Constructing Subtle Faults Using Higher Order Mutation Testing

Yue Jia
King's College London
Strand, London
WC2R 2LS, UK
yue.jia@kcl.ac.uk

Mark Harman
King's College London
Strand, London
WC2R 2LS, UK
mark.harman@kcl.ac.uk

Abstract

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. This paper investigates Higher Order Mutants (HOMs). It 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 into subsuming HOMs, using six benchmark programs. This is the largest study of mutation testing to date. To overcome the exponential explosion in the number of mutants considered, the paper introduces a search based approach to the identification of subsuming HOMs. Results are presented for a greedy algorithm, a genetic algorithm and a hill climbing algorithm.

1. Introduction

Mutation testing is a fault-based software testing technique initially proposed by DeMillo et al. [8] and Hamlet [11]. 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 [1], to generate effective test data [3] and to compare techniques for verification [5, 13].

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 [10]. Indeed, traditional source code analysis has been proposed as a technique to address this equivalent mutant problem [4, 12, 20, 23], 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. [2]. 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. 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.

Table 1. A Example of Mutating Operation

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

This paper introduces the concept of subsuming HOMs. A subsuming HOM 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 HOM.

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.36

A strongly subsuming HOM is only killed by a subset of the intersection of test cases that kill each FOM from which it is constructed.

Consider a strongly subsuming HOM, 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.

2. The algorithms we introduce target all subsuming HOMs, rather than specifically searching for strongly subsuming HOMs. However, the results reveal that approximately 15% of the HOMs found turn out to be strongly subsuming, suggesting that these highly valuable HOMs are not as rare as one might think.

3. We report on the relationship between subsumption and mutant order. The results reveal that the more FOMs a program has, the higher is the order at which peak subsumption occurs. This is an important finding because it means that large programs, with larger sets of FOMs also tend to have HOMs that can subsume a lager number of the FOMs. That is, as the problem scale increases the ability of the solution approach to tackle scale also increases. This finding provides evidence that higher order mutation testing may turn out to be far more scalable that first order mutation testing.

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 presents a search-based approach and explains three meta-heuristic algorithms used to find subsuming HOMs. Section 4 explains the experimental setting, while the results are discussed in Section 5. Section 6 discusses threats to validity of experiment. Section 7 introduces related work, and the paper concludes with Section 8.

2. High 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).

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.

Formal Definitions

a. Strongly Subsuming and Coupled

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

b. Weakly Subsuming and Coupled

$|T_h| < |\bigcup_i T_i| \ , \quad T_h \neq \emptyset$

and $\ T_h \cap \bigcup_i T_i \neq \emptyset$

c. Weakly Subsuming and De-coupled

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

and $\ T_h \cap \bigcup_i T_i = \emptyset$

d. Non-Subsuming and De-coupled

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

and $\ T_h \cap \bigcup_i T_i \neq \emptyset$

e. Non-Subsuming and De-coupled

$T_h = \emptyset$ (Equivalent)

f. Non-Subsuming and Coupled

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

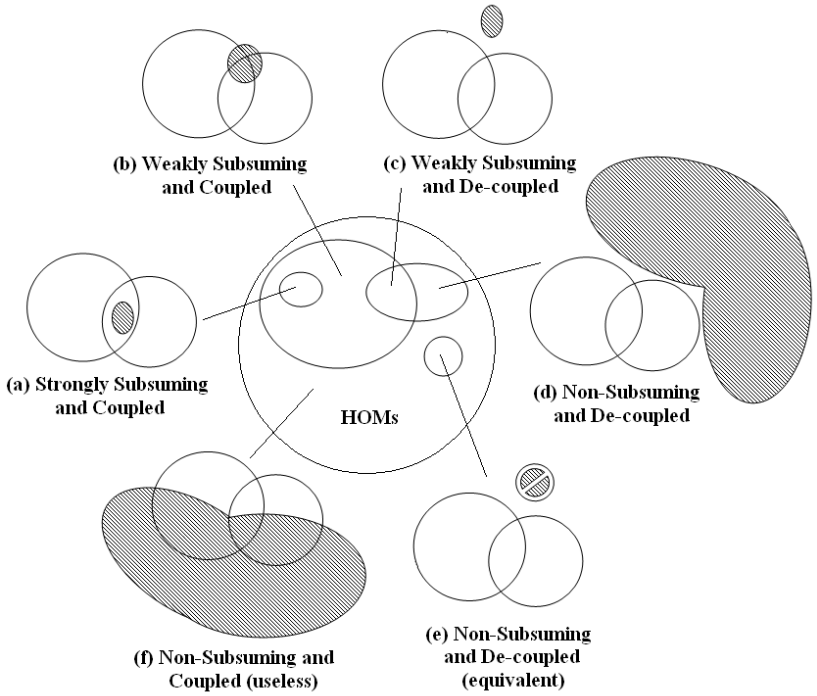

Figure 1. HOMs Classification. The central Venn Diagram depicts important subclasses into which HOMs fall, while the outer diagrams depict killing test sets for the HOMs (shaded) and their constituent FOMs (unshaded). For ease of exposition, the diagrams illustrate only the second order case, whereas 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. For a HOM h, constructed from FOMs $f_1, ..., f_n$, the test set T_h contains all the test cases that kill h, while the test sets $T_1, ..., T_n$ are the test sets that kill that kill $f_1, ..., f_n$ respectively.

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. Suppose a program contains n FOMs; it would have n^n HOMs. For example, even a small program like Triangle has 50 lines of code (LoC) and approximately 500 FOMs. Therefore, it is impossible to generate all of the HOMs, because there are 10^{1349} in all; considerably more numerous than the number of atoms in the known universe.

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 [8, 18, 19] 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 pro-

cess. 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**.

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 6.1 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.

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 [4, 12, 20, 23], 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 [19], 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 1 (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 2 (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 [7]. 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.

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. The pseudo-code is shown in Algorithm 2.

```
   Input  : Running Time Limit: limit
   Output: Mutation vector homlist
1  set counter = 0
2  while counter < limit do
3      set hom = generateRandFOM()
4      foreach FOM m of Program do
5          temp_hom = combine(hom,m)
6          if fitness(temp_hom) >
           fitness(hom) then
7              hom = temp_hom
8          end
9          archvie(temp_hom)
10     end
11 end
```

Algorithm 1: Optimized Greedy Algorithm

```
   Input  : Running Time Limit: limit
   Output: Mutation vector homlist
1  set counter = 0
2  foreach Mutation m in population do
3      set m = generateRandHOM()
       fitness(m)
4  end
5  while counter < limit do
6      createMtPool(population)
7      archvie(population)
8      crossover(population)
9      mutate(population)
10     fitness(population)
11     counter ++
12 end
```

Algorithm 2: Optimized Genetic Algorithm

```
   Input  : Running Time Limit: limit
   Output: Mutation vector homlist
1  set counter = 0
2  set hom = generateRandFOM()
3  while counter < limit do
4      temp_hom = getNeighbor(hom)
5      if fitness(temp_hom) <
       fitness(hom) then
6          hom = temp_hom
7          archvie(hom)
8      end
9      hom = generateRandFOM()
10     counter ++
11 end
```

Algorithm 3: Optimized Hill Climbing Algorithm

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 connot 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

The first research question addresses the main objective of this work; how prevalent are subsuming HOMs? If there are very few then there would be no future for higher order mutation testing. The research questions 2, 3 and 4 are based on question 1. They further study of these subsuming HOMs.

RQ1: How numerous are subsuming HOMs?

RQ2: What proportion of subsuming HOMs are strongly subsuming?

RQ3: What is the relationship between mutant order and subsumption?

RQ4: Which algorithms perform best at finding subsuming HOMs?

5.2 Test Programs

The experiments use six benchmark C programs with branch adequate test sets from the Software-artifact Infrastructure Repository (SIR) [9], 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 [15]. `TCAS` is a program used to avoid an aircraft collision. `Schedule2` is a program that prioritizes schedulers. `Totinfo` is a program that computes statistics from input data. `Printtokens` is a lexical analyser 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 (100 LoC). 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 Test Program

Program	Scale	# of FOM	# of SHOM
`Triangle`	50 LoC	584	47
`TCAS`	150 LoC	679	98
`Schedule2`	350 LoC	1,014	78
`Totinfo`	500 LoC	2,570	320
`Printtokens`	750 LoC	866	67
`Space`	6,000 LoC	7,570	522

5.3 Mutation Operators

The study of Agrawal et al. introduces 77 mutation operators for the C language [2]. However, not all of the mutation operators increase the effectiveness of mutation testing. Offutt [22, 25] 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 (28 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 [28]. 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 testings 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 Experiment Procedure

To answer the proposed research questions RQ1—RQ4, the experiments are divided into two steps. The first step aims to investigate RQ1, RQ2, and RQ3. For each candidate program, the experiments ran the genetic algorithm (Section 4.4) to find subsumed HOMs from order 2 to 13.

The second step aims to investigate the most suitable algorithm for finding the subsumed HOMs efficiently, which addresses RQ4. RQ3 establish that for all but one program

(Space) the subsuming HOMs have orders ranging from 2 to 10. Therefore, a limit on each search for HOMs was set to order 10. These four were run on every candidate test program to find subsumed HOMs of order 2 to 10. To treat the random and other search-based algorithms in a fair way for comparison, a running-counter is placed in each of their fitness functions. This is used to capture the effort each algorithm expends on optimization.

6. Results and Analysis

Before getting into the detailed quantitative results we first present (in Section 6.2) a case study of a single second order HOM. This HOM illustrates the kind of subtle fault that higher order mutation testing can reveal.

6.1 Case Study

The Triangle is a small C program (50 LoC) that has been studied for at least 30 years [8]. 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 a triangle's $trian$ value equals 0, and the side lengths satisfy the side length constraint, it is a 'valid scalene' triangle.

Program 4 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 3 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.

Program: Triangle
Input : Three sides a, b, c
Output : Types of Triangle

```
1  int trian
2  if (a <= 0 || b <= 0 || c <= 0) then
3      return INVALID
4  trian = 0
5  if (a == b) then trian = trian + 1
6  if (a == c) then trian = trian + 2
7  if (b == c) then trian = trian + 3
8  if (trian == 0) then
9      if (a + b < c || a + c < b || b + c < a) then
10         return INVALID
11     else return SCALENE
12 if (trian > 3) then return EQUILATERAL
13 if (trian == 1 && a + b > c) then
14     return ISOSCELES
15 else if (trian == 2 && a + c > b) then
16     return ISOSCELES
17 else if (trian == 3 && b + c > a) then
18     return ISOSCELES
19 return INVALID
```

Mutant : FOM_i

```
13 if (trian > 1 && a + b > c) then
14     return ISOSCELES
15 else if (trian == 2 && a + c > b) then
16     return ISOSCELES
17 else if (trian == 3 && b + c > a) then
18     return ISOSCELES
19 return INVALID
```

Mutant : FOM_j

```
13 if (trian == 1 && a + b <= c) then
14     return ISOSCELES
15 else if (trian == 2 && a + c > b) then
16     return ISOSCELES
17 else if (trian == 3 && b + c > a) then
18     return ISOSCELES
19 return INVALID
```

Mutant : HOM_ij

```
13 if (trian > 1 && a + b <= c) then
14     return ISOSCELES
15 else if (trian == 2 && a + c > b) then
16     return ISOSCELES
17 else if (trian == 3 && b + c > a) then
18     return ISOSCELES
19 return INVALID
```

Program 4: 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 3 depicts the corresponding killing test cases.

6.2 Analysis

To begin the analysis, the last two columns of Table 2 present the overall results for sum of all subsuming HOMs found in all six subject programs by our GA algorithm with 100,000 fitness evaluations. From the smallest

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 3. Killing Test Cases for the Triangle HOM and its FOMs

`Triangle` program (50LoC) to the largest `Space` program (6,000Loc), there exist subsuming HOMs. Furthermore, the quantity of the HOMs increases substantially as the number of FOMs increases. For instance, there are 584 FOMs and 47 subsuming HOMs in `Triangle` while there are 7,570 FOMs and 522 HOMs in the program `Space`.

Of all subsuming HOMs found in our experiments, approximately 15% of these were found to be of the highly valuable, strongly subsuming type. This is a very encouraging finding. It means that there may be many cases where a set of first order mutants may be replaced by a higher order mutant, thereby reducing the number of mutants required overall.

The chart in Figure 2 presents the overall order distribution for all six test programs. In the chart, each type of line represents one of the studied programs. The x-axis shows the mutant order number and the y-axis shows the number of subsuming HOMs found. Therefore, the peak value of each line represents the largest number of subsuming HOMs that can be found by the order that shows on the x-axis. The figure answers RQ3. It shows that subsuming HOMs are unevenly distributed. This peak order is named peak subsumption in this paper; one that is correlated with the total number of its FOMs. There are insufficiently many programs for us to meaningfully apply Spearman Rank correlation test but observe that the evidence for correlation is compelling; the ordering by peak subsumption and the ordering number of FOMs are *identical*.

The chart in Figure 3 presents the result of comparison of the four algorithms (see Section 4), which answers the RQ4. 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. In Figure 3, the x-axis shows the four algorithms, and y-axis shows the percentage of oracle HOMs found. From the chart, the bar for the randomized algorithm is the lowest, as expected. The genetic algorithm bar is the highest. We believe that the 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. The optimized hill climbing algorithm is second best; slightly better than greedy.

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.

7. 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, 28 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 [21, 22, 25], 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 [9], 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. [1]. This will allow us to co-evolve test sets adequate to kill the co-evolving HOM set.

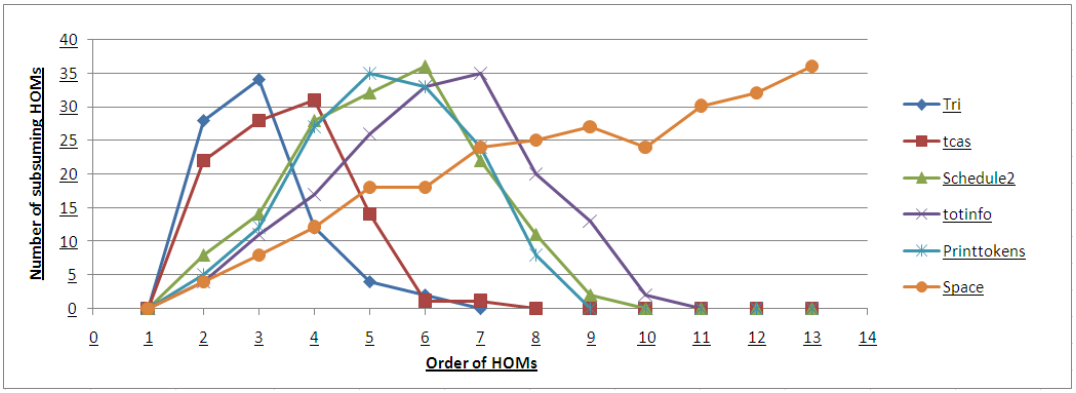

Figure 2. Overall Order Distribution of Valuable HOMs The key lists the 6 programs in ascending order determined by their scale. Note the correlation between this 'order of problem scale' and the order denoted by the sequence of points of maximal subsumption.

The last threat is equivalent mutants. Although the problem of equivalent mutants has been studied by numerous researches [12, 20, 24], 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.

8. Related Work

The main research area related to this work is the coupling effect hypothesis. Although the coupling effect has been studied by many researches [6, 17, 18, 19, 26, 27], these studies all focus on verifying or disproving the coupling effect, rather than finding subsuming HOMs.

The experimental studies presented by Offutt [18, 19] show results that support the mutation coupling effect. However, Offutt modifies Demillo et al.'s statement that *all* HOMs are coupled to weaken it to suggest that the vast majority, are coupled. 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 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 [14] and Morell[17].

The validity of the mutation coupling effect has also been considered in a theoretical study by Wah [26, 27]. 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 are interested in de-coupled HOMs (the exceptions to the coupling effect hypothesis), where these decoupled HOMs are also subsuming. However, these HOMs are only a part of our story. We are also interested in coupled HOMs; these obey the coupling effect hypothesis, but may be strongly subsuming. Our studies indicate that up to 15% of all subsuming HOMs may be strongly subsuming. This is an extremely encouraging finding. It indicates that higher order mutation testing may have been incorrectly overlooked by previous work on mutation testing.

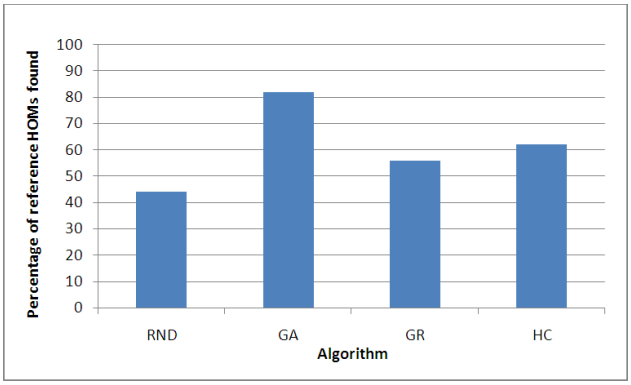

Figure 3. Comparison Of Four Algorithms

9. Conclusion

This paper has introduced the concept of subsuming HOMs and higher order mutation testing. The paper has also introduced a search-based approach to find these subsuming HOMs and presented an empirical study that compares a greedy algorithm, a genetic algorithm and a hill climbing algorithm.

The experimental results 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.

Future work will investigate strongly subsuming HOMs in detail and evaluate the extent to which they can reduce mutation testing effort and increase its effectiveness.

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] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *GECCO (2)*, volume 3103 of *Lecture Notes in Computer Science*, pages 1338–1349. Springer, 2004.

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

[3] K. Ayari, S. Bouktif, and 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*, pages 1074–1081, New York, NY, USA, 2007. ACM.

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

[5] J. S. Bradbury, J. R. Cordy, and 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*, pages 210–222, Washington, DC, USA, 2007. IEEE Computer Society.

[6] T. A. Budd. *Mutation analysis of program test data*. PhD thesis, Yale University, New Haven , CT , USA, 1980.

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

[8] 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, Apr. 1978.

[9] 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.

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

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

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

[13] S.-S. Hou, L. Zhang, T. Xie, H. Mei, and J.-S. Sun. Applying interface-contract mutation in regression testing of component-based software. pages 174–183, October 2007.

[14] R. J. Lipton and F. G. Sayward. The status of research on program mutation. In *Digest for the Workshop on Software Testing and Test Documentation*, pages 355–373. IEEE/NBS, IEEE Computer Society Press, 1978.

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

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

[17] L. J. Morell. *A Theory of Error-Based Testing*. PhD thesis, University of Maryland, College Park , MD , USA, 1984.

[18] A. J. Offutt. The coupling effect: fact or fiction. *Proceedings of the ACM SIGSOFT '89 third symposium on Software Testing, And Verification*, 14(8):131–140, 1989.

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

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

[21] A. J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutation operators. Technical report, Sept. 01 1994.

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

[23] A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. In *Annual Conference on Computer Assurance (COMPASS 96), IEEE Computer Society Press*, pages 224–236, Gaithersburg, MD, June 1996.

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

[25] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *International Conference on Software Engineering*, pages 100–107, 1993.

[26] K. S. H. T. Wah. A theoretical study of fault coupling. *Software Testing, Verification & Reliability*, 10(1):3–45, 2000.

[27] K. S. H. T. Wah. An analysis of the coupling effect I: Single test data. *Science of Computer Programming*, 48(2-3):119–161, 2003.

[28] M. H. Yue Jia. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Testing: Academic & Industrial Conference*, Windosor, UK, August, 2008 To appear.