Constructing Subtle Higher Order Mutants for Java and AspectJ Programs

Elmahdi Omar, Sudipto Ghosh, and Darrell Whitley
Colorado State University
Fort Collins, CO, USA
Email: Elmahdi.Omar@colostate.edu, ghosh@cs.colostate.edu, whitley@cs.colostate.edu

***Abstract*—One goal of higher order mutation testing is to produce higher order mutants (HOMs) that represent subtle faults. We define subtle HOMs as those that are not killed by an existing test set that kills all the first order mutants of a given program. The fault detection effectiveness of the test set can be improved by adding test cases that kill subtle HOMs. However, finding subtle HOMs can be costly even for small programs because of the large space of candidate HOMs. Moreover, a large majority of HOMs are killed by test sets that kill all first order mutants, making the subtle ones relatively rare.**

We introduce three search-based algorithms (Genetic Algorithm, Local Search, and Random Search) for finding subtle HOMs in Java and AspectJ programs. All three algorithms found subtle HOMs for all studied programs but Local Search was more successful in finding subtle HOMs than Genetic Algorithm and Random Search.

***Keywords*—Higher order mutation testing; aspect-oriented programming; search-based software engineering; genetic algorithm; local search; random search; software testing;**

I. Introduction

Search-based software engineering techniques are becoming popular because they are often able to find optimal or near optimal solutions for large-scale software engineering problems, and yet their cost is much less than exhaustive and exact methods. We investigate the use of search-based techniques in the area of higher order mutation testing.

One goal of higher order mutation testing is to produce Higher Order Mutants (HOMs) that can be used to improve the fault detection effectiveness of test sets. Because the majority of First Order Mutants (FOMs) represent trivial faults that are often easily detected, researchers have proposed the use of higher order mutation to identify combinations of single faults that are harder to detect than each fault individually [1]–[6]. Harder to detect faults are those that fewer test cases can detect. HOMs are useful for developing test cases that are more effective in detecting faults.

Because HOMs are created by inserting two or more faults (i.e., combining two or more FOMs) in a program, the number of candidate HOMs is exponentially large. The majority of HOMs are *coupled* to their constituent FOMs such that they are killed by any test set that kills all the FOMs [1], [7]. However, there exist some HOMs that are not killed by the same test set which kills all of the constituent FOMs for a given program. Such HOMs are referred to as *subtle HOMs* in this paper. The subtle HOMs represent cases where single faults interact (e.g., by masking each other) to produce new faulty behavior that has not been tested. Such HOMs can be used as the basis to generate new test inputs to improve the fault detection effectiveness of an existing test set.

Although such HOMs are usually rare [1]–[3], they are important to the testing process. Research has shown that the majority of real faults not detected during testing are complex faults that cannot be simulated using FOMs [8]. In fact, fixing 90% of these reported faults requires making multiple changes to the source code. Such faults can be simulated using HOMs but not FOMs.

Jia and Harman [1]–[3], [6] used search based optimization techniques to explore the space of all candidate HOMs to seek out *strongly subsuming HOMs* in C programs. Strongly Subsuming HOMs are those that can replace their constituent FOMs without loss of test effectiveness. Such HOMs can reduce testing effort and cost by reducing the total number of mutants that need to be compiled and executed. However, in the available higher order mutation testing literature [1]–[3], [5], [6], [9], [10], there is a lack of techniques for generating HOMs that are not killed by an existing test set.

A brute force approach that evaluates every HOM in the search space by constructing, compiling, and executing the HOM using the given test set is unrealistic. In this paper, we propose three algorithms, Genetic Algorithm, Local Search, and Random Search, for finding subtle HOMs for a given Java/AspectJ program.

We compare the effectiveness of the proposed algorithms in terms of their ability to find subtle HOMs. We also analyze the HOMs produced by each algorithm to understand how each algorithm explored the search space.

The rest of the paper is organized as follows. Section II summarizes concepts pertinent to higher order mutation testing and aspect-oriented programs. Section III presents the objective function, the proposed search-based algorithms, and the implementation details. Section IV presents the experiment goals and set up. Section V discusses the experimental results. Section VI presents related work in the areas of higher order mutation testing and mutation testing for aspect-oriented programs. Section VII concludes the paper and outlines directions for future work.

II. Background

Mutation testing involves modifying a program by creating syntactic changes that simulate errors made by programmers. The syntactic changes are based on well-defined operators

978-1-4799-2366-3/13/$31.00 ©2013 IEEE

called mutation operators. In traditional mutation testing, mutants are generated by creating a single syntactic change in the original program to produce a First Order Mutant (FOM). In higher order mutation testing, mutants are generated by creating multiple syntactic changes in the original program to produce a Higher Order Mutant (HOM). A HOM of degree n is created by making n syntactic changes (i.e., combining n FOMs). A mutant is said to be killed by a test case if the mutant produces a different output than the original program. A mutant is considered to be an *equivalent mutant* if it is semantically identical to the original program and thus, there is no test case that can distinguish between the outputs of the mutant and the original program.

Jia and Harman [1] classified HOMs in terms of their coupling relationships with FOMs. An HOM is considered to be coupled to its constituent FOMs "if a test set that kills the FOMs also contains test cases that kill the HOM" [1]. Otherwise the HOM is decoupled. Decoupled HOMs are valuable to the testing process because they represent different faults than their constituent FOMs and can be used to improve fault detection effectiveness.

In previous work [9], we used four approaches to construct HOMs for Java and AspectJ programs. The approaches are based on AOP fault models [5], [12], [14], which describe faults that can occur in base classes, in aspects (pointcut, inter-type declarations, aspect declaration, and advice), or in the interaction between the base classes and aspects. The construction approaches are also used in this paper and are as follows:

1) Single Base Class or Aspect Approach (SCA): Each HOM is constructed by inserting two or more mutation faults into a single base class or by inserting two or more mutation faults into a single aspect.
2) Dispersed Base Class Approach (BC): Each HOM is constructed by inserting two or more mutation faults in two or more different base classes.
3) Dispersed Aspect Approach (AS): Each HOM is constructed by inserting two or more mutation faults in two or more different aspects.
4) Dispersed Base Class and Aspect Approach (BC&AS): Each HOM is constructed by inserting at least one fault in a base class and at least one fault in an aspect.

III. Approach

We first describe our approach for generating, compiling, and executing both FOMs and HOMs. We then introduce the objective function used by the three algorithms to measure the fitness of HOMs. Finally, we describe the search-based algorithms for finding subtle HOMs.

A. Higher Order Mutation Testing Tool

We developed a prototype tool for higher order mutation testing for Java and AspectJ programs. The tool automates the process of generating, compiling, and executing both first order and higher order mutants. The tool also implements the proposed search algorithms.

The tool takes as an input the program under test along with the test set. The tool then generates Java and AspectJ FOMs, and compiles and executes them against the given test set. Before the search process starts, the tool verifies that the given test set kills all the generated FOMs. Equivalent mutants need to be manually identified and removed. The tool then starts the search process based on the selected search algorithm seeking out FOM combinations that are not killed by the given test set. When the search process stops, the tool presents a list of subtle HOMs that were found during the search process. Below we describe the key features of the tool.

1) Generating FOMs: To generate FOMs, our tool uses the same approach and implementation described in previous work [9]. Our tool utilizes MuJava [17] to create base class FOMs and AjMutator [15] to create pointcut FOMs. To create advice and inter-type declaration FOMs, we adopted an approach similar to Wedyan and Ghosh [5]. Our tool uses Java Decompiler project [18] to decompile aspect `.class` files into `.java` files, and then invokes MuJava to mutate the resulting Java files.

Our tool produces XML files that provide complete information about all the generated FOMs. The XML files are called *FOM metadata*. Each XML record represents an FOM containing the applied mutation operator, the line number of the mutated statement, the Java method (method signature) containing the mutated statement, the class or aspect name, and the actual mutation (i.e., the original code fragment and the mutated code fragment).

2) Creating HOMs: HOMs are created by combining two or more FOMs from the FOM metadata. Once the corresponding FOM metadata records are selected, the HOM is created by copying all class and aspect files of the program, line by line, while replacing each line corresponding to a selected FOM with the mutated statement of the FOM (obtained from the actual mutation field in the FOM metadata record). In this study we only consider inserting one mutation fault per line of code.

3) Compiling and executing mutants and recording the results: Because mutated statements may be dispersed among different class and aspect files, it is necessary to keep a complete version of the program with all base class and aspect files for each HOM. The process of compiling and executing the mutants involves creating a folder for the subject program, creating each mutant as described in Section III-A2 one at a time, compiling and executing the mutant against the given test set, and recording the execution result in the output text files. The execution result for a mutant includes a list of identifiers of all the test cases that kill the mutant. A test case is said to kill a mutant if the test case fails when executed on the mutant.

During the search process, a particular HOM might be created more than once. This would require compiling and executing the same HOM against the same test set every time it is created. To avoid the cost of compiling and executing the same HOM more than once, the tool checks for duplicates.

B. Objective Function

The primary goal of our objective function is to identify subtle HOMs. Each subtle HOM represents an optimal solution and we want as many distinct optimal solutions as possible.

The objective function also needs to identify the HOMs that have a better chance of becoming subtle HOMs if an FOM is added or removed. We explain our notation below and provide a formal definition of the objective function.

- $F = \{f_1, \ldots, f_f\}$ is the set of all the FOMs for the program under test.
- T is the universe of all possible test cases.
- $T_s = \{t_1, \ldots, t_t\}$ is the set of all test cases under consideration (the given test set), $T_s \subseteq T$ and T_s kills all the FOMs in F.
- H is the space of all candidate HOMs. $H = \mathcal{P}(F)$
- $h_i^n \in H$ is an HOM constructed from n FOMs. The notation can be simplified to $h_i = h_i^n$ without confusion.
- T_i is a subset of T_s and contains all test cases that kill any of the n FOMs used to construct h_i.
- Let T_{h_i} denote the set of those test cases in T_s ($T_{h_i} \subset T_s$) that kill h_i. $T_{h_i} = \emptyset$ if none of the test cases in T_s kill h_i.

Objective Function

$$fitness(h_i) = \frac{|(T_i \cup T_{h_i})| - |(T_i \cap T_{h_i})|}{|T_i \cup T_{h_i}|} + subtlety(h_i) \quad (1)$$

Subtlety Function

$$subtlety(h_i) = \begin{cases} 1 & \text{if } T_{h_i} = \emptyset \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

To calculate the fitness of an HOM, our objective function first measures the fault detection difference between the HOM and its constituent FOMs (first term in Equation 1) and then measures its subtlety. The fault detection difference quantifies the variation between the set of test cases that kill the HOM and the union of all sets of test cases that kill the individual constituent FOMs. The subtlety value shows whether the HOM is killed by the given test set or not. The fitness value of an HOM lies between 0 and 2. Note that a solution which has an evaluation of 2 represents a global optimum in the search space, and there are potentially many globally optimal solutions. Our goal is not only to find a global optimum, but to find as many globally optimal solutions as possible. HOMs with higher fitness values are favored in the selection process. Each FOM has a zero fitness value. HOMs are classified based on their fitness values as follows:

1) Entirely Coupled HOMs: An HOM with a fitness value of 0 (worst value) is entirely coupled to its constituent FOMs. There is no difference between the set of test cases that kill the HOM and the union of all sets of test cases that kill the individual constituent FOMs. An entirely coupled HOM is considered to be useless because the single faults used to create the HOM do not interact to produce new faulty behavior.
2) Partially Coupled HOMs: An HOM with a fitness value greater than 0 but less than 1 is called partially coupled. There is a difference between the set of test cases that kill the HOM and the union of all sets of test cases that kill the individual constituent FOMs. A partially coupled HOM represents the case when some single faults interacted to mask each other and/or produced new faulty behavior. Partially coupled HOMs are considered to be more promising than entirely coupled HOMs and have the potential to develop into subtle HOMs when an FOM is added or removed.
3) Decoupled HOMs: An HOM with a fitness value of 1 is called a decoupled HOM. Such HOMs are killed by a set of test cases that is totally different from the union of all sets of test cases that kill the individual constituent FOMs. Decoupled HOMs are considered to be important because their constituent faults masked each other and produced new faulty behavior that could not be detected by the test sets that killed their individual constituent FOMs. However, the new faulty behavior was detected by the given test set. Decoupled HOMs are considered to be more promising than partially coupled and entirely coupled HOMs.
4) Subtle HOMs: An HOM with a fitness value of 2 is called a subtle HOM (an optimal solution). Such an HOM represents new faulty behavior that has not been tested because it was not killed by any test case in the given test set.

C. Genetic Algorithm

The Genetic Algorithm (GA) evolves sets of HOMs over a number of iterations allowing the survival of HOMs that are considered to be more promising to produce subtle HOMs. Below we describe the GA operators and discuss some implementation issues. The pseudocode is shown in Algorithm 1.

Algorithm 1 Genetic Algorithm

Require: $FOMsFile$, $crossPoint$, $popSize$, $mutationRate$, $eliteHOM$, $firstPopDegree$
1: $t \leftarrow 0$
2: $pop[t] \leftarrow$ createFirstPopulation($FOMsFile$, $popSize$, $firstPopDegree$)
3: $subtleHOMsList \leftarrow \emptyset$
4: **while** not (terminationCondition()) **do**
5: $pop[t+1]$.add($pop[t]$,$eliteHOM$)
6: **while** $pop[t+1]$.size $<$ $pop[t]$.size **do**
7: $parents \leftarrow$ Selection($pop[t]$)
8: $offspring \leftarrow$ Crossover($parents, crossPoint$)
9: $offspring \leftarrow$ Mutation($offspring, mutationRate$)
10: $pop[t+1]$.add($offspring$)
11: **end while**
12: $t \leftarrow t+1$
13: $pop[t]$.executeMutants()
14: $pop[t]$.calculateFitness()
15: $subtleHOMsList$.add($pop[t]$.getSubtleHOMs())
16: **end while**
17: **return** $subtleHOMsList$

1) *GA Inputs*: The GA takes as input the FOM metadata ($FOMsFile$) for the program under test,

number of crossover points ($crossPoint$), mutation rate ($mutationRate$), the population size ($popSize$), the degree of HOMs in the first population ($FirstPopDegree$), and the number of elite HOMs that get carried over at each iteration ($eliteHOM$).

2) *Chromosome Representation*: Each chromosome (HOM) is represented as a one dimensional array of strings such that each element in the array represents a line of code (one Java/AspectJ statement) of the program under test. All program statements (from all the classes and aspects) are included in the chromosome. This representation makes it easy to manipulate the program statements and to implement GA crossover and mutation operators.
3) *GA First Population*: Each HOM in the first population is created by combining a number of randomly selected FOMs from $FOMsFile$. All HOMs in the first population are of the same degree, which is configurable using the parameter, $FirstPopDegree$.
4) *Selection*: The selection process involves selecting HOMs that are allowed to produce offspring using crossover and mutation. The GA uses *tournament selection* to implement generational replacement of the population. Tournament selection selects four random HOMs and then selects the two HOMs with the highest fitness value to be parents. The two parents then produce offspring that are passed on to the next generation. The offspring replace the parents. However, a certain number (specified by $eliteHOM$) of HOMs with the highest fitness values in the current population are automatically carried over (copied) to the next generation. This "elitist" selection strategy ensures that copies of the best HOMs are not lost when moving from one generation to the next generation.
5) *Crossover*: The crossover happens between two selected HOMs (parents) that are recombined to produce two HOMs (offspring). The crossover depends on the number of crossover points, which could be one or many, based on the configurable parameter, $crossPoint$.
6) *Mutation*: The GA mutation operator applies mutation to an existing HOM by either adding or removing an FOM. The FOMs to be added to an HOM are randomly selected from the FOM metadata and the FOMs to be removed from an HOM are randomly selected from the HOM's constituent FOMs. The configurable parameter, $mutationRate$, determines how many FOMs are added to or removed from HOMs.
7) *Evaluating HOMs*: Evaluating HOMs in a population involves two steps. All HOMs in the population are created, compiled, and executed as described in Section III-A3. The objective function then evaluates each HOM and assigns a fitness value as shown in Equation 1.
8) *Stopping condition and output*: The GA maintains a list of distinct, subtle HOMs ($subtleHOMsList$) that were found during the search process. That list is returned to the tester after a stopping condition is met. The stopping condition is configurable by the tester. The GA can be stopped if the maximum number of generated HOMs is reached or the tester chooses to stop GA using the termination flag.

D. Local Search

The Local Search (LS) selects the most promising HOM at each iteration. It starts by selecting a random HOM (incumbent HOM) and then searches for the neighboring HOM that has the best fitness value. If no better HOM is found, LS restarts by selecting a new HOM. Below we describe LS and the pseudocode is shown in Algorithm 2.

Algorithm 2 Local Search

Require: $FOMsFile$, $incumbentHOMDegree$
1: $shouldRestart \leftarrow true$
2: $subtleHOMsList \leftarrow \emptyset$
3: **while** not (terminationCondition()) **do**
4: **if** $shouldRestart$ **then**
5: $incumbentHOM \leftarrow$ generateHOM($FOMsFile$, $incumbentHOMDegree$)
6: executeMutant($incumbentHOM$)
7: evaluateFitness($incumbentHOM$)
8: **end if**
9: $neighborsList \leftarrow$ generateNeighbors($incumbentHOM$)
10: executeMutants($neighborsList$)
11: evaluateFitnesses($neighborsList$)
12: **if** thereIsAbetterNeighbor($incumbentHOM$, $neighborsList$) **then**
13: $incumbentHOM \leftarrow$ getBestHOM($neighborsList$)
14: $shouldRestart \leftarrow false$
15: **else**
16: $shouldRestart \leftarrow true$
17: **end if**
18: $subtleHOMsList$.add(getSubtleHOMs($neighborsList$))
19: **end while**
20: **return** $subtleHOMsList$

1) *LS Inputs*: LS takes as input the FOM metadata file ($FOMsFile$) of the program under test and the degree of the incumbent HOM ($incumbentHOMDegree$).
2) *LS starting point*: LS starts by generating an incumbent HOM using a number of randomly selected FOMs from the list of FOMs. The incumbent HOM degree is set using the value of the parameter, $incumbentHOMDegree$. The algorithm then evaluates the incumbent HOM by compiling and executing it against the test set, records the execution result, and calculates the fitness value using Equation 1.
3) *Generating neighboring HOMs*: After the incumbent HOM is generated and evaluated, LS generates all the HOMs neighboring the incumbent HOM. The neighboring HOMs are those that vary by one FOM (one step) from the incumbent HOM. The neighboring HOMs are maintained in a list called $neighborsList$. The neighboring HOMs can be formally defined as follows. Let $F = \{f_1, \ldots, f_f\}$ denote the set of all FOMs for the program under test. H is the space of all candidate HOMs, $H = \mathcal{P}(F)$. h_i^n is an HOM

created from n number of FOMs, such that $n \geq 2$ and $h_i^n \in H$. Let h_j^m be a neighbor of h_i^n such that $h_j^m \in H$ and $m = n$, or $m = n + 1$, or $m = n - 1$; furthermore, one of the following conditions holds:

a) $h_i^n \subset h_j^{n+1}$

b) $h_j^{n-1} \subset h_i^n$

c) $|h_i^n - h_j^n| = 1$ and $|h_j^n - h_i^n| = 1$

Figure 1 shows an example of each of the three cases with $n = 3$.

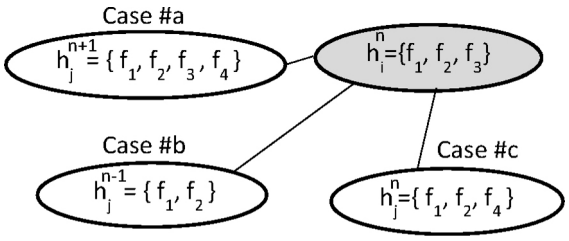


Fig. 1. Example of neighboring HOMs

4) *Evaluating neighboring HOMs*: Each HOM in the $neighborsList$ is created, compiled, and executed against the test set, its execution result is recorded, and its fitness value is assigned using Equation 1.
5) *Next iteration*: After the HOMs in the $neighborsList$ are evaluated, LS looks for the best neighboring HOM that has an equal or higher fitness value than the incumbent HOM. If such an HOM exists, it becomes the next incumbent HOM, and the process starts all over again. If no better HOM exists, the algorithm restarts by selecting a new incumbent HOM.
6) *Stopping condition and output*: LS maintains a list of all distinct, subtle HOMs ($subtleHOMsList$) that were found during the search process. The algorithm returns that list after it stops executing. LS stops when the maximum number of generated HOMs is reached or the tester chooses to stop the algorithm at any point.

E. Random Search

Random Search (RS) explores the space of all candidate HOMs by randomly selecting HOMs, one at a time, seeking out the HOMs that are not killed by the given test set. The algorithm iterates the process of generating an HOM ($randomHOM$) with a set of randomly selected FOMs from the FOM metadata ($FOMsFile$). The degree of $randomHOM$ is controlled by the maximum HOM degree allowed ($maxHOMDegree$).

Each generated $randomHOM$ is compiled and executed, its execution result is recorded, and its fitness value is calculated. Subtle HOMs are stored in the $subtleHOMsList$. RS repeats this process until a stopping condition is reached. The stopping condition is that either the required number of HOMs has been generated or the tester has chosen to stop the algorithm. The pseudocode for RS is provided in Algorithm 3.

Algorithm 3 Random Search

Require: $FOMsFile$, $maxHOMDegree$
1: $subtleHOMsList \leftarrow \emptyset$
2: **while** not (terminationCondition()) **do**
3: $randomHOM \leftarrow$ generateHOM($FOMsFile$, $maxHOMDegree$)
4: executeMutant($randomHOM$)
5: evaluateFitness($randomHOM$)
6: **if** isAsubtleHOM($randomHOM$) **then**
7: $subtleHOMsList$.add($randomHOM$)
8: **end if**
9: **end while**
10: **return** $subtleHOMsList$

IV. Experiment Goals and Set Up

This section describes the goals of the experimental evaluation of the algorithms.

A. Experiment Goals

We conducted an experiment to answer the following research questions (RQ):

- *RQ1: Can the proposed algorithms find subtle HOMs?* The goal is to evaluate the ability of the algorithms in terms of finding optimal solutions. We also report the number of equivalent mutants generated for each benchmark and each algorithm.
- *RQ2: How do the proposed algorithms compare in terms of the number of subtle HOMs found?* We compare the relative effectiveness of the proposed algorithms in terms of their ability to find non-equivalent, subtle HOMs.
- *RQ3: What are the differences between the HOMs produced using different algorithms?* To understand in what ways the search algorithms explored the space of all candidate HOMs and how that affected the discovery of subtle HOMs, we investigate the number of generated HOMs and subtle HOMs with respect to the degree of the HOMs as well as the construction approach used to create the HOMs.

B. Subject Programs

We used four AspectJ programs of different sizes. The programs implement various AspectJ constructs. They contain before, after, and around advices, inter-type declarations, as well as primitive and composed pointcuts. The programs also have base classes that contain Java constructs to which we could apply MuJava operators. Table I provides information about the subject programs, their sizes, and test sets.

The *Banking* program [19] is a bank account management system that contains two classes, *Customer* and *Account*, and two aspects, *MinimumBalance* and *OverdraftProtection*. The aspects are responsible for checking the balance and controlling the overdraft fee.

The *Movie Rental* program[1] contains three classes, *Movie*, *Customer*, and *Rental*, and one aspect, *Updates*, which implements functionality to enable the addition of new movie classifications and pricing strategies. Since there is only one aspect, the AS approach cannot be applied to create HOMs.

[1]Downloadable from http://www.cs.colostate.edu/~elman/Aspects.html

TABLE I. SUBJECT PROGRAM AND TEST SET CHARACTERISTICS

Subject program	LOC	# of classes	# of aspects	# of test sets	# of test cases per test set	# of advices	# of point-cuts	# of ITDs
Banking	243	2	2	3	15	2	2	1
Kettle	125	1	2	3	18	4	3	2
Movie Rental	191	3	1	3	26	8	9	0
Telecom	928	10	3	3	30	9	12	9

The *Kettle* application [20] simulates the functionality of an electric kettle for heating water. It contains one class, *Kettle*, and two aspects, *HeatControl* and *SafetyControl*, to optimize the power consumption and temperature control of the kettle. Since there is only one class, the BC approach cannot be applied to create HOMs.

The *Telecom* program [11] simulates a telephone system. It allows customers to make, accept, merge, and hang up both local and long distance calls. *Telecom* contains three aspects, *Timing*, *Billing*, and *TimerLog* and ten classes, *AbstractSimulation*, *BasicSimulation*, *BillingSimulation*, *Call*, *Connection*, *Customer*, *Local*, *LongDistance*, *Timer*, and *TimingSimulation*.

C. Test Sets

We developed a large pool of JUnit test cases for each subject program. We developed our own random test case generator for each subject program because RANDOOP [21] does not support AspectJ. Each random test case contains code that exercises the class constructors and methods in a random sequence and creates assertions on the execution results. The size of the generated test cases randomly varied from two to 20 method calls per test case. We also used RANDOOP to generate even more test cases. In order to make RANDOOP work with AspectJ, we rewrote the aspects using AspectJ annotation style and placed the intertype methods inside interfaces implemented by the corresponding Java classes. The test cases generated by RANDOOP represent less than 10% of the set of all test cases for each subject program.

For each subject program, the pool of JUnit test cases achieved statement coverage, coverage of equivalence classes and boundary values of the input domain, and killed all non-equivalent FOMs.

For each subject program, we generated three test sets of the same size by randomly selecting test cases from the large pool of JUnit test cases. Each test set contains test cases that achieved statement coverage and killed all FOMs. Table I shows information about the subject programs and the associated test sets.

D. Experimental Configuration

We used a standard office personal computer with Intel Core(TM)2 1.86GHz to run the experiment. For each subject program, we ran each algorithm 10 times with the stopping condition of generating 50,000 HOMs. This process was repeated three times for each subject program using a different test set each time. We obtained 4.5 million HOMs for each subject program using the three algorithms with three test sets. The data reported in Section V is based on the analysis of all generated HOMs. To generate and evaluate 50,000 HOMs, it took RS 13.7 seconds, LS 16.8 seconds, and GA 22.5 seconds. The times do not include compilation and execution times.

The configuration of each algorithm can affect its performance. In the early stages of this study we ran the GA and LS with different configurations and finally selected the configurations that produced the highest number of subtle HOMs.

The GA was configured as follows: $crossPoint = 2$, $mutationRate = 3$, $eliteHOM = 15$, $popSize = 300$, and $FirstPopDegree = 2$. LS was configured with the incumbent HOM degree as 2. For RS, the maximum HOM degree was set at 25.

V. RESULTS AND ANALYSIS

Below we present the answers to each research question.

A. RQ1: Can the proposed algorithms find subtle HOMs?

TABLE II. FOMS, GENERATED HOMS, AND SUBTLE HOMS FOR EACH SUBJECT PROGRAM

Programs	# FOMs	# Generated HOMs	# Subtle HOMs
Banking	92	4,500,000	57
Kettle	125	4,500,000	81
Movie	316	4,500,000	30
Telecom	188	4,500,000	48

Table II shows for each subject program the number of FOMs, the number of generated HOMs, and the number of distinct, subtle HOMs. The algorithms successfully found subtle HOMs for each program. In some cases the number of subtle HOMs found was more than half the number of FOMs of that program. For example, for the Kettle program, 81 subtle HOMs were found that were not killed by test sets that killed the 125 FOMs. We checked that the subtle HOMs were not equivalent and that new test cases can be developed to kill them.

Some of the subtle HOMs in Table II can be killed by test cases contained in the large pool of JUnit test cases that we developed for each program. However, the majority of produced subtle HOMs required developing test cases with specific input values and sequences of method calls that did not exist in the large pool of the test cases.

TABLE III. EQUIVALENT HOMS PRODUCED BY THE SEARCH ALGORITHMS

Programs	Genetic Algorithm (GA)	Local Search (LS)	Random Search (RS)
Banking	16	15	11
Kettle	7	8	3
Movie	6	14	7
Telecom	48	59	26
Total	77	96	47

Table III shows the number of equivalent HOMs produced by each algorithm for the four subject programs. Although the FOM metadata did not include any equivalent FOMs, the algorithms generated equivalent HOMs. Equivalent HOMs

were manually identified and removed after the algorithms were stopped and they are not included in the count of subtle HOMs. Equivalent HOMs result when two or more faulty statements interact to eliminate the effect of their faults. For example, suppose that the test case assertion is based on the variable, `customerPayments`, in the two consecutive statements shown below:

```
customerAccount -= payment;
customerPayments += payment;
```

When these two statements are mutated as follows, the second order mutant is equivalent to the original program.

```
customerAccount -= payment++;
customerPayments += --payment;
```

GA and LS produced more equivalent HOMs than Random Search. This is because the selection mechanism in both GA and LS favors HOMs with higher fitness values and the equivalent HOMs were assigned the highest fitness value of 2 because they were not killed by any test case. In other words, they were treated as if they were subtle HOMs.

B. RQ2: How do the proposed algorithms compare in terms of the number of subtle HOMs found?

For each subject program, we ran each algorithm 10 times for each of the three test suites. We then calculated the maximum, average, median, and minimum number of subtle HOMs found per run for all 30 runs as shown in Table IV.

TABLE IV. SUBTLE HOMS PRODUCED BY DIFFERENT SEARCH ALGORITHMS

Programs	Algorithm	# HOMs	# Subtle HOMs			
			Max	Avg	Med	Min
Banking	Genetic (GA)	50,000	25	13	19	1
	Local (LS)	50,000	31	22.2	21	16
	Random (RS)	50,000	8	4	3	0
Kettle	Genetic (GA)	50,000	18	12.3	13	7
	Local (LS)	50,000	35	18	17	7
	Random (RS)	50,000	6	2.2	2	0
Movie	Genetic (GA)	50,000	11	3	2	0
	Local (LS)	50,000	19	8	7	0
	Random (RS)	50,000	3	1	1	0
Telecom	Genetic (GA)	50,000	14	8.3	7	5
	Local (LS)	50,000	17	5.7	4	1
	Random (RS)	50,000	6	3.4	5	0

Both GA and LS produced a higher number of subtle HOMs than RS but LS was the most successful in finding subtle HOMs for all subject programs. RS may have produced better results for certain other configurations of the algorithms. In this study, RS is used as a baseline to compare the results. In the next research question, we investigate different factors that explain the results.

C. RQ3: What are the differences between the HOMs produced using different algorithms?

Figure 2 shows the distribution of the distinct, subtle HOMs and all generated HOMs according to their degree. As Figure 2(a) shows, the majority of subtle HOMs produced by all three algorithms are of lower degree (second and third). This is because, in general, increasing the degree of an HOM by adding more FOMs just makes it easier to kill. However, subtle HOMs of up to degree seven were produced. LS was more successful in finding subtle HOMs of lower degrees but the GA was more successful in finding subtle HOMs of higher degrees.

As Figure 2(b) shows, the majority of HOMs generated by LS are of second and third degrees, and this explains why LS was more successful in finding a higher number of subtle HOMs of lower degrees. Because the parameter *incumbentHOMdegree* was set at two, it became harder for LS to move to the space of higher degree HOMs. LS selects the HOM with the best fitness value at each iteration and generally, higher degree HOMs usually have lower fitness values because they are easier to kill. However, LS still generated a small number of HOMs of up to degree six. Configuring the *incumbentHOMdegree* parameter differently is likely to produce different results.

Although the objective function of the GA favors HOMs of lower degrees, GA managed to generate more higher degree HOMs (ten and higher) than LS, and thus, GA was more successful in finding subtle HOMs of higher degrees.

RS generated a uniform number of HOMs of all considered degrees but only produced subtle HOMs of lower degrees (up to degree four). Since RS uniformly sampled HOMs of all degrees (up to 25) in an unbiased fashion, this strongly suggests that it is easier to find subtle HOMs of lower degrees.

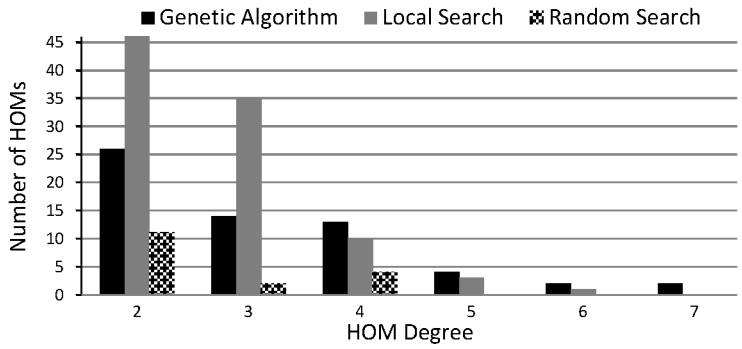
Figure 3 shows the number of distinct, subtle HOMs and all generated HOMs found by each algorithm using the four construction approaches. Data for the Telecom and Banking programs was used because only these two programs have a sufficient number of base classes and aspects to enable the use of all four approaches.

As Figure 3(a) shows, subtle HOMs were created by all four construction approaches. However, the majority of the distinct, subtle HOMs were a result of the construction approaches BC&AS and SCA. BC&AS HOMs are created by inserting multiple faults where at least one is in a base class and at least one in an aspect. SCA HOMs are created by inserting multiple faults in a single base class or in a single aspect. The BC&AS HOMs seem to be the most predominant construction approach of HOMs by all algorithms. However, RS produced even more HOMs using BC&AS than the other two algorithms because of the high maximum HOM degree allowed ($maxHOMDegree$), which was set at 25. Higher degree HOMs are more likely to be of type BC&AS.

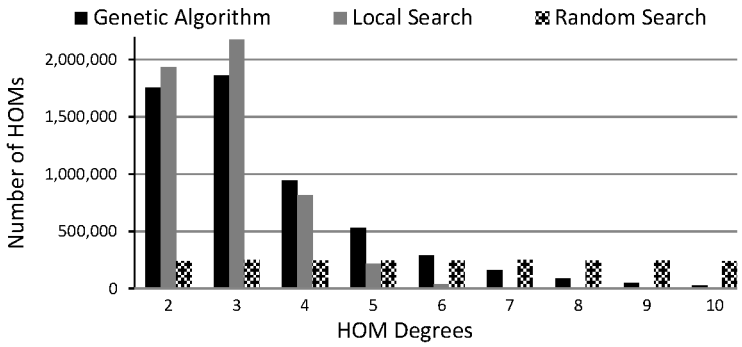
Although all three algorithms generated a much higher number of BC&AS HOMs than SCA HOMs (see Figure 3(b)), the number of subtle BC&AS HOMs is comparable to the number of subtle SCA HOMs for the three algorithms. For example, RS, which is the most unbiased algorithm, produced 9 BC&AS subtle HOMs and 7 SCA subtle HOMs while it generated mostly BC&AS HOMs. Further investigation revealed that most of the distinct, subtle HOMs were those where multiple faults are inserted in class methods that call each other, or in class methods and their woven aspect advices.

D. Threats to Validity

One threat to external validity is that the four studied programs may not be representative of AspectJ programs

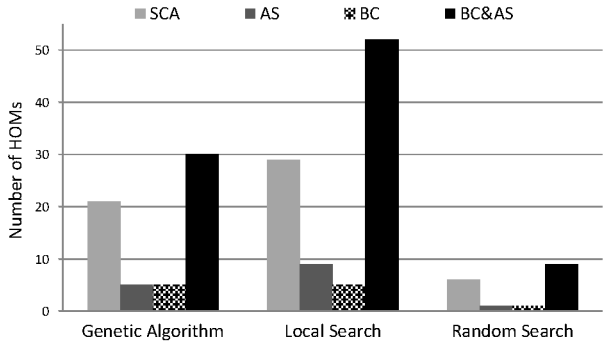


(a) Distinct Subtle HOMs

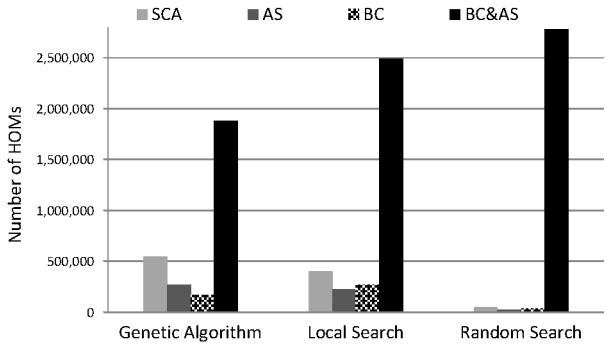


(b) All Generated HOMs

Fig. 2. Distribution of HOMs According to Degree for the Search Algorithms. Random Search continued to find a similar number of HOMs at all levels up to $maxHOMdegree = 25$.



(a) Distinct Subtle HOMs



(b) All Generated HOMs

Fig. 3. Distribution of HOMs According to Construction Approaches for the Search Algorithms

in general, and thus, the results of the study may not be generalizable to all AspectJ programs. However, the selected programs use many of the constructs of AspectJ.

A threat to internal validity stems from the configurations of the algorithms. Different configurations can lead to different results. For example, since all of the subtle HOMs are of degree 7 or less, Random Search might be reconfigured so that $maxHOMdegree = 7$. However, the goal of the study is to find subtle HOMs and we wanted to use Random Search with a uniform unbiased distribution to make sure we did not miss easy-to-find HOMS of higher degrees. The implications of the different operators and their configurations are left for future work.

A threat to internal validity stems from the quality of the test sets and the number of test cases used. The test sets achieve statement coverage and kill all the FOMs, but there is a possibility of getting different results if we use test cases created with different test objectives.

VI. Related Work

Researchers have proposed new fault models for AspectJ programs and described new testing challenges [5], [12]. Researchers have also explored first order mutation testing techniques in the context of AOP [5], [13], [13]–[16]. However, more work is needed to explore the benefits and challenges of higher order mutation testing in the context of AOP [5], [9].

Jia and Harman [1] conducted an experimental study using three search based optimization approaches (Genetic Algorithm, Greedy Algorithm, and Hill Climbing Algorithm) to find *Strongly Subsuming HOMs* in C programs. The results indicated that Genetic Algorithm performed best at finding the strongly subsuming HOMs because the subsuming HOMs are easier to generate from existing subsuming HOMs.

Langdon et al. [3] used the multi-objective Pareto optimal approach and Monte Carlo sampling Genetic Algorithms to search for HOMs that are hard-to-kill and are syntactically similar to the original program. The aim of the study was to investigate the relationship between the mutants' syntax and semantics. The search algorithm produced HOMs that were harder-to-kill than their constituent FOMs and also HOMs that were constructed from single faults masking each other.

Anbalagan et al. [16] used mutation testing to test the strength of pointcuts. Their work involved developing a framework that generates relevant pointcut mutants and detects equivalent mutants.

Delamare et al. [15] presented the AjMutator tool that implements several mutation operators (proposed by Ferrari [14]) to generate pointcut mutants. The tool classifies and compiles the mutants in addition to executing the test cases on the mutants.

Ferrari et al. [13] presented the Proteum/AJ tool for mutation testing of AspectJ programs. The tool automates a set of aspect-oriented mutation operators proposed in their previous work [14] and supports the basic steps of mutation testing. Proteum/AJ included 24 mutation operators, 15 of which are pointcut related operators.

Polo [10] presented three algorithms for constructing second order mutants (SOMs) from FOMs: LastToFirst, Different Operators, and Random Mix. The LastToFirst algorithm constructs SOMs by combining the first mutant in the FOMs list with the last, the second with the previous and so on. The Different Operators algorithm combines FOMs resulting from different mutation operators. The Random Mix algorithm randomly combines FOMs using each mutant once.

DiGiuseppe and Jones [22] presented a study of the effects of the interaction of different faults within a program. The goal was to reveal the nature of fault interaction in programs with different numbers of faults. Their results showed that multiple real faults could hide the impact of each other leading to fewer failures than expected. They also reported that fault obfuscation was the most prominent of all fault interaction types.

Debroy and Wong [23] presented a study of the implication of fault interference. They examined the status of passing and failing test cases as more faults were added to the program. Based on the principles of wave theory, they classified fault interference into four different types. Their results showed that failure masking was more frequent in the programs they studied.

VII. Conclusions and Future Work

We presented three search-based algorithms for finding subtle HOMs that can help testers improve fault detection effectiveness of test sets. The algorithms used a new objective function to identify subtle HOMs and to recognize the HOMs that are considered promising for developing into subtle HOMs.

All three algorithms found subtle HOMs. Random Search was able to find subtle HOMs of lower (second or third) degrees, which indicates that subtle HOMs of lower degrees can be relatively easy to discover. The majority of the produced subtle HOMs were of lower degrees and Local Search was the most successful overall in finding them. However, the Genetic Algorithm found subtle HOMs of higher degrees.

All four construction approaches produced subtle HOMs. However, the majority of subtle HOMs resulted from inserting multiple faults in the same base class or the same aspect, or from inserting multiple faults in base classes and aspects.

In the future, we will further analyze the HOMs produced by each algorithm and develop heuristics that can more effectively guide the search algorithms in finding subtle HOMs. We will also study characteristics of FOMs to find out which FOM combinations are more likely to produce subtle HOMs.

Further research is needed to investigate the implications of the configurations of the search algorithms. Larger subject programs and test sets need to be used in the evaluations. Finally, the impact of subtle HOMs on test generation techniques needs to be studied.

References

[1] Y. Jia and M. Harman, "Higher order mutation testing," *Information & Software Technology*, vol. 51, no. 10, 2009.

[2] ——, "Constructing subtle faults using higher order mutation testing," in *International Working Conference on Source Code Analysis and Manipulation*, 2008, pp. 249–258.

[3] W. B. Langdon, M. Harman, and Y. Jia, "Efficient multi-objective higher order mutation testing with genetic programming," *Journal of Systems and Software*, vol. 83, no. 12, pp. 2416–2430, 2010.

[4] M. Papadakis and N. Malevris, "An empirical evaluation of the first and second order mutation testing strategies," in *International Conference on Software testing, verification, and validation workshops*. IEEE, 2010, pp. 90–99.

[5] F. Wedyan and S. Ghosh, "On generating mutants for AspectJ programs," *Information and Software Technology*, vol. 54, no. 8, pp. 900–914, 2012.

[6] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 212–222.

[7] J. Offutt, "Investigations of the software testing coupling effect," *Transactions on Software Engineering and Methodology*, vol. 1, no. 1, pp. 5–20, 1992.

[8] R. Purushothaman and D. E. Perry, "Toward understanding the rhetoric of small source code changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 511–526, 2005.

[9] E. Omar and S. Ghosh, "An exploratory study of higher order mutation testing in aspect-oriented programming," in *International Symposium on Software Reliability Engineering*, 2012, pp. 1–10.

[10] M. Polo, M. Piattini, and I. G. R. de Guzmán, "Decreasing the cost of mutation testing with second-order mutants," *Software Testing, Verification and Reliability*, vol. 19, no. 2, pp. 111–131, 2009.

[11] The AspectJ Team, "AspectJ Compiler 1.5.4," http://www.eclipse.org/aspectj/, October 2008.

[12] J. S. Baekken and R. T. Alexander, "A candidate fault model for AspectJ pointcuts," in *International Symposium on Software Reliability Engineering*, 2006, pp. 169–178.

[13] F. Ferrari, E. Nakagawa, A. Rashid, and J. Maldonado, "Automating the mutation testing of aspect-oriented Java programs," in *Workshop on Automation of Software Test*, 2010, pp. 51–58.

[14] F. C. Ferrari, J. C. Maldonado, and A. Rashid, "Mutation Testing for Aspect-Oriented Programs," in *International Conference on Software Testing, Verification, and Validation*, 2008, pp. 52–61.

[15] R. Delamare, B. Baudry, and Y. Le Traon, " AjMutator: A Tool for the Mutation Analysis of AspectJ Pointcut Descriptors," in *International Conference on Software Testing, Verification, and Validation Workshops*, 2009, pp. 200–204.

[16] P. Anbalagan and T. Xie, "Automated generation of pointcut mutants for testing pointcuts in AspectJ programs," in *International Symposium on Software Reliability Engineering*, 2008, pp. 239–248.

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

[18] S. Forge, "The Java Decompiler project," http://dcompiler.sourceforge.net/, October 2002.

[19] R. Laddad, *AspectJ in action*. Manning Publications Co, 2003, vol. 512.

[20] F. Wedyan and S. Ghosh, "A dataflow testing approach for aspect-oriented programs," in *IEEE International Symposium on High-Assurance Systems Engineering*, 2010, pp. 64–73.

[21] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *International Conference on Software Engineering*, 2007, pp. 75–84.

[22] N. DiGiuseppe and J. Jones, "Fault interaction and its repercussions," in *International Conference on Software Maintenance*, 2011, pp. 3–12.

[23] V. Debroy and W. E. Wong, "Insights on fault interference for programs with multiple bugs." in *International Symposium on Software Reliability Engineering*, 2009, pp. 165–174.