilla was not able to generate any tests. In those situations where Godzilla produced a test set significantly less than adequate, additional tests were generated using random number generators. Beyond that, it was almost always necessary to construct a few cases by hand to kill the last few mutants and obtain an adequate test set. The number of mutants killed by hand generated tests was typically two or three percent.

Non-redundant test sets were produced by repeatedly executing cyclic permutations of an effective sequence against a given set of mutants, each time discarding ineffective tests. This was continued until each remaining test had been executed as the last of a sequence. Recall that the last test in an effective sequence is non-redundant. This property, together with the fact that if a test is non-redundant with respect to a set of tests then it is also non-redundant with respect to any subset of those tests, ensures that the test sets obtained by this procedure are non-redundant.

<table>
<thead>
<tr>
<th>Op.</th>
<th>Score mean</th>
<th>Score sd</th>
<th>Cost mean</th>
<th>Cost sd</th>
<th>Gen Cost mean</th>
<th>Gen Cost sd</th>
<th>Eq Stmt Cost mean</th>
<th>Eq Stmt Cost sd</th>
<th>Eq Mut Cost mean</th>
<th>Eq Mut Cost sd</th>
<th>Mutants (%) mean</th>
<th>Mutants (%) sd</th>
</tr>
</thead>
<tbody>
<tr>
<td>uoi</td>
<td>98.1</td>
<td>1.1</td>
<td>5.7</td>
<td>4.1</td>
<td>4.5</td>
<td>2.0</td>
<td>11.1</td>
<td>11.9</td>
<td>2.8</td>
<td>2.8</td>
<td>6.5</td>
<td>2.1</td>
</tr>
<tr>
<td>svr</td>
<td>98.0</td>
<td>1.2</td>
<td>11.2</td>
<td>5.6</td>
<td>12.7</td>
<td>5.9</td>
<td>14.8</td>
<td>11.9</td>
<td>4.6</td>
<td>4.6</td>
<td>18.5</td>
<td>9.1</td>
</tr>
<tr>
<td>csr</td>
<td>97.6</td>
<td>1.3</td>
<td>10.1</td>
<td>5.7</td>
<td>9.4</td>
<td>5.1</td>
<td>16.2</td>
<td>14.4</td>
<td>5.4</td>
<td>4.0</td>
<td>12.2</td>
<td>5.0</td>
</tr>
<tr>
<td>abs</td>
<td>96.9</td>
<td>3.5</td>
<td>43.0</td>
<td>5.3</td>
<td>10.8</td>
<td>3.0</td>
<td>84.9</td>
<td>11.5</td>
<td>65.7</td>
<td>14.2</td>
<td>7.0</td>
<td>2.3</td>
</tr>
<tr>
<td>scr</td>
<td>95.6</td>
<td>4.6</td>
<td>7.3</td>
<td>6.5</td>
<td>5.7</td>
<td>2.4</td>
<td>13.9</td>
<td>17.5</td>
<td>4.1</td>
<td>5.4</td>
<td>8.3</td>
<td>3.4</td>
</tr>
<tr>
<td>ror</td>
<td>95.5</td>
<td>4.9</td>
<td>4.8</td>
<td>1.9</td>
<td>1.8</td>
<td>0.8</td>
<td>12.1</td>
<td>5.9</td>
<td>3.5</td>
<td>2.1</td>
<td>2.5</td>
<td>1.1</td>
</tr>
<tr>
<td>asr</td>
<td>95.2</td>
<td>4.5</td>
<td>6.3</td>
<td>4.9</td>
<td>6.8</td>
<td>3.5</td>
<td>8.7</td>
<td>15.2</td>
<td>2.8</td>
<td>4.0</td>
<td>12.5</td>
<td>4.8</td>
</tr>
<tr>
<td>crp</td>
<td>94.7</td>
<td>4.6</td>
<td>2.1</td>
<td>1.8</td>
<td>1.2</td>
<td>0.6</td>
<td>4.9</td>
<td>5.5</td>
<td>1.0</td>
<td>1.2</td>
<td>1.8</td>
<td>0.8</td>
</tr>
<tr>
<td>sdl</td>
<td>94.7</td>
<td>3.6</td>
<td>0.9</td>
<td>0.8</td>
<td>0.7</td>
<td>0.4</td>
<td>1.7</td>
<td>2.0</td>
<td>0.4</td>
<td>0.6</td>
<td>1.2</td>
<td>0.4</td>
</tr>
<tr>
<td>src</td>
<td>92.4</td>
<td>6.0</td>
<td>4.9</td>
<td>6.8</td>
<td>3.3</td>
<td>2.9</td>
<td>10.4</td>
<td>20.5</td>
<td>2.5</td>
<td>4.2</td>
<td>5.9</td>
<td>5.3</td>
</tr>
<tr>
<td>acr</td>
<td>91.4</td>
<td>7.7</td>
<td>3.7</td>
<td>4.5</td>
<td>3.1</td>
<td>1.8</td>
<td>6.4</td>
<td>10.7</td>
<td>2.4</td>
<td>5.5</td>
<td>6.0</td>
<td>3.4</td>
</tr>
<tr>
<td>aar</td>
<td>90.1</td>
<td>6.7</td>
<td>1.3</td>
<td>1.4</td>
<td>2.0</td>
<td>1.8</td>
<td>1.1</td>
<td>2.6</td>
<td>0.2</td>
<td>0.4</td>
<td>3.1</td>
<td>2.3</td>
</tr>
<tr>
<td>sar</td>
<td>89.6</td>
<td>7.2</td>
<td>1.4</td>
<td>1.4</td>
<td>2.4</td>
<td>1.5</td>
<td>0.8</td>
<td>2.5</td>
<td>0.1</td>
<td>0.3</td>
<td>4.7</td>
<td>1.4</td>
</tr>
<tr>
<td>cnr</td>
<td>88.7</td>
<td>6.0</td>
<td>0.9</td>
<td>1.6</td>
<td>0.7</td>
<td>0.8</td>
<td>2.1</td>
<td>4.5</td>
<td>0.3</td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>aor</td>
<td>87.9</td>
<td>12.2</td>
<td>0.8</td>
<td>1.0</td>
<td>0.9</td>
<td>0.7</td>
<td>1.3</td>
<td>2.9</td>
<td>0.2</td>
<td>0.4</td>
<td>1.7</td>
<td>1.6</td>
</tr>
<tr>
<td>san</td>
<td>86.9</td>
<td>9.3</td>
<td>0.3</td>
<td>0.2</td>
<td>0.5</td>
<td>0.4</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.2</td>
<td>0.7</td>
</tr>
<tr>
<td>car</td>
<td>85.0</td>
<td>12.1</td>
<td>0.7</td>
<td>0.6</td>
<td>1.4</td>
<td>1.1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>2.7</td>
<td>1.3</td>
</tr>
<tr>
<td>rsr</td>
<td>84.4</td>
<td>12.5</td>
<td>0.6</td>
<td>0.6</td>
<td>0.6</td>
<td>0.4</td>
<td>1.0</td>
<td>1.7</td>
<td>0.2</td>
<td>0.4</td>
<td>1.4</td>
<td>0.5</td>
</tr>
<tr>
<td>lcr</td>
<td>82.4</td>
<td>18.6</td>
<td>1.7</td>
<td>1.6</td>
<td>0.4</td>
<td>0.5</td>
<td>4.7</td>
<td>4.5</td>
<td>1.4</td>
<td>1.3</td>
<td>0.6</td>
<td>0.7</td>
</tr>
<tr>
<td>glr</td>
<td>81.8</td>
<td>19.6</td>
<td>1.5</td>
<td>2.2</td>
<td>1.0</td>
<td>1.8</td>
<td>3.2</td>
<td>4.4</td>
<td>0.9</td>
<td>1.4</td>
<td>1.3</td>
<td>2.6</td>
</tr>
<tr>
<td>der</td>
<td>72.3</td>
<td>22.1</td>
<td>3.7</td>
<td>3.8</td>
<td>0.3</td>
<td>0.4</td>
<td>12.2</td>
<td>13.1</td>
<td>1.9</td>
<td>1.7</td>
<td>0.3</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 3: Mean score and costs (as a %) over 11 programs of individual mutation operators using non-redundant test sets, the standard deviation is also shown
### Results and analysis

Table 3 shows the mean score and costs of each mutation operator together with the standard deviation for each of the 11 programs studied using non-redundant test sets. The Cost column shows the overall cost resulting from the generation, equivalent statement and equivalent mutant cost. The two rightmost columns show the proportion, and standard deviation of all mutants generated by each operator. The following observations may be made.

- svr, asr and csr are the operators that generate the most mutants. The overall distribution of mutants is similar to that found by Mathur (?) who studied a sample of 28 programs.

- The scores of the highest scoring operators is close to one. This result is consistent with the results obtained from earlier studies of selective mutation (??; Offutt et al. 1993) that indicate a considerable amount of redundancy in the mutants generated by Mothra.

- The standard deviation of the score of the three highest scoring operators is small. This can be interpreted as a crude measure of the reliability of these score values. The scores of the lower ranking mutation operators are more influenced by the program under test.

- The cost of abs is by far the highest and is due mainly to the large number of equivalent mutants it generates. In general, there is some correlation between the generation cost and the number of mutants but note that the generation cost of abs is higher than the relatively low number of abs mutants might suggest. Note also that the generation cost of asr is lower than the relatively high number of asr mutants might suggest.

- There is some correlation between score and cost as can be seen from Figure 2 which plots the mean scores and costs of each operator. Cost is plotted on a log scale in order to differentiate more clearly between operators with low but similar costs.

The score and cost of each operator were also calculated using effective test sequences and the results are shown in Table 4. The scores obtained using effective sequences are higher as expected and generally so are the standard deviation.
deviations of comparable scores. The generation cost of non-redundant tests are slightly lower in most cases but since costs are relative, little difference would be expected.

For a given program and operator, there was little variation in the sizes of the 10 adequate test sets, the mutation scores achieved and their costs. The standard deviation varied somewhat with the program and operator but for mutation scores over 90% the standard deviation was typically a fraction of 1%. For both score and cost, there was certainly much less variation between the 10 test sets for a given program than between the means of these test sets across the 11 programs.

4 Comparing the efficiency of mutation operators and selective mutation strategies

The most suitable operators for selective mutation analysis would be expected to have a high mutation score and a low cost and so a potential efficiency measure is the ratio of score to cost, i.e.

$$\frac{\text{score}(o, T^p_o, P)}{\text{cost}(o, T^p_o, P)}$$

The advantage of this measure is that as a continuous scalar value it produces a total order and thereby allows each operator to be directly compared with all other operators. But there are also disadvantages. Since there is far more variability in the cost than in the score (most scores are close to one) the efficiency can be maximised almost entirely by reducing cost, irrespective of the score value. This is not useful in practice since a tester is concerned not only with maximising efficiency but also with achieving a given adequacy criterion.

Secondly, the cost measured in this study is that of unit testing only. As mentioned in the introduction, there are other important software costs, for example maintenance costs, which increase as the score and test cost decrease with the result that test cost savings may well be cancelled out. These unknown costs can reasonably be factored out, however, when comparing two operators with equal scores, or more generally, if the score of one operator
exceeds that of another but has a lower cost then it is reasonable to assume that the overall cost of the higher scoring operator is also lower.

Mutation operator \( o \) is said to be as efficient as operator \( r \) with respect to mutation adequacy and given samples of test sets and programs, if it can at least equal the score of operator \( r \) but is no more costly, i.e.

\[
asEfficient(o, r, T^o, T^r, P) \iff \text{score}(o, T^o, P) \geq \text{score}(r, T^r, P) \land \text{cost}(o, T^o, P) \leq \text{cost}(r, T^r, P)
\]

Note that \( asEfficient \) is anti-symmetric and transitive and thus a reflexive partial order. The irreflexive subset is a more useful relation for the purposes of this study. Operator \( o \) is said to be more efficient than operator \( r \) with respect to mutation adequacy if operator \( o \) is as efficient as operator \( r \) but operator \( r \) is not as efficient as operator \( o \), i.e.

\[
moreEfficient(o, r, T^o, T^r, P) \iff asEfficient(o, r, T^o, T^r, P) \land \neg asEfficient(r, o, T^r, T^o, P)
\]

The efficiency relationships between individual operators can be seen in Figure 2. In this graph, operator \( o \) is more efficient than operator \( r \) if \( o \) lies to the left of and/or above \( r \).

Clearly efficiency relationships are statistical and Figure 3 gives an indication of the reliability of these results. Each rectangle in this figure shows the variability associated with an operator’s score and cost. The width of a rectangle is equal to the standard deviation of the cost and the height is equal to the the standard deviation of the score. It is evident that with a few notable exceptions (san, sdl, der, uoi and abs), most operators cannot be reliably said to be more or less efficient than any other operator.
5 Experiment to determine efficient selective mutation strategies

Although a few individual operators achieve high mutation scores, a practical selective mutation strategy will typically employ a number of operators. The aim of this experiment is to use the score and cost information for each individual operator to find an efficient but relatively high scoring subset of the Mothra mutation operators. This was done by adopting the heuristic that operators that are efficient individually will also produce an efficient set of operators.

From the information shown in Figure 2 it can be seen that the maximal elements of the partial order moreEfficient (i.e. those operators for which there are no more efficient operators) are san, aor, sdl, ror and uoi and may thus be considered to be the most efficient operators. These operators were therefore applied collectively as a set, denoted in this study by eff.

Other sets with which to compare the performance of eff were also constructed. Experience with the eff set of operators showed that test sets that were adequate for eff mutants failed to kill a disproportionately high number of abs mutants when applied to all the mutants of a given program. For this reason, the abs operator was added to the eff set to produce the set called here efa. Offutt et al. (1996) recommend the set of expression operators, i.e. abs, uoi, lcr, aor and ror for selective mutation and so the performance of the sets eff and efa was compared to the expression set of operators, called exp in this study.

The three sets of operators, eff, efa and exp, were also compared to x%-selective mutation where x was chosen separately for each program to force the construction of a test set that when adequate on x% of the mutants is able to achieve comparable full mutation scores. For example, test sets adequate for the exp set achieved a score of 99.4% in the CALVAL program and to achieve this score, a test set adequate for 15% selective mutation was required. In the MINV program, the exp set achieved a score of 99.6% and here 26% selective mutation was required to achieve this score. Table 5 shows the particular percentage values required for x%-selective mutation to achieve the score achieved by the exp set, this is labelled xex, and also the percentage values required for x%-selective mutation to achieve the score achieved by the eff set, this is labelled xef.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Op.</th>
<th>ARSG</th>
<th>BANK</th>
<th>CALV</th>
<th>FCNT</th>
<th>MINV</th>
<th>PA10</th>
<th>RPC</th>
<th>SEQs</th>
<th>SORT</th>
<th>STRE</th>
<th>TRET</th>
<th>mean</th>
<th>sd</th>
</tr>
</thead>
<tbody>
<tr>
<td>xex</td>
<td>24</td>
<td>26</td>
<td>15</td>
<td>29</td>
<td>26</td>
<td>44</td>
<td>23</td>
<td>30</td>
<td>38</td>
<td>26</td>
<td>27</td>
<td></td>
<td>28.0</td>
<td>7.6</td>
</tr>
<tr>
<td>xef</td>
<td>12</td>
<td>11</td>
<td>7</td>
<td>14</td>
<td>11</td>
<td>21</td>
<td>10</td>
<td>15</td>
<td>17</td>
<td>11</td>
<td>13</td>
<td></td>
<td>12.9</td>
<td>3.8</td>
</tr>
</tbody>
</table>

Table 5: Percentages required for each program for x%-selective mutation to achieve the mutation scores achieved by the exp and eff set of mutation operators

5.1 Experiment method

This experiment was carried out in the same way as the single operator experiment described previously except that sets of operators replaced individual operators and equivalent mutants had already been identified. In particular, the same method was used to create a sample of 10 adequate test sets for the mutants of each set of operators and x% of the mutants.

The previously given definitions for score and cost may be applied to sets of operators and to x%-selective mutation by extending them in a natural way to apply to test sets that are adequate for the mutants of a set of operators or the mutants of x%-selective mutation. Similarly, the relations asEfficient and moreEfficient may be generalised to sets of operators and x%-selective mutation and in general to any testing criterion for which coverage and cost measures are available.

5.2 Results and analysis

For each of the 11 programs studied, Table 6 shows the mutation score achieved by the five selective mutation strategies studied using non-redundant test sets. The two rightmost columns show the mean and standard deviation. The efa scores are very similar to the exp set. Note that Offutt et al. (1996) also obtained a mutation score of 99.5%
for the \textit{exp} set of operators but by using effective sequences. The score of the \textit{eff} set is significantly lower than \textit{exp} or \textit{efa}.

For each of the 11 programs studied, Table 7 shows the costs of mutation analysis for the five selective mutation strategies studied and again the two rightmost columns show the mean and standard deviation. It can be seen that the \textit{exp} and \textit{efa} strategies have very similar costs. Since these two sets have similar scores, neither is more efficient than the other. The cost of \textit{xex}, however, is significantly lower than \textit{exp}. Given that \textit{xex} and \textit{exp} have similar scores, it follows that \textit{xex} is more efficient than \textit{exp}. The cost of \textit{eff} is significantly lower than \textit{xef}, the \textit{x\%}-selective strategy with comparable score. Although the score of \textit{eff} is slightly lower than \textit{xef} the difference is much less than the standard deviation and so it is reasonable to conclude that \textit{eff} is more efficient than \textit{xef}. The efficiency relationships in general can be determined from Figure 4. Again, each rectangle in this figure shows the variability associated with an operator’s score and cost. The width of a rectangle is equal to the standard deviation of the cost and the height is equal to the the standard deviation of the score.

The cost saving of \textit{xex} over \textit{exp} is due entirely to the lower equivalent mutant cost of \textit{xex}. The same is true of the cost saving of \textit{eff} over \textit{xef}. This can be seen from Table 8 which shows that on generation cost alone, \textit{xex} and \textit{exp} are as efficient as each other and the same is true of \textit{eff} and \textit{xef}.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Op.</th>
<th>ARSG</th>
<th>BANK</th>
<th>CALV</th>
<th>FCNT</th>
<th>MINV</th>
<th>PAT0</th>
<th>RPC</th>
<th>SEQS</th>
<th>SORT</th>
<th>STRE</th>
<th>TRET</th>
<th>mean</th>
<th>sd</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>99.9</td>
<td>98.4</td>
<td>99.4</td>
<td>99.7</td>
<td>99.6</td>
<td>98.8</td>
<td>99.8</td>
<td>99.8</td>
<td>99.8</td>
<td>99.8</td>
<td>99.8</td>
<td>99.8</td>
<td>99.5</td>
<td>0.5</td>
</tr>
<tr>
<td>efa</td>
<td>99.9</td>
<td>98.3</td>
<td>99.3</td>
<td>99.8</td>
<td>99.5</td>
<td>98.0</td>
<td>99.8</td>
<td>99.8</td>
<td>99.8</td>
<td>99.8</td>
<td>99.8</td>
<td>99.8</td>
<td>99.4</td>
<td>0.6</td>
</tr>
<tr>
<td>xef</td>
<td>99.7</td>
<td>99.2</td>
<td>99.4</td>
<td>99.5</td>
<td>99.6</td>
<td>99.8</td>
<td>99.7</td>
<td>99.8</td>
<td>99.8</td>
<td>99.8</td>
<td>99.6</td>
<td>99.0</td>
<td>99.4</td>
<td>0.6</td>
</tr>
<tr>
<td>xex</td>
<td>99.7</td>
<td>97.0</td>
<td>99.0</td>
<td>99.5</td>
<td>99.0</td>
<td>98.0</td>
<td>99.8</td>
<td>99.4</td>
<td>99.2</td>
<td>99.2</td>
<td>99.2</td>
<td>98.5</td>
<td>98.9</td>
<td>0.8</td>
</tr>
<tr>
<td>eff</td>
<td>99.9</td>
<td>97.0</td>
<td>99.0</td>
<td>99.5</td>
<td>99.0</td>
<td>98.0</td>
<td>99.8</td>
<td>99.4</td>
<td>99.2</td>
<td>99.2</td>
<td>99.2</td>
<td>98.5</td>
<td>99.4</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 6: Scores of selective mutation strategies for each program using non-redundant test sets

Table 9 shows the mean scores and costs obtained with the use of effective sequences for the five selective
mutation strategies studied. Compared to the use of non-redundant test sets, the scores are slightly higher, as would be expected and the generation costs are slightly higher but otherwise the results obtained with non-redundant test sets are confirmed.

<table>
<thead>
<tr>
<th>Score</th>
<th>Cost</th>
<th>Gen Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op.</td>
<td>mean</td>
<td>sd</td>
</tr>
<tr>
<td>exp</td>
<td>99.7</td>
<td>0.3</td>
</tr>
<tr>
<td>efa</td>
<td>99.6</td>
<td>0.6</td>
</tr>
<tr>
<td>xex</td>
<td>99.7</td>
<td>0.1</td>
</tr>
<tr>
<td>eff</td>
<td>99.2</td>
<td>0.6</td>
</tr>
<tr>
<td>xef</td>
<td>99.4</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Table 9: Mean score and costs (as a %) of selective mutation strategies using effective test sequences

6 Discussion

Non-redundant test sets were used in this study in the hope that they would permit clearer differentiation of high scoring selective mutation strategies. Examination of the results of both experiments obtained using non-redundant test sets with those obtained using effective test sequences show that there is little to distinguish them and so there is no real evidence that non-redundant test sets are more useful for the kind of experiment carried out in this study.
Table 11: Relative equivalent mutant costs of expression operators and $x\%$-selective mutation for each program

<table>
<thead>
<tr>
<th>Programs</th>
<th>ARSG</th>
<th>BANK</th>
<th>CALV</th>
<th>FCNT</th>
<th>MINV</th>
<th>PAT0</th>
<th>RPC</th>
<th>SEQS</th>
<th>SORT</th>
<th>STRE</th>
<th>TRET</th>
<th>mean</th>
<th>sd</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>72.3</td>
<td>57.5</td>
<td>79.2</td>
<td>81.6</td>
<td>55.7</td>
<td>93.2</td>
<td>70.6</td>
<td>91.8</td>
<td>67.4</td>
<td>59.0</td>
<td>81.0</td>
<td>73.6</td>
<td>13.1</td>
</tr>
<tr>
<td>eta</td>
<td>72.3</td>
<td>56.8</td>
<td>75.7</td>
<td>82.6</td>
<td>55.7</td>
<td>93.2</td>
<td>68.2</td>
<td>92.5</td>
<td>65.2</td>
<td>56.4</td>
<td>80.0</td>
<td>72.6</td>
<td>13.6</td>
</tr>
<tr>
<td>xex</td>
<td>25.1</td>
<td>24.7</td>
<td>14.6</td>
<td>28.6</td>
<td>23.6</td>
<td>40.7</td>
<td>24.2</td>
<td>29.1</td>
<td>39.1</td>
<td>23.1</td>
<td>26.7</td>
<td>27.2</td>
<td>7.3</td>
</tr>
<tr>
<td>eff</td>
<td>9.4</td>
<td>5.5</td>
<td>9.7</td>
<td>8.2</td>
<td>8.1</td>
<td>3.4</td>
<td>8.5</td>
<td>4.5</td>
<td>9.8</td>
<td>1.9</td>
<td>6.7</td>
<td>6.9</td>
<td>2.7</td>
</tr>
<tr>
<td>xef</td>
<td>11.6</td>
<td>12.3</td>
<td>6.2</td>
<td>13.3</td>
<td>9.7</td>
<td>15.2</td>
<td>12.8</td>
<td>11.9</td>
<td>15.2</td>
<td>7.0</td>
<td>9.5</td>
<td>11.4</td>
<td>3.0</td>
</tr>
</tbody>
</table>

The findings reported in this paper naturally depend on the specific definition of cost that has been adopted. Unlike related studies, this study measures the cost of detecting equivalent mutants. If the cost of detecting equivalent mutants is ignored then it can be seen from Table 8 that selective mutation strategies that achieve similar scores have similar costs and there are no efficiency gains between the strategies considered in this paper.

The cost of detecting equivalent mutants was measured as the sum of two equally weighted components, namely the relative number of statements with equivalent mutants and the relative number of equivalent mutants. The equal weighting of these two components is somewhat arbitrary. The results contained in Tables 10 and 11 show that the lower costs of the efficient strategies, xex and eff is more pronounced in the relative number of equivalent mutants. If therefore, the relative number of statements with equivalent mutants is given more weight, the overall cost will increase for these strategies. Table 12 shows the costs of the selective mutation strategies when the relative number of equivalent mutants is ignored. These costs show that xex and eff are still more efficient than their respective competitors. It is thus reasonable to conclude that the findings of this paper are not unduly sensitive to the specific weighting of the two components of equivalent mutant cost.

<table>
<thead>
<tr>
<th>Cost</th>
<th>Op.</th>
<th>mean</th>
<th>sd</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>56.6</td>
<td>7.2</td>
<td></td>
</tr>
<tr>
<td>eta</td>
<td>57.5</td>
<td>7.3</td>
<td></td>
</tr>
<tr>
<td>xex</td>
<td>47.8</td>
<td>9.9</td>
<td></td>
</tr>
<tr>
<td>eff</td>
<td>15.0</td>
<td>4.7</td>
<td></td>
</tr>
<tr>
<td>xef</td>
<td>27.4</td>
<td>7.5</td>
<td></td>
</tr>
</tbody>
</table>

Table 12: Mean costs (as a %) of selective mutation strategies using non-redundant test sets and equivalence cost measured as equivalent statement cost only

The findings reported in this paper may well depend on the size of the programs considered. The costs might be more vulnerable to scale effects since these are directly affected by the number of mutants generated. Offutt et al. (1996) showed that the number of mutants generated by a program is of the order of the product of the number of values and the number of value references whereas the number of mutants generated by the exp operators is of the order of the number of value references, a metric that is approximately linear with respect to program size. The implication of this is that the cost of a selective mutation strategy is likely to depend on the size of the program under test. Mutation scores are also vulnerable to program size. It is not known, for example, if the exp set of operators maintains its 99.5% mutation score as program size increases.
7 Conclusions

This paper differs from previous work on selective mutation in that instead of looking for strategies to simply reduce cost with the implicit assumption that a small loss of test set adequacy is acceptable, no assumptions are made about what is an acceptable loss of test set adequacy and considers, instead, the efficiency of selective mutation strategies. Since cost reductions from more efficient strategies are not made at the expense of test set effectiveness, efficiency comparisons are likely to be more useful to software testers.

The work reported in this paper also differs from previous work that has considered the cost benefits of selective mutation in that it considers the cost of detecting equivalent mutants, a significant part of the cost of mutation testing. This cost is measured not only in terms of the number of equivalent mutants but also in terms of the number of program statements that must be examined.

This paper defines the score and cost of an individual mutation operator, and then reports the results of an experiment to establish these properties. By comparing operators in terms of both score and cost it is possible to identify the most efficient individual operators. Efficiency relationships are of course statistical. However, by adopting the heuristic that operators that are efficient individually will also produce an efficient set of operators, an efficient set of operators was proposed for selective mutation testing. The score and cost of this set was compared against other sets of operators and $x\%$-selective mutation.

The appropriate selective mutation strategy to use in practice will, of course, depend on the testing requirements and the resources available. However, the results of this paper show that if a mutation score very close to 100\% is required then $x\%$-selective mutation is likely to be more efficient than selective mutation based on only the expression set of operators i.e. \{abs, aor, lcr, ror, uoi\}, and also more efficient than selective mutation based on only the set of operators \{abs, aor, san, sdl, ror, uoi\} which achieves a similar mutation score. The results also show that if less stringent test coverage is required then selective mutation based on a restricted set of efficient operators i.e. \{aor, san, sdl, ror, uoi\} is likely to be more efficient than $x\%$-selective mutation. These cost savings result from differences in the cost of equivalent mutant detection.

On reflection, it seems intuitive that if a high mutation score is required then the sampling distribution for efficient selective mutation should be similar to the population distribution. This suggests that there are a few mutants of each operator that cannot in general be killed by tests adequate for other operators. If, however, the tester is willing to allow such mutants to remain alive then significant savings can be made by using the most efficient operators.

8 Further work

The experiments reported in this paper were very time consuming and so only 11 programs were used. It is difficult to know if 11 programs is a sufficiently large sample from which to generalise and so similar studies on larger sets of programs will be useful. In particular, it would be interesting to see how the findings reported here depend on the size of the program under test.

All the results of this study have been obtained using the set of mutation operators used by Mothra. Clearly, these results cannot be applied directly to mutation systems that use different operators. Efficiency relationships will, nonetheless, be present between any set of operators and it would be interesting to know if selective mutation based on an efficient set of operators and $x\%$-selective mutation are generally related in the way reported in this paper.

ACKNOWLEDGEMENTS

The authors would like to thank the Software Engineering Research Centre at Purdue University for allowing us to use the Mothra mutation tool. The authors also would like to thank Prof. A. J. Offutt at ISSE Department, George Mason University for providing a very useful Mothra interface. Finally, the authors would like to thank the anonymous referees who made a number of helpful comments.

References


Lipton, R. J. and F. G. Sayward (1978, December). The status of research on program mutation. In Digest for the workshop on software testing and test documentation, pp. 355–373.


Appendix A: Subject Programs

The eleven programs used in this study are described below. Source listings are available from the contact author.

AREASG calculates the areas of the segments formed by a rectangle inscribed in a circle. The areas are calculated analytically and also estimated using a monte-carlo method. The number of iterations of the monte-carlo method is input. The estimated areas are returned together with the overall percentage error of the estimates at ten equally spaced points in the iteration.

BANKER This function (Offutt et al. 1993) implements a deadlock avoidance algorithm for processes competing for resources and detects whether a process’s request for resources might possibly lead to deadlock. The algorithm simulates granting the request, and then ensures that upon doing so, all other processes can somehow complete even if they make their maximum claim on resources.

CALVAL calculates the number of days between the two given days which must be in the same year. Adapted from the program CAL originally taken from Budd’s thesis, (Budd 1980). CALVAL extends CAL to validate the input data and returns a negative number of days if the first input day follows the second input day.

FINDCNT This program is a version of the FIND subroutine due to C.A.R.Hoare modified by DeMillo et al. (1978) and used in several studies (Mathur and Wong 1994; Offutt et al. 1996; Offutt et al. 1993). FIND subroutine partitions an integer array. It takes two inputs, an array A and an index F, and permutes the elements of A so that elements to the right of position F are greater than or equal to A[F] and elements to the left of position F are less than or equal to A[F]. For the purpose of this study, the FIND subroutine was modified to ensure that all the mutation operators used in Mothra were applicable. This modified version, named FINDCNT, stores the elements of A in reverse order in another array B. It also counts the elements that lie within a specific range of A and stores the result in a variable COUNT.

MINV This program (Borse 1991) computes the inverse of the square N by N matrix A. The matrix is destroyed in the process. The determinant is calculated and returned as DET. If DET is non-zero, the inverse is computed and returned in matrix AINV.

PAT0 is a function that decides if a given sequence of integers, the pattern, is embedded within another sequence, the subject. The position in the subject where the pattern starts is returned and the elements in the subject that are not part of the pattern are set to zero. If the pattern is not in the subject, 0 is returned.

RPCALC calculates the value of a reverse polish expression using a stack. The expression is input as an integer array, a parallel input array contains tag values to identify operators and operands. In addition to the value of the expression, an error value reports division by zero, stack underflow and syntax errors.

SEQSTR locates sequences of integers within an input array and copies them to an output array. The input array contains integers in the range [0, 9] and an input sequence is defined as a sequence of consecutive integers ending in an integer less than three. Input sequences are inserted in the output array in ascending order. If no input sequences are present, the elements of the output array are set to minus one.

SORT This program (Borse 1991) executes a selection sort using an index. It takes as input an array X. The array X is sorted, smallest to largest, and is not destroyed. The proper sequence of elements will be contained in the array INDEX(). Thus, X(INDEX(1)) and X(INDEX(n)) will be the smallest and largest elements of the array. The program was modified such that the squares of all the elements of X are computed and stored, in ascending order, in an array Z.
**STREQL** compares two strings after replacing consecutive white space characters with a single space. If both strings are empty or consist of a single space then zero is returned. If both strings are non-empty and match then their common length is returned. A return value of minus one indicates no match and a return value of minus two indicates an illegal input.

**TRETRV** performs an in-order traversal of a binary tree of integers to produce a sequence of integers. If this sequence is monotonic increasing, the output is three. If the sequence is monotonic decreasing, the output is two. If the sequence is both monotonic non-decreasing and non-increasing the output is one. A non-monotonic sequence produces an output of minus one. The binary tree is input as three arrays, NODE, LEFTLINK and RIGHTLINK. An illegal input in which a node has more than one parent produces an output of minus two and an invalid link produces an output of minus three.