Higher Order Mutation Testing

Yue Jia and Mark Harman

King's College London, CREST Centre
Strand, London WC2R 2LS, UK

Abstract

This paper introduces a new paradigm for Mutation Testing, which we call Higher Order Mutation Testing (HOM Testing). Traditional Mutation Testing considers only first order mutants, created by the injection of a single fault. Often these first order mutants denote trivial faults that are easily killed. Higher order mutants are created by the insertion of two or more faults. The paper introduces the concept of a subsuming HOM; one that is harder to kill than the first order mutants from which it is constructed. By definition, subsuming HOMs denote subtle fault combinations. The paper reports the results of an empirical study of HOM Testing using ten programs, including several non trivial real–world subjects for which test suites are available.

Key words:
Mutation Testing, Higher Order Mutant

1. Introduction

The paper introduces a new form of Mutation Testing, called Higher Order Mutation Testing (HOM Testing). The underlying motivation is to seek to find those rare but valuable higher order mutants that denote subtle faults. This is achieved using automated search based optimization techniques to seek out combinations of simple faults that partially mask one another, so that the combination of faults is harder to detect than any of the individual constituent faults.

Such combinations of faults are relatively rare. As one might expect, adding more faults to a faulty program tends to make it more likely that a program will fail and, therefore, more likely that testing will reveal the fault combination. However, the rare exceptions to this rule are very interesting and, we argue, valuable.

Also, though the combinations we seek are rare, the large number of possible fault combinations creates a set of candidate combinations that is exponentially large. As the paper shows, search based optimization techniques can be used to manage the inherent combinatorial explosion involved in locating the rare but valuable cases from amongst the enormous candidate set. This makes the algorithmic complexity of seeking out such valuable fault combinations tractable. Indeed, the empirical study reported in this paper concerns programs that are several orders of magnitude larger than those reported upon in previous studies of (first order) Mutation Testing. This scale up is made possible by the exploitation of a search based optimization approaches.

The approach advocated in this paper is a new form of Mutation Testing, which we call 'Higher Order' because almost all approaches to traditional Mutation Testing have been conducted by inserting single small faults in a program to create first order mutants, whereas our approach treats these first order mutants as merely special cases of higher order mutants.

Mutation Testing has a long history, tracing back to the 1970s. It was first proposed by DeMillo et al. [1] and Hamlet [2]. Just like other fault-based testing techniques, the main purpose of Mutation Testing is to measure the quality of a test set. However, it can also be used to reduce the size of test set [3], to generate effective test data [4] and to compare techniques for verification [5, 6].

The mutation paradigm brings source code manipulation to bear within the realm of software testing. In the parlance of source code analysis and manipulation, each mutant is created by a source-to-source transformation of the original program. However, the goal is to insert a simulated fault. Therefore, the transformation should be non-meaning preserving, while meaning preserving transformations are eschewed by Mutation Testing because they create equivalent mutants [7]. Indeed, traditional source code analysis has been proposed as a technique to address this equivalent mutant problem [8, 9, 10, 11], thereby further highlighting the link between Mutation Testing and source code analysis and manipulation.

In Mutation Testing, from a program p, a set of faulty programs p', called mutants, is generated by injecting faults into the original program p. The motivation for Mutation Testing is that injected faults should represent mistakes that programmers often make. Traditionally, a mutant is generated by a single small change to the original program. For example, Table 1 shows the mutant p' generated by changing the *and* operator (&&) of the original program p into the *or* operator (||) of the mutant p'.

A transformation rule that generates a mutant from the original program is known as a mutation operator. Table 1 contains only one example of a mutation operator; there are many others. In this paper we adopt the 77 mutation operators for the C programming language introduced by Agrawal et al. [12]. Each mutant p' will be run against a test set T. If the result of running p' is different from the result of running p for any test case in T, then the mutant p' is said to be "killed", otherwise it is said to have "survived". The adequacy level of the test set T can be measured by a mutation score that is computed in terms of the number of mutants killed by T.

Table 1: A Example of Mutation Operation

Program p	Mutant p'
...	...
if ($a > 0$ && $b > 0$)	if ($a > 0$ || $b > 0$)
return 1;	return 1;
...	...

Mutants can be classified into two types: First Order Mutants (FOMs) and Higher Order Mutants (HOMs). FOMs are generated by applying mutation operators only once. HOMs are generated by applying mutation operators more than once.

This paper introduces the concept of subsuming HOMs. A subsuming HOMs is harder to kill than the FOMs from which it is constructed. As such, it may be preferable to replace the FOMs with the single HOM. In particular, the paper introduces the concept of a strongly subsuming HOMs. A subsuming HOMs is only killed by a subset of the intersection of test cases that kill each FOM from which it is constructed.

Consider a subsuming, h, constructed from the FOMs $f_1, ..., f_n$. The set of test cases that kill h also kill each and every FOM $f_1, ..., f_n$. Therefore, h can replace all of the mutants $f_1, ..., f_n$ without loss of test effectiveness. The converse does not hold; there exist test sets that kill all FOMs $f_1, ..., f_n$ but which fail to kill h. The FOMs cannot, even taken collectively, replace the HOM without possible loss of test effort. This is the sense in which h can be said to 'strongly subsume' $f_1, ..., f_n$.

In order to overcome the inherent computational cost that comes with the large number of HOMs, the paper introduces a search-based optimization approach to identify these subsuming HOMs efficiently.

The main contributions of the paper are as follows:

1. We introduce the Higher Order Mutation Testing paradigm. We categorize the various kinds of HOM and introduce a search-based optimization approach to overcome the exponential explosion in the number of HOMs. We set out a ‘manifesto’ for Higher Order Mutation Testing that clarifies the differences between the Higher Order Mutation Testing paradigm and the First Order Mutation Testing paradigm, as previously practiced and studied. As this manifesto shows, the higher order paradigm overcomes some limitations and overturns some previously held ‘myths’ of Mutation Testing.
2. We explore the proportion of all HOMs that are subsuming and strongly subsuming. The results show that a large proportion of HOMs are subsuming and that a small proportion of these are strongly subsuming. Though the proportion of strongly subsuming mutants is small, the number of strongly subsuming mutants is large, because the number of HOMs increases exponentially. The search based algorithms were able to find significant numbers of strongly subsuming HOMs in all of the ten programs studied.
3. We investigate the relationship between mutation killing set intersection and mutant order. The results explore the degree to which higher order mutants contain first order mutants that are completely decoupled.
4. The paper introduces three algorithms for finding optimal HOMs. The results indicate that the genetic algorithm performs best overall. However, they also reveal that each algorithm targets a different kind of HOM, so all three algorithms are useful.

The rest of this paper is organized as follows. Section 2 introduces the idea of a subsuming HOM formally. Section 3 discussed the advantage of Mutation Testing. Section 4 presents a search-based approach and explains three meta-heuristic algorithms used to find subsuming HOMs. Section 5 explains the experimental setting, while the results are discussed in Section 6. Section 7 presents a manifesto for Higher Order Mutation testing in the form of a polemic that discusses what we call the ‘myths’ of Mutation Testing and how the higher order paradigm overturns these myths. Section 8 discusses threats to validity of experiment. Section 9 introduces related work, and the paper concludes with Section 10.

2. Higher Order Mutant Classification

HOMs can be classified in terms of the way that they are 'coupled' and 'subsuming', as shown in Figure 1. In Figure 1, the region area in the central Venn diagram represents the domain of all HOMs. The sub-diagrams surrounding the central region illustrate each category. For sake of simplicity of exposition these examples illustrate the second order mutant case; one that assumes that there are two FOMs f_1 and f_2, and h denotes the HOM constructed from the FOMs f_1 and f_2. The two regions depicted by each sub-diagram represent the test sets containing all the test cases that kill FOMs f_1 and f_2. The shaded area represents the test set that contains all test cases that kill HOM h. The areas of the regions indicate the proportion of the domain of HOMs for each category.

Following the coupling effect hypothesis, if a test set that kills the FOMs also contains cases that kill the HOM, we shall say that the HOM is a 'coupled HOM', otherwise we shall say it is a 'de-coupled HOM'. Therefore, in Figure 1, the sub-diagram is a coupled HOM if it contains an area where the shaded region overlaps with the unshaded regions. For example the sub-diagrams (a), (b) and (f). Since the shaded region from sub-diagrams (c) and (d) do not overlap with the unshaded regions, (c) and (d) are de-coupled HOMs. Sub-diagram (e) is a special case of a de-coupled HOM, because there is no test case that can kill the HOM; there is no overlap, the HOM is an equivalent mutant.

Subsuming HOMs, by definition, are harder to kill than their constituent FOMs. Therefore, in Figure 1, the subsuming HOMs can be represented as those where the shaded area is smaller than the area of the union of the two unshaded regions, such as sub-diagrams (a), (b) and (c). By contrast, (d), (e) and (f) are non-subsuming. Furthermore, the subsuming HOMs can be classified into strongly subsuming HOMs and weakly subsuming HOMs. By definition, if a test case kills a strongly subsuming HOM, it guarantees that its constituent FOMs are killed as well. Therefore, if the shaded region lies only inside the intersection of the two unshaded regions, it is a strongly subsuming HOM, depicted in (a), otherwise, it is a weakly subsuming HOM, depicted in (b) and (c).

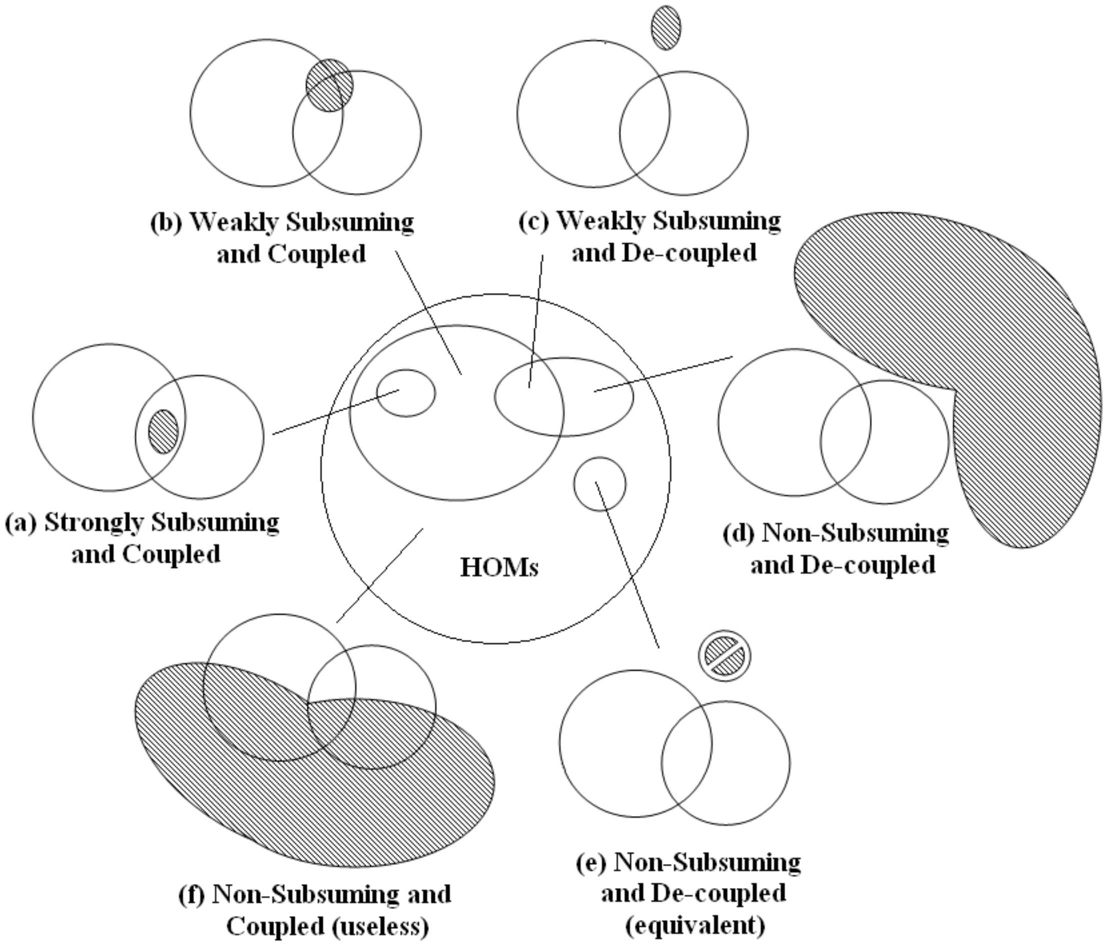

(b) Weakly Subsuming and Coupled
(c) Weakly Subsuming and De-coupled
(a) Strongly Subsuming and Coupled
HOMs
(d) Non-Subsuming and De-coupled
(f) Non-Subsuming and Coupled (useless)
(e) Non-Subsuming and De-coupled (equivalent)

Figure 1: HOMs Classification. The central Venn Diagram depicts important subclasses into which HOMs fall, while the outer diagrams depict killing test sets for the HOMs (shaded) and their constituent FOMs (unshaded). For ease of exposition, the diagrams illustrate only the second order case, whereas the definitions cover arbitrary order. HOMs of type (a), (b) and (c) are harder to kill than their constituent FOMs, thereby capturing subtler faults. In particular, type (a) are both subtle and useful; they can replace their constituent FOMs because they are killed by a subset of the intersection of test cases that kill their constituents.

According to the combination of subsuming and de-coupled HOM types, the six possibilities we considered are: strongly subsuming and coupled (a), weakly subsuming and coupled (b), weakly subsuming and de-coupled (c), non-subsuming and de-coupled (d), non-subsuming, de-coupled which is equivalent (e), and non-subsuming and coupled (f) which is useless, as shown in Figure 1.

The formal definitions of these HOMs are defined below. Let h be a HOM, constructed from FOMs $f_1, ..., f_n$. We assume the existence of a test set T. T is the set of all test cases under consideration. T_h is the subset of T that kills the HOM h, while $T_1, ..., T_n$ are the subsets of T that kill the constituent FOMs $f_1, ..., f_n$ respectively.

Definition 1 (Strongly Subsuming and Coupled)**.**

$$T_h \subset \bigcap_i T_i \ \ and \ \ T_h \neq \emptyset$$

Definition 2 (Weakly Subsuming and Coupled)**.**

$$|T_h| < |\bigcup_i T_i| \ , \ \ T_h \neq \emptyset \ \ and \ \ T_h \cap \bigcup_i T_i \neq \emptyset$$

Definition 3 (Weakly Subsuming and De-coupled)**.**

$$|T_h| < |\bigcup_i T_i| \ , \ \ T_h \neq \emptyset \ \ and \ \ T_h \cap \bigcup_i T_i = \emptyset$$

Definition 4 (Non-Subsuming and De-coupled)**.**

$$|T_h| \geq |\bigcup_i T_i| \ , \ \ T_h \neq \emptyset \ \ and \ \ T_h \cap \bigcup_i T_i \neq \emptyset$$

Definition 5 (Non-Subsuming and De-coupled)**.**

$$T_h = \emptyset \qquad (Equivalent)$$

Definition 6 (Non-Subsuming and Coupled)**.**

$$|T_h| \geq |\bigcup_i T_i| \qquad (Useless)$$

3. Advantages of Higher Order Mutation Testing

At first sight, any move from FOMs to HOMs brings with it an exponential explosion. Since a HOM is constructed by combining different FOMs, the number of HOMs can be computed from the number of FOMs. For such HOMs, let $p_{1...n}$ be the places that can be mutated, and $m_{1...n}$ be the number of changes that can be applied at place $p_{1...n}$. The number of the FOMs is $\sum_{i=0}^{n} m_i$. The number of the HOMs is $\sum_{i=2}^{n} \binom{i}{n} m^i$.

Because of this exponential explosion, Higher Order Mutation Testing has previously been considered to be so computationally expensive as to be impractical. Furthermore, the coupling hypothesis [1, 13, 14] suggests that the vast majority of HOMs will be coupled to FOMs, such that test sets that kill all FOMs will also kill almost all HOMs.

However, the few HOMs that are not coupled to their constituent FOMs may be very important; they are killed by a different set of test cases than their constituent FOMs. For decoupled mutants, the act of combining FOMs *shifts* the fault-revealing test set. Suppose that the act of combining FOMs to form a decoupled HOM not only shifts the fault-revealing set, but also reduces its size, so that the HOM is harder to kill than its constituent FOMs. Surely such a HOM would be potentially valuable in testing. In the nomenclature we introduce in this paper, it would be a “subsuming decoupled HOM”.

De-coupling is not the only way to produce a subsuming HOM. Strongly subsuming HOMs are, by definition, coupled, since the test sets that kill them are subsets of those that kill each of their constituent FOMs. Therefore, both coupled and decoupled HOMs may turn out to be harder to kill than the FOMs from which they are constructed, making them potentially valuable to the Mutation Testing process. In this paper we focus on the subsuming HOMs in general, and the strongly subsuming HOMs in particular, since a strongly subsuming HOM can always be used as a substitute for its constituent FOMs. We believe that Higher Order Mutation Testing offers three important benefits: Increased subtlety, reduced effort and reduced number of equivalent mutants, as explained below.

Increased Subtlety: The vast majority of FOMs are killed by a few very simple test cases, because many FOMs denote trivial faults. For instance, a mutant is unlikely to remain alive for very long if it is created by deletion of a frequently-executed statement or the transformation of '+' to '-' on a path to an output statement. Even in the presence of the most perfunctory testing activity, these 'dumb' mutants will not survive long.

However, by their very nature, the subsuming HOMs we study in this paper are more subtle; they denote faults that more elaborate testing may not reveal and, in so-doing, they drive the test data generator to consider the more difficult 'corner cases', where undiscovered faults often reside. In Section 6.4 we give an example of just such a subtle HOM that our search based algorithms revealed to be constructible from the very simple and widely studied benchmark program: `Triangle`.

Reduced Test Effort: One might think that since there are exponentially more HOMs than FOMs, Higher Order Mutation Testing would be much more expensive. However, it can be *less* expensive. We overcome this apparent paradox by specifically targeting those HOMs, the strongly subsuming HOMs, each of which can be used to replace more than one FOM. Fewer (but better) mutants means fewer (but better) test cases. Our higher order approach avoids dumb mutants in favour of subtle ones. Of course, in order to find the subtle HOMs we have to first construct *all* of their constituent FOMs. However, this process is entirely automated by the search-based optimization approach.

By contrast, the process of checking the original program's output for each the mutant-killing test cases often requires a (human) oracle. This oracle cost is often the most expensive part of the overall the test activity. The oracle cost can be reduced by reducing the size of the test suite. By moving from the first order to the higher order paradigm we seek to reduce the number of mutants considered (simultaneously increasing their quality). This has the potential to reduce test effort while improving its effectiveness.

Figure 2 illustrates an simple example of using SSHOM to reduce test effort and to increase test effectiveness at the same time. Suppose there is a SSHOM h which is constructed from FOMs f_a and f_b. The two regions T_a and T_b in Figure 2 represent the test sets containing all the test cases that kill FOMs f_a and f_b respectively, while the region T_h represents the test set containing all test cases that kill SSHOM h. In traditional mutation testing, it is not hard to find test cases like t_a and t_b which kill both FOM f_a and f_b. However the test case t_h that kills SSHOM h is a better choice, because it kills FOM f_a and f_b both separately and in combination, so a human oracle need only check one test output. Reduction of test effort can also be achieved by some 'smart' techniques with slightly more effort. For example, clustering test cases to identify the intersection of T_a and T_b. Although any test case selected from this intersection can achieve the same test effort as the test cases that kill the SSHOM h, such a test case like t_{ab} might not able to find the subtle fault represented by SSHOM h, thereby losing test effectiveness.

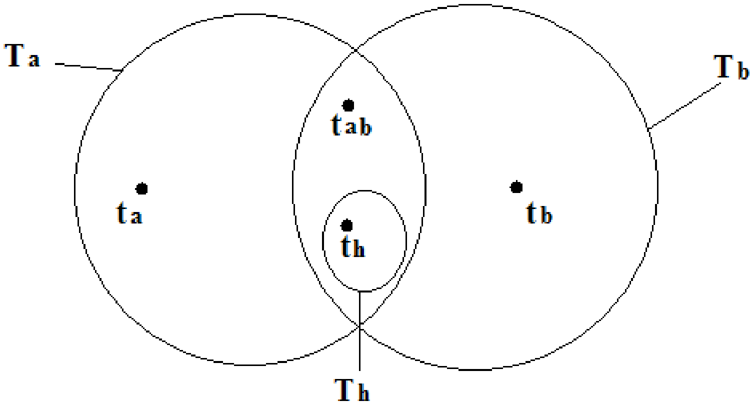


Figure 2: Test Effort Reduction Example

Reduced Number of Equivalent mutants: A mutant is said to be an 'equivalent mutant' if there does not exist a test input that kills it. Unfortunately, it is undecidable, in general, whether a mutant is equivalent. The equivalent mutant problem has been a bugbear for Mutation Testing for several decades. Although, several authors have proposed ways to partially detect equivalent mutants [8, 9, 10, 11], the core difficulty is the undecidability of the underlying problem.

One, hitherto largely overlooked, aspect of Offutt's empirical study of second order Mutation Testing [14], was the comparatively low density of equivalent mutants found in the second order paradigm, compared to that found in the first order paradigm. Offutt reported that approximately 1% of the second order mutants were found (by human examination) to be equivalent, whereas approximately 10% of the corresponding first order mutants were found to be equivalent. Furthermore, the search-based approach we advocate specifically searches the HOM space for *non-equivalent* HOMs, thereby further reducing the impact of this problem.

4. Algorithms

Due to the large number of HOMs, the cost in finding valuable HOMs could turn out to be extremely expensive. Therefore, using a normal undirected search is not efficient enough to find subsuming HOMs. In order to find the subsuming HOMs more effectively, our approach uses three meta-heuristic algorithms (GR, GA, HC). This section will introduce the representation and fitness function first, and then explain the three meta-heuristic algorithms in detail.

4.1. Representation

To identify a HOM uniquely, two types of value need to be specified: the position at which to mutate and the mutation operator to be applied. In our approach, HOMs are represented as a vector of integers. Each element of the vector denotes an application of a mutation operator, while indices indicate the position at which to apply the mutation operator.

4.2. Fitness Function

In order to measure the fitness of the HOM, a value is needed that measures the ease with which a FOM or HOM can be killed. Let T be a set of test cases, $\{M_1,...,M_n\}$ be a set of mutants, and the $kill(\{M_1,...,M_n\})$ function returns the set of test cases which kill mutants $M_1,...,M_n$. We shall define fragility for a set of mutants so that a single definition caters for individual mutants (which may be either first order *or* higher order), but also for sets of individual mutants. That is the fragility of a mutant shall be defined as follows:

Definition 7 (fragility)**.**

$$fragility(\{M_1,...,M_n\}) = \frac{|\bigcup_{i=1}^{n} kill(M_i)|}{|T|}$$

The value of fragility lies between 0 and 1. When it equals 0 this means that there is no test case that can kill this mutant, which indicates that this mutant is potentially an equivalent mutant. As the value of fragility increases from 0 to 1, the mutant is assessed to be weaker, until the value equals 1, which means that the mutant is so weak that it can be killed by any of the test cases. In the following, we use $M_{1...n}$ to denote a HOM consisting of the FOMs F_1 to F_n. The fitness function for a HOM is defined as follows.

Definition 8 (Fitness Function)**.**

$$fitness(M_{1...n}) = \frac{fragility(\{M_{1...n}\})}{fragility(\{F_1, ..., F_n\})}$$

That is the fitness of a HOM is defined to be the ratio of the fragility of its HOM to the fragility of the constituent FOMs. From the definition, if the fitness is greater than 1, it means the HOM is weaker than the constituent FOMs (i.e. it is useless). As the fitness decreases from 1 to 0, the HOM becomes gradually stronger than its constituent FOMs. However, when the fitness value reaches 0, it is considered as a potential equivalent HOM, and so all such zero-valued HOMs are discarded. All of the following algorithms use this fitness function to evaluate the fitness of HOMs.

4.3. Greedy Algorithm

A greedy algorithm is an algorithm that makes local optimized choices at each stage with the hope of achieving a near global optimum [15]. The general procedure of the greedy algorithm starts from solving the first sub-problem by selecting the solution with maximum current fitness. It then repeats the action to solve the rest of the problem. Therefore, it can only be used to solve a problem that can be divided into sub-problems, and can only provide a single solution. In order to apply the greedy approach to finding more than one subsuming HOM, several optimized changes have been made. An initial FOM is chosen at random as a starting point. Subsequently, the normal greedy algorithm process is performed to incrementally augment with additional the correct solution FOMs. An archive operation is used to store the subsuming HOMs found. The overall algorithm is iterated with repeated randomized initial position, much like a random-restart hill climbing algorithm. The pseudo-code is shown in Algorithm 1.

```
Input  : Running Time Limit: limit
Output: Mutation vector homlist
set counter = 0
while counter < limit do
    set hom = generateRandFOM()
    foreach FOM m of Program do
        temp_hom = combine(hom,m)
        if fitness(temp_hom) > fitness(hom) then
            hom = temp_hom
        end
        archvie(temp_hom)
    end
end
```

Algorithm 1: Optimized Greedy Algorithm

```
Input  : Running Time Limit: limit
Output: Mutation vector homlist
set counter = 0
foreach Mutation m in population do
    set m = generateRandHOM() fitness(m)
end
while counter < limit do
    createMtPool(population)
    archvie(population)
    crossover(population)
    mutate(population)
    fitness(population)
    counter ++
end
```

Algorithm 2: Optimized Genetic Algorithm

```
Input   : Running Time Limit: limit
Output: Mutation vector homlist
set counter = 0
set hom = generateRandFOM()
while counter < limit do
    temp_hom = getNeighbor(hom)
    if fitness(temp_hom) < fitness(hom) then
        hom = temp_hom
        archvie(hom)
    end
    hom = generateRandFOM()
    counter ++
end
```

Algorithm 3: Optimized Hill Climbing Algorithm

4.4. Genetic Algorithm

A genetic algorithm is an algorithm that simulates the process of natural genetic selection according to the Darwinian theory of biological evolution [16]. In a genetic algorithm, every possible solution within the solution domain will be represented as a chromosome, and crossover and mutation operation will be performed on chromosomes to produce new solutions repeatedly, until one member of the population denotes a suitably ‘good’ solution. In the proposed genetic algorithm, each gene of a chromosome represents the position and possible types of mutation operator (see Section 4.1), and in additional to crossover and mutation operators, an archive operator is used to store the subsuming HOMs found. The pseudo-code is shown in Algorithm 2.

4.5. Hill Climbing Algorithm

A hill climbing algorithm is a local search algorithm in which the next solution considered will depend on both the fitness value and distance to the current solution. The process starts from random initial solution. By comparing the current solution and its neighbour solution’s fitness, the greater one becomes the new current solution, until fitness cannot be further improved. Our optimized algorithm is based on a random-restart hill climbing algorithm, which chooses a random starting solution for each run. The pseudo-code is shown in Algorithm 3.

5. Experiment Set Up

This section describes the set of experiments which are designed to explore properties of subsuming HOMs. Section 5.1 discusses the research questions that the study will address. Section 5.2 describes the subject programs used in this study. Sections 5.3 and 5.4 briefly overview the selected mutation operators and the mutation tool used to implement these experiments. Section 5.5 explains the experimental procedure.

5.1. Research Questions

This section sets out the research questions addressed in the empirical study and for which the next section provides answers.

RQ1: How numerous are subsuming HOMs?
The main goal of this paper is to introduce and study subsuming HOMs. Therefore, the natural first research question is how prevalent are subsuming HOMs?

RQ2: What proportion of subsuming HOMs have entirely decoupled constituent FOMs?
Since we seek ways in which first order mutants combine to make valuable higher order mutants that partially mask each other, we are also interested to know what proportion of higher order mutants contain first order mutants whose killing sets do not overlap. Where there is no intersection between the killing sets of the first order mutants, these first order mutants cannot combine in ways that partially mask one another. In RQ2 we explore this issue, by repeated sampling of HOMs to determine the relative proportion (for each program studied) of the HOMs that consists of entirely decoupled FOMs. This allows us to approximate the overall proportion of 'decoupled HOMs' and the degree to which this proportion varies per program studied.

RQ3: What proportion of subsuming HOMs are strongly subsuming?
As introduced in Section 2, strongly subsuming HOMs are the most valuable HOMs that can be applied in higher order mutation testing directly. RQ3 studies the proportion of the strongly subsuming HOMs found in all subsuming HOMs.

RQ4: What do strongly subsuming HOMs look like?
In order to understand HOMs better, we examined several of those found by our algorithms to find the simplest example of a strongly subsuming HOM.

This illustrates the way in which faults may partially mask one another so that the set of test cases that kill all FOMs is a subset of the intersection of the test sets that kill the FOMs. To our surprise, our algorithms even managed to find such an example in the familiar `Triangle` program. Our initial intuition had been that such a program would have been too small and simple to allow for the construction of a strongly subsuming HOM.

RQ5: Which algorithms perform best at finding subsuming HOMs? The paper introduces three algorithms for finding subsuming HOMs. RQ5 asks how these algorithms perform relative to one another.

5.2. Subject Programs

The experiments use ten benchmark C programs with branch adequate test sets from the Software-artifact Infrastructure Repository (SIR) [17], as described in the first two columns of Table 2. The `Triangle` program is a small program that is used to determine the type of triangle from the length of its sides. This version is the one used by Michael and McGraw in their test data generation study [18].

The seven programs `Replace`, `TCAS`, `Schedule2`, `Schedule`, `Totinfo`, `Printtokens` and `Printtokens2` are collectively known as the ‘Siemens Suite’, which is widely used as a benchmark for software testing techniques. `TCAS` is a program used to avoid an aircraft collision. `Schedule2` and `Schedule` are programs that prioritize schedulers. `Totinfo` is a program that computes statistics from input data. `Printtokens` and `Printtokens2` are lexical analyzers. `Replace` performs pattern matching and substitution.

Besides the `Triangle` program and the Siemens suite, there are two other ‘real world’ programs: `Gzip` and `Space`. `Gzip` is a widely used compression program and `Space` is an interpreter for an array definition language.

There are two reasons for choosing these programs. The first reason is that previous studies of HOMs are limited to programs on a small scale. By contrast, this study is able to consider programs from 50 to 6,000 lines of code. The second reason is that, in order to measure the fitness of HOMs precisely, the HOMs have to be executed against a set of reasonably high quality test cases. The SIR provides branch adequate test sets, thereby achieving this aim. So far as we are aware this is the largest study of Mutation Testing (first order *or* higher order) to date.

Table 2: Selected Subject Programs: Scale shows the size of the programs, No. of FOMs is a count of all FOMs generated for each program. The 'possible equivalent' FOMs are those not killed by any test cases, while the 'dumb FOMs' are those killed by all test cases.

Programs	Scale	No. of Test Cases	No. of FOMs	No. of possible Equivalent FOMs	No. of Dumb FOMs
`Triangle`	50 LoC	60	601	62	35
`TCAS`	150 LoC	1,608	744	239	60
`Schedule2`	350 LoC	2,710	1,603	238	970
`Schedule`	400 LoC	2,650	1,213	155	810
`Totinfo`	500 LoC	1,052	2,316	245	1,100
`Replace`	550 LoC	5,542	4,195	486	3,133
`Printtokens2`	600 LoC	4,054	1,714	345	569
`Printtokens`	750 LoC	4,071	1,237	557	210
`Gzip`	5,500 LoC	228	12,027	1,124	5,770
`Space`	6,000 LoC	13,498	68,843	26,401	5,378

5.3. Mutation Operators

As explained in section 2. The total number of HOMs are correlated to number of FOMs. Therefore, in order to reduce the runtime cost, selective mutation technique are used. The study of Agrawal et al. describe the mutation operator for C language into 77 set. However, not all of the mutation operators increase the effectiveness of Mutation Testing. Offutt [19, 20] shows that 5 of 22 FORTRAN mutation operators used by Mothra are sufficient to carry out Mutation Testing effectively. In our experiment, only the subset of the C mutation operators (23 of 77) which falls into Offutt's 5 categories will be used.

5.4. Experiment Tool: MiLu

In spite of several existing Mutation Testing tools, there is none designed for studying HOMs. Therefore, a new Mutation Testing infrastructure called MiLu has been developed [21]. MiLu is specially designed for the study the HOMs in C programs, and supports general purpose of Mutation Testing as well higher order study. The objective of MiLu is to allow users to focus, on either algorithms for generating FOMs and HOMs, or on analysing the experimental results. MiLu currently supports 70 of the 77 mutation operators for the C language, and provides a source code analysis and program testing environment to support full Mutation Testing with either FOMs, HOMs or both. All of the experiments are performed within the MiLu mutation infrastructure. MiLu supports the full C language. A full description of the tool is beyond the scope of the present paper. We plan to make the tool publicly available and to publish implementation details.

5.5. Experimental Procedure

```
for each subject program do
    generate all possible FOMs
    filter out the FOMs that are killed by all test cases
    filter out the FOMs that are killed by non test cases
    store rest FOMs as the set: ‘non trivial FOMs’
    apply search based optimization to generate subsuming HOMs
    from non trivial FOMs
    for 100 trails, from all non-trivial FOMs, allow the algorithm
    to consider 10,000 HOMs from which its optimization procedure
    finds as many subsuming HOMs as possible, guided by the
    fitness function do
        count the percentage of Subsuming HOMs within the
        HOMs (SHOMs)
        count the percentage of Strongly Subsuming HOMs
        (SSHOMs) within the subsuming HOMs
        count the percentage of Non-Intersection HOMs (NIHOMs)
        within the subsuming HOMs
    end
end
```

Algorithm 4: Experimental procedure

Algorithm 4 sets out the steps involved in the experimental procedure. Trivial mutants are first filtered out from the set of all FOMs to remove from consideration those killed by all test cases and those killed by none. The remaining ‘non trivial mutants’ are used to generate subsuming higher order mutants. The set of all possible subsuming higher order mutants is infeasibly large, but we use search–based optimization to locate them so this size is not a problem. Rather, it provides a rich set from which to seek useful HOMs.

However, in order to answer questions about relative proportions, a kind of sampling approach is required to approximate the answers. Each ‘sample' is a set of subsuming HOMs, constructed by one of the search based optimization algorithms from an allowed 'budget of consideration' of 10,000 HOMs. The particular algorithm used is a parameter to the procedure.

In answering RQ5, we report results for the performance of four algorithms: A Greedy Algorithm, a Hill Climber, a Genetic Algorithm and (for base line comparison) a random search. However, to answer the questions about proportions of HOMs that have the properties captured by RQ1–RQ3, we use only the genetic algorithm, since this was found to locate the most subsuming HOMs. From each set of 10,000 HOMs we compute the proportion of HOMs constructed by the genetic algorithm that were subsuming. From the set of subsuming HOMs we compute both the proportion that were strongly subsuming and the proportion that is entirely decoupled. These proportions are reported as percentages. In order to factor out possible effects from sampling, thereby arriving at a more accurate approximation to the true proportion, we repeat the entire process for 100 trials per program and report average per program over the 100 trials.

6. Results and Analysis

In this section we present the answer to each research question in turn, indicating how the results answer each.

6.1. Answer to RQ1

RQ1 is designed to investigate the quantity of the subsuming HOMs. To begin the analysis, the second and third columns of Table 3 present the overall results for sum of percentage subsuming HOMs found in each subject programs by our GA algorithm with 10,000 fitness evaluations, repeated for 100 trials (giving 1,000,000 fitness evaluations in total per program). From the smallest `Triangle` program (50LoC) to the largest `Space` program (6,000Loc), there exist subsuming HOMs.

6.2. Answer to RQ2

1.2 RQ2 is designed to investigate the proportion of entirely decoupled subsuming HOMs. Figure 3 shows the percentage of HOMs that are constructed of non intersecting FOMs on the vertical axis against the order of the HOM concerned on the horizontal axis. For instance, a point at (x, y) means that $y\%$ of all HOMs of order x are non–intersecting. That is, their

Table 3: This table shows the proportion of HOMs which are Subsuming HOMs (SHOMs) and the proportion of these SHOMs that are Strongly Subsuming HOMs (SSHOMs) and Non–Intersecting HOMs (NIHOMs).

Program	Non trivial FOMs	% of SHOMs	% of SSHOMs	% of NIHOMs
`Triangle`	504	81.6%	0.24%	80.4%
`TCAS`	445	89.5%	0.11%	97.2%
`Schedule2`	395	57.5%	0.27%	77.2%
`Schedule`	248	75.1%	0.39%	64.1%
`Totinfo`	971	58.2%	0.24%	49.3%
`Replace`	576	67.5%	0.31%	62.2%
`Printtokens2`	800	47.0%	0.10%	31.2%
`Printtokens`	470	52.2%	0.01%	50.9%
`Gzip`	5,133	71.4%	0.08%	43.3%
`Space`	39,064	77.5%	0.21%	32.4%

FOMs are entirely decoupled; there is no pairwise intersection between any of the sets of test cases that kill each of the constituent FOMs.

As the figure shows, there is a tendency for decoupling to increase as the order of the HOM increases (for all programs studied). However, the figure reveals that this property is very different for different programs. For instance, for the program `totinfo`, only about 5% of 9th order HOMs are composed of entirely decoupled FOMs, whereas about 90% of the 9th order HOMs for `triangle` and `TCAS` consist of entirely decoupled FOMs.

The rightmost column of Table 3 shows the proportion of all HOMs constructed that were found to be composed of entirely decoupled FOMs. Notice that the number of NIHOMs appears to decrease as the number of FOMs increases. We performed a Spearman Rank Correlation statistical test to investigate this observation more rigorously. The test showed a strong rank correlation between the proportion of subsuming HOMs that are NIHOMs and the number of FOMs and also between the proportion of subsuming HOMs that are NIHOMs and the number of non-trivial FOMs.

6.3. Answer to RQ3

RQ3 is designed to investigate the proportion of strongly subsuming HOMs. Of all subsuming HOMs found in our experiments, between approximately 0.01% to 0.4% of these were found to be of the highly valuable,

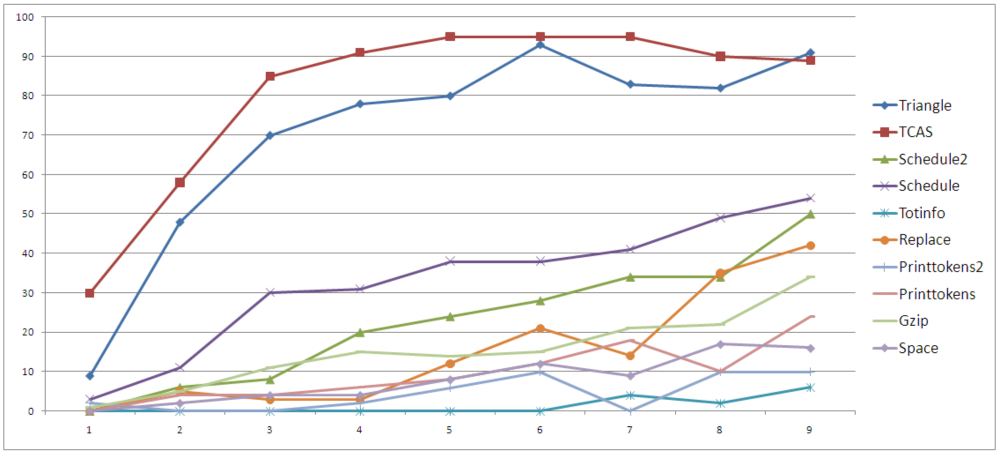


Figure 3: Overall Type Distribution

strongly subsuming type[1]. This may be a very small proportion, but there are a very large number of subsuming HOMs because the proportion of all HOMs that are subsuming HOMs is very large and so the *numbers* of strongly subsuming HOMs is high.

6.4. Answer to RQ4

RQ4 focused on the study of strongly subsuming HOMs. To answer RQ4 we present a case study of a strongly subsuming HOM that our genetic algorithm found in the `Triangle` program. The `Triangle` is a small C program (50 LoC) that has been studied for at least 30 years [1]. The program takes the length of sides of a triangle, and outputs whether the triangle is a valid and whether it is equilateral, isosceles or scalene. Program 4 shows the source of the Triangle program. There are two main factors to decide the type of the triangle. The first is the side length constraint; the sum of any two sides has to be greater than the third. The second is captured by the variable *trian*, whose value is used to specify the type of the triangle. For instance, if

[1] In the conference version of this paper, we allowed only a short computation time before a mutant was deemed to be killed due to non–termination. However, in the journal version, we re-implemented the system to allow for testing of mutants in parallel. This increased the available computing power and allowed us to let mutants run on far longer before we deemed them to have failed to terminate. As a result we found far fewer FOMs were killed and this increased the precision of the results that we are able to report here in the extended version of the paper for I&ST.

a triangle's *trian* value equals 0, and the side lengths satisfy the side length constraint, it is a 'valid scalene' triangle.

Program 5 presents the source code of `Triangle` program, two FOMs and the subsuming HOM constructed from them, which was found by our optimized genetic algorithm. The way in which the HOM strongly subsumes the two FOMs is subtle and involves an interplay between the validity and type-of-triangle tests in the original program. We believe that it is just this sort of subtle interaction that leads to faults that may go unnoticed in less rigorous testing.

Table 4 summaries the reasons why this is an instance of strong subsumption. From the table, only three types of test cases can kill FOM_i while two types of test cases can kill FOM_j. However, careful consideration reveals that HOM_ij can only be killed by test cases of the form $(a == b \;\&\&\; a + b > c)$. Test cases of this form also kill FOM_i and FOM_j. There is no other test case that can kill HOM_ij. Therefore, we can use strongly subsuming HOM_ij to replace both FOM_i and FOM_j in Mutation Testing.

6.5. Answer to RQ5

RQ5 is designed to investigate the effectiveness of proposed algorithms. The chart in Figure 4 presents the result of comparison of the four algorithms, which answers RQ5. We use an oracle of all subsuming HOMs found, to provide a reference against which each algorithm is assessed. The oracle contains the union of resulting subsuming HOMs from each algorithm. The greater the percentage of this oracle an algorithm can find, the better is the algorithm is deemed to perform. In Figure 4, the x-axis shows the four algorithms, and the y-axis shows the percentage of oracle HOMs found. The genetic algorithm bar is the highest. We believe that the GA algorithm performs best, because the subsuming HOMs are easier to generate from existing subsuming HOMs. In the genetic algorithm, this observation favours crossover, which is one of the genetic algorithm's distinguishing features.

Although the genetic algorithm found more of the subsuming HOMs, the hill climbing algorithm and the greedy algorithm also have their advantages. The hill climbing algorithm always finds the highest fitness HOMs, because its subroutine repeatedly improves the fitness of HOMs, while the greedy algorithm finds the highest order HOMs, because it starts from a random FOM, and tries to achieve as high an order as possible. Therefore, the results reveal that genetic algorithm is the best performing algorithm and the greedy

```
Program: Triangle
Input  : Three sides a, b, c
Output : Types of Triangle
int trian
if (a <= 0 || b <= 0 || c <= 0) then
    return INVALID
trian = 0
if (a == b) then trian = trian + 1
if (a == c) then trian = trian + 2
if (b == c) then trian = trian + 3
if (trian == 0) then
    if (a + b < c || a + c < b || b + c < a) then
        return INVALID
    else return SCALENE
if (trian > 3) then return EQUILATERAL
if (trian == 1 && a + b > c) then
    return ISOSCELES
else if (trian == 2 && a + c > b) then
    return ISOSCELES
else if (trian == 3 && b + c > a) then
    return ISOSCELES
return INVALID

Mutant : FOM_i ——————————————————————
if (trian > 1 && a + b > c) then
    return ISOSCELES
else if (trian == 2 && a + c > b) then
    return ISOSCELES
else if (trian == 3 && b + c > a) then
    return ISOSCELES
return INVALID

Mutant : FOM_j ——————————————————————
if (trian == 1 && a + b <= c) then
    return ISOSCELES
else if (trian == 2 && a + c > b) then
    return ISOSCELES
else if (trian == 3 && b + c > a) then
    return ISOSCELES
return INVALID

Mutant : HOM_ij ——————————————————————-
if (trian > 1 && a + b <= c) then
    return ISOSCELES
else if (trian == 2 && a + c > b) then
    return ISOSCELES
else if (trian == 3 && b + c > a) then
    return ISOSCELES
return INVALID
```

Program 5: The `Triangle` program together with a strongly subsuming HOM and its two constituent FOMs. As this case study demonstrates, even from this trivially small program, extremely subtle strongly subsuming HOMs can be constructed. Table 4 depicts the corresponding killing test cases.

Mutant	Test Case	Original Result	Mutant Result
M_1	$a == b$ && $a + b > c$	Isosceles	Invalid
	$a == c$ && $a + b > c$ && $a + c <= b$	Invalid	Isosceles
	$b == c$ && $a + b > a$ && $b + c <= a$	Invalid	Isosceles
M_2	$a == b$ && $a + b > c$	Isosceles	Invalid
	$a == b$ && $a + b <= c$	Invalid	Isosceles
M_{12}	$a == b$ && $a + b > c$	Isosceles	Invalid

Table 4: Killing Test Cases for the Triangle HOM and its FOMs

algorithm and hill climbing algorithm can also be used to augment results and to search for extreme cases. The results also show that even random search can find a large number of subsuming HOMs, indicating that there are a large number of available subsuming HOMs that are relatively easy to find.

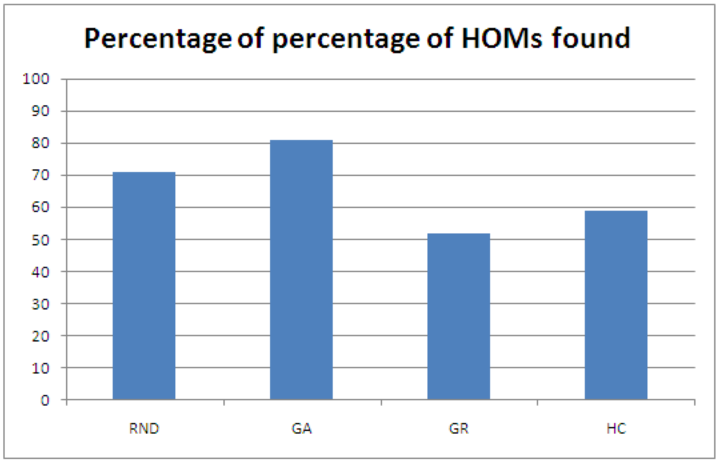


Figure 4: Algorithm comparison

7. Myths of Mutation Testing

The leap from first order Mutation Testing, as traditionally practiced for over three decades, to higher order mutation presents challenges to several widely held beliefs about Mutation Testing. It also denotes a shift in thinking about the philosophy underlying Mutation Testing. This section aims to address these philosophical issues head-on in the form of a polemic against some of the 'received wisdom' and 'folklore' of Mutation Testing. The section constitutes a manifesto for Higher Order Mutation Testing.

The section characterises the points of departure, at which HOM testing diverges from this received wisdom. In order to focus attention on these points of departure, the section is constructed as a set of 'myths of Mutation Testing'. That is, those aspects of the Mutation Testing folklore that the authors consider to be 'myths' about the nature of Mutation Testing, why they are misconceived and the way in which Higher Order Mutation Testing challenges these myths.

Real Fault Representation Myth (RFR)
Mutants denote faults that a typical programmer might make.

The misconception underlying this myth: Many of the mutants created by first order mutation are not real faults, certainly not those that would be likely to be committed by a 'competent programmer'. For instance, arbitrarily replacing a plus symbol with a minus symbol might represent a *few* real faults in a *few* spacial cases, but it is likely to create a large number of faults which no competent programmer would commit. Most mutants do not denote real faults at all. This is also true of most higher order mutants.

How Higher Order Mutation Testing challenges The RFR Myth
Higher Order Mutation Testing is not based on any claim that higher order mutants are inherently more realistic than first order mutants. Rather, the approach is based on the philosophy that all mutation, whether higher order or first order, should be considered to be a *search process*, in which the goal is to seek out a set of mutants that represent some aspect of interest. This aspect of interest could be some measure of realism (perhaps guided by a fault model), or it could be a measure of subtlety (as explored in the results presented in this paper). There are many possible choices for what might be sought from a set of mutants. The approach advocated here is that these should be captured by a fitness function, so that search based optimization techniques can be applied to automate the process of locating high quality mutant suites according to the chosen fitness function.

Unscalability of Mutation Testing Myth (UMT)
The Unscalability of Mutation Testing Myth states that Mutation Testing cannot scale to larger programs because of the large number of mutants created. This myth originated in the observation that a large number of first

order mutants that were created for even very trivial programs (for instance 601 first order mutants are created from the 50 lines of the triangle program). The UMT Myth is implicitly acknowledged by the Mutation Testing community; much effort has been devoted to reducing the number of mutants to be considered [19, 20] by sampling and selection and in reducing Mutation Testing effort [22, 23, 24, 25, 26, 27, 28, 29, 30].

The misconception underlying this myth: The UMT Myth derives from the assumption that all mutants that can be created should be tested. This misconceived assumption is captured in the All Mutants and Equal Equality Myth below.

All Mutants are Equal (AME)
The AME Myth states that all mutants are created equal and effort must be put into trying to kill them all. This is implicitly how all Mutation Testing tools work. They create a set of mutants that another tool or another part of the creation tool should seek to kill. There may be some attempt to remove equivalent mutants, but all those deemed to be potentially non-equivalent are otherwise considered to be equal.

Notice that the AME Myth has not been rejected, despite the more recent attention in the literature to work on mutant sampling [31, 22] and selective mutation [20, 19]. Both these techniques seek to reduce the number of mutants created. However, both implicitly respect the AME Myth. That is, in mutant sampling, all mutants are equal, but since there are too many to test all of them, we sample randomly from the set of all possible mutants. In selective mutation, we consider all mutants equal, but focus attention on the operators that generate mutants, seeking to reduce the number of mutants created by permanently removing certain mutant generation operators. This reduces the set of operators that are applied, but the remaining operators produce a set of mutants, all of which are considered to be of equal value.

The misconception underlying this myth: All mutants are not of equal value; though one cannot differentiate *a priori* between mutant operators, it *is* possible to differentiate between different mutants of a *given program p*. Some mutants will be better at denoting faults *in p* than others. Some will present a tougher challenge to the tester. To take an extreme example, consider a mutant that is killed by every possible test case that could be applied to *p*. Testing with any test suite apart from the empty test suite will kill it. Putting such a mutant into the testing process is a waste of testing effort. Such a mutant is worthless and, therefore, should be considered to have lower value than some other mutant that is not killed by at least one test for p.

To illustrate, consider Figure 5. This figure shows the cumulative numbers of all mutants that are killed by different degrees of test set. At the leftmost end of the horizontal axis, the figure shows the number of all mutants that are killed by all test cases (the mutants that we call 'dumb' mutants). As we move across the horizontal axis we decrease this lower threshold on the proportion of all test cases considered, until at the rightmost end of the horizontal axis we show the number of mutants killed by 0% or more of the test cases (that is all mutants). A point (x, y) on these figures means that y mutants are killed by $x\%$ or more of the test cases.

The figures show that all the programs have *some* dumb mutants. They also reveal that many of the programs have *large numbers* of dumb mutants. Figure 6 shows the growth trend over all first order mutants from all programs.

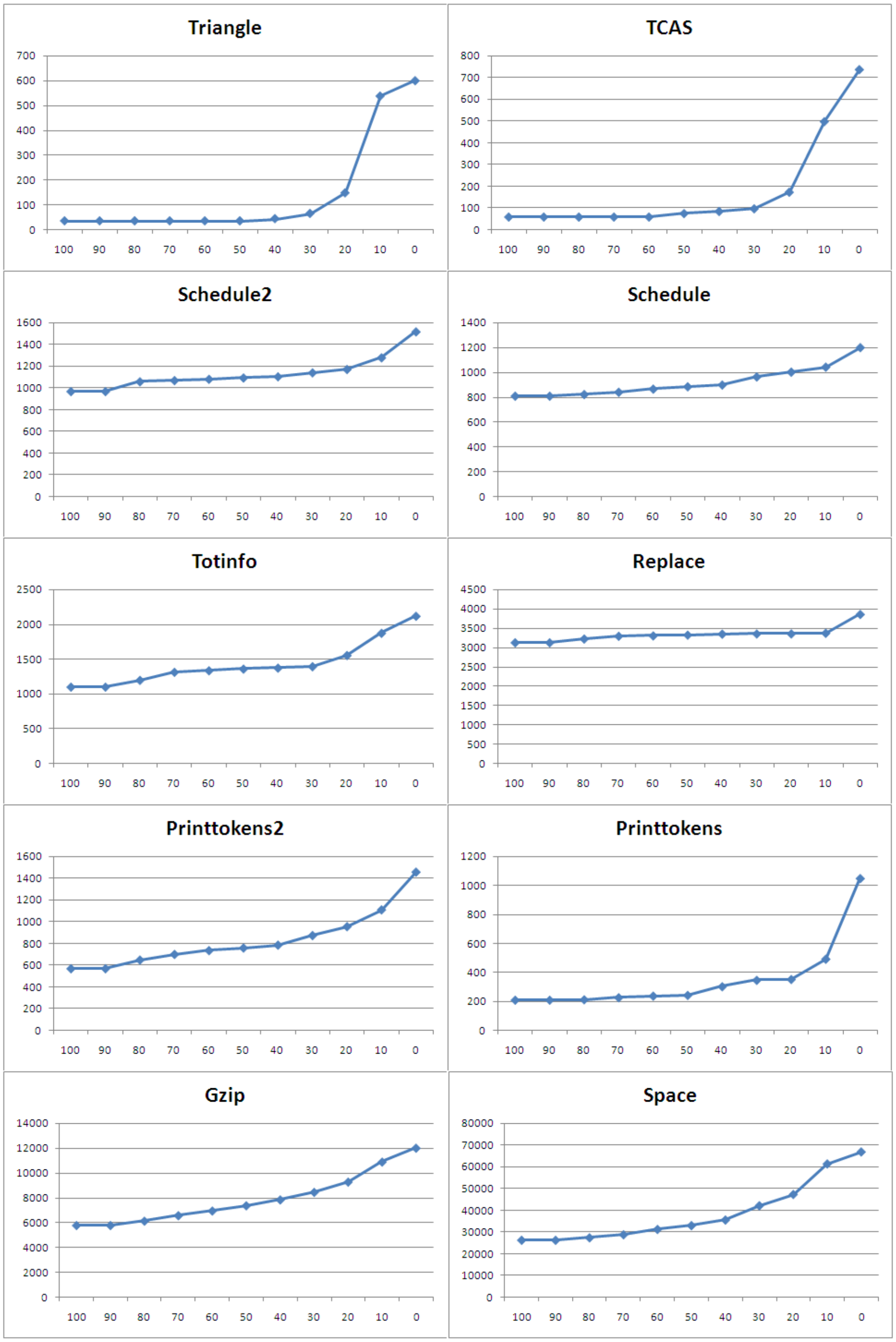
How Higher Order Mutation Testing challenges The UMT and AME myths

In Higher Order Mutation Testing, not all mutants that can be created are equal. Rather than pre–determining a set of mutation operators, in HOM Testing, we first decide upon a criterion of mutant quality. This criterion is captured by a fitness function, which is able to determine which is the better or two candidate mutants according to which better meets the criterion of interest. Mutation testing now becomes a process of *fitness guided selection* from the vast set of candidate mutants. The selection process is formulated as an optimization problem, using Search Based Software Engineering, guided by the fitness function.

The reformulation of Mutation Testing as a search based selection problem denotes a significant shift that finds its origins in the work on mutant sampling [3] and selective mutation [32]. However, the key difference in the approach advocated by HOM Testing is that *all* mutants are considered as candidates for selection; both first order and higher order and that the selection should is tailored to the program under test; it is not possible to define the fitness function without reference to the program under test p. This second observation is captured in the Global Mutant Operator Myth below.

Global Mutant Operator Myth (GMO)

The Global Mutant Operator Myth (GMO) assumes that the best way to create mutants is to define a set of global mutantion operators *before* any


Triangle

TCAS

Schedule2

Schedule

Totinfo

Replace

Printtokens2

Printtokens

Gzip

Space

Figure 5: The growth in the number of first order mutants killed for each program. In these graphs, y mutants are killed by $x\%$ or more of the test cases.

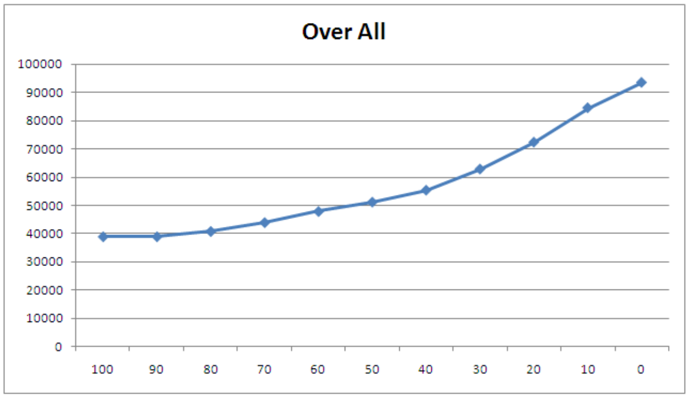


Figure 6: The growth in the number of mutants killed for all first order mutants created. In these graphs, y mutants are killed by $x\%$ or more of the test cases

programs under test have been encountered and then to apply this *same* set of global mutation operators to all programs, without any difference in application based on the programs under test themselves.

The misconception underlying this myth: According to the GMO Myth, all programs are alike and they should be subjected to the same set of mutation operators. However, all programs are not alike. No human tester would attempt to test every program using exactly the same testing procedure, so why should an automated process behave in this inflexible manner?

How Higher Order Mutation Testing challenges The GMO Myth According to HOM testing, each program should have its own set of mutants, purpose built, taking into account the syntactic structure and semantics of the program under test. The mutants are tailored by a fitness function that guides the search for good mutants. The fitness function is not merely a function of 'globally' determined aspects of 'mutant quality'. Rather, it is partly determined by 'local' consideration of the specific program under test. For example, in this paper, we seek to construct mutants that are hard to kill. This is inherently a property of a mutant that *depends* upon the syntax and semantics of program under test.

Competent Programmer Hypothesis Myth (CPH)

The Competent Programmer Hypothesis states that programmers are generally competent engineers who make some mistakes, but only relatively few and that, *therefore*, their programs are within a few keystrokes of being correct.

The Syntactic Semantic Size Myth (SSS)
In traditional Mutation Testing small syntactic changes are made. The philosophy underlying this approach is that these small syntactic changes denote the relatively small ways in which programs may be faulty. The origin of the SSS Myth can be traced to an incorrect extrapolation from the Competent Programmer Hypothesis, which is formulated by Offutt [33] as follows

> "The competent programmer hypothesis states that competent programmers tend to write programs that are 'close' to being correct. Although a program written by a competent programmer may be incorrect, it will only differ from a correct version by relatively few faults."

The SSS Myth extrapolates from this, with the (incorrect) assumption that relatively few faults means relatively few minor syntactic changes. The SSS Myth is implicit in all previous work on Mutation Testing, and is sometimes made explicit in the literature:-

> "The Competent Programmer Hypothesis states that programmers are generally competent and produce a program close to a correct program. A correct program can be constructed from an incorrect program by making changes that are composed of minor alterations." [34]

> "A competent programmer can be viewed as someone who creates programs close to being correct. If a competent programmer creates an incorrect program, the correct version may differ by a small amount, perhaps a single program lexeme." [35]

> "An important concept is the competent programmer hypothesis, a theory proposed by DeMillo, asserting that good programmers will write code that is within few keystrokes of what is correct. Therefore, 'simple' mutants that are a few keystrokes off of the original should be similar in nature to typical programmer faults." [36]

The misconception underlying this myth: A small semantic change in a program may be denoted by a small syntactic change, but it is not necessarily the case that small semantic changes are always denoted by small syntactic changes, far from it. A small variation in semantics can be produced by a programmer's selection of entirely the wrong data structure for the central database in their program. The incorrect program may behave correctly almost all of the time, thereby representing a small semantic change from the correct program. However, to correct the fault, an entirely different data structure is required.

The HOM Testing approach advocated in this paper is not the first to draw attention to the difference between a fault's syntactic and semantic size. For example, Offutt and Hayes addressed precisely this issue in 1996 [37]. However, the SSS myth is characterised here as a myth, because it remains an implicit assumption that lies at the heart of the way in which traditional Mutation Testing is practiced and it is challenged by HOM Testing, in a way that agrees, for example, with Offutt and Hayes [37], but which disagrees with those works cited above ([34, 35, 36]) and which significantly differs from Mutation Testing as currently practiced within the first order paradigm.

How Higher Order Mutation Testing challenges The CPO and SSS myths

HOM Testing accepts that *many* small syntactic changes may need to be made in order to construct a fault. However, although this paper goes some way towards the idea that large syntactic changes *may* be required to affect a small semantic change, the changes we consider are based on a set of initial FOMs that are the 'traditional' *tiny* syntactic changes of First Order Mutation Testing. Further work is required to consider macro level changes that denote significant syntactic changes to the program, in order to see whether a set of such large macro syntactic changes could be combined to create a mutant that is hard to kill, because it denotes a relatively small semantic change.

Coupling Hypothesis Extension Myth (CHE)

The coupling hypothesis is stated by Offutt [14] as follows:

> "Complex faults are coupled to simple ones in such a way that test data which find all simple ones will detect high percentage of complex faults."

Observe that Offutt does not claim that test data which finds all simple faults will find *all* complex faults. That is, by ignoring higher order mutants, there may be a small percentage of faults that are not tested.

The Coupling Effect Extension is a false corollary of the Coupling Effect Hypothesis that states that, because of the Coupling Effect, higher order mutants need not be considered. This is a convenient myth; in a world where all mutants have to be tested (according to the AME Myth) the CHE Myth renders Mutation Testing tractable, when it would otherwise have been prohibitively expensive. That is, if one accepts the AME Myth, then one is almost forced to also accept the CHE Myth, because of the infeasibily large number of mutants that would otherwise result from the inclusion of sets of higher order mutants.

The misconception underlying this myth: In this case, the misconception is simply to over extend the reach of the Coupling Effect Hypothesis. As formulated by Offutt, the hypothesis only claims that *most* HOMs are unnecessary, not that *all* HOMs are unnecessary.

How Higher Order Mutation Testing challenges The CHE Myth In HOM testing we consider higher order mutants and first order mutants as equally valid candidates from which to select. In this paper we focus on selecting from this candidate set, those HOMs that are killed by a subset of the test cases that kill the FOMs. Such HOMs are clearly coupled to their FOMs and yet they may strongly subsume the FOMs from which they are constructed. The paper directly challenges the CHE Myth, by seeking to find HOMs that can strongly subsume the FOMs from which they are constructed. This has the potential to reduce the number of mutants that need be considered with no loss test effectiveness. The paper does not claim that FOMs should be ignored, but merely that FOMs should be regarded as a special case of HOMs in which the mutant order is 1.

In the paper we further categorised the ways in which FOMs and HOMs can be coupled, distinguishing between the different categories, developing the theory that underlies the coupling effect. Further work will consider the other categories, including HOMs that are partly coupled or de coupled. These may also denote interesting faults.

8. Threats to Validity and Limitations

This section considers the threats to validity of our experiments. Although due to limitations of the experiments, the following threats may affect some of the results, for instance, the distribution and classification of subsumed HOMs, it should be noted that they do not affect the proof of the existence of strongly subsuming HOMs found by the experiments.

The selection of mutation operators is the first threat. In order to reduce the computational cost, in our experiment, 23 of 77 C mutation operators were selected to generate HOMs. However, the selected subset belongs to the five selective mutation operator categories suggested by Offutt [19, 20], so it is typical and also widely used by other researchers. The threat to validity will be overcome by future work which will investigate the relationship between HOMs and mutation operators.

The quality of the test sets is another potential threat. Since the fitness of HOMs is computed in terms of their fragility, low quality test sets may affect the results. Although the test sets provided by SIR achieve brunch coverage [17], given a different test set as input, the experiment may lead to different results in terms of distribution and classification. To overcome this threat we plan, in future work, to combine Higher Order Mutation Testing with the co-evolutionary Mutation Testing approach of Adamopoulos et al. [3]. This will allow us to co-evolve test sets adequate to kill the co-evolving HOM set.

The last threat is equivalent mutants. Although the problem of equivalent mutants has been studied by numerous researches [9, 10, 38], there is no approach that can solve it in both an effective and a precise way. In order to avoid this problem, the fitness function for finding interesting HOMs is designed to filter out potential equivalent mutants. With a low quality test set, some of the 'stubborn decoulped' HOMs may be wrongly treated as equivalent mutants. However, this would only reduce the number of HOMs found, so our results can be considered to be a lower bound on the number of subsuming HOMs to be found.

9. Related Work

This paper focused on an investigation of the higher order mutants and their relationship to first order mutants. Since a FOM can be taken as a case of a HOM, we have been able to use and adapt many existing techniques associate with previous work on traditional (first order) Mutation Testing. This section discusses this related work.

In our study the number of generated FOMs clearly affects the running cost. Therefore, we applied selective mutation techniques to reduce the number of FOMs. The idea of selective mutation is to choose a subset of mutant without significant loss of test effectiveness in Mutation Testing. It was first suggested as ‘constrained mutation’ by Wong et al. [39] in 1994. Offutt et al. extended this idea calling that selective mutation by only applying a selected set of mutation operators. [20, 19]

In addition to selective mutation, there are other techniques to reduce the number of FOMs. For example, mutant sampling, which was first proposed by Acree [40] and Budd [41]. The idea of mutant sampling is to randomly select a subset of mutants to be executed. In Wong’s [22] experiments, the results show that using only 10% of all mutants will only reduce test effectiveness by 16%.

In this paper we use a very simplistic selection technique to remove trivial higher order mutants that cannot help us to build useful subsuming higher order mutants. We remove first order mutants that are killed by no test cases and those killed by all test cases. This is a kind of ‘biased mutation sample’.

In order to apply Mutation Testing to real world programs, strong Mutation Testing is adopted by our experiments. In strong Mutation Testing, a mutant is killed if its final output is different from the original program. Therefore, each mutant is executed until it terminates or is killed. In order to reduce the running cost, previous work also considered weak Mutation Testing, first proposed by Howden in 1982 [24]. In weak Mutation Testing, mutants are evaluated immediately after execution of their mutation point. This is faster than strong Mutation Testing but at the expense of imprecision. There are also other approaches that lie between strong and weak Mutation Testing, known as firm Mutation Testing [25, 26].

In this paper, since this is the first paper to consider Higher Order Mutation Testing as a valid alternative to First Order Mutation Testing, we prefer the full precision of strong Mutation Testing. Weak and firm Higher Order Mutation Testing remain interesting and potentially important topics for future work.

The closest research area related to this work is the previous work on the coupling effect hypothesis. Although the coupling effect has been studied by many researches [13, 14, 25, 41, 42, 43], these studies all focus on verifying or disproving the coupling effect, rather than finding subsuming HOMs, which could be thought of as special cases.

The experimental studies presented by Offutt [13, 14] show results that support Offutt’s version of the mutation coupling effect. However, Offutt modifies Demillo et al.’s original statement of the coupling effect [1], which was:

> “ Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors [1]. ”

The original formulation appears to suggest that *all* HOMs are coupled, whereas Offutt [14] weakens this to suggest that a ‘large percentage’, are coupled:

> “ Complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants in a program will detect a large percentage of the complex mutants [14]. ”

The stronger formulation of the coupling effect is termed the CHE Myth in the present paper.

Some of our ‘subsuming HOMs’ are drawn from the minority ‘de-coupled’ mutant set. Offutt’s experiments were based on three small FORTRAN77 programs (16-28 LOC). All of the second order and some of the third order mutants of these programs were generated by the Mutation Testing tool Mothra. The results suggested that the selected adequate test set which killed all the first order mutants, killed over 99% of the second and third order mutants. This study implied that the mutation coupling effect is valid in the most general case, which also agreed with the empirical study by Lipton and Sayward [44] and Morell[25].

The validity of the mutation coupling effect has also been considered in a theoretical study by Wah [42, 43]. A simple theoretical model, the q function model, considers a program to be a set of finite functions. By applying test sets of order 1 and order 2 to this model, the results indicated that the average survival ratio of high-order mutants is $1/n$ and $1/n^2$ respectively, which is also similar to the estimated results of empirical studies mentioned above. However, compared to a real world program, this model is very simplistic. In real programs, the data and control flow between functions are more complex and unpredictable.

In this paper we proposed using strongly subsuming HOMs in mutation testing. This idea has been partly proved by Polo et al.’s work [45]. In their experiment, they focused on a specific order of HOMs, namely the second order mutants. They proposed different algorithms to combine first order mutants to generate the second order ones. By applying the second order mutants, test effort was reduced around 50%, without much loss of test effectiveness. However, Polo et al. did not use search based optimization and so they were limited to small number of lower orders. Future work will consider the question of whether search can find arbitrary order HOMs that can reduce test effort.

10. Conclusion

This paper introduces the paradigm of Higher Order Mutation Testing (HOM Testing). The paper introduced the concept of subsuming higher order mutant; a HOM that is hard to kill than its constituent FOMs. In terms of fragility, the whole is greater than the sum of its parts. That is, the HOM is greater than the collection of FOMs from which it is constructed because it is less fragile. The paper introduced a search-based approach to find these subsuming HOMs and presented an empirical study that compared a greedy algorithm, a genetic algorithm and a hill climbing algorithm.

The experimental results from ten programs indicate that there exist many subsuming HOMs in each studied program. The results also reveal that genetic algorithm is the most efficient algorithm for finding those subsuming HOMs, while the greedy algorithm and hill climbing algorithm can also be used to improve the quality of the results.

The paper introduced the concept of a Strongly Subsuming HOM. A Strongly Subsuming HOM is only killed by a subset of the intersection of the set of test cases that kill its constituent FOMs. Therefore, a Strongly Subsuming HOM is one that is so much harder to kill than the FOMs from which it is constructed that one can replace all the FOMs with the SHOM without any loss of test effectiveness.

The paper showed that the search based approach was able to find a number of these SSHOMs in every one of the 10 programs studied. Though the proportion of all HOMs that are SSHOMs is small, the size of the HOM set grows exponentially so the number of these valuable SSHOMs is relatively high. The paper illustrated the intricate interplay between faults that SSHOMs exhibit by describing one of the HOMs found by the genetic algorithm and the test sets that kill it and its constituent FOMs in detail.

There remains much work to do in developing the field of Higher Order Mutation Testing. Future work will consider different categories of HOMs and the way in which HOMs can be used to reduce mutation test effort; replacing a whole set of FOMs by a HOM that strongly subsumes them. Further work is also required to develop techniques for generating test data able to kill HOMs. For this we plan to use a co-evolutinary approach along the lines first suggested by Adamopoulos et al. [3]. The approach will co-evolve sets of SSHOMs and the test cases that are able to kill them with the goal of generating a set of very subtle faults and a set of test data that is sufficient to reveal them.

Acknowledgement

The authors would like to thank Jeff Offutt and John Clark for many helpful discussions on Higher Order Mutation and for commenting on an earlier draft of this paper.

Yue Jia is supported by the EPSRC grant, EP/D050863 (SEBASE) and the ORSAS. Mark Harman is partly supported by the EPSRC grants EP/F059442/1, EP/F010443/1, EP/E002919/1, EP/D050863/1 and GR/T22872/01.

References

[1] R. A. DeMillo, R. J. Lipton, F. G. Sayward, Hints on test data selection: Help for the practicing programmer, IEEE Computer 11 (4) (1978) 34–41.

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

[3] K. Adamopoulos, M. Harman, R. M. Hierons, How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution, Genetic and Evolutionary Computation GECCO 2004 3103 (2004) 1338–1349.

[4] K. Ayari, S. Bouktif, G. Antoniol, Automatic mutation test input data generation via ant colony, in: GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM, New York, NY, USA, 2007, pp. 1074–1081.

[5] J. S. Bradbury, J. R. Cordy, J. Dingel, Comparative assessment of testing and model checking using program mutation, in: TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, IEEE Computer Society, Washington, DC, USA, 2007, pp. 210–222.

[6] S.-S. Hou, L. Zhang, T. Xie, H. Mei, J.-S. Sun, Applying interface-contract mutation in regression testing of component-based software, in: Proc. 23rd IEEE International Conference on Software Maintenance(ICSM 2007), 2007.

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

[8] D. Baldwin, F. Sayward, Heuristics for determining equivalence of program mutations, Research Report 276, Department of Computer Science, Yale University (1979).

[9] R. M. Hierons, M. Harman, S. Danicic, Using program slicing to assist in the detection of equivalent mutants, Software Testing, Verification & Reliability 9 (4) (1999) 233–262.

[10] A. J. Offutt, W. M. Craft, Using compiler optimization techniques to detect equivalent mutants, Software Testing, Verification & Reliability 4 (3) (1994) 131–154.

[11] A. J. Offutt, J. Pan, Detecting equivalent mutants and the feasible path problem, in: Proceedings of the 1996 Annual Conference on Computer Assurance, IEEE Computer Society Press, Gaithersburg, Maryland, 1996, pp. 224–236.

[12] H. Agrawal, R. Demillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, E. Spafford, Design of mutant operators for the C programming language, Tech. Rep. SERC-TR-41-P, Software Engineering Research Centre (Mar. 1989).

[13] A. Offutt, The coupling effect: fact or fiction, SIGSOFT Softw. Eng. Notes 14 (1989) 131–140.

[14] A. J. Offutt, Investigations of the software testing coupling effect, ACM Transactions on Software Engineering and Methodology 1 (1) (1992) 5–20.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, Second Edition, The MIT Press, 2001.

[16] M. Mitchell, An Introduction to Genetic Algorithms, The MIT Press, 1996.

[17] H. Do, S. G. Elbaum, G. Rothermel, Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact., Empirical Software Engineering: An International Journal 10 (4) (2005) 405–435.

[18] C. C. Michael, G. McGraw, M. Schatz, Generating software test data by evolution, IEEE Transactions on Software Engineering 27 (12) (2001) 1085–1110.

[19] A. J. Offutt, G. Rothermel, C. Zapf, An experimental evaluation of selective mutation, in: Proceedings of the Fifteenth International Conference on Software Engineering, IEEE Computer Society Press, Baltimore, Maryland, 1993, pp. 100–107.

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

[21] Y. Jia, M. Harman, Milu: A customizable, runtime-optimized higher order mutation testing tool for the full C language, in: 3rd Testing Academia and Industry Conference - Practice and Research Techniques TAIC PART'08, Windosor, UK, 2008.

[22] W. E. Wong, On mutation and data flow, Phd thesis, Purdue University, West Lafayette, Indiana (December 1993).

[23] R. H. Untch, A. J. Offutt, M. J. Harrold, Mutation analysis using mutant schemata, in: Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis, Cambridge, Massachusetts, 1993, pp. 139–148. doi:10.1145/154183.154265.

[24] W. E. Howden, Weak mutation testing and completeness of test sets, IEEE Transactions on Software Engineering SE-8 (4) (1982) 371–379.

[25] L. J. Morell, Theoretical insights into fault-based testing, in: Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, IEEE Computer Society Press, Banff Alberta, Canada, 1988, pp. 45–62. doi:10.1109/WST.1988.5353.

[26] M. R. Woodward, K. Halewood, From weak to strong, dead or alive? an analysis of some mutationtesting issues, in: Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, IEEE Computer Society Press, Banff Albert, Canada, 1988, pp. 152–158. doi:10.1109/WST.1988.5370.

[27] J. R. Horgan, A. P. Mathu, Weak mutation is probably strong mutation, Technical Report SERC-TR-83-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana (December 1990).

[28] R. A. DeMillo, E. W. Krauser, A. P. Mathur, Compiler-integrated program mutation, in: Proceedings of the Fifteenth Annual Computer Software and Applications Conference, IEEE Computer Society Press, Tokyo, Japan, 1991, pp. 351–356.

[29] M. E. Delamaro, J. C. Maldonado, Proteum-a tool for the assessment of test adequacy for c programs, in: Proceedings of the Conference on Performability in Computing Systems, New Brunswick, New Jersey, 1996, pp. 79–95.

[30] B. Choi, A. P. Mathur, High-performance mutation testing, Journal of Systems and Software 20 (2) (1993) 135–152. doi:10.1016/0164-1212(93)90005-I.

[31] M. Sahinoglu, E. H. Spafford, A bayes sequential statistical procedure for approving software products, in: W. Ehrenberger (Ed.), Proceedings of the International Federation for Information Processing Conference on Approving Software Products, Elsevier Science, Garmisch Partenkirchen, Germany, 1990, pp. 43–56.

[32] Y. Zhan, J. A. Clark, Search-based mutation testing for simulink models, in: Proceedings of the 2005 conference on Genetic and evolutionary computation, Washington DC, USA, 2005, pp. 1061–1068.

[33] A. Offutt, S. Lee, An empirical evaluation of weak mutation, IEEE Transactions on Software Engineering 20 (5) (1994) 337–344. doi:An empirical evaluation of weak.

[34] R. T. Alexander, J. M. Bieman, S. Ghosh, B. Ji, Mutation of java objects, in: ISSRE '02: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02), 2002, p. 341.

[35] P. May, K. Mander, J. Timmis, Software vaccination: An artificial immune system approach to mutation testing, in: Artificial Immune Systems Second International Conference, ICARIS 2003, 2003, pp. 81–92.

[36] I. Stamelos, Detecting associative shift faults in predicate testing, Journal of Systems and Software 66 (2003) 57–63.

[37] A. J. Offutt, J. H. Hayes, A semantic model of program faults, in: ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis, 1996, pp. 195–200.

[38] A. J. Offutt, J. Pan, Automatically detecting equivalent mutants and infeasible paths, Software Testing, Verification & Reliability 7 (3) (1997) 165–192.

[39] W. E. Wong, M. E. Delamaro, J. C. Maldonado, A. P. Mathur, Constrained mutation in c programs, in: Proceedings of the 8th Brazilian Symposium on Software Engneering, Curitiba, Brazil, 1994, pp. 439–452.

[40] A. T. Acree, On mutation, Phd thesis, Georgia Institute of Technology, Atlanta Georgia (1980).

[41] T. A. Budd, Mutation analysis of program test data, Phd thesis, Yale University, New Haven, Connecticut (1980).

[42] K. S. H. T. Wah, A theoretical study of fault coupling, Software Testing, Verification & Reliability 10 (1) (2000) 3–46.

[43] K. S. H. T. Wah, An analysis of the coupling effect i: single test data, Science of Computer Programming 48 (2003) 119–161.

[44] R. J. Lipton, F. G. Sayward, The status of research on program mutation, in: Digest for the Workshop on Software Testing and Test Documentation, 1978, pp. 355–373.

[45] M. Polo, M. Piattini, I. G. Rodriguez, Decreasing the cost of mutation testing with second-order mutants, Software Testing, Verification and Reliability 234 (2008) 234.