Mutant Hierarchies Support Selective Mutation

Kalpesh Kapoor
Department of Mathematics,
Indian Institute of Technology Guwahati 781 039, India.
E-mail: kalpesh@iitg.ernet.in

Keywords: fault-based testing, mutation testing, first-order mutants, selective mutation

Received: October 25, 2009

Mutation testing attempts to assess the quality of a test set by its ability to distinguish the program under test from its mutants. One of the main difficulties faced in practice is due to the large number of mutants that can be generated for a program under test. Earlier research to solve this problem has suggested variants of mutation testing, and finding an effective set of mutation operators referred to as selective mutation. This paper presents an alternative approach for reducing the cost of testing by identifying hierarchies among first-order mutants. The key idea is to evaluate the strength of a mutant with respect to other mutants and ignore "weaker" mutants during testing. Unlike previous approaches, our method is formal and it is guaranteed that the effectiveness of a test suite will be identical with that can be achieved using all mutants. The theory described here is also applicable to the quantitative assessment of testing effort and can be used to guide successive testing steps in fault-based testing. We present an empirical evaluation to find reduction in the test effort using mutant classification and show that it supports selective mutation.

Povzetek: Metodo testiranja z mutanti so izboljšali s hierarhijo mutacij, ki izloča slabše mutante.

1 Introduction

Fault-based [23] testing approach relies on generating test set that can guarantee to detect all hypothesized faults in a Program Under Test (PUT). A fault is a manifestation of an error, for example, misunderstanding about the semantics of an operator. A failure is the inability of the system or a system component to perform a function as dictated by the specification [15, 37]. In other words, a fault is locally incorrect (computational or control) operation that, when propagated, results in a failure [29, 35].

Mutation testing [4, 6] is a fault-based approach to test programs written in an imperative programming language. In mutation testing, a set of programs is generated by making a single (well-defined) syntactic change in a given PUT. This set of programs, referred to as first-order mutants, are used for evaluating a test set. A test set is a collection of test cases where each test case is a set of inputs with expected output values for a PUT.

A mutant is said to be *killed* by a test set if it can distinguish the mutant to be different from a PUT. Given a test set, its effectiveness, defined as *mutation score*, is measured by computing the percentage of first-order mutants that it killed with respect to the total number of non-equivalent mutants. Since the semantics of a PUT is not considered while generating mutants, some of them could be semantically equivalent to the PUT. Such mutants have to be identified manually or by other methods [26] and their number is reduced from the total number of mutants before computing the mutation score. In rest of the paper we refer to non-equivalent first-order mutants as simply mutants.

The two test hypotheses [11, 13] that form the basis for mutation testing are competent programmer; and coupling effect [6]. The former assumes that programmers are competent, however, they make small mistakes while writing programs. These small mistakes can be modelled as syntactic changes such as replacing $>$ by $\leq$ etc. and are referred to as mutation operators. In practice, it is expected that the set of mutation operators either represent the commonly occurring faults or they enable generation of test cases that can expose complex faults. Thus the benefits of mutation analysis depends on the mutation operators that are used to generate the mutants from a PUT.

During mutation testing only those mutants are considered that can be obtained by making single syntactic change. Those mutants that can be obtained by making multiple changes, called as higher-order mutants, in a PUT are ignored. The basis for this is the coupling effect hypothesis which states that if a test set can guarantee killing first-order mutants then it is also likely to guarantee the same for higher-order mutants. Coupling hypothesis has been investigated both theoretically [19, 36] and empirically [25] and is found to hold for several fault classes. However, there is a recent research on higher order mutants by Jia and Harman [17], which suggests that some strongly subsuming higher order mutants are in fact harder to kill than some

A preliminary version of the theoretical foundation of this paper appeared in: K. Kapoor and J. P. Bowen, Ordering mutants to minimise test effort in mutation testing, proceedings of 4th International Workshop on Formal Approaches to Testing of Software (FATES), Springer-Verlag, LNCS, Volume 3395, pages 195–209, September 2004.

first order mutants.

If a PUT gives correct results for all the test cases in a test set with 100% mutation score then it is concluded that the PUT is correct with respect to the faults represented by the mutation operators. Thus such a test set is good at distinguishing a program from its mutants and, if the program is faulty, the test set is also likely to be good at distinguishing the program from a correct program [13].

Thus, mutation testing provides a means of evaluating a test criteria and test sets. However, with the increase in size of implementation, it is computationally expensive or infeasible to consider all possible mutants that can be hypothesized as the number of possible mutants is usually very large which makes the testing process expensive [14, 28, 27, 39].

The number of possible mutants is proportional to the product of the number of variables with the count of number of times they are referred either in a definition or in use[27, 38]. A consequence of the generation of a large number of possible mutant programs is that they need to be executed in each step of the testing phase till an adequate test set is obtained. To overcome this problem, a number of approaches have been suggested, such as finding an effective set of mutation operators, referred to as selective mutation [27], and variants of mutation testing [14, 40]. The original idea, as described above, is referred to as *strong mutation testing*.

This paper presents an alternative approach for reducing the cost of mutation testing by the identification of hierarchies among mutants. Let P_k and P_j be two mutants of a given PUT. It is possible to reduce test effort by considering only P_k if it can be deduced that a test set which can kill P_k is also be guaranteed to kill P_j.

The approach described here is also applicable to the quantitative assessment of testing effort and can be used to guide successive testing steps in fault-based testing. In particular, the objectives of this paper are:

a. To give theoretical foundation for identifying the relationship among mutants and show that mutation operator cannot be ordered without reference to a PUT;

b. To empirically evaluate the reduction in test effort that can be achieved by identifying the relationship among mutants;

c. To find if selective mutation study [27] is supported by our study.

The rest of the paper is organized as follows. The next section presents formal definitions in the context of mutant hierarchies. Section 3 describes the properties and conditions to identify the relationship among mutants. Section 4 gives an overview of the related work. An empirical study conducted to evaluate our approach and compare it with selective mutation is described in Section 5. Finally, conclusions are presented in Section 7.

2 Mutant Hierarchies

The theoretical and empirical study presented in this paper is done for the programs written in a subset of C programming language which include constructs such as loop, array and function calls. However, this does not impose any restriction on the use of those language constructs in programs that are not included in the subset.

For the purpose of theoretical analysis below, we assume that the statements, and predicates in conditional statements (such as `if` and `while`), of a program are uniquely labelled. Boolean conditions are used solely for deciding the branch to be followed in the next step of the execution and therefore are assumed not to modify the state of a program during execution. The assumption is justified as program transformation techniques can be used to achieve this and make programs more testable [12].

A label when given to a Boolean condition is said to be a *p-location*, otherwise it is said to be a *c-location*. Note that a condition in an **if** or **while** statement is given a unique label (i.e., different from the labels that are given to statements that appear inside the **then-else** or body of a **while** statement, respectively). Let l_k and l_j be two locations in P that are mutated to obtain mutants P_k and P_j respectively. These mutants will be known as *intra-location* mutants of P if $l_k = l_j$, otherwise they will be referred to as *inter-location* mutants.

We denote the output obtained on execution of a program M with an input x by $M(x)$. The notation $M = M'$ will be used to signify that a program M is semantically equivalent to another program M'. The state of a program under execution at an instant is set of pairs of variable and their corresponding values. A test case is a pair of input and the expected output. For simplicity, we will use input and test case interchangeably.

Definition Let P be a PUT and P_k be a first-order mutant that differs from P at a location. A test case, t, is said to kill a mutant P_k if one of the following conditions hold:

a. $P(t) \neq P_k(t)$, where both $P(t)$ and $P_k(t)$ are non-erroneous states.

b. either $P(t)$ or $P_k(t)$, but not both, results in an erroneous state.

c. both $P(t)$ or $P_k(t)$ results in different erroneous states.

We say a test set kills P_k if it includes a test case that kills P_k. Thus, a test case that kills a mutant identifies that a PUT and its mutant represent two distinct functions. For comparison, we observe the final internal state of programs.

During an execution of a program, it may fail, for example due to division by zero or insufficient memory, we call such a state an erroneous state. It may happen that one of the program fails while the other does not, in which case they are obviously distinguishable as stated in the definition 2(b) above. We classify non-termination of a program among the erroneous state. The definition 2(c) includes the case where both programs result in distinguishable erroneous states such as a floating point exception and a memory fault.

Our analysis remains applicable even if we change the definition with respect to other variants: weak [14] and firm [40] mutation testing which allow to distinguish a PUT and its mutant by observing their internal states that are not final. A partial order between mutant programs can be defined using the following relation.

Definition [Relation between Mutants] Let P_k and P_j be the two mutants of P and t be a test case. Then P_k is said to be stronger than P_j denoted by $P, t \vdash P_k \geq_m P_j$ if

$$\exists\, t \mid t \text{ kills } P_k \Rightarrow t \text{ kills } P_j$$

The notation $P, t \vdash P_k \not\geq_m P_j$ will be used to indicate that P_k is not stronger than P_j. For a pair of mutants, P_k and P_j, of P if both $P, t \vdash P_k \not\geq_m P_j$ and $P, t \vdash P_j \not\geq_m P_k$ hold for all test cases t in a test set then both P_k and P_j must be considered during mutation testing.

Definition [Mutant Class] A set of mutants, S, of a PUT is said form a mutant class if there exists a test case that kills all the mutants in S.

Note that the relation among the mutants and the definition of mutant class are within the context of one test case. In other words, the mutant classes are induced by a test set. Therefore, for a given set of mutants, the mutant classes may be different with respect to different test sets. An alternative way to define the mutant relation, $\geq_m$, could be on the basis of comparing the constraints of all possible test inputs that kills a mutant rather than just by one single test case. This would make mutant classes unique for a given program and independent of test cases. However such a strong requirement will be hard and expensive to analyse in comparison to our weaker definition.

Fact 1. *[26, 29, 35] A test case, t, can kill P_k provided the following necessary and sufficient conditions hold on executing P and P_k with input t:*

a. the execution must reach location l (reachability);

b. the evaluation of expressions at location l in P and P_k must result in different values at least once (infection);

c. the final states on termination of execution of P and P_k must be different (propagation).

Condition (b) (i.e., infection) has been referred to as *necessity* in [26], and the *original state failure condition* in [29] consisting of an *origination condition* and *computational transfer conditions.*

Given a mutant P_k which is obtained by applying a mutation operator at a location l in a PUT with input domain D, let subdomain $D^r_k \subseteq D$ be the set of inputs which reaches location l; similarly, $D^i_k \subseteq D$ be the set of inputs that can cause the original and mutated expression at the location l to result in different values and $D^p_k \subseteq D$ be the set that causes P and P_k to result in different final outcomes.

```
    int fun(int x, int i) {
L1:   while (i <= 2) {     fun1          fun2
L2:     if (x <= 4)      | if (x < 4) |  | if (x > 4) |
L3:       x = x + 1;
L4:     else
L5:       x = x + 2;
L6:     i = i + 1;
L7:   }
L8:   return x;
    }
```

Figure 1: An example to illustrate the insufficiency of infection conditions.

Fact 2. *[26] Given P, a test case, t, will kill P_k iff $t \in D^p_k$ which implies $t \in D^r_k \cap D^i_k$ and $D^p_k \subseteq D^r_k \cap D^i_k$.*

Note that there may be test cases in D^i_k that does not satisfy the reachability condition. The computation of a test case that can kill a mutant is undecidable as the sets D^r_k, D^i_k and D^p_k cannot be computed, in general. However, in practice it is often possible to find such test cases using approximation techniques. On one hand, to compute (whenever feasible) the set D^i_k requires only analysis of the expression at location l. On the other hand, computation of D^r_k is more expensive and complex as it requires analysis of the paths that can reach location l.

Proposition 1. *$\forall t (P, t \vdash P_k \geq_m P_j) \Leftrightarrow D^p_k \subseteq D^p_j$, where P_k and P_j are mutants of P and t represents a test case.*

Proof. The proof follows from the definitions. □

3 Identifying Mutant Hierarchies

A brute-force method to identify $\geq_m$ relation is by checking if $D^p_k \subseteq D^p_j$. It is also possible to restrict the test cases to be selected from the set $D^p_k \cap D^p_j$, provided that this set is not empty, in which case killing P_k will also guarantee the same for P_j.

The objective of our analysis is to identify a subset of mutants with the same effectiveness as the whole set without generating all mutants. Therefore, if possible, the $\geq_m$ relation between mutants should be established during their generation itself, thereby only producing the strongest mutants. This approach is an improvement over the method where mutants are first generated explicitly and then an attempt to establish a partial order among them is made.

3.1 Intra-location Mutants

Let P be an implemented program and P_k and P_j be two mutants of P that are obtained by applying mutation operator at location l in P. Let C_k and C_j be the predicates that correspond to the sets D^i_k and D^i_j, respectively.

Now consider the example program shown in Figure 1. The mutants fun$_1$ and fun$_2$ are shown in boxes and are obtained by applying mutation operator at location $L2$, where

Input		Output		
x	i	fun	fun$_1$	fun$_2$
3	1	5	6	6
4	1	7	8	7

Table 1: A counter example for Theorem 1 based on the program shown in Figure 1.

```
     int Comp(int x) {  Comp1        Comp2
L1:    x = x + 1;      | x = x − 1; | | x = x + 2; |
L2:    if (x == 5 | | x == 7)
L3:      x = 9;
L4:    else
L5:      x = 6;
L6:    return x;
     }
```

Figure 2: An example for Theorem 1.

$<=$, $<$ and $>$ are relational operators. The conditions (obtained by taking exclusive-or of two expressions, for instance, $C_{fun_1} \equiv x <= 4 \oplus x < 4$, where $\oplus$ is exclusive-or operator) C_{fun_1} and C_{fun_2}, in this case are x = 4 and **true**, respectively. Although C_{fun_1} implies C_{fun_2}, it is not sufficient to claim a hierarchy between fun$_1$ and fun$_2$, in general. This is illustrated by the following theorem.

Theorem 1. *Let P be a PUT and P_k and P_j be its two mutants obtained by mutating a statement at location l then, $D^i_k \subseteq D^i_j$ does not guarantee $\forall t(P, t \vdash P_k \geq_m P_j)$.*

Proof. The statement holds for both a p-location and a c-location. We give counter-examples as a proof. Table 1 gives the output of two test cases for the program and its two mutants that are shown in the Figure 1. The first row shows that the mutants are not equivalent to the original program, whereas the second row shows that the conjecture is true for p-locations as fun_1 is killed and fun_2 remains live. Note that, as mentioned above, $C_{fun_1} \Rightarrow C_{fun_2}$ i.e. $D^i_{fun_1} \subseteq D^i_{fun_2}$.

Figure 2 shows a concrete example that illustrates the above theorem for a c-location. Consider the two mutants, Comp$_1$ and Comp$_2$, obtained by applying mutation operator at location $L1$ in Comp (see Figure 2). The D^i sets for both mutants is the whole input domain D since the mutated statements would result in different values for any integer input. In other words, the conditions C_1 and C_2 are *true*. However, input $x = 8$ will kill Comp$_1$; whereas input $x = 3$ will kill Comp$_2$ (see Table 2). Thus, mutants Comp$_1$ and Comp$_2$ are not related under $\geq_m$ with respect to all possible test cases. □

Input (x)	Output		
	Comp	Comp$_1$	Comp$_2$
8	6	9	6
3	6	6	9

	Comp$_1$	Comp$_2$
D^i	D	D
D^p	$\{4, 8\}$	$\{3, 4, 5, 6\}$

Table 2: Input, output and subdomains for the programs shown in Figure 2.

The following formal observation gives insight into Theorem 1.

$$\begin{array}{ll} D^p_k \subseteq D^r_k \cap D^i_k & \text{(Fact 2)} \\ D^p_j \subseteq D^r_j \cap D^i_j & \text{(Fact 2)} \\ D^r_k = D^r_j & \text{(Same location mutants)} \\ D^i_k \subseteq D^i_j & \text{(given)} \end{array}$$

The most favourable conclusion that can be drawn from the above statements is that both D^p_k and D^p_j are subsets of $D^r_k \cap D^i_k$. But this does not guarantee $D^p_k \subseteq D^p_j$ (as required by Proposition 1).

Theorem 2. *Let P, P_k, P_j and l be the entities as stated in Theorem 1. If $D^i_k \cap D^i_j = \emptyset$ then P_k and P_j are not related under $\geq_m$ P or P is equivalent to P_k or P_j.*

Proof. The first three conditions are identical with the above formal observation.

$$\begin{array}{lll} & D^i_k \cap D^i_j = \emptyset & \text{(given)} \\ \Rightarrow & D^p_k \cap D^p_j = \emptyset & \text{(set theory)} \\ \Rightarrow & D^p_k = \emptyset \vee D^p_j = \emptyset \vee & \\ & (D^p_k \not\subseteq D^p_j \wedge D^p_j \not\subseteq D^p_k) & \text{(set theory)} \\ \Leftrightarrow & P = P_k \vee P = P_j \vee & \\ & \quad \forall t\ (P, t \vdash P_k \ngeq_m P_j \wedge & \\ & \qquad P, t \vdash P_j \ngeq_m P_k) & \text{(Fact 2 \& Prop. 1)} \end{array}$$

□

This is particularly helpful in *isolating* those mutants that definitely need to be considered during testing. However, a hierarchy can be established between mutants under certain conditions. These conditions are discussed below.

Theorem 3. *Let P be a given PUT and l be a p-location that corresponds to a condition, c. Further, let c be mutated to c′ and c′′ giving mutants P_k and P_j, respectively. If ($c' \Leftrightarrow c''$) then one of the mutant P_k or P_j need not be considered during testing.*

Proof. As per Fact 1 (c), P and P_r ($r \in \{k, j\}$) must follow different paths after reaching location l (sometime during an execution) in order to be killed.

The condition ($c' \Leftrightarrow c''$) ensures that condition c' and c'' always evaluate to the same Boolean value. Thus, for a given test case, the path followed by P_k and P_j will always be the same, ensuring that the infection and propagation conditions for both P_k and P_j will be identical i.e. $\forall t(P, t \vdash P_1 \geq_m P_2 \wedge P, t \vdash P_2 \geq_m P_1)$. □

The above condition in Theorem 3 is a very strong requirement. However, the property was found to be helpful in reducing the test effort during empirical study described in Section 5.

Remark 1. *Why do we need $c' \Leftrightarrow c''$ condition to hold in general? To answer this question, let us consider the two mutants P_k and P_j, obtained by mutating a condition for a* **while** *loop of P. For a given test case, let i, i_k and i_j be* *the number of times the* **while** *loop is executed in P, P_k and P_j, respectively. Thus, the necessary conditions for killing P_k and P_j are $i \neq i_k$ and $i \neq i_j$ respectively. To establish $\forall t(P, t \vdash P_k \geq_m P_j)$, one of the following properties must hold:*

a. $i_k = i_j$, or

b. the resulting states must be the same after executing the body of **while** *loop i_k and i_j times.*

The condition in Theorem 3 is equivalent to (a) above. However, the condition (b) is equally acceptable, but requires analysis of the program segment to guarantee that it holds for any test case that kills P_k; this may be difficult to establish. It is also possible to weaken the requirement in Theorem 3 under certain conditions as described by the following theorem.

Theorem 4. *Let P be a PUT, t be a test case and l be a p-location that corresponds to a condition, c, in P. Further, let c be mutated by two operators to c' and c'' giving mutants P_k and P_j, respectively. If $(c \oplus c') \Rightarrow (c \oplus c'')$, where $\oplus$ is exclusive-or operator, and the label l is reached during the execution exactly once then $P, t \vdash P_k \geq_m P_j$.*

Proof. The programs P and P_k must follow different paths after reaching location l, sometime during an execution, in order to be distinguished. The condition $(c \oplus c') \Rightarrow (c \oplus c'')$ ensures that P_k and P_j will follow the same path, if P and P_k take different paths.

The necessity for the criterion of checking the internal state can be explained as follows. Assume that the Boolean condition is evaluated twice and c' differs from c in the second execution. However, it is possible that only c'' may differ from c in the first execution but not in the second execution, in which case it is not guaranteed that killing P_k will also ensure the same for P_j. □

Thus, Theorem 3, Remark 1 and Theorem 4 give three different possibilities to identify the hierarchies among mutants and also present the reasoning for the conditional requirements associated with them.

Note that in Theorem 4, c and c'' may differ, but c and c' may evaluate to the same values, in which case P_j may be killed but P_k will not. Thus P_j could be killed by more test cases than P_k. This can also be observed by noting that $c \oplus c'$ defines the subdomain D_k^i and considering the implication as a subset relation.

3.2 Inter-location Mutants

The above analysis is restricted to mutations in p-locations. For mutants that can be generated by mutating c-locations, we need to symbolically propagate the effect to nearest variable use in a predicate and then use theorems mentioned above to identify the relationship among mutants. We illustrate this by an example:

Consider the fragment of a PUT, R, shown in Figure 3. Let R_1, R_2 and R_3 be three mutants of R as shown in the figure. If the definition of variable b at location L_a is guaranteed to reach location L then we can compare R_1, R_2 and R_3 under $\geq_m$. Let `orig`, `r1`, `r2` and `r3` be predicates as defined below:

```
                          R1
La: b = -y;          | b = y; |
     :                               R2                   R3
L:  if (x + y == a + b) | x - y == a + b | | x + y == a - b |
        St
    else
        Se
    end
```

Figure 3: Fragment of a program R.

```
orig: x + y == a + b
  r1: x + y == a + y
  r2: x - y == a + b
  r3: x + y == a - b
```

By observing that the predicate

$$b == -y \Rightarrow ((orig \oplus r2) \Rightarrow (orig \oplus r3))$$

is valid, we conclude that $R_2 \geq_m R_3$. To check the relation of R_1 with R_2, we propagate the assignment at L_a to L which give us the following predicates:

```
orig': x + y == a - y
  r2': x - y == a - y
```

Since the predicate $(orig' \oplus r2') \Rightarrow (orig' \oplus r1)$ is also valid, $R_2 \geq_m R_1$. Thus, if the mutant R_2 can be killed, the other two mutants need not be considered during the testing. Note that, here we have also used known properties of the PUT in establishing the hierarchies. To consider the mutant R_1, the impact of mutation is propagated to location L. This can also be seen as making the control conditions more explicit and has been studied in the context of test data generation in [12].

Let P_j and P_k be two mutants of a program P that are obtained by mutating location l_j and l_k respectively. Consider the set $R = D_j^r \cap D_k^r$. If $R = \emptyset$, there is no relationship between P_j and P_k and both of them must be considered during mutation testing. The checking of condition $R = \emptyset$ does not necessarily require explicit computation of the reachability sets. An example where such a condition holds is when l_j and l_k appear in **then** and **else** branches of an **if** statement that is reached exactly once in an execution with a test case.

Consider the other case when $R \neq \emptyset$; i.e., there may be an execution that passes through both l_j and l_k. In this case in order to find if the two mutants are related, it is necessary to evaluate the impact of mutation at l_j at location l_k (or vice versa). If this symbolic evaluation can be done then the hierarchies among the mutants can be identified with the aid of theorems mentioned above.

Such an analysis, in which the internal state is observed and the effect of a fault is propagated, is studied in the literature in terms of propagation conditions. For example, the local propagation of origination conditions is referred to as transfer conditions in [29].

The deductions required can be done automatically using existing tools such as the CVC3 [5], which can check the validity of quantifier-free first-order formulas over several interpreted theories including real linear arithmetic, arrays, uninterpreted functions, constants and abstract data types.

In the next section we give an overview of the related work.

4 Related Work

A number of research studies have been conducted to test the feasibility of mutation testing in an industrial context. For a recent survey on mutation testing see [18]. These studies concluded that mutation testing is difficult as the number of mutants generated and the time taken to kill each mutant when executed against the original program is myriad [28]. These research studies have proposed several variations of mutation testing. The two variants suggested by them are weak mutation [14] and firm mutation [40].

In weak mutation, the outputs of the original and mutated programs are compared immediately after the execution of the mutated statement. Firm mutation is a kind of "in-between" approach where the output of the original and mutated program can be compared at any location after the mutated statement and the end of the program. A short survey is presented in [28]. The variants of firm mutation testing can save some time during the execution and comparison phase of a PUT with its mutants, since the whole program need not be executed when comparing the outputs of the original program with the mutants.

An approach of computing the detection conditions for hypothesized faults has been extensively studied, for example, in *constraint-based testing* [7] and computation of failure conditions in [23, 29]. This require the sets D_k^p and D_j^p to be computed by using symbolic execution techniques, which give a constraint on inputs that must be satisfied by a distinguishing test case. Let C_k and C_j be two constraints that correspond to the subdomains D_k^p and D_j^p respectively. Then, $C_k \Rightarrow C_j$ will also guarantee that $\forall t(P, t \vdash P_k \geq_m P_j)$.

Another alternative to reduce the test effort is *selective mutation* [27] which attempts to identify a subset of mutation operators without significantly affecting the effectiveness. The analysis of Boolean expressions has been extensively studied in the literature; see for example [33, 37]. In [21, 34], a hierarchy between different types of faults that can arise in Boolean specifications is analyzed. These results are applicable in the context of Theorem 4.

It was found that the mutation operators, ABS, AOR, LCR, ROR and UOI, were sufficient for generating mutants. It was also observed that among those generated by these operators 57% mutants were equivalent mutants.

A similar approach, but complementary to that presented in this paper, has been suggested in [9, 30], involving the determination of an optimal ordering for the relational operators. The key idea can be stated as follows. Let P be a given program and P_k and P_j be two of its mutants that are obtained by replacing a relational operator, say RO, in P by other relational operators, RO_k and RO_j respectively, where RO_k is higher in the optimal ordering relation than RO_j [30]. Then, for a given input, if P_k remains live then P_j will also remain live. Thus, when attempting to kill mutants, P_k should be tried before P_j. However, Woodward in [39] has shown a fallacy in the above argument by providing a counter example and suggested the following in the conclusion:

One final point is that the fallacious argument ... is, in a sense, the opposite of that which a mutation tester really wants. The argument that "since this test data kills this mutant, it must be good and will kill these other mutants", would offer even greater potential benefits.

This paper contributes towards the above argument.

In the next section, we describe the empirical study performed to evaluate the cost savings obtained by identifying the relationship among mutants.

5 Experimental Evaluation

The programs considered in this study are written in C programming language. As mentioned earlier, a mutation operator can be defined to be a rule for generating mutants by making a single syntactic change in PUT for example, changing operator '$+$' to '$-$'.

5.1 Experimental Setup

We have used 23 mutation operators that are adapted from operators defined in [20] for FORTRAN programming language. The summary of these operators is given in Table 3. Since mutation testing is applied at a unit or component level, in our study we have mutated all the functions in a PUT except the `main` function.

As mentioned before, in common with a number of program analysis problems such as reachability of a location, identifying every possible $\geq_m$ relation is undecidable in general. Nevertheless, in the restricted setup of mutation testing, where a program differs from its mutants in a well-defined way, it is possible to find the relationship between some, if not all, mutant programs. The consequence of any technique being inherently incomplete is that it may not always be able to deduce the $\geq_m$ relation between two given mutants. However, this is not harmful except that the number of mutants to be considered during testing will not

Operator	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	do/while statement end replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	return statement replacement
SAN	statement analysis (replacement by TRAP)
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

Table 3: Mutation operators [20]

be minimal. The symbolic evaluation is not done for the complete unit under test but for a segment consisting of the mutated line upto nearest p-location as mentioned in the section 2.

However, generating the set of strongest mutants may lead to problems due to the presence of equivalent mutants already in the set. Note that the property $P, t \vdash P_k \geq_m P_j$ holds trivially for all j if P is equivalent to P_k (i.e., when $D_k^p = \emptyset$). The statistics, as reported in [26], indicate that the number of equivalent mutants is typically in the range of 7 to 12% of the total number of mutants and the automatic detection rate using symbolic execution varies from 12 to 84% of the total number of equivalent mutants. Another recent study [31] further explores the identification of equivalent mutants. Therefore in our empirical study we rely on the fact that a randomly selected mutant from the set of all mutants is likely to be non-equivalent.

Thus, although explicit generation of all mutants may not be required, the information regarding them must still be maintained. This information about $P, t \vdash P_k \geq_m P_j$ will be required whenever it is deduced or suspected that $P = P_k$, for example when a significant amount of effort is spent in killing P_k without success (such as, large size of the test set and large number of times reachability and infection conditions are met). The instantiation order for mutants is guided by the hierarchies among them.

We considered terminating sequential programs. Since application of a mutation operator can generate a program which may not terminate on execution, we kept a threshold of 5 seconds to decide if the execution is in an infinite loop. This approach is similar to that implemented in MuJava tool[22]. The comparison is made on the output as in the case of strong mutation testing.

The analysis for empirical study was done manually with the aid of CVC [5] tool. To analyze a given program, we used symbolic execution techniques which have been used in a wide variety of problems, such as, test data generation [7] and detecting equivalent mutants [26]. In symbolic execution, a program is executed with the symbolic values representing arbitrary values, instead of actual input values. Such an execution results in a tree in which every node consists of symbolic values of the variables and the path constraint that must be true to reach that node.

The steps to carry out the experiments are given in Table 4. Let P be a PUT and $\mathcal{M}$ be the set of mutants of P. We start with the execution of an instrumented mutant using a test case that kills the selected mutant. The instrumentation is done to enable the observation of internal state which are used to identify other mutants in $\mathcal{M}$ that can also be killed using the same test case. Such mutants form a class. We restart the process with the remaining mutants and continue until no mutants are left to be classified or they are identified as equivalent.

We illustrate the approach using an example shown in Figure 4. Consider the three possible mutants marked as $\min_1$ to $\min_3$. Assume that we explicitly generate $\min_1$ and would like to know if a test set that kills $\min_1$ will also kill any of the other two mutants. Since $\min_1$ and $\min_2$ are obtained by mutating a c-location, we will propagate the effect to the p-location at `L3`. The necessary (but not sufficient) condition at location `L3` to kill these mutants are given below.

$Cmin_1$: `a != b` $\vee$ `(a > b` $\oplus$ `b > b)`

$Cmin_2$: `(a != -abs(a))` $\vee$ `(a > b` $\oplus$ `-abs(a) > b)`

1. *Create a list of all mutants for the given PUT in increasing order of line number of the PUT. Let N be the number of possible mutants.*
2. *Set i = 0*
3. *Let P_i be the i^{th} mutant.*
4. *If P_i is already classified to be equivalent or in a mutant class then goto step 9.*
5. *Identify the condition that are needed to establish the hierarchies for other intra-location and inter-location mutants with P_i.*
6. *Check the conditions with theorem prover if they are valid. If yes, mark the corresponding mutants and P_i to belong to a single class.*
7. *Generate P_i with statements to observe satisfiability of conditions identified in the step 5.*
8. *Identify a test case to kill P_i. Mark the mutants to belong to a single class for which conditions identified in step 5 are satisfied while executing P_i with the test case that kills P_i.*
9. *Set i = i + 1*
10. *If i ≠ N goto step 3.*

Table 4: Steps for Experimental Study.

```
int min(int x, int y) {
L1:   float m;
                 min1          min2
L2:   m = a;   | m = b; |   | m = -abs(a); |
                     min3
L3:   if (m > b)   | if (m != b) |
L4:     m = b;
L5:
L6:   return m;
}
```

Figure 4: An example to illustrate the experimental approach.

$Cmin_3$: (a > b ⊕ a != b)

If the test case $\{x = -6, y = 2\}$ is used to kill mutant min_1 then it is guaranteed that the same test case will also kill min_2 and min_3. However, if the test case $\{x = 0, y = 1\}$ is used to kill mutant min_1 then only min_3 will be killed. This can be observed by checking the satisfiability of conditions given above for these test cases. Therefore, if we use latter test case, min_1 and min_2 will be the same class, whereas with former all three mutants will belong to the same class.

In step 6, the identification of hierarchies require checking validity of first-order quantifier-free logical formulas. Let C be such a formula that we want to validate to determine if it holds at a location l in a program P. There are three possibilities for the property to hold: (a) C is valid;(b) C is satisfiable for some test cases; (c) C is false. For case (c), we did not the put the required checks in the generated mutant.

The following programs were considered for the empirical study.

compare_str The program is taken from [1] and is intended to compares two strings. The specification requires to return true if the given input strings are identical, otherwise false. However, the implementation returns correct output for unequal length strings and strings of equal length with the identical last characters. Thus, the strings "cat" and "mat" are reported to be identical. The subtle issue, as mentioned in [1], is that the probability of random selection of input which will reveal the bug is very low. Probability that n strings will expose the fault is $1 - (25/26)^n$ [1]. In our study, a strong mutant forced to select such an input. The implemented program is a first-order mutant of the intended program.

find This is an implementation of Hoare's find algorithm. This program and its buggy version are studied earlier in [3, 10]. We have used the correct version of find algorithm. One of the strong mutant forced selection of input that detected it to be different from the incorrect version. As noted in [3], for the buggy version, it is extremely difficult to identify test case using random selection. The buggy version has two faults and therefore is a second order mutant. Unlike previous studies, we decided to consider the correct version and check if the test set with 100% mutation score can also kill the second order buggy mutant.

gcd This program computes the greatest common divisor of given two positive integers.

iroot The program is taken from [16] and computes integer square root of a given positive integer. This was given as an exercise to the students of first course in structured programming. We observed that it was difficult to get a correct implementation.

min Computes minimum of two numbers and is also studied in [26, 27].

prime Tests if a given positive integer is prime. The implementation has a subtle bug which causes it to give incorrect result for only one input. The implemented (faulty) program is directly representable as a first order mutant of the intended program. It was required to generate the test case that detects the fault.

selection This is a faulty implementation of selection sort algorithm. The program is taken from the book [2] which contains program with a single, hard-to-detect but realistic bug. In this case also, a test case detected the fault.

triangle This is the famous program, first mentioned in Myers [24], for testing whether the three given integers form a triangle and its classification.

tcas TCAS (Traffic Alert and Collision Avoidance System) is an aircraft conflict detection and resolution system. The SIR Siemens suite [8, 32] includes an ANSI C version of the resolution advisory component of TCAS system along with 41 faulty versions. The "Siemens" programs were assembled by Tom Ostrand and colleagues at Siemens Corporate Research. Our experiments include the 41 faulty versions and some other mutants.

5.2 Results Discussion

For the above programs, we studied number of equivalent mutants, number of mutant classes, operators with respect to those mutants in each mutant class that cannot be killed by the test cases used for other mutant classes. The statistics for these programs is given in Table 5 and 6. The savings are calculated using the formula given below.

$$\frac{\text{number of mutant classes}}{\text{number of mutants} - \text{equivalent mutants}} \times 100$$

As can be observed in Table 6 the overall savings is above 90% in all the cases.

The maximum number of equivalent mutants and mutant classes were found for the *triangle* program. The reason for the higher number of equivalent mutants is due to the redundant test $a \le 0 \vee b \le 0 \vee c \le 0$ in the presence of another check that sum of two sides is more than the third side. Also, the application of *abs* operator after initial testing of values to be positive since all values are guaranteed to be positive afterwards. Finally, the application of SVR operator on test for equilateral triangle $a = b \wedge b = c$ also generates equivalent mutants.

For each of the above example, once the identification of mutant classes was complete, we identified those mutants in each class that cannot be killed by the test cases of all other classes. The table reports the mutation operators associated with such mutants.

In selective mutation study [27], it was found that the mutation operators, ABS, AOR, LCR, ROR and UOI, were sufficient for generating mutants. In our study, we also find that the strong mutant were generated by these operators. However, we also observed that some strong mutants were generated by SVR operator. In one case we found that DER operator too generated a strong mutant.

6 Threat to Validity

The threat to validity of our empirical results could be due the set of programs used in the study. Although the programs are selected from the variety of sources, they cannot be claimed to be representatives of set of all programs. Our strategy requires pre-analysis to identify the relationship among mutants.

In comparison, the selective mutation [27] does not require any pre-analysis to generate mutants for a program although it may generate more mutants. The mutants are generated using a subset of set of operators defined in the previous study [20]. In contrast, our approach does not classify operators but attempts to identify relationship among mutants to avoid ignoring any mutant. Thus it is guaranteed to ensure the quality of a test set. Although this enables inclusion of new operators without affecting the effectiveness but increases the overall cost of testing.

We expect the cost of our technique to be more than selective mutation but less compared to full mutation testing. From the effectiveness point of view, if selective mutation chooses appropriate set of mutants than it may be as effective as full mutation testing. Also, since the program analysis in our study was done manually there is a possibility that the results may differ with a completely automated analysis. Nevertheless, the results of empirical study provide some confidence in the approach.

7 Conclusions

Mutation testing is a powerful testing approach that can not only ensure the checking of hypothesized faults but also the generation of test data satisfying common structural coverage criteria. The main difficulty faced in mutation testing is due to the large number of mutant programs that can be generated for a given implemented program. We have given a strategy that suggests the ordering of the mutants such that if a mutant is stronger than another, then killing the stronger will automatically kill the weaker. This approach can significantly reduce the cost of mutation testing. Although our approach ensures the same effectiveness as full mutation testing, it is expensive than selective mutation testing. To obtain the exact cost comparison we will require extensive research on automation of our approach, which we plan to pursue in future.

Identification of such hierarchies is also useful in the quantitative assessment of the quality of fault detection effectiveness, since, with the knowledge about mutant hierarchies, it is possible to reason if the mutation-adequacy score includes *strong* mutants. This is particularly helpful in directing the test effort with every step of the testing process.

The issue in identifying all possible orderings is mainly due to the undecidability of the problem and it also depends on the complexity of the program. However as our empirical study shows, a significant cost saving can be achieved even if one can identify some of the possible orderings among the mutants.

We have presented various conditions to identify the relationship between mutants that can be analyzed locally

Program	Lines of code	Mutants	Equivalent mutants
compare_str	26	111	9
find	65	464	36
gcd	23	136	13
iroot	26	134	5
min	21	53	3
prime	26	99	10
selection	55	263	38
triangle	37	524	59
tcas	174	85	2

Table 5: Statistics for programs considered in empirical study

Program	Mutant classes	Operators for strong mutants	Savings %
compare_str	7	ABS, ROR, RSR, SVR, UOI	93
find	5	ABS, ROR, UOI, SVR	99
gcd	4	ABS, AOR, ROR, UOI, SVR	97
iroot	4	AOR, CSR, UOI, DER	96
min	4	ABS, SVR, UOI	92
prime	4	CSR, ROR, UOI	95
selection	4	ABS, AOR, ROR, UOI	96
triangle	24	ABS, ROR, SVR	93
tcas	17	CRP, LCR, ROR, SVR	79

Table 6: Results for programs considered in empirical study

and thus can be evaluated in an effective way. Our work gives theoretical proof and the empirical evaluation with the examples taken from previous studies shows the feasibility of idea and significant cost savings that can be achieved. We found that the operators associated with strong mutants were the same as those identified in selective mutation. However in addition SVR and DER operators also generated strong mutants in some cases.

As the kind of analysis required to establish hierarchies is already part of various program analysis such as constant propagation, and transformation tools, and also tools like CVC3 [5] are already available, the given approach should be practical.

References

[1] R. C. Backhouse. *Program Construction and Verification.* Prentice-Hall International, 1986.

[2] A. Barr. *Find the Bug: A Book of Incorrect Programs.* Addison-Wesley, 2004.

[3] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT — a formal system for testing and debugging programs by symbolic execution. In *International conference on Reliable software*, pages 234–245, 1975.

[4] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and Empirical Studies on using Program Mutation to Test the Functional Correctness of Programs. In *7th Symposium on Principles of Programming Languages*, pages 220–233. ACM, 1980.

[5] CVC3: An Automatic Theorem Prover, 2007. http://www.cs.nyu.edu/acsys/cvc3/ (last accessed: Dec. 2007).

[6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[7] R. A. DeMillo and A. J. Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[8] H. Do, S. G. Elbaum, and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[9] I. M. M. Duncan and D. J. Robson. Ordered Mutation Testing. In *ACM SIGSOFT Software Engineering Notes*, volume 15, pages 29–30, April 1990.

[10] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, September 1997.

[11] M.-C. Gaudel. Testing can be Formal too. In *Theory and Practice of Software Development (TAPSOFT)*, volume 915, pages 82–96, March 1995.

[12] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability Transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[13] R. M. Hierons. Comparing test sets and criteria in the presence of test hypotheses and fault domains. *ACM Transactions on Software Engineering and Methodology*, 11(4):427–448, October 2002.

[14] W. E. Howden. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering*, 8:371–379, July 1982.

[15] IEEE. *IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Std.* 610.12. New York: Institute of Electrical and Electronics Engineers, 1990.

[16] J. Jacky. *The Way of Z: Practical Programming with Formal Methods.* Cambridge University Press, 1997.

[17] Y. Jia and M. Harman. Higher Order Mutation Testing. *Information and Software Technology*, 51(10):1379–1393, October 2009.

[18] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, To appear, 2010.

[19] K. Kapoor. Formal Analysis of Coupling Hypothesis for Logical Faults. *Innovations in Systems and Software Engineering (A NASA Journal)*, 2(2):80–87, July 2006.

[20] K. N. King and A. J. Offutt. A Fortran Language System for Mutation-Based Software Testing. *Software Practice and Experience*, 21(7):685–718, 1991.

[21] D. R. Kuhn. Fault Classes and Error Detection Capability of Specification-based Testing. *ACM Transactions on Software Engineering and Methodology*, 8(4):411–424, October 1999.

[22] Y.-S. Ma, A. J. Offutt, and Y. R. Kwon. MuJava: An Automated Class Mutation System. *Software Testing, Verification and Reliability*, 15(2):97–133, June 2005.

[23] L. J. Morell. A Theory of Fault-based Testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[24] G. Myers. *The Art of Software Testing.* Wiley-Interscience, 1979.

[25] A. J. Offutt. Investigations of the Software Testing Coupling Effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, January 1992.

[26] A. J. Offutt and J. Pan. Automatically Detecting Equivalent Mutants and Infeasible Paths. *Software Testing, Verification and Reliability*, 7(3):165–192, September 1997.

[27] A. J. Offutt, G. Rothermel, R. H. Untch, and C. Zapf. An Experimental Determination of Sufficient Mutant Operator. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, April 1996.

[28] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the Orthogonal. In W. E. Wong, editor, *Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55. Kluwer, October 2000.

[29] D. J. Richardson and M. C. Thompson. An Analysis of Test Data Selection Criteria using the RELAY Model of Fault Detection. *IEEE Transactions on Software Engineering*, 19(6):533–553, June 1993.

[30] I. J. Riddell, M. A. Hennell, M. R. Woodward, and D. Hedley. Practical Aspects of Program Mutation. Technical report, University of Liverpool, UK, 1982.

[31] D. Schuler and A. Zeller. (Un-)Covering Equivalent Mutants. In *3rd International Conference on Software Testing Verification and Validation (ICST'10)*, Paris, France, April 2010.

[32] SIR: Software-artifact Infrastructure Repository (SIR), 2010. http://sir.unl.edu/ (last accessed: Oct. 2010).

[33] K.-C. Tai. Theory of Fault-based Predicate Testing for Computer Programs. *IEEE Transactions on Software Engineering*, 22(8):552–562, August 1996.

[34] T. Tsuchiya and T. Kikuno. On Fault Classes and Error Detection Capability of Specification-based Testing. *ACM Transactions on Software Engineering and Methodology*, 11(1):58–62, January 2002.

[35] J. M. Voas. PIE: A Dynamic Failure-Based Technique. *IEEE Transactions on Software Engineering*, 18(2):717–727, August 1992.

[36] K. S. H. T. Wah. An Analysis of the Coupling Effect I: Single Test Data. *Science of Computer Programming*, 48:119–161, 2003.

[37] E. J. Weyuker, T. Goradia, and A. Singh. Automatically Generating Test Data from a Boolean Specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.

[38] W. E. Wong and A. P. Mathur. Reducing the Cost of Mutation Testing: An Empirical Study. *Journal of Systems and Software*, 31(3):185–196, December 1995.

[39] M. R. Woodward. Concerning Ordered Mutation Testing of Relational Operators. *Software Testing, Verification and Reliability*, 1(3):35–40, October 1991.

[40] M. R. Woodward and K. Halewood. From Weak to Strong, Dead or Alive? An Analysis of some Mutation Testing Issues. In *2nd Workshop on Software Testing, Verification, and Analysis*, pages 152–158, July 1988.