DISSERTATION

CONSTRUCTING SUBTLE HIGHER ORDER MUTANTS FROM JAVA AND ASPECTJ PROGRAMS

Submitted by

Elmahdi Omar

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 2015

Doctoral Committee:

Advisor: Sudipto Ghosh
Co-Advisor: Darrell Whitley

James M. Bieman
Daniel E Turk

Copyright by Elmahdi Omar 2015

All Rights Reserved

ABSTRACT

CONSTRUCTING SUBTLE HIGHER ORDER MUTANTS FROM JAVA AND ASPECTJ PROGRAMS

Mutation testing is a fault-based testing technique that helps testers measure and improve the fault-detection effectiveness of their test suites. However, a majority of traditional First Order Mutants (FOMs), which are created by making a single syntactic change to the source code, represent trivial faults that are often easily detected (i.e. killed). Research has shown that the majority of real faults not detected during testing are complex faults that cannot be simulated with FOMs because fixing these faults requires making multiple changes to the source code. Higher Order Mutants (HOMs), which are created by making multiple syntactic changes to the source code, can be used to simulate such faults.

The majority of HOMs of a given program are killed by any test suite that kills all the FOMs. We refer to HOMs that are not killed as *subtle HOMs*. They represent cases where single faults interact by masking each other with respect to the given test suite and produce complex faulty behavior that cannot be simulated with FOMs. The fault-detection effectiveness of the given test suite can be improved by adding test cases that detect the faults denoted by subtle HOMs.

Because subtle HOMs are rare in the exponentially large space of candidate HOMs, the cost of finding them can be high even for small programs. A brute force approach that evaluates every HOM in the search space by constructing, compiling, and executing the HOM against the given test suite is unrealistic.

We developed a set of search techniques for finding subtle HOMs in the context of Java and AspectJ programming languages. We chose Java because of its popularity, and the availability of experimental tools and open source programs. We chose AspectJ because of its unique concepts and constructs and their consequent testing challenges.

We developed four search-based software engineering techniques: (1) Genetic Algorithm, (2) Local Search, (3) Test-Case Guided Local Search, (4) Data-Interaction Guided Local Search. We also developed a Restricted Random Search technique and a Restricted Enumeration Search technique. Each search technique explores the search space in a different way and that affects

the type of subtle HOMs that can be found by each technique. Each of the guided local search techniques uses a heuristic to improve the ability of Local Search to find subtle HOMs.

Due to the unavailability of higher order mutation testing tools for AspectJ and Java programs, we developed HOMAJ, a Higher Order Mutation Testing tool for AspectJ and Java programs for finding subtle HOMs. HOMAJ implements the developed search techniques and automates the process of creating, compiling, and executing both FOMs and HOMs.

The results of our empirical studies show that all of the search techniques were able to find subtle HOMs. However, Local Search and both the Guided Local Search techniques were more effective than the other techniques in terms of their ability to find subtle HOMs.

The search techniques found more subtle HOMs by combining faults created by primitive Java mutation operators than by combining faults created by Java class level operators and AspectJ operators.

Composing subtle HOMs of lower degrees generated by Restricted Enumeration Search is an effective way to find new subtle HOMs of higher degrees because such HOMs are likely to exist as compositions of multiple subtle HOMs of lower degrees. However, the search-based software engineering techniques were able to find subtle HOMs of higher degrees that could not be found by combining subtle HOMs of lower degrees.

ACKNOWLEDGEMENTS

My great gratitude to my advisers, Dr. Sudipto Ghosh and Dr. Darrell Whitley, for their continuous guidance and support. I am thankful to them for their valuable input on how to improve my work and research skills. A big thanks to both of them for pushing me further and inspiring me to be a better thinker and writer.

My appreciation also goes to my advisory committee members, Dr. James Bieman and Dr. Dan Turk, for taking the time to give me valuable feedback. I would like to thank the entire Computer Science Department staff for being available and supportive.

A special thanks to my wife and son for being patient, understanding, and supportive, to my parents for their continuous support and encouragement, and to all my friends for their support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Developing test cases that are effective at revealing hard-to-find faults is essential for creating reliable software applications. However, due to the growing complexity of software, developing test cases that can expose faults is becoming an even more challenging task. Testers generally rely on software specifications, documentation, and source code to develop their test cases, but this is usually not sufficient to deliver a correct and reliable software application [1]. Therefore, researchers have developed systematic testing techniques, such as mutation testing, to help testers measure and improve the ability of their test cases to detect faults [2, 3, 4, 5].

Mutation testing is a fault-based testing technique that involves creating faulty versions of the program under test and then measuring the ability of the given test suite to distinguish each faulty version from the original program. The faulty versions of the program are called *mutants*. Each mutant is created by making a single syntactic change to the original program. The syntactic changes are based on *mutation operators*, which simulate typical programming errors. A mutant that produces a different output than the original program is said to be killed, which means that the syntactic change (i.e. mutation fault) was detected by the given test suite. The *mutation score* is the ratio of the number of killed mutants over the total number of non-equivalent mutants. It predicts the ability of the given test suite to detect faults.

Equivalent mutants are those that are semantically identical to the original program and cannot be killed by any test case. The non-equivalent mutants that are not killed help testers generate new test inputs that improve the fault-detection effectiveness of the given test suite.

1.1 Problem Description

Mutation testing has been shown to be effective for improving the quality of test suites [6, 7]. However, a majority of traditional First Order Mutants (FOMs), which are created by making a single syntactic change to the source code, represent trivial faults that are often easily detected [8].

Purushothaman and Perry [9] showed that the majority of real faults not detected during testing are complex faults that cannot be simulated using FOMs. Their study reported that fixing 90% of these complex faults required making multiple changes to the source code. Moreover, Gopinath et al. [10] investigated real faults in 6,000 programs in four different programming languages and reported that a typical real fault involves about three to four tokens. They also reported the large majority of typical faults cannot be simulated with a single traditional mutation operator. Therefore, there is a need for techniques that can produce mutants that can simulate real complex faults.

Higher Order Mutants (HOMs), which are created by making multiple syntactic changes to the source code (i.e. combining multiple FOMs), can simulate real complex faults. Wedyan and Ghosh [11] used HOMs to produce mutants that cover all Aspect-Oriented Programming [12] fault types. Researchers have used HOMs to reduce the cost of mutation testing. Jia and Harman [8, 13, 14] developed techniques to identify HOMs that can replace their constituent FOMs without loss of test effectiveness. The goal of such HOMs, which are referred to as *strongly subsuming HOMs*, is to reduce the total number of FOMs that need to compiled and executed, thereby reducing test execution cost. Kintis et al. [15] used second order mutants to detect equivalent FOMs and to reduce the total number of equivalent FOMs that need to be inspected by testers.

We introduced the notion of *subtle HOMs* [16, 17, 18], which denote non-trivial and complex faults that are not detected by the given test suite that kills all the non-equivalent FOMs for the program under test. Subtle HOMs represent HOMs whose constituent FOMs interact by masking each other to produce new faulty behavior that cannot be simulated using all the FOMs for the program under test separately.

The mutation faults simulate typical programming errors and research has shown that mutation faults can be representative of real faults [19]. Subtle HOMs can simulate complex real faults. The fault-detection effectiveness of the given test suite can be improved by adding test cases that kill subtle HOMs.

The cost of finding subtle HOMs can be high even for small programs [16]. The number of HOMs is exponentially large because each HOM is created by combining two or more FOMs,

which result in a combinatorial explosion in the number of HOMs that can be created. Subtle HOMs are rare because a large majority of the HOMs are killed by any test suite that kills all the FOMs for the program under test; this is known as the *coupling effect* [20, 21].

Another factor that adds to the cost of finding subtle HOMs is the high computational cost of evaluating mutants. The process of evaluating mutants involves the compilation and execution of each mutant against the given test suite to determine whether the mutant is killed by the given test suite or not. Using a brute force approach to explore the large space of HOMs, which requires evaluating all the HOMs, is unrealistic. Therefore, there is a need for search techniques that can effectively explore the search space and find subtle HOMs.

Researchers used Search-Based Software Engineering (SBSE) techniques to solve various software engineering problems [22]. Harman and Jones [23] reported that 59% of the published SBSE techniques targeted software testing problems, such as the automation of test input generation [24] and bug fixing [25]. Search-Based Software Engineering techniques can be used to find subtle HOMs.

Jia and Harman [8] used search-based software engineering techniques to explore the space of all HOMs and find strongly subsuming HOMs in the context of the C programming language. Because strongly subsuming HOMs are defined as those that are killed by a subset of the union of the sets of test cases that kill the constituent FOMs, Jia and Harman defined their fitness measure such that it favors HOMs that are killed by fewer test cases than their constituent FOMs.

The fitness measure defined by Jia and Harman [8] is not sufficient for finding subtle HOMs. This is because subtle HOMs are defined as those that their constituent FOMs can interact to produce new faulty behavior that cannot be detected by the given test set. Therefore, a fitness measure for finding subtle HOMs should favor HOMs with high level of interaction amongst their constituent FOMs. A new faulty behavior results when multiple faults interact to mask each other with respect to the given test set. Nonetheless, a fitness measure for finding subtle HOMs should also favor HOMs that are killed by fewer test cases than their constituent FOMs. Subtle HOMs by definition are harder to kill than their constituent FOMs.

1.2 Approach and Contributions

In this dissertation we introduce the notion of subtle HOMs and developed a set of search techniques for finding subtle HOMs in the context of Java and AspectJ programming languages. We chose Java because of its popularity, and the availability of experimental tools and open source programs. We chose AspectJ because of the unique concepts and introduced by the Aspect-Oriented Programming (AOP) paradigm. The unique constructs cause new types of interactions between the program elements, which result in new testing challenges [11, 26]. The main contributions of this dissertation are as follows.

1- Search techniques for finding subtle HOMs

We developed three types of search techniques for finding subtle HOMs: (1) search-based software engineering techniques, (2) a restricted random search technique, and (3) a restricted enumeration search technique. Each search technique explores the search space in a different way.

We developed an objective function that provides a metric to measure the fitness of HOMs. The objective function uses information about the sets of test cases that kill the HOM and those that kill its constituent FOMs to calculate the fitness value of the HOM. The objective function is designed such that it favors HOMs with high level of interaction amongst their constituent FOMs as well as those HOMs that are killed by fewer test cases than their constituent FOMs. The developed search techniques use the objective function to identify subtle HOMs as well as HOMs that have the potential to develop into subtle HOMs when the right FOMs are added or removed.

The search-based software engineering techniques are a Genetic Algorithm, Local Search, and two Guided Local Search techniques. The Genetic Algorithm evolves a set of HOMs over a number of iterations using combinatorial operators, *crossover* and *mutation*. The Genetic Algorithm uses a *selection* operator to favor HOMs with better fitness values for reproduction.

Local Search starts with an arbitrary solution and iteratively improves it by searching the neighborhood for HOMs with better fitness values. Local Search uses a *neighborhood graph* to determine the neighborhood of an HOM.

An initial experimental evaluation [16, 17, 18] of the Genetic Algorithm and Local Search provided insights into the search space and enabled us to identify ways in which to improve the ability of the search techniques to find subtle HOMs. The evaluation showed that certain FOM combinations were more likely to produce subtle HOMs than others.

Because Local Search was the most effective technique in terms of its ability to find subtle HOMs, we propose two guided versions of Local Search. Each version uses a heuristic to improve the ability of Local Search to find subtle HOMs. The first version, Data-Interaction Guided Local Search, utilizes program structural information, such as data flow, to identify the FOM combinations that are more likely to produce subtle HOMs. The second version, Test-Case Guided Local Search, utilizes information about the given test suite, such as the set of test cases that kills each FOM, to identify the FOM combinations that are more likely to produce subtle HOMs.

Restricted Random Search is a constrained version of a random search technique. It iteratively explores the search space of HOMs by selecting a random number of FOMs at each iteration to create an HOM. Restricted Random Search uses a parameter that allows it to limit the degree of the explored HOMs. The majority of the subtle HOMs found by the search techniques in the initial experimental evaluation were HOMs of lower degrees [16, 17, 18].

Restricted Enumeration Search is a guided version of brute force approach. It explores the space of all HOMs in a predefined order starting with the areas of the search space where more subtle HOMs are expected to be found. Restricted Enumeration Search enumerates HOMs in an increasing order of their degrees starting with second order mutants.

2- Higher order mutation testing tool

The second contribution of the dissertation is the automation of the process of finding subtle HOMs. Due to the unavailability of higher order mutation testing tools for AspectJ and Java programs, we developed HOMAJ, a Higher Order Mutation Testing tool for AspectJ and Java programs. HOMAJ consists of five main components that automate the process of creating, compiling, and executing both FOMs and HOMs and implement the search techniques presented above. HOMAJ uses a selective compilation process to optimize the compilation of HOMs and reduce

the high computational cost associated with finding subtle HOMs. HOMAJ takes as an input the program under test along with a number of test suites. It creates AspectJ and Java FOMs, and compiles and executes them against the given test suite. HOMAJ starts the search process based on the selected search technique. When the search process stops, HOMAJ presents a list of all subtle HOMs that were found.

To promote the use of higher order mutation testing among researchers and practitioners, HOMAJ is designed such that it can be easily extended to include new search strategies and objective functions.

3- Evaluation studies

We performed a set of empirical studies to evaluate the effectiveness of the proposed search techniques in term of their ability to find subtle HOMs and investigated different factors that impact the creation of subtle HOMs. The empirical studies are as follows:

- *Measuring the relative effectiveness of the search technique*

  In this study we compared the relative effectiveness of the search techniques in terms of their ability to find subtle HOMs. The goal is to determine which technique can find a higher number of distinct, subtle HOMs. A higher number of distinct, subtle HOMs can be more beneficial for improving the fault-detection effectiveness of test suites.

  We measured the effectiveness in terms of the average number of subtle HOMs that were found. We investigated how the effectiveness of the search techniques compare over time and investigated the effectiveness of the search techniques with respect to the degree of subtle HOMs that were found.

- *Comparing sets of subtle HOMs found by different search techniques*

  The search techniques use different operators to generate HOMs, which can lead to different results. We investigated the overlap between the sets of subtle HOMs found by different techniques to determine what set of subtle HOMs can be uniquely found by each search technique and what set of subtle HOMs can be found by all techniques.

In a practical setting, a tester may not have the time nor the resources to run all the search techniques to find subtle HOMs. Therefore, knowing what set of subtle HOMs can be uniquely found by each search technique and what set of subtle HOMs can be found by all techniques can help testers select and prioritize the search techniques based on the desired type of subtle HOMs.

- *Impact of programming language constructs on creating subtle HOMs*

  We investigated the impact of the different constructs provided by both AspectJ and Java on the creation of subtle HOMs. We also investigated the impact of the aspectual behavior on the creation of subtle HOMs. The goal is to determine if subtle HOMs are more likely to be found when combining FOMs that correspond to a specific set of constructs or locations of the program under test.

- *Cost of finding subtle HOMs*

  We measured the cost of finding subtle HOMs in terms of the time taken by each search technique to find subtle HOMs. Because equivalent HOMs cannot be killed by test suites, they are treated as subtle HOMs. Equivalent HOMs can increase the cost of finding subtle HOMs because of the additional human effort needed to identify equivalent mutants. We investigated the proportion of subtle HOMs that represent equivalent HOMs and investigated the difficulty of killing non-equivalent subtle HOMs using randomly generated test cases.

- *Composition and decomposition relationships between subtle HOMs*

  Our initial investigation showed that the search techniques found a low number of subtle HOMs of higher degrees (four and higher). Thus, we investigated alternative techniques that can be more effective for finding subtle HOMs of higher degrees. We investigated how composing subtle HOMs of lower degrees could create subtle HOMs of higher degrees. We also analyzed subtle HOMs of higher degrees that were found by the search techniques to determine if subtle HOMs of higher degrees can only exist as a composition of other subtle HOMs.

The rest of the dissertation is organized as follows. Chapter 2 summarizes concepts pertaining to traditional and higher order mutation testing, aspect-oriented programming, and search-based software engineering. Chapter 3 presents related work in the areas of mutation testing for Java and AspectJ programs, cost reduction techniques in traditional mutation testing, higher order mutation testing, and search-based software engineering testing techniques. Chapter 4 describes the search techniques. Chapter 5 presents the design and use of HOMAJ. Chapter 6 presents the experimental setup for the empirical evaluation. Chapter 7 presents a study to measure the effectiveness of the search techniques. Chapter 8 presents a study to compare the sets of Subtle HOMs found by different search techniques. Chapter 9 presents a study to investigate the impact of the programming language constructs on the creation of subtle HOMs. Chapter 10 presents a study to measure the cost of finding subtle HOMs. Chapter 11 presents a study to investigate the composition and decomposition relationships between subtle HOMs. Chapter 12 discusses threats to validity. Chapter 13 presents the conclusions and Chapter 14 outlines directions for future work.

Chapter 2

Background

This chapter describes the concepts and terminology pertaining to traditional and higher order mutation testing, provides a brief introduction into AspectJ concepts and constructs, and introduces background information related to search-based software engineering techniques.

2.1 Mutation Testing

The first documented work of mutation testing can be found in 1971 in a student paper by Lipton [2]. In the late 1970s, DeMillo et al. [3] and Hamlet [5] presented the first published work in the field of mutation testing. Since then, mutation testing has gained a lot of interest and it has been increasingly used for testing software. Recently, Jia and Harman [6] stated that the field of mutation testing is reaching a mature state and showed evidence suggesting that mutation testing is becoming a practical testing approach.

There are two hypotheses underlying the work of mutation testing: *the competent programmer hypothesis* and *the coupling effect* [6, 20]. The competent programmer hypothesis states that programmers tend to develop programs that are almost correct (i.e., close to being correct). Therefore, a few syntactic changes can fix the faults in the program. The coupling effect states that complex faults and simple faults are coupled in a way that detecting simple faults can lead to the detection of many of the complex faults. Below we describe the key concepts and processes involved in mutation testing.

- **Mutation operator**: A mutation operator makes specific syntactic changes to the program to generate mutants. The syntactic changes simulate the mistakes often made by programmers. Researchers have defined mutation operators for various programming languages, such as Fortran [27], C [28], and Java [29, 30]. A single mutation operator might generate different mutants when applied to the same part of a program, each with a unique syntactic change.

For example, applying *Relational Operator Replacement* (ROR) to a Java statement, such as $if(x > y)$, produces the following mutated statements: $if(x >= y)$, $if(x <= y)$, $if(x < y)$, $if(x == y)$, and $if(x != y)$.

- **Mutant compilation and execution**: Mutant compilation refers to the translation of the source code of the generated mutants into object code that can be executed. Mutant execution refers to the process of executing the compiled mutant against the given test suite to assess the fault-detection effectiveness of the test suite. A mutant that produces a different execution result than the original program for some test case is considered killed.

- **Equivalent mutant:** An equivalent mutant is one that cannot be killed by any test case because it always produces the same output as the original program. Determining whether or not a mutant is equivalent is an undecidable problem [6, 31]. The presence of equivalent mutants is a major obstacle for the practical use of mutation testing because of the additional human effort needed to manually identify them, which can be high even for small programs.

- **Stubborn mutant:** Stubborn mutants [32] are those not killed by a high quality test suite. Although they are not equivalent to the original program, they are hard to kill because they require a specific set of test inputs or a specific set of test input combinations that are difficult to identify. The number of remaining stubborn mutants depends on the effectiveness of the test suite. Increasing the effectiveness of the test suite will result in fewer stubborn mutants. Exhaustive testing, which is infeasible to achieve in most cases, will result in no stubborn mutants.

- **Result analysis**: Result analysis involves calculating the *mutation score*, which is a measure of the fault-detection effectiveness of the test suite. The mutation score is the ratio of the number of killed mutants over the total number of non-equivalent mutants.

- **Mutation testing tool:** A mutation testing tool automates the process of generating and compiling mutants, executing them against the given test suite, and calculating the mutation score. Jia and Harman [6] reported 36 mutation tools that were implemented for different

programming languages. Some mutation tools can detect certain types of equivalent mutants [30, 33]. For example, AjMutator [33], which is a mutation tool for AspectJ programs, uses static analysis to automatically detect some of the equivalent mutants.

2.2 Higher Order Mutant Classification

Jia and Harman [8] classified HOMs in terms of their coupling and subsumption relations with FOMs. Below we summarize the formal definitions of HOM classifications.

- $F = \{f_1, \ldots, f_f\}$ is the set of all the FOMs for the program under test.
- H is the space of all candidate HOMs. $H = \mathcal{P}(F)$, where $\mathcal{P}$ is a power set.
- U is the universe of all possible test cases.
- $T = \{tc_1, \ldots, tc_t\}$ is the set of all test cases under consideration (the given test set), $T \subset U$.
- $h_i^n \in H$ is an HOM constructed from n FOMs, such that $h_i = \{f_{i_1}, \ldots, f_{i_n}\}$. The notation can be simplified to $h_i = h_i^n$ without confusion.
- Let $T_{h_i} \subseteq T$ denote the set of those test cases in T that kill h_i.
- There are n test sets $T_{i_1}, \ldots, T_{i_n}$, $\forall j \in [1, \ldots, n], T_{i_j} \subseteq T$ and T_{i_j} contains all test cases that kill f_{i_j} in h_i
- TU_i is a test set such that
$$TU_i = \bigcup_{j=1}^{n} T_{i_j}$$
- TI_i is a test set such that
$$TI_i = \bigcap_{j=1}^{n} T_{i_j}$$

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" [8], otherwise the HOM is decoupled. Formally, an HOM is defined as coupled to its constituent FOMs if:

$$T_{h_i} \cap TU_i \neq \emptyset \tag{2.1}$$

A decoupled HOM is that killed by a different set of test cases than its constituent FOMs. Decoupled HOMs are valuable to the testing process because they represent different faults than their constituent FOMs. An HOM is defined as decoupled from its constituent FOMs if:

$$T_{h_i} \cap TU_i = \emptyset \tag{2.2}$$

A subsuming HOM is one that is killed by a test set smaller in size than the test set that kills all FOMs used to construct the HOM. This means that the subsuming HOM is harder to kill than its constituent FOMs. A strongly subsuming HOM (SSHOM) is defined as one that is "only killed by a subset of the intersection of test cases that kill each FOM from which it is constructed" [8]. In other words, if a test case kills an SSHOM, it also kills all the constituent FOMs.

An SSHOM can replace its constituent FOMs without loss of test effectiveness. Therefore, SSHOMs reduce test effort by reducing the total number of FOMs to be compiled and executed and also by reducing the total number of test cases that need to be executed to kill the FOMs. An HOM is defined as a strongly subsuming HOM if:

$$T_{h_i} \subset TI_i \; and \; T_{h_i} \neq \emptyset \tag{2.3}$$

A weakly subsuming HOM is one that is killed by a test set that is smaller in size than the union of the test sets that kill each FOM from which the HOM is constructed. An HOM is defined as a weakly subsuming HOM if:

$$|T_{h_i}| < |TU_i| \; and \; T_{h_i} \neq \emptyset \tag{2.4}$$

A weakly subsuming and decoupled HOM is considered valuable because it is killed by fewer and different test cases than the constituent FOMs. An HOM is defined as a weakly subsuming and decoupled HOM if:

$$|T_{h_i}| < |TU_i|, \; T_{h_i} \neq \emptyset, \; and \; T_{h_i} \cap TU_i = \emptyset \tag{2.5}$$

A weakly subsuming and coupled HOM is harder to kill than the constituent FOMs, but it cannot replace the constituent FOMs without possible loss of test effectiveness. An HOM is defined as a weakly subsuming and coupled HOM if:

$$|T_{h_i}| < |TU_i|, \; T_{h_i} \neq \emptyset, \; and \; T_{h_i} \cap TU_i \neq \emptyset \tag{2.6}$$

2.3 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a programming paradigm that supports software modularity and composition by providing the means to modularize crosscutting concerns [12, 34]. Crosscutting concerns are aspects of the program that affect the primary functionalities of the system, which are called core concerns [34]. Crosscutting concerns, such as synchronization, logging, and memory allocation, need to be addressed in many if not all system units. In procedural programming and object-oriented programming, the code that handles a crosscutting concern is scattered or duplicated across various related locations, resulting in poor system modularity, maintainability, and evolvability [34, 35]. An AOP language provides mechanisms to support the separation and composition of such crosscutting concerns.

AspectJ [12] is an AOP extension for the Java programming language and it is the most widely used AOP language. It introduces the following concepts and constructs to handle the crosscutting concerns.

- **Join point**: A join point is well-defined point during the execution of a program where the execution can be augmented or altered. A join point can be a call or execution of a method or constructor, initialization of a class or an object, field read and write access, or execution of an exception handler.

- **Pointcut**: A pointcut is a set of join points. A pointcut descriptor is expressed as a predicate that matches a set of join points. AspectJ pointcuts can be expressed as a set of primitive pointcuts combined using logical operators where each primitive pointcut targets the desired join points.

- **Advice**: An advice contains the actions to be applied at the identified join point(s). It contains code that implements the additional behavior to be executed when program execution reaches a join point. The additional behavior represents the functionalities that implement the crosscutting concern. AspectJ provides three types of advice, which can be executed around, before, and after a join point.

- **Inter-Type Declarations**: Inter-Type Declarations (ITD) are used to introduce new fields and methods into classes and interfaces. Inter-type declarations allow developers to add methods, constructors, state variables to classes, and concrete implementation to an interface. They can also be used to alter the class and interface hierarchies and to declare aspect precedence.

- **Aspects**: An aspect is a modular unit that encapsulates pointcuts, advices, and inter-type declarations.

2.4 Search-Based Software Engineering

Search Based Software Engineering (SBSE) refers to the body of work in which optimization and search techniques, such as genetic algorithms and local search, are applied to software engineering problems [36]. SBSE techniques have attracted a lot of attention within the software engineering community because many software engineering problems can be easily formulated as optimization problems and the evaluation of candidate solutions can be automated [36, 37]. Software testing problems were the first type of problems to be tackled by SBSE techniques and the majority of the available SBSE research publications targeted testing problems [36, 38].

2.4.1 Applying Search-Based Techniques to Software Engineering Problems

Harman and Jones [23] presented three steps that need to be taken to reformulate a software engineering problem into a search-based problem.

- **Defining solution representation**: A candidate solution is represented as a scheme that is amenable to symbolic manipulation. The choice of the representation is a key for the success of the SBSE technique. For many software engineering problems, the original representation of a candidate solution can be directly used by the search techniques. Frequently used representations are binary strings and floating point numbers. Data structures, such as arrays and trees, have also been used.

- **Defining objective function**: An objective function provides a metric to measure the quality of solutions.

- **Defining operators**: The operators of a search technique are responsible for transforming one solution to another. For example, a Genetic Algorithm applies mutation and crossover operators to a set of solutions to produce new solutions. The design and implementation of the operators depend on the selected solution representation.

2.4.2 Commonly Used Search Based Optimization Techniques

Genetic Algorithms and Local Search are the most commonly used SBSE techniques [22]. Random Search have also been frequently used in the software engineering literature [36]. In the remainder of this section, we briefly describe the distinguishing features of these techniques.

1- Genetic Algorithm

The earliest work of Evolutionary Algorithms (EA) can be dated back to the late 1950s [39]. Holland et al. [40, 41] introduced the model known as Genetic Algorithm. Genetic algorithms have been shown to perform well for many problem domains, including large-scale software engineering problems [22, 42, 43, 44].

A Genetic Algorithm starts with a set of initial solutions, called *chromosomes*, which can be randomly generated. A set of chromosomes is called a *population*, and the set of the initial solutions is called the *first population*. A Genetic Algorithm applies a set of reproduction operators to a population to produce a new population. The reproduction operators are *crossover* and *mutation*. The crossover operator combines two or more chromosomes, which are called parents, to produce new chromosomes, which are called offspring. The mutation operator alters a small part of a chromosome to maintain diversity in a population. A Genetic Algorithm iteratively applies the reproduction operators. In each iteration, the *selection* operator is used to favor better chromosomes for reproduction. The better chromosomes are those that have better fitness values, which get assigned by the objective function. By favoring the better chromosomes, a Genetic Algorithm is more likely to find even better chromosomes in each iteration.

2- Local Search

A Local Search technique moves in the search space by applying local changes to a solution until an optimal solution is found [45]. Hill Climbing and Simulated Annealing are the most commonly used Local Search techniques in SBSE literature [22, 36]. Hill Climbing is an iterative search technique that starts with an arbitrary solution, called the *incumbent solution*. Hill Climbing searches for a better solution using an objective function to examine the set of candidate solutions in the neighborhood. The neighborhood graph determines which candidate solutions are considered neighbors and which are not.

When Hill Climbing finds a better neighboring solution, it becomes the incumbent solution. If no better neighboring solution is found, the incumbent solution is considered to be a *locally optimal* solution, which may not be *globally optimal*. The process can be restarted in order to find potentially better solutions [45].

Hill Climbing uses different strategies to move in the search space. The *steepest ascent* usually requires evaluating all candidate solutions in the neighborhood and then selecting the solution that provides the best improvement. *First ascent* requires evaluating the candidate neighboring solutions, one at a time, and then selecting the first improvement [22].

Simulated Annealing is based on the annealing process that happens in metallurgy where a material is heated beyond its melting point and then cooled to change its structural properties [46]. Unlike Hill Climbing, Simulated Annealing uses a probabilistic measure when selecting a neighboring solution. The probabilistic measure can lead to the selection of a solution of a lower fitness value than the incumbent solution [47].

3- Random Search

Random Search iteratively moves in the search space randomly to locate candidate solutions in the search space [48, 49]. Pure (unguided) Random Search techniques usually fail to find globally optimal solutions [36]. However, many Random Search techniques have been successfully applied to many software engineering problems, such as generation of test inputs [22, 50].

Chapter 3

Related Work

In this chapter, we summarize related work in the area of mutation testing in the context of Java and AspectJ programs. We describe existing mutation cost reduction techniques. We present related work in the area of higher order mutation testing and fault interactions. We highlight some of the reported uses of search-based software engineering techniques in software testing.

3.1 Mutation Testing for Java Programs

The earliest work of mutation testing targeted the programming language, Fortran [51]. Researchers designed mutation operators and implemented mutation systems for procedural programming languages, such as Fortran [27, 52, 53] and C [28, 54]. However, the mutation operators that were proposed for the procedural languages did not cover the programming concepts and constructs introduced in Object-Oriented Programming (OOP) languages, such as Java. Such concepts and constructs result in different program structure and new fault types that did not exist in programs developed using procedural languages.

Moore [55] developed the first mutation tool for Java programs, Jester. The tool makes four types of syntactic modifications to the source code. It modifies literal numbers, switches boolean values, and makes the condition of an if-statement always true or always false. Jester can also introduce syntactic changes to JUnit test cases to help testers identify errors in their testing code.

Kim et al. [56] proposed 20 Java mutation operators and grouped them into six categories: types and variables, names, classes/interface declarations, blocks, expressions and others. Kim et al. [57, 58] later added 15 class mutation operators that target faults related to object-oriented features. The class operators were classified as polymorphic, method overloading, information hiding, and exception handling.

Ma et al. [59] proposed 24 Java mutation operators, which were based on previous studies of object-oriented fault models [60, 61, 62, 63, 64]. The operators were classified as information hid-

ing, inheritance, polymorphism, overloading, Java-specific features, and common programming mistakes. Ma et al. [30, 65] developed MuJava, a mutation tool for Java programs. The tool automates the process of generating FOMs, compiling and executing them against the given test suite, and calculating the mutation score. MuJava provides graphical user interfaces and implements approaches to automatically detect some types of equivalent FOMs and to reduce the cost of mutation testing.

Ma et al. [66] conducted an experimental study to examine the numbers and kinds of mutants that can be generated using MuJava. Ma et al. also examined whether or not the faults introduced by class mutation operators can be detected by test cases that detect statement level faults. The number of class mutants is relatively smaller than statement level mutants. A similar observation was reported by Offutt et al. [67]. Some class mutation operators can be omitted because their faults can be easily detected by test cases that detect statement level faults.

Offutt et al. [67] presented an empirical study for object-oriented programs using MuJava. They used six open-source programs and described techniques for eliminating some equivalent mutants. Although the number of class level mutants was far less than the number of statement level mutants, there were more equivalent class level mutants than equivalent statement level mutants.

Chevalley and Fosse [68] developed the first Java mutation tool, JavaMut, with a graphical user interface support. The tool implements 26 mutation operators. Kim et al. [69] introduced a technique and a prototype tool for mutation testing called MUGAMMA. The tool gathers information about the program states to be used later for regression testing. MUGAMMA implements only a small set of Java mutation operators.

Irvine et al. [70] developed Jumble, a bytecode level mutation testing tool. Jumble implements heuristics to speed the execution and analysis processes of FOMs. Schuler et al. [71] presented the JAVALANCHE framework to mutate Java bytecode. JAVALANCHE uses a small set of mutation operators, which have been shown to be sufficient for mutation testing [67]. The tool also uses techniques to reduce the number of generated FOMs and the number of test cases that need to be executed on each FOM.

More recently, Madeyski and Radyk [29] presented a new approach to mutation testing, called Fast Aspect-oriented Mutation Testing Approach (FAMTA). The approach takes advantage of the pointcut and advice mechanisms provided by AspectJ to enhance the performance of the mutation testing process. Madeyski and Radyk developed Judy, a tool that implements the new approach and supports 16 mutation operators.

Kirk [72] developed PIT, which is a bytecode level mutation testing tool for Java. PIT was intended to be used by development teams rather than academic researchers. PIT implemented a set of 14 mutation operators that were designed to be produce harder to kill mutants while minimize the number of equivalent mutants.

3.2 Mutation Testing for Aspect-Oriented Programs

Because of the unique concepts and constructs introduced by Aspect-Oriented Programming languages, researchers proposed new fault models to target the new types of faults that may rise. According to the proposed fault models [11, 26, 73, 74, 75, 76], faults can be classified as those that can occur in base classes, aspects (pointcut, inter-type declarations, aspect declarations, and advice), or in the interaction between the base classes and aspects.

Ferrari et al. [73] presented four aspect-oriented fault types, which were based on specific AspectJ constructs, and designed a set of mutation operators based on these fault types. The fault types are: pointcut faults (8 faults), inter-type declaration faults (9 faults), advice faults (6 faults), and base program related faults (3 faults). Ferrari et al. defined 26 mutation operators and divided them into three groups. Group 1 contains 15 mutation operators that model pointcut expression faults. Group 2 contains five operators that model Java generics faults. Group 3 contains six operators that model advice definition and implementation related faults. Ferrari et al. also conducted a cost analysis to measure the cost of using their mutation operators. Pointcut expression operators resulted in the largest number of mutants comparing to other AspectJ constructs.

Delamare et al. [33] developed AjMutator, an AspectJ mutation tool that implements the pointcut mutation operators proposed by Ferrari [73]. The tool classifies and compiles the mutants in addition to executing the test cases on the mutants. The mutants are classified by comparing the

set of join points they match with the set of join points matched by the original pointcut. The tool outputs an XML file that contains information about every mutant handled (e.g., mutant status, pointcut ID and aspect ID). The tool was evaluated using two different AspectJ systems. The experiments showed that AjMutator was able to generate and compile a large number of mutants on large systems and perform mutation analysis to obtain a mutation score.

Ferrari et al. [77] developed a tool for mutation testing of AspectJ programs called Proteum/AJ. The tool automates a set of aspect-oriented mutation operators proposed in their previous work [73] and supports the basic steps of mutation testing. Proteum/AJ implements 24 mutation operators, 15 of which are pointcut related operators. The tool allows the tester to select and manage mutation operators in several ways.

Wedyan and Ghosh [11] modified the existing AspectJ fault models to address some of the shortcomings. The modifications included moving some existing faults from one fault category to another and adding new data-flow interaction based fault types. Wedyan and Ghosh also studied three problems with the use of mutation analysis: manual identification and removal of equivalent mutants, generating mutants that cover the all fault types, and measuring the difficulty of killing the generated mutants. They used higher order mutation to produce HOMs that cover all pointcut fault types. In the subject programs they studied, the pointcut mutants that match some of the intended join points and some of the unintended join points at the same time could only be obtained with the help of HOMs.

Anbalagan et al. [78] used mutation testing to test the strength of pointcuts to help developers detect pointcut faults. They developed a framework that generates relevant pointcut mutants and detects equivalent mutants. Relevant mutants are those that resemble the original pointcut expression closely, reflecting the kind of mistakes developers may make. Equivalent mutants are those matching the same set of join points as the original pointcut. The framework classifies the mutants and ranks them using distance measures to reduce the total number of mutants that need be inspected. Developers can inspect the highly ranked mutants and their differentiating join points to determine pointcut correctness and robustness. The framework implementation was applied on selected pointcuts from aspects in different AspectJ benchmarks.

3.3 Mutation Cost Reduction Techniques

Due to the high computational cost associated with the compilation and execution of a large number of mutants, researchers proposed different mutation cost reduction techniques for traditional FOMs.

Jia and Harman [6] classified the mutation cost reduction techniques into two types. The first type of techniques reduces the number of generated mutants. Acree and Budd [4, 79] presented a *mutant sampling* approach, which requires randomly selecting a small number of FOMs from the large number of generated FOMs for mutation analysis. Only the randomly selected FOMs are compiled, executed, and analyzed and the remaining FOMs are discarded. Hussain [80] introduced the *mutant clustering* approach, which involves classifying FOMs into different clusters based on the test cases that kill them, then selecting a small number of FOMs from each cluster, and discarding the rest. The *selective mutation* approach [81, 82] involves using a smaller set of mutation operators to generate a smaller number of FOMs that can be used for mutation testing without significant loss of test effectiveness. *Strongly subsuming HOMs* have also been used to reduce the total number of mutants to be compiled and executed without loss of test effectiveness [8].

Cost reduction techniques of the second type optimize the compilation and execution process of FOMs by adopting smart and fast mechanisms. King and Offutt [27, 52] presented an *interpreter-based* technique that interprets the FOM execution result directly from the source code. The original program is translated into an intermediate form, and mutation and interpretation are performed at this intermediate code level.

DeMillo et al. [83] proposed a *compiler-integrated* technique where an instrumented compiler generates the executable code for the original program along with FOM patches. Each patch contains the binary code of the mutated code, which can be applied directly to the original executable code to produce the executable FOM. Untch [84, 85] presented a similar approach called the *mutant schema generation* approach. This approach generates a meta-program (i.e; super mutant), which represents all generated FOMs.

3.4 Higher Order Mutation Testing

Jia and Harman [8] utilized three search based optimization approaches, a Genetic Algorithm, a Greedy Algorithm, and a Hill Climbing Algorithm, to explore the search space seeking out the strongly subsuming HOMs in C programs. They introduced a fitness function that uses a fragility function, which measures the difficulty of killing a mutant. The fitness of an HOM is measured by the ratio of the fragility of the HOM to the fragility of the constituent FOMs. HOMs with lower fitness value are less fragile (harder to kill) than their constituent FOMs. Jia and Harman [8] conducted an experimental study using 10 C programs. Subsuming HOMs were found for all the studied programs. Although the proportions of strongly subsuming HOMs were low, the actual numbers were large because the total numbers of all generated HOMs were extremely large. The Genetic Algorithm performed the best at finding strongly subsuming HOMs because they were easier to generate from existing subsuming HOMs.

Jia and Harman [87] developed a mutation testing tool, *MILU*, for both first order and higher order mutation testing. In first order mutation testing mode, testers can use a pre-defined set of mutation operators or customize their own mutation operators. The tool implements 77 C mutation operators and provides a flexible scripting language for the customization of operators. In higher order mutation testing mode, testers can use one of the predefined search-based techniques, a Genetic Algorithm, Greedy Algorithm, or Hill Climbing Algorithm, or specify their own. MILU provides graphical user interfaces and introduces a test harness technique that reduces the cost of running mutants. The technique involves compiling each mutant into a shared library, and then using the given test suite to generate a test harness that runs all the test cases by dynamically invoking the shared mutant library.

Langdon et al. [13] presented a study to explore the relationship between mutated source code syntax and its semantics. They used a Multi-Objective Pareto Optimal Genetic Programming approach to search for HOMs that were hard to kill and were syntactically similar to the original program. The syntactic distance between the original and mutated source code is measured in terms of the number of the syntactic differences weighted by a constant. The semantic distance

is measured in terms of the number of test cases that kill the mutant. The Pareto search produced HOMs that were harder to kill than any of the FOMs for the two studied programs. Exploring the search space of HOMs could provide more insights into the different structure and behavior of test cases in the test suite.

Polo et al. [88] presented three algorithms for constructing second order mutants from FOMs: LastToFirst, Different Operators, and Random Mix. The LastToFirst algorithm constructs second order mutants 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 FOM once. The number of mutants was reduced to half and the number of equivalent second order mutants was significantly reduced.

DiGiuseppe and Jones [89] 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. They reported four significant types of fault interaction: fault synergy, which occurs when fault interaction produces new faults, fault obfuscation, which occurs when fault interaction produces less faults, fault independence, which occurs when fault interaction does not produce more or less faults, and fault multi-type interaction, which occurs when fault interaction results in both synergy and obfuscation of faults. Fault obfuscation was the most prominent fault type and this fault type was even more prominent among the larger programs they studied. The authors argued that the prevalence of fault obfuscation can affect many of the existing test practices and emphasized the needs for models that can predict such fault interactions.

Debroy and Wong [90] 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. Their results showed that failure masking was more frequent in the programs they studied.

Omar and Ghosh [91] presented an exploratory study of higher order mutation testing in the context of AspectJ programs. They proposed four approaches to constructing second order mutants in AspectJ programs.

The approaches classify HOMs based on the proposed AOP fault models [11, 26, 73]. Faults in AspectJ programs can occur in the base classes, in the aspects, or in the interactions between base classes and aspects. The construction approaches 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.

Omar and Ghosh also developed a prototype tool that automates the process of generating, compiling, and executing both first and second order mutants. They evaluated the approaches in terms of their ability to create second order mutants that result in higher test effectiveness and lower test effort compared to FOMs. All approaches produced second order mutants that can be used to increase test effectiveness and reduce test effort. However, the first approach produced a larger percentage of higher order mutants that were harder to kill than the constituent FOMs as compared to the last three approaches. The first approach lowered the total number of mutants to be compiled and executed to a greater extent than the last three approaches. The last three approaches produced a lower density of equivalent mutants. We can use second order mutants to reduce the number of FOMs that need to be compiled and executed by 17.6% without loss of test effectiveness.

Kintis et al. [15] used second order mutants to isolate possible first order equivalent mutants. The proposed technique, I-EQM, aims to reduce the number of possible equivalent FOMs that need to be inspected by the tester. I-EQM classifies FOMs that are not killed by the given test set into two sets. The first set contains FOMs that are considered more likely to be *killable*, which

the tester needs to inspect. The second set contains FOMs that are considered more likely to be equivalent, which the tester can ignore. The technique combines an already killed FOM with a possibly equivalent FOM and compares the execution result of the resulting second order mutant to the execution result of the already killed FOM. If the results differ, the possibly equivalent FOM is classified as a possibly killable FOM, otherwise it is classified as an equivalent FOM. Kintis et al. presented a case study using four subject programs to evaluate the effectiveness of the proposed technique. The proposed technique was able to correctly classify 82% of the killable mutants.

3.5 Search-Based Software Engineering for Testing Problems

In the available literature, most of the proposed SBSE testing techniques are concerned with test input generation for fault discovery. Researchers also proposed SBSE techniques for fixing bugs [25] and facilitate debugging [92]. However, other than the work presented by Jia and Harman [8, 13, 14], which is described in detail in Section 3.4, none of the proposed SBSE testing techniques tackle similar testing goals as the techniques presented in this proposal. We present some of the reported uses of SBSE techniques in software testing.

Xanthakis et al. [24] was the first to use an SBSE technique to solve a software engineering problem. They used Genetic Algorithms to generate test input for structural coverage. Since then, SBSE techniques have been applied to automate test input generation for the coverage of specific program structures, exercising specific program features, and to verify non-functional properties. Search-based techniques have also been used to generate test input that kill mutants [93, 94, 95, 96].

Davies et al. [97] presented a Genetic Algorithm approach for test input generation. The goal of the algorithm was to find test inputs for which an expert system for supporting combat pilots performs most poorly. Other researchers used SBSE testing techniques to search for test inputs that degrade software performance (stress testing) and cause exceptions to be raised [98, 99].

Ferguson and Korel [100] proposed a Local Search technique for automated software test input generation. They used data dependence analysis to guide the test input generation process to seek inputs that traverse hard-to-cover predicate branches. The technique was effective in generating test inputs that traverse specified branches.

Lakhotia et al. [101] introduced the first multi-objective approach for structural test input generation. All previous techniques aimed to find test inputs that traverse a specific branch. The proposed approach seeks to maximize both program coverage and dynamic memory consumption at the same time. The authors presented five case studies to compare the performance of three search techniques: (1) a random search, (2) a Pareto Genetic Algorithm, and (3) a weighted Genetic Algorithm. The weighted Genetic Algorithm outperformed the other two algorithms in most cases. The authors also argued that a hybrid approach between Pareto and weighted Genetic Algorithms may offer the best overall results.

Unlike procedural languages, the test inputs for class methods in object-oriented programs can include objects that need to be created and their internal state might need to be changed to satisfy some criterion [102]. Further, determining neighboring inputs can be difficult because a small change in a state variable can produce totally different program behavior. Therefore, researchers proposed SBSE techniques for test input generation for object-oriented programs. Tonella [102] used a Genetic Algorithm to generate test inputs for unit testing of classes. The chromosome representation included information about the objects to be created, methods to be invoked, and input values to be used. A tool called eToc was implemented to conduct an experimental study using some classes from the standard Java library. The generated test cases were able to cover hard-to-reach code and revealed a known fault in the standard Java library.

Harman et al. [103] introduced a search-based technique for test input generation for AspectJ programs. They also introduced a domain reduction technique to improve the performance of the proposed technique. They used dependence analysis based on slicing to reduce the test input space and remove irrelevant parts that cannot affect the output. Harman et al. conducted an empirical study using 14 AspectJ programs to demonstrate the effectiveness of their technique. They compared their technique with an existing random technique for test input generation for AspectJ programs [104, 105]. The proposed technique produced significantly better results than the random technique. The domain reduction technique increased both effectiveness and efficiency of the developed system.

SBSE testing techniques have been used for regression testing for the selection of representative test cases from a large pool of test cases, prioritization of the execution of test cases within a test suite, and minimization of the size of test suites without loss of test effectiveness [106, 107, 108].

SBSE testing techniques have also been used to automate the process of software repair [109, 110, 111]. Forrest et al. [109] used genetic programming and program analysis methods to repair faults in C programs. The approach uses negative test cases to exercise the fault and positive test cases to encode the required behavior of the program. A successful repair passes all test cases. Reported results showed that 11 bugs in over 60,000 lines of code were repaired.

Chapter 4

Approach

In this chapter, we present the search techniques for finding subtle HOMs. Section 4.3 presents the objective function used by the search techniques to measure the quality of HOMs. Sections 4.2 through 4.6 present the Genetic Algorithm, Local Search, the Guided Local Search techniques, Restricted Random Search, and Restricted Enumeration Search respectively.

4.1 Objective Function

The goal of our objective function is to provide a metric to measure the quality of HOMs. The objective function identifies subtle HOMs, which represent optimal solutions. The metric should also identify the HOMs that have the potential to develop into subtle HOMs. The search techniques aim to find as many distinct optimal solutions as possible. Below we provide the notation and formal definition of the objective function.

- $F = \{f_1, \ldots, f_f\}$ is the set of all non-equivalent FOMs for the program under test.
- H is the space of all candidate HOMs. $H = \mathcal{P}(F)$, where $\mathcal{P}$ is a power set.
- U is the universe of all possible test cases.
- $T = \{tc_1, \ldots, tc_t\}$ is the set of all test cases under consideration (the given test suite), $T \subset U$ and T kills all the FOMs in F.
- $h_i^n \in H$ is an HOM constructed from n FOMs, such that $h_i = \{f_{i_1}, \ldots, f_{i_n}\}$. The notation can be simplified to $h_i = h_i^n$ without confusion.
- Let $T_{h_i} \subseteq T$ denote the set of those test cases in T that kill h_i. $T_{h_i} = \emptyset$ if none of the test cases in T kill h_i.

- There are n test sets $T_{i_1}, \ldots, T_{i_n}, \forall j \in [1, \ldots, n], T_{i_j} \subseteq T$ and T_{i_j} contains all test cases that kill f_{i_j} in h_i.

- TU_i is a test set such that

$$TU_i = \bigcup_{j=1}^{n} T_{i_j}$$

Based on these notations we first present two measures, fault detection difference between HOM and its constituent FOMs and difficulty of killing HOM, which are both used by the objective function below.

Fault detection difference between HOM and its constituent FOMs

$$FDD(h_i) \;=\; \frac{|(TU_i \cup T_{h_i})| - |(TU_i \cap T_{h_i})|}{|TU_i \cup T_{h_i}|} \tag{4.1}$$

The goal of this measure is to capture the level of interaction between the constituent FOMs of an HOM. The level of interaction between the constituent FOMs is measured in terms of the difference between the set of test cases that kill the HOM and the union of all sets of test cases that kill each individual constituent FOMs. FOMs that are killed by a different set of test cases when combined than when individually are those that can interact to mask each other. Subtle HOMs represent the case where the constituent FOMs interact to completely mask each other and produce new faulty behavior that cannot be detected by the given test suite. Therefore, an HOM with a greater difference between the set of test cases that kill the HOM and the union of all the sets of test cases that kill its constituent FOMs should be assigned a higher fitness value and favored in the selection process of the search techniques.

The value of the fault detection difference in Equation 4.1 lies between 0 and 1. An HOM that is killed by a totally different set of test cases than its constituent FOMs will have the highest value of 1. This includes subtle HOMs, which are not killed by any of the test cases in the given test suite. An HOM that is killed by the same set of test cases that kill all its constituent FOMs will have the lowest value of 0.

This measure does not take into consideration the size of the set of test cases that kills the HOM as long as this set is totally different from the union of all the sets of test cases that kill the constituent FOMs. For example, two HOMs that are each killed by totally different set of test cases than their constituent FOMs will be assigned value of 1 regardless of the size of the set of test cases that kill each one. Although both HOMs are desired because their constituent FOMs can interact and form different faulty behavior, the HOM that is killed by a smaller set of test cases is more desirable because it is harder to kill. This aspect of measuring the fitness of an HOM is captured by the following measure, called difficulty of killing HOM.

Difficulty of killing HOM

$$DOK(h_i) \ = \ \frac{|(TU_i \cup T_{h_i})| - |(T_{h_i})|}{|TU_i \cup T_{h_i}|} \tag{4.2}$$

The goal of this measure is to capture how hard it is to kill an HOM with respect to its constituent FOMs. A harder to kill HOM is killed by a set of test cases that is smaller in size (i.e fewer test cases) than the union of the sets of test cases that kill each individual constituent FOMs. Subtle HOMs by definition are harder to kill than their constituent FOMs. In fact, they are the hardest to kill because they are not killed by any test case in the given test suite. Therefore, HOMs that are harder to kill than their constituent FOMs should be assigned higher fitness values so that they can be favored in the selection process of the search techniques.

The value of the difficulty of killing an HOM in Equation 4.2 lies between 0 and 1. An HOM that is not killed by any test case in the given test suite will have the highest value of 1 and an HOM that is killed by all the test cases that kill its constituent FOMs will have the lowest value of 0.

Objective function

$$fitness(h_i) \ = \ \alpha \ * \ DOK(h_i) \ + \ (1 - \alpha) \ * \ FDD(h_i) \tag{4.3}$$

The objective function assigns the fitness value of an HOM based on the sum of the two weighted terms, (1) difficulty of killing HOM and (2) fault detection difference, in Equation 4.3. The value of α, which lies between 0 and 1, determines the weight of the two terms.

The fitness value of an HOM lies between 0 and 1. HOMs with higher final fitness values represent better solutions and are favored in the selection process. Note that an HOM which has a fitness value of 1 represents a global optimum in the search space, and there are potentially many globally optimal solutions. FOMs have 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. An entirely coupled HOM represent the case where 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 each individual constituent FOMs. An entirely coupled HOM is considered to be useless because its constituent FOMs cannot interact to form new faulty behavior and the HOM is as hard to kill as its constituent FOMs. The value of α has no impact on the entirely coupled HOMs because the values of the difficulty of killing and fault detection difference are equal to 0.

2. **Promising HOMs**:

   An HOM with a fitness value greater than 0 but less than 1 is considered to be a promising HOM because it has the potential to develop into a subtle HOM when the right FOMs are added or removed.

   A promising HOM represent the case where there is an interaction between the constituent FOMs because the set of test cases that kill the HOM and the union of all sets of test cases that kill each individual constituent FOM are not the same. The value of the fault detection difference for a promising HOM will be higher than 0 depending on the difference between the two sets of test cases. However, the value of the difficulty of killing for a promising HOM will be less than 1 depending on the number of test cases that kill the HOM. The value of the difficulty of killing will be 0 if the HOM is easier to kill (i.e. killed by more test cases) than its constituent FOMs.

   The value of α, which is determined by the tester, affects the fitness values of promising HOMs. A higher value for α means harder to kill HOMs will have higher fitness values than

those with higher level of interaction between their constituent FOMs. The optimal value for α is one that leads to finding the highest number of subtle HOMs and that is not easy to determine. In this dissertation we used experimental evaluation to determine the value of α that leads to best results.

3. **Subtle HOMs**:

   An HOM with a fitness value of 1 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. The values of the difficulty of killing and fault detection difference are equal to 1 and the value of α has no impact in this case.

4.2 Genetic Algorithm

The Genetic Algorithm (GA) evolves a set 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 Genetic Algorithm operators and discuss some implementation issues. The pseudocode is shown in Algorithm 1.

1- Inputs

Our Genetic Algorithm takes as input the list of FOMs ($FOMsList$) for the program under test, number of crossover points ($numCrossPoint$), mutation rate ($mutationRate$), the population size ($populationSize$), the maximum degree of HOMs in the first population ($firstPopulationMaxDegree$), and the number of elite HOMs that get carried over at each iteration ($numEliteHOM$). These parameters are configured as follows.

- $1 \leq numCrossPoint \leq NOS$, where NOS represents the number of code statements in the program under test.
- $0 \leq mutationRate \leq 1$.
- $populationSize > 0$.

- $2 \leq firstPopulationMaxDegree \leq |F|$, where F represents the set of all non-equivalent FOMs for the program under test.
- $0 \leq numEliteHOM < populationSize$.

The configuration of these parameters is set by the user. However, the optimal configurations, which can lead to the highest number of subtle HOMs are not easy to determine. We used experimental evaluation to determine the configuration that gave the best results.

2- Chromosome representation

Each chromosome corresponds to an HOM. The chromosome is represented as one-dimensional array of strings such that each element in the array represents a Java/AspectJ statement from the program under test. All program statements from all the classes and aspects are included in each chromosome. Each mutated statement corresponds to a constituent FOM. This representation makes it easy to manipulate the program statements and to implement crossover and mutation operators.

3- First population

Each HOM in the first population is created by combining a number of randomly selected FOMs from $FOMsList$. The degree of each HOM in the first population is randomly selected such that the degree is between 2 and the maximum degree allowed ($firstPopulationMaxDegree$) for the first population.

4- Selection

The selection process involves selecting HOMs that are allowed to produce offspring using crossover and mutation. Our Genetic Algorithm uses *tournament selection* to implement generational replacement of the population. Our 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 of HOMs with the highest fitness values in the current population are automatically carried over (copied) to the next generation. This number is specified by

$numEliteHOM$. The elitist selection strategy ensures that copies of the best HOMs are not lost when moving from one generation to the next.

5- Crossover

The crossover happens between two selected parent HOMs that are recombined to produce two offspring HOMs. The crossover depends on the number of crossover points, which could be one or many, based on the configurable parameter, $numCrossPoint$. The crossover operator is designed such that it does not produce an offspring with a single mutation fault. An offspring HOM with a single mutation is combined with a random mutation to become a second order mutant.

6- Mutation

Our Genetic Algorithm 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 $FOMsList$ 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. The mutation operator is designed such that it does not cause an HOM to become an FOM.

7- Evaluation

Evaluating HOMs in a population involves two steps. All HOMs in the population are created, compiled, and executed against the given test suite. The objective function shown in Equation 4.3 evaluates each HOM and assigns a fitness value.

8- Stopping condition and output

Our Genetic Algorithm 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 tester can define the maximum number of distinct HOMs that the Genetic Algorithm is allowed to explore, the time limit that the Genetic Algorithm is allowed to run for, and the number of required subtle HOMs.

Algorithm 1 Genetic Algorithm

Require: $FOMsList$, $numCrossPoint$, $populationSize$, $mutationRate$, $numEliteHOM$, $firstPopulationMaxDegree$

1: $t \leftarrow 0$
2: $subtleHOMsList \leftarrow \emptyset$
3: $pop[t] \leftarrow$ createFirstPopulation($FOMsList$, $populationSize$, $firstPopulationMaxDegree$)
4: $pop[t]$.repairMutants()
5: $pop[t]$.executeMutants()
6: $pop[t]$.calculateFitness()
7: $subtleHOMsList$.add($pop[t]$.getSubtleHOMs())
8: **while** not (Termination_Condition()) **do**
9: $pop[t+1]$.add($pop[t]$,$numEliteHOM$)
10: **while** $pop[t+1]$.size $<$ $pop[t]$.size **do**
11: $parents \leftarrow pop[t]$.select(4,2)
12: $offSpring \leftarrow parents$.crossover($numCrossPoint$)
13: $offSpring$.mutate($mutationRate$)
14: $pop[t+1]$.add($offSpring$)
15: **end while**
16: $t \leftarrow t+1$
17: $pop[t]$.repairMutants()
18: $pop[t]$.executeMutants()
19: $pop[t]$.calculateFitness()
20: $subtleHOMsList$.add($pop[t]$.getSubtleHOMs())
21: **end while**
22: **return** $subtleHOMsList$

4.3 Local Search

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

1- Inputs

Local Search takes as input the list of FOMs for the program under test ($FOMsList$).

2- Starting point

Local Search starts by selecting an incumbent HOM from the list of all candidate Second Order Mutants (SOMs), $SOMsList$. The list of all SOMs contains pairs of FOMs and it is generated prior to the search process. Local Search then evaluates the incumbent HOM by (1) compiling and executing it against the given test suite, (2) recording the execution result, and (3) calculating the fitness value using Equation 4.3.

3- Neighborhood graph

After the incumbent HOM is generated and evaluated, Local Search generates all the HOMs neighboring the incumbent HOM. A neighborhood graph defines HOMs that are considered to be neighbors. Our neighborhood graph defines neighbors as those that vary by one FOM (one step) from the incumbent HOM. The neighboring HOMs are maintained in a list called $neighborsList$. Our neighborhood graph can be formally defined as follows.

- $F = \{f_1, \ldots, f_f\}$ is the set of all the non-equivalent FOMs for the program under test.
- H is the space of all candidate HOMs. $H = \mathcal{P}(F)$.
- $h_i^n \in H$ is an HOM constructed from n FOMs, such that $n \geq 2$.
- $h_j^m \in H$ is an HOM constructed from m FOMs, such that $m \geq 2$.

Algorithm 2 Local Search

Require: $FOMsList$
1: $shouldRestart \leftarrow true$
2: $subtleHOMsList \leftarrow \emptyset$
3: $restrictedList \leftarrow \emptyset$
4: $SOMsList \leftarrow$ createListOfSOMs($FOMsList$)
5: **while** not (Termination_Condition()) **do**
6: **if** $shouldRestart$ **then**
7: $i \leftarrow$ genererateRandomInt($SOMsList$.getSize())
8: $incumbentHOM \leftarrow SOMsList$.remove($i$)
9: executeMutant($incumbentHOM$)
10: evaluateFitness($incumbentHOM$)
11: **end if**
12: $neighborsList \leftarrow$ generateNeighbors($incumbentHOM$)
13: executeMutants($neighborsList$)
14: evaluateFitnesses($neighborsList$)
15: **if** ($incumbentHOM$.isSubtle()) **then**
16: $subtleHOMsList$.add($incumbentHOM$)
17: **end if**
18: $subtleHOMsList$.add($neighborsList$.getSubtleHOMs())
19: $restrictedList$.add($incumbentHOM$))
20: $bestHOM \leftarrow$ getBestHOM($neighborsList$)
21: **if** ($bestHOM$.getFitnessValue() $\geq$ $incumbentHOM$.getFitnessValue() && !$restrictedList$.contains($bestHOM$)) **then**
22: $incumbentHOM \leftarrow bestHOM$
23: $shouldRestart \leftarrow false$
24: **else**
25: $shouldRestart \leftarrow true$
26: **end if**
27: **end while**
28: **return** $subtleHOMsList$

- h_j^m is a neighbor of h_i^n such that $m = n$, or $m = n + 1$, or $m = n - 1$. Furthermore, one of the following conditions holds:

  1. $h_i^n \subset h_j^{n+1}$
  2. $h_j^{n-1} \subset h_i^n$
  3. $|h_i^n - h_j^n| = 1$ and $|h_j^n - h_i^n| = 1$

Figure 4.1 shows an example of each of the three cases with $n = 3$.

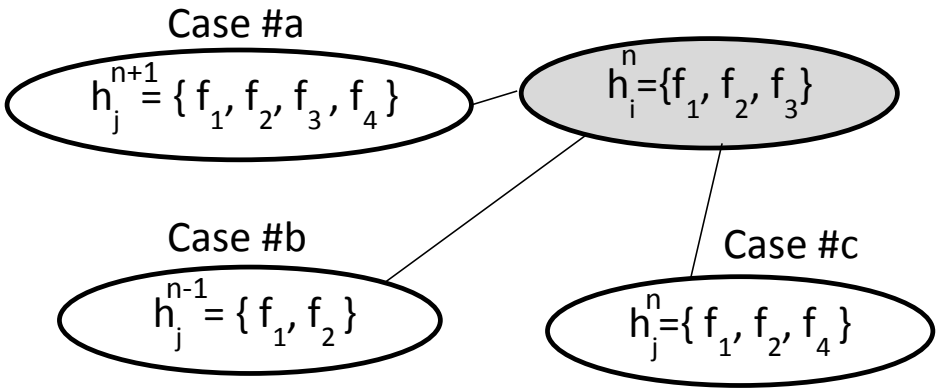


Figure 4.1: Example of neighboring HOMs

4- Evaluating neighboring HOMs

Each HOM in the $neighborsList$ is created, compiled, and executed against the test suite, its execution result is recorded, and its fitness value is assigned using Equation 4.3.

5- Iterations

After the HOMs in the $neighborsList$ are evaluated, Local Search looks for the best neighboring HOM that has an equal or higher fitness value than the incumbent HOM. The Local Search maintains a list of HOMs that have been selected as incumbent HOMs, which is called $restrictedList$. This list prevents the local Search from selecting the same incumbent HOM twice and allows the Local Search to explore different parts of the search space at each iteration. If a better neighboring HOM is found, it becomes the next incumbent HOM, and the process starts all over again. If no

better neighboring HOM is found, Local Search restarts by selecting a different SOM from the list of SOMs, $SOMsList$, to be the new incumbent HOM.

6- Stopping condition and output

Local Search maintains a list of all distinct, subtle HOMs ($subtleHOMsList$) that were found during the search process. The tester can define the maximum number of distinct HOMs that Local Search is allowed to explore, the time limit that Local Search is allowed to run for, and the number of required subtle HOMs that need to be found by Local Search.

4.4 Guided Local Search

We present two guided versions of the Local Search that have the same steps as Local Search. However, these techniques use different heuristics. The first version, called Data-Interaction Guided Local Search, utilizes program structural information to favor combining FOMs with data interactions because they are more likely to produce subtle HOMs. The second version, called Test-Case Guided Local Search, utilizes information about the test sets that kill each FOM for the program under test to avoid combining FOMs that are killed by totally different test sets because they are not expected to produce subtle HOMs.

4.4.1 Data-Interaction Guided Local Search

Data-Interaction Guided Local Search (DIGLS) generates a smaller set of neighboring HOMs than Local Search. It aims to avoid creating and evaluating HOMs that are expected to be entirely coupled to their constituent FOMs.

During the preliminary empirical studies that set the foundation for this work [16, 17], we found that the majority of HOMs that have a fitness value greater than zero (promising and subtle HOMs) are those where the mutated statements shared a common variable, such as local, instance, static, and method parameters of classes, and variables and parameters defined in aspects. For example, for the Kettle Program, we found that in 96% of HOMs with fitness value greater than zero, at least two of their mutated statements shared at least one common instant variable. On the

other hand, 60% of the HOMs with a fitness value equal to zero, their mutated statements shared some common instant variables. Although the presence of a common variable among the mutated statements in the HOM cannot guarantee that the HOM is not entirely coupled, the initial data we obtained for most subject programs motivated us to utilize this knowledge in the search process.

When an incumbent HOM is selected and evaluated, DIGLS explores only neighboring HOMs where at least two of their mutated statements share at least one common variable. That is, there exists at least one pair of mutated statements where both statements read and/or write to at least one program variable. DIGLS considers all type of variables.

In addition, DIGLS maintains a list of all SOMs that were found and evaluated during the search process. This list contains the fitness values for the evaluated SOMs. DIGLS uses the list of SOMs to further reduce the size of the neighborhood by ignoring all HOMs that contain pairs of mutated statements that share a common variable when all these pairs were found to result in entirely coupled SOMs.

4.4.2 Test-Case Guided Local Search

Test-Case Guided Local Search (TCGLS) generates a smaller set of neighboring HOMs than Local Search. When an incumbent HOM is selected and evaluated, TCGLS explores only neighboring HOMs that their constituent FOMs are killed by similar test cases. That is, there exists at least one pair of constituent FOMs that are killed by at least one common test case. TCGLS uses the heuristic that FOMs that are killed by test sets that do not intersect (i.e. overlap) cannot be combined in a way to mask one another. This is because such FOMs usually represent independent faults that cannot interact.

TCGLS maintains a list of all SOMs that were found and evaluated during the search process. TCGLS uses this list to further reduce the size of the neighborhood by ignoring all HOMs who contain pairs of FOMs that are killed by common test cases when all these pairs were found to result in entirely coupled SOMs.

4.5 Restricted Random Search

Restricted Random Search (RRS) 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 suite. RRS iterates the process of generating an HOM ($randomHOM$) with a set of randomly selected FOMs from the list of FOMs. The pseudocode for RRS is provided in Algorithm 3.

Algorithm 3 Restricted Random Search

```
Require: FOMsList, maxHOMDegree
 1: subtleHOMsList ← ∅
 2: while not (Termination_Condition()) do
 3:     randomHOM ← generateHOM(FOMsList, maxHOMDegree)
 4:     executeMutant(randomHOM)
 5:     evaluateFitness(randomHOM)
 6:     if (randomHOM.isSubtle()) then
 7:         subtleHOMsList.add(randomHOM)
 8:     end if
 9: end while
10: return subtleHOMsList
```

RRS uses a configurable parameter, $maxHOMDegree$, to allow it to control the maximum degree of explored HOMs. This parameter allows RRS to limit the search to a smaller part of the search space where subtle HOMs are more likely to be found. Our preliminary investigation [16, 18] showed that most of the discovered subtle HOMs were of lower degrees (less than six).

Setting the configurable parameter, $maxHOMDegree$, to *null* allows RRS to be unrestricted and capable of exploring any candidate HOM in the search space. The maximum degree of explored HOMs in this case is controlled by the maximum number of FOMs for the program under test that can be combined together.

Each generated $randomHOM$ is compiled and executed, its execution result is recorded, and its fitness value is calculated. If the $randomHOM$ is subtle, it is stored in $subtleHOMsList$. RRS repeats this process until a stopping condition is reached. The stopping conditions, which can be configured, include the maximum number of distinct HOMs RRS is allowed to explore, time limit, and the number of subtle HOMs required to be found.

4.6 Restricted Enumeration Search

Restricted Enumeration Search (RES) examines candidate HOMs in the search space in a predefined sequence until a defined stopping condition is met. Restricted Enumeration Search takes as input the list of FOMs for the program under test. It creates and evaluates all candidate SOMs, followed by third order mutants, and so on until a stopping condition is met.

Algorithm 4 Restricted Enumeration Search

```
Require: FOMsFile
 1: subtleHOMsList ← ∅
 2: degree ← 2
 3: HOM ← null
 4: while not (Termination_Condition()) do
 5:     HOM ← getNewHOM(degree)
 6:     if HOM.isNull() then
 7:         degree ← degree + 1
 8:         HOM ← getNewHOM(degree)
 9:     end if
10:     executeMutant(HOM)
11:     evaluateFitness(HOM)
12:     if (HOM.isSubtle()) then
13:         subtleHOMsList.add(HOM)
14:     end if
15: end while
16: return subtleHOMsList
```

The method $getNewHOM(degree)$ randomly selects a new HOM based on the required degree. This method does not select the same HOM more than once and returns $Null$ if no new HOM of the specified degree is found. Restricted Enumeration Search is designed to start the search in the space of lower degree HOMs because the majority of subtle HOMs found in an initial experimental evaluation were HOMs of lower degrees [16, 18]. Further, adding more faults to an HOM in general makes it easier to be killed and that makes subtle HOMs of higher degrees harder to find than subtle HOMs of lower degrees.

Chapter 5

Implementation

This chapter presents the design and use of the Higher Order Mutation Testing tool for AspectJ and Java programs, called HOMAJ. The goal of HOMAJ is to automate the (1) generation of FOMs and HOMs, (2) compilation and execution of FOMs and HOMs against the given test suite, and (3) finding subtle HOMs. The process of finding subtle HOMs for a given program requires as input a set of non-equivalent FOMs and a test suite that contains test cases that kill all the FOMs in that set.

HOMAJ can be used by researchers to perform studies involving higher order mutation. For example, HOMAJ can classify HOMs based on their coupling and subsumption relations with FOMs [8] and their construction approaches [16]. To promote the use of higher order mutation testing among researchers and practitioners, HOMAJ is designed with flexibility in mind; new evaluation measures and search techniques can be easily added.

5.1 Higher Order Mutation Testing Tool

HOMAJ consists of five main components. Figure 5.1 shows the architecture and main components of HOMAJ. Third-party components, which are used by the main components, are marked with a triangle. In the remainder of this chapter we describe the components and their functionality.

5.1.1 I/O Services

This component is responsible for receiving inputs from testers and presenting outputs. HOMAJ takes as inputs (1) the program under test, (2) JUnit test suites, and (3) a set of configuration parameters for the search technique selected by the tester.

Using the sub-component “Program Setup Services”, HOMAJ creates a set of folders and sub-folders to maintain information about the program under test. For each program, HOMAJ maintains the files of the original source code, the files of the given test suites, XML files that

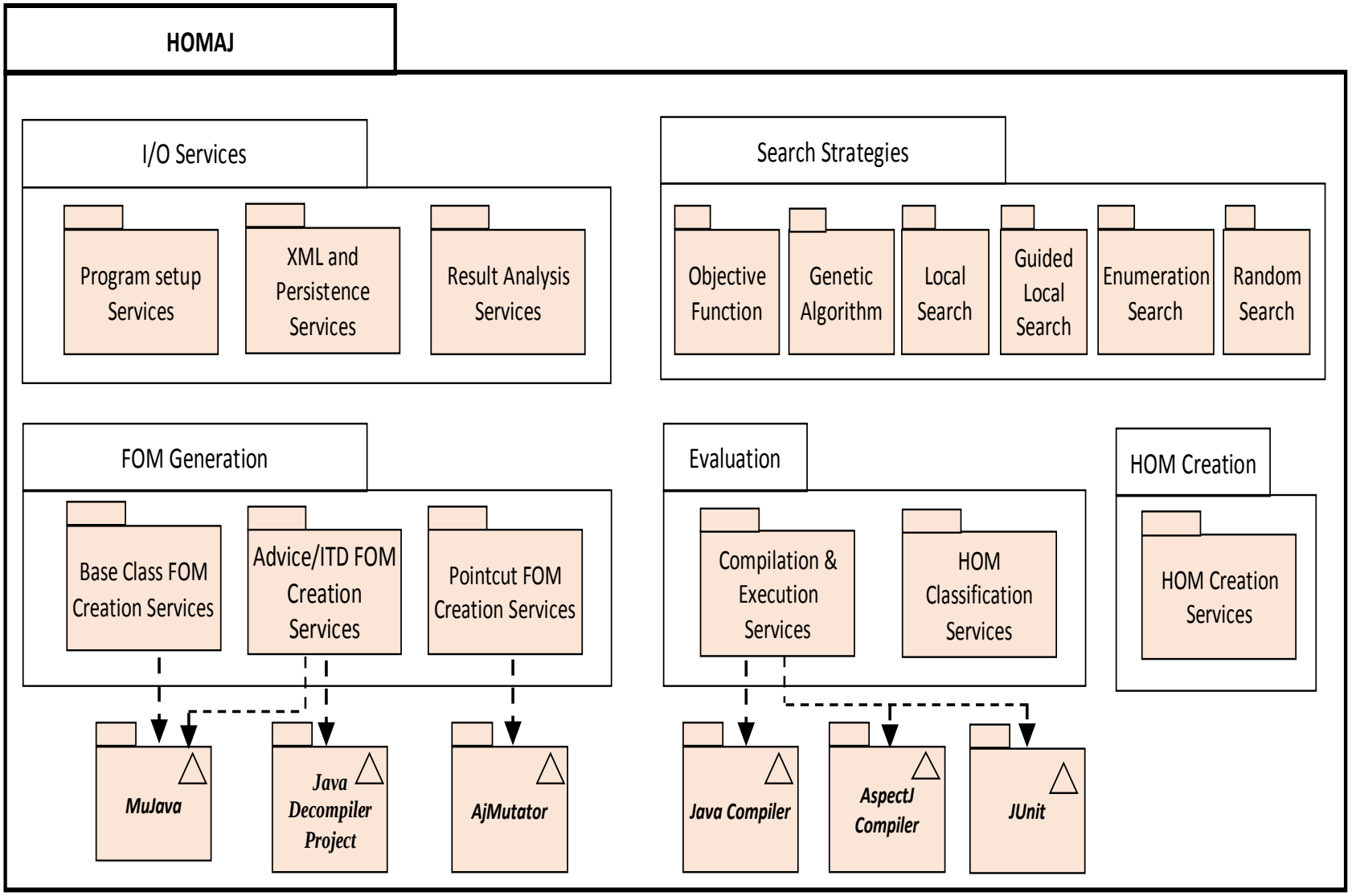


Figure 5.1: Architecture of HOMAJ

contain metadata about the generated FOMs and subtle HOMs, and files that contain the execution and classification results of HOMs. The sub-component "XML and Persistence Services" is responsible for maintaining and persisting the XML records. The sub-component "Result Analysis Services" is responsible for running queries on the HOM execution results and producing reports.

5.1.2 FOM Generation

HOMAJ utilizes all the operators of two available traditional mutation testing tools, MuJava [30] and AjMutator [33], to generate FOMs. MuJava generates both traditional method-level and class-level FOMs for Java. It can also compile and execute these mutants using a Java runtime environment. AjMutator implements only pointcut mutation operators out of all the AspectJ mutation operators proposed by Ferrari et al. [73]. It can compile the mutants and execute test cases in conjunction with an AspectJ compiler and a Java runtime environment.

HOMAJ uses the public methods in the `MuJava.MutationSystem` class to create Java base class FOMs. This process requires the name of the class to be mutated, the source and result folders, and the set of mutation operators to be applied. After creating the mutants, MuJava generates a log file for each mutated class that contains metadata, such as the applied mutation operator, the line number of the mutated statement, the mutated method or operator, and the performed mutation (i.e., the original code statement and the mutated code statement). HOMAJ extracts the information and produces a file that contains metadata records in XML, such that each record provides complete information about a generated FOM. Figure 5.2 shows a file containing three FOM metadata records.

To create pointcut FOMs from an aspect, the AjMutator command line interface is provided the names of the original program source folder and the result folder. AjMutator log files do not provide sufficient information about the generated mutants. We developed functionality in HOMAJ to iterate through the generated mutant folders and sub-folders, and extract the necessary mutant metadata. Some of the extracted information, such as the applied mutation operator, line number of the mutated statement, mutated pointcut, and mutated aspect name are obtained from the generated mutant's path, which is obtained from the AjMutator log files. The performed mutation (i.e., the

```
<FOM> <!--Base class FOM -->
   <Mutation Operator> AOIS</Mutation Operator>
   <Line Number>45 </Line Number>
   <Mutated Method>char_getType() </Mutated Method>
   <Mutation>movieType => movieType++ </Mutation>
   <Class>Movie.java </Class>
   <Mutant Path> .\movieRental\...\AOIS_7\Movie.java </Mutant Path>
</FOM>
<FOM> <!--Pointcut Descriptor FOM -->
     <Mutation Operator>PCCE </Mutation Operator>
     <Line Number>13 </Line Number>
     <Mutated Pointcut>pointcut newCustomer()  </Mutated Pointcut>
     <Mutation> execution(Customer.new(..)) => call(Customer.new(..))
       </Mutation>
     <Aspect>Updates.aj </Aspect>
     <Mutant Path> .\movieRental\....\PCCE000\Updates.aj</Mutant Path>
</FOM>
<FOM> <!--Aspect Advice FOM -->
   <Mutation Operator>AORB </Mutation Operator>
   <Line Number>56</Line Number>
   <Mutated Advice> void around(double charges) </Mutated Advice>
   <Mutation>charges -= (charges*1/4) => charges += (charges*1/4)
     </Mutation>
   <Aspect>Updates.aj </Aspect>
   <Mutant Path> .\movieRental\....\AORB56\Updates.aj</Mutant Path>
</FOM>
```

Figure 5.2: FOM Metadata file Example

original code statement and the mutated code statement) is obtained from both the mutated file and the original file. After extracting all the information, HOMAJ produces metadata records for the pointcut FOMs as shown in Figure 5.2.

To mutate aspect advice and inter-type declarations, we used a technique previously used by Wedyan and Ghosh [11]. The technique requires the use of a Java decompiler. We used *The Java Decompiler Project* [112] to decompile AspectJ files into Java files on which MuJava can be used. HOMAJ copies the mutated statements into the original aspect files to produce FOMs. The metadata records for the aspect advice and inter-type declaration FOMs are produced in the same way as for base class FOMs.

5.1.3 HOM Creation

An HOM metadata record is generated by combining two or more selected FOM metadata records. The HOM is created by first creating a folder using information from the HOM metadata record and then copying all class and aspect files of the program, statement by statement, while replacing each statement corresponding to a selected FOM record with the mutated statement of the FOM (obtained from the mutation field in the FOM metadata record). HOMAJ supports multiple faults in a single statement if the selected FOMs mutate different tokens in that statement. HOMAJ can be configured to create HOMs by using a particular HOM construction approach.

5.1.4 Search Strategies

The component "Search Strategies" includes the implementations of the search techniques for finding subtle HOMs. Currently HOMAJ implements all the search techniques described in Chapter 4.

HOMAJ uses XML files and records to set the values of the configurable parameters for each of the search techniques. For example, the tester can use the Genetic Algorithm configuration file to set the number of crossover points, mutation rate, and the stopping condition.

5.1.5 Evaluation

This component is responsible for the compilation and execution of both FOMs and HOMs as well as the classification of HOMs. HOMAJ requires Java and AspectJ compilers to compile mutants and uses JUnit to execute test cases on mutants.

The process of compiling and executing a large number of HOMs against the given test suite is computationally expensive. For example, our preliminary studies [16, 17] showed that an unrestricted Random Search took 27 hours to explore 50,000 HOMs. Local Search and Genetic Algorithm took 16 and 15 hours respectively. Both Genetic Algorithm and Local Search explored more duplicate HOMs than Random Search. Duplicate HOMs are evaluated only once. Thus, their execution time was less than that of Random Search. On the other hand, when HOM compilation and execution time was not included, Random Search, Local Search, and Genetic Algorithm, took

13.7, 16.8, and 22.5 seconds respectively. Both Local Search and Genetic Algorithm use more operators to create HOMs, such as neighborhood graph, crossover, and mutation, during the search, which results in their higher execution time.

Because of the high computational cost of compiling and executing HOMs, we extended HOMAJ to use the utility program, *Make* [113], and perform selective compilation of Java and AspectJ files. Make automates the process of building executable programs and determining which pieces of a program need to be recompiled. Make reads a file, called *makefile*, that specifies how to derive and compile the target program. Make uses Java and AspectJ compilers to compile mutants and it only recompile the files that had changed since the last compilation.

To compile and execute mutants, HOMAJ creates an execution folder for the program under test and copies packages and classes of the original source code to that folder. An HOM is created by modifying the source files in the execution folder and changing the program statements corresponding to the constituent FOMs.

For each program, HOMAJ automatically generates and executes a makefile to compile the HOMs. HOMAJ runs the $java$ command to execute each HOM. The execution result of an HOM includes a list of identifiers of all the test cases that kill the HOM. HOMAJ maintains a list of all compiled and executed HOMs. This list is used to check for duplicates, and to compile and execute each distinct HOM only once.

HOMAJ uses results obtained from executing both HOMs and FOMs to classify the HOMs based on the definitions of subsumption and coupling relationships proposed by Jia and Harman [8].

Chapter 6

Experimental Setup

This chapter presents the setup for the empirical studies conducted in this dissertation. We present the subject programs used in the studies and describe the approach used to develop the test suites for these programs. We also present the configurations that were selected for the search techniques.

6.1 Subject Programs

We used five Java programs that varied in size and implemented various Java constructs. The selected programs contain various constructs, such as loops and multithreading. The programs also implement various Object-Oriented concepts, such as interfaces and inheritance. Table 6.1 summarizes the information about the subject programs, their sizes, and size of their test suites. The programs are as follows:

1. *Coordinate*:

   The Coordinate Program shown in Appendix 1 simulates a rectangular grid for the world, such grids are commonly used for games. It contains two classes, *Coordinate.java* and *World.java*. The program allows the world to wrap from top to bottom and from left to right.

2. *Roman*:

   The Roman program shown in Appendix 2 converts Roman numbers to Hindu-Arabic numbers and vice versa. It contains two classes *Roman.java* and *InvalidRomanNumberException.java*.

3. *Cruise Control*:

   The Cruise Control program [114] simulates a car and its cruise and speed controllers. We used two versions of the cruise Control program. One used Java and the other one used

Table 6.1: Subject Programs

Subject program	Type	LOC	# of FOMs	# of classes	# of aspects	# of advices	# of pointcuts	# of ITDs	# of test cases
Coordinate	Java	121	242	2	0	0	0	0	14
Roman Numbers	Java	179	208	2	0	0	0	0	11
Cruise Control	Java	917	129	6	0	0	0	0	18
Elevator	Java	1046	249	17	0	0	0	0	14
XStream	Java	14,388	1216	318	0	0	0	0	96
Kettle	AspectJ	125	125	1	2	4	3	2	12
Movie Rental	AspectJ	191	316	3	1	8	9	0	15
Banking	AspectJ	243	92	2	2	2	2	1	9
Telecom	AspectJ	928	152	10	3	9	12	9	10
Cruise Control	AspectJ	1008	215	9	3	18	19	15	26

AspectJ. The Java version contains six classes that implement the main features of the car and its cruise and speed controllers, such as starting the car, setting and resetting the cruise and speed controls.

4. *Elevator*:

   The Elevator program [115] contains 17 classes that simulate a number of elevators servicing a number of floors. An elevator accepts travel requests from one floor to another. The program uses multi-threading.

5. *XStream*:

   The XStream program [116] is an open source library for object serialization. It contains 318 classes that serialize Java objects into XML records. XStream can also be used to transform XML records back to Java objects.

We used five AspectJ programs that varied in size and implemented various Java and AspectJ constructs. They contain before, after, and around advices, inter-type declarations, as well as primitive and composed pointcuts. These programs also contain base classes that contain Java constructs to which we could apply MuJava operators. Table 6.1 provides information about these programs, their sizes, and size of their test suites. The AspectJ programs are as follows.

1. *Kettle*:

   The Kettle program [117] 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.

2. *Movie Rental*:

   The Movie Rental program shown in Appendix 3 simulates some of the functionality of a movie rental kiosk. It 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.

3. *Banking*:

   The Banking program [34] is a bank account management system that contains two classes, *Customer* and *Account*, and two aspects, *MinimumBalance* and *OverdraftProtection*. The aspects implements additional functionalities for checking the balance and controlling the overdraft fee.

4. *Telecom*:

   The Telecom program [12] simulates a telephone system. It allows customers to make, accept, merge, and hang up both local and long distance calls. *Telecom* contains ten classes to perform the basic telephone system functionalities and three aspects that implement functionalities for measuring the call durations, maintaining call logs, and generating bills.

5. *Cruise Control*:

   This is an AspectJ version of the Cruise Control program [114]. It contains three aspects that enforce the pre and post-conditions for cruise and speed controllers, which are implemented in nine Java classes.

6.2 Test Sets

We developed our own random test case generator to develop a large pool of JUnit test cases for each subject program. 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. 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 the non-equivalent FOMs.

For each subject program, we generated a test suite by randomly selecting test cases from the large pool of JUnit test cases. Each test suite contains test cases that achieved statement coverage and killed all non-equivalent FOMs.

Unlike other programs, the XStream program came with 60 test suites, which contained a large number of test cases that achieved statement coverage. MuJava generated more than 40,000 FOMs

for the XStream program and all of the test cases in the 60 test suites killed only 588 FOMs. We manually developed another 18 test cases that killed another 628 FOMs. We eventually used 1216 FOMs and all 63 test suites that contained 96 test cases for the XStream program.

6.3 Configuration of the Search Techniques

We used 64-bit Linux machines with Intel Core^{TM}4x3.3G and 8 Gb memory. The configuration used for each technique can affect its performance. In the early stages of this study we ran the search techniques with different configurations and finally selected the configurations that produced the highest number of subtle HOMs.

Due to the stochastic nature of the proposed search techniques, we ran each technique 30 times per subject program and calculated different statistical measures to compare the effectiveness of the search techniques. The search techniques were configured in the same way for all subject programs as follows.

1. *Objective Function*: the value of α of the objective function was set at 0.75.

2. *Stopping Condition*: we used the stopping condition of exploring 50,000 distinct HOMs.

3. *Genetic Algorithm*

   - Number of crossover points was set at two and the number of elite HOMs was set at 15.
   - Population size was set at 600.
   - Mutation rate was set at 0.01 of the number of mutable statements of the program.
   - HOMs of the first population degrees ranged from two to three.

4. *Restricted Random Search*: the maximum HOM degree allowed was set at six.

5. *Restricted Enumeration Search*: restricted Enumeration Search does not have any configurable parameters besides the stopping condition.

Chapter 7

Measuring the Relative Effectiveness of the Search Techniques

This chapter presents a study to measure the effectiveness of the search techniques in terms of their ability to find subtle HOMs. First, we present the research questions motivating the study. Second, we present the general findings for each research question. Last, we discuss the results for each subject program.

7.1 Research Questions

This section presents three research questions and the metrics used to measure the effectiveness of the search techniques.

RQ1: What is the relative effectiveness of the search technique in terms of their ability to find subtle HOMs?

We measured the effectiveness in terms of the average number of distinct, subtle HOMs that were found by each techniques. We compared the effectiveness of the search techniques to determine which technique can find a higher number of distinct, subtle HOMs. A higher number of distinct, subtle HOMs can be more beneficial for improving the fault-detection effectiveness of test suites.

In addition, we investigated the variation in the ability of search techniques to find subtle HOMs. We calculated the maximum, average, standard deviation, median, and minimum number of distinct, subtle HOMs that were found over the 30 runs for each search technique. We used box plots to show the distribution of the number of distinct, subtle HOMs that were found by each technique. We also used the Analysis of Variance (ANOVA) test to analyze the number of subtle HOMs that were found by each technique and determine if there is a significant difference between the ability of the search techniques to find subtle HOMs.

According to Harman et al. [23], a pure random search technique can be used as a base line to validate the use of search-based software engineering techniques. Restricted Random Search is not purely random and our initial experimental evaluation [16] showed that Restricted Random Search found a higher average number of subtle HOMs than a pure random search technique. Nonetheless, we used Restricted Random Search as a base line measure for the other five techniques. That is, we consider Local Search, both the Guided Local Search techniques, the Genetic Algorithm, and Restricted Enumeration Search to be effective at finding subtle HOMs if they can find a higher average number of distinct, subtle HOMs than Restricted Random Search.

We also compared the average number of subtle HOMs that were found by Local Search, both the Guided Local Search techniques, and the Genetic Algorithm with the average number of subtle HOMs that were found by Restricted Enumeration Search. The former four techniques are search based software engineering techniques while the latter technique is an exact method for finding subtle HOMs. The four techniques are not required to find a higher average number of subtle HOMs than Restricted Enumeration Search in order to be considered effective at finding subtle HOMs. However, we investigated whether the four techniques can find distinct, subtle HOMs that are not found by Restricted Enumeration Search after all the techniques have explored the same number of distinct HOMs.

RQ2: How does the relative effectiveness of the search techniques compare over time?

We investigated the growth in the average number of distinct, subtle HOMs that were found as the search techniques explored more distinct, subtle HOMs. The number of explored, distinct HOMs is considered a quasi-representation of the time taken by each search technique. The goal is to determine which of the search techniques is more effective when a tester has limited time and resources for finding subtle HOMs.

RQ3: How does the relative effectiveness of the search techniques compare with respect to the degree of subtle HOMs that were found?

We evaluated the effectiveness of the search techniques in terms of their ability to find subtle HOMs of specific degrees. For each program, we investigated the number of explored HOMs and subtle HOMs that were found with respect to the degree of the HOMs. The goal is to determine in what ways the search techniques explored the search space and how that affects the degree of the subtle HOMs that were found. We also used the Chi-square test to determine if there is a significant difference between the number of subtle HOMs that were found by the search techniques with respect to the degree of HOMs.

7.2 Results and Analysis

First, we highlight the general findings and present the answers for the three research questions. Second, we discuss and explain the results for each subject program.

7.2.1 RQ1: What is the relative effectiveness of the search technique in terms of their ability to find subtle HOMs?

Tables 7.1 through 7.19 show the maximum, average, standard deviation, median, and minimum number of distinct, subtle HOMs that were found over the 30 runs for each technique. ANOVA results 7.2 through 7.20 show the test result for each program with *alpha level* of 0.05. Box Plots 7.1 through 7.28 show the distribution of the number of distinct, subtle HOMs that were found over the 30 runs for each technique.

Although all six techniques found subtle HOMs for all subject programs, there is a significant difference in terms of the ability of the search techniques to find subtle HOMs. For all programs, the *P-value* was much lower than 0.05 and *F statistic* was higher than *F critical*.

Local Search, both the Guided Local Search techniques, the Genetic Algorithm, and Restricted Enumeration Search found higher average and median numbers of subtle HOMs than Restricted Random Search for all ten programs. This indicates that the five techniques were effective in terms of their ability to find subtle HOMs.

Local Search and both the Guided Local Search techniques were more effective than the other techniques at finding subtle HOMs. The average and median numbers of subtle HOMs that were found by Local Search were higher than the average and median numbers of subtle HOMs that were found by Restricted Enumeration Search for nine out of the ten programs. Data-Interaction Guided Local Search and Test-Case Guided Local Search were more effective than Restricted Enumeration Search for eight and seven programs, respectively. The combination of the fitness evaluation and the neighborhood graph (i.e. selection mechanism) seem to be a better strategy for finding subtle HOMs.

Data-Interaction Guided Local Search was more effective than Test-Case Guided Local Search. Data-Interaction Guided Local Search found a higher average number of subtle HOMs than Local Search for six programs, while Test-Case Guided Local Search found a higher average number of subtle HOMs than Local Search for four programs. Test-Case Guided Local Search found a higher average number of subtle HOMs than Data-Interaction Guided Local Search for only one program, Roman. For all other program, either Data-Interaction Guided Local Search found a higher average number of subtle HOMs than Test-Case Guided Local Search, which is the case for four programs, or both techniques found almost the same average number of subtle HOMs, which is the case for five programs. The maximum number of subtle HOMs that were found by Data-Interaction Guided Local Search was higher than the maximum number of subtle HOMs that were found by Test-Case Guided Local Search for all programs. The maximum number of subtle HOMs in Tables 7.1 to 7.19 represent the best result out of the 30 runs for each technique.

The Genetic Algorithm was more effective for AspectJ programs than for Java programs. It found a higher average number of subtle HOMs than Restricted Enumeration Search for four out of the five AspectJ programs and found a lower average number of subtle HOMs than Restricted Enumeration Search for four out of the five Java programs.

Although Restricted Random Search was the least effective at finding subtle HOMs, it found subtle HOMs for all programs. Limiting the search to the space of lower degree HOMs is a good strategy because it allowed Restricted Random Search to find a higher average number of subtle HOMs than pure random search [16].

7.2.2 RQ2: How does the relative effectiveness of the search techniques compare over time?

Line Charts 7.2 through 7.29 show the growth in the average number of distinct, subtle HOMs that were found as the search techniques explored more distinct HOMs. The vertical line in these charts shows where Restricted Enumeration Search finished enumerating all Second Order Mutants (SOMs) and started enumerating third order mutants.

For nine out of the ten subject programs, Restricted Enumeration Search enumerated all SOMs. Restricted Enumeration Search did not enumerate all third order mutants for any subject programs. This is because the number of third order mutants was much more than the number of HOMs that the search techniques were allowed to explore (50,000 distinct HOMs) for all subject programs.

Restricted Enumeration Search was more effective during the early stages of the search process than the latter stages. This is because Restricted Enumeration Search explores the space of SOMs first where more subtle HOMs are expected to be found.

For nine out of the ten programs, Data-Interaction Guided Local Search was more effective than Local Search during the early stages of the search process. For six out of the ten programs, Test-Case Guided Local Search was more effective than Local Search during the early stages of the search process.

The Genetic algorithm was less effective than Restricted Enumeration Search during the early stages of the search process for all programs. However, the Genetic Algorithm was eventually more effective than Restricted Enumeration Search for five programs. Restricted Random Search was less effective than Restricted Enumeration Search during all stages of the search process for all subject programs.

7.2.3 RQ3: How does the relative effectiveness of the search techniques compare with respect to the degree of subtle HOMs?

Column Charts 7.3 through 7.30 show the average number of the distinct, subtle HOMs and all explored HOMs with respect to their degrees.

For five programs, the Chi-Square test produced a *P-value* that is less than the significance level of 0.05. For such programs, there is a significant difference between the number of subtle HOMs that were found by the search techniques with respect to the degree of HOMs.

Our investigation showed that 89% of the subtle HOMs that were found for all programs were of second or third degrees. Subtle HOMs of higher degrees were harder to find because increasing the degree of an HOM by adding more FOMs makes it easier to kill in most cases, and that can cause the number of subtle HOMs of higher degrees to be low.

For nine out of the ten programs, Restricted Enumeration Search was able to find all subtle SOMs because it fully explored the space of SOMs. For programs with a small number of SOMs (e.g., Banking with 4065 SOMs) Restricted Enumeration Search was able to quickly find subtle HOMs. The XStream program contained about a half million SOMs, and thus Restricted Enumeration Search could not find all subtle SOMs.

Data-Interaction Guided Local Search was also effective at finding subtle SOMs. On average for nine programs, Data-Interaction Guided Local Search found 94% of the subtle SOMs while it explored only 16% of the space of SOMs. Local Search, the Genetic Algorithm, Test-Case Guided Local Search, and Restricted Random Search found 90%, 85%, 83%, and 55% of the subtle SOMs while exploring 80%, 64%, 34%, and 50% of the space of SOMs, respectively.

For all programs, Restricted Enumeration Search was less effective than Local Search, both the Guided Local Search techniques, and the Genetic Algorithm at finding subtle HOMs of degree three and higher. Although Restricted Enumeration Search explored more distinct third order mutants than the four techniques for five programs, it still found a lower average number of third order subtle HOMs than the four techniques.

7.2.4 Discussion

This section discusses and explains the results for each subject program.

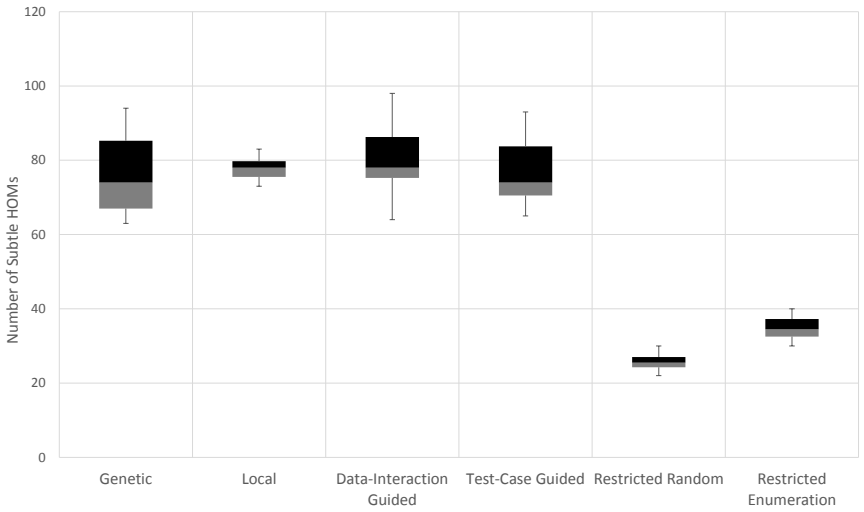
1- Cruise Control (Java) Program

Table 7.1: Number of subtle HOMs that were found for Cruise Control (Java)

Technique	# Subtle HOMs				
	Maximum	Average	St. Deviation	Median	Minimum
Genetic	94	76.1	10.8	74	63
Local	83	77.8	3	78	73
Data-Interaction Guided	98	80.7	10.7	78	64
Test-Case Guided	93	77.1	9.6	74	65
Restricted Random	30	25.8	2.3	25.5	22
Restricted Enumeration	40	34.8	3.4	34.5	30

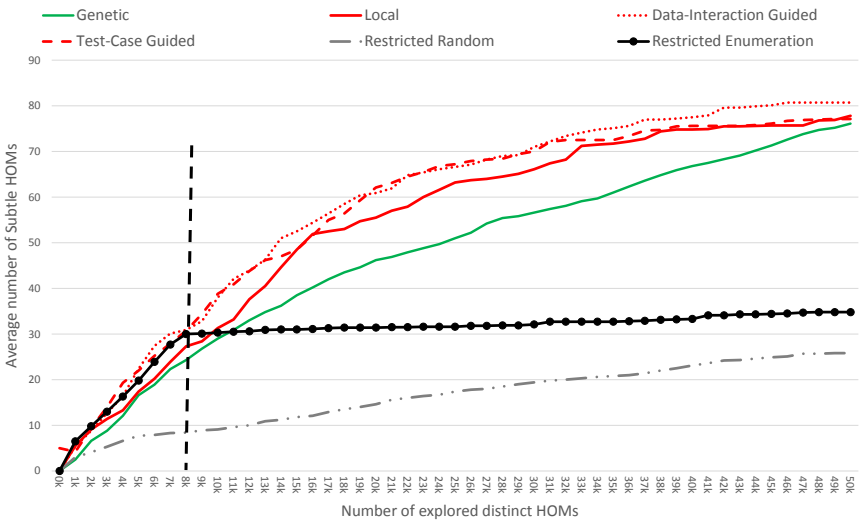
ANOVA Result 7.2: Cruise Control (Java)

Source of Variation	SS	df	MS	F statistic	P-value	F critical
Between Groups	30764.2	5	6152.8	106.3	1.11495E-26	2.4
Within Groups	3126.7	54	57.9			
Total	33890.9	59				



Box Plot 7.1: Distribution of the number of subtle HOMs that were found for Cruise Control (Java)

Table 7.1 and Box Plot 7.1 show that Local Search, both the Guided Local Search techniques, and the Genetic Algorithm were more effective than the other techniques for Cruise Control (Java)

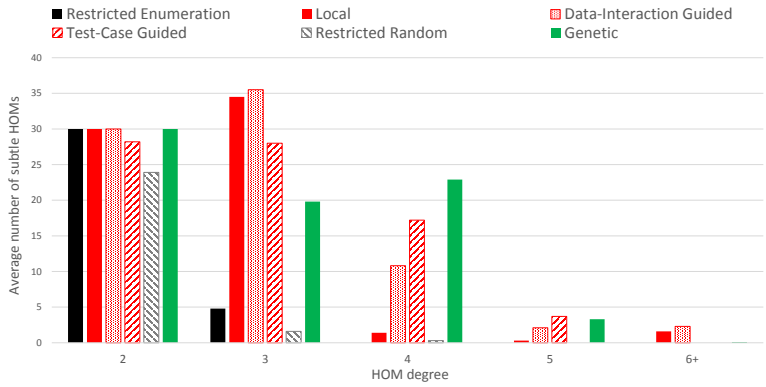


Line Chart 7.2: Growth in the average number of subtle HOMs that were found over the number of explored HOMs for Cruise Control (Java)

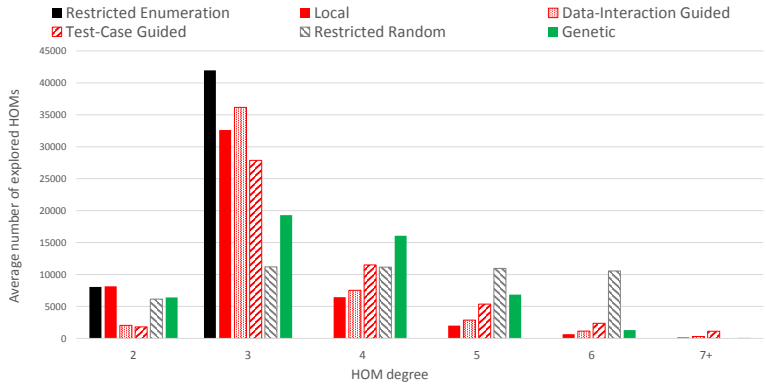
Program. The average number of subtle HOMs that were found by these four techniques is higher than the average number of subtle HOMs that were found by Restricted Enumeration Search. This is true for the median number of subtle HOMs as well. However, the average numbers of subtle HOMs that were found by these four techniques deviated by less than four subtle HOMs, which means the effectiveness of these four techniques is comparable.

The Chi-Square test produced a *P-value* of 1.4E-14, which is less than the significance level of 0.05. This indicates that there is a significant difference between the number of subtle HOMs that were found by the search techniques with respect to the degree of HOMs.

Column Chart 7.3 shows that subtle SOMs were relatively easy to find by all techniques. The Java Cruise Control program has a low number of SOMs (8042 HOMs) with respect to the other programs. Restricted Enumeration Search was able to quickly find all subtle SOMs. Local Search, Data-Interaction Guided Local Search, and the Genetic Algorithm found all the subtle SOMs. Test-Case Guided Local Search and Restricted Random Search found 28 and 24 out of the 30 subtle SOMs, respectively. However, Data-Interaction Guided Local Search and Test-Case Guided Local Search explored less than 25% of the space of SOMs.



(a) Distinct Subtle HOMs



(b) All Explored HOMs

Column Chart 7.3: Number of HOMs with respect to the degree for Cruise Control (Java)

2- Movie Rental Program

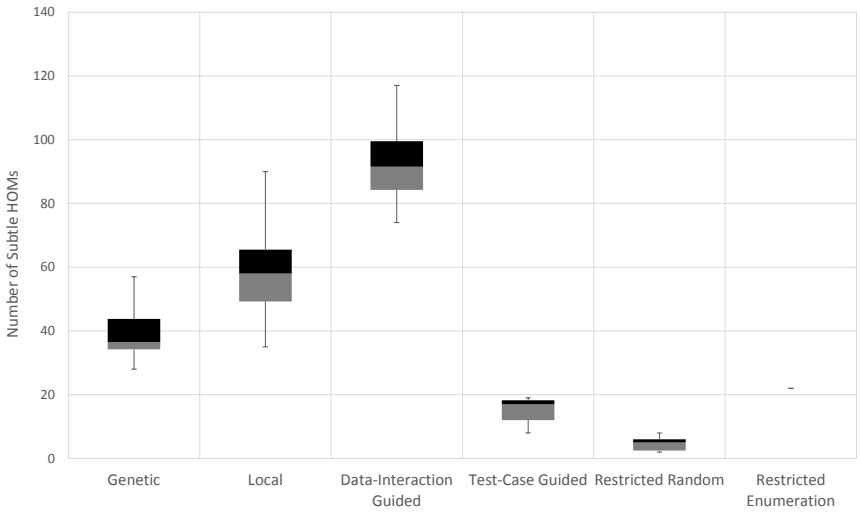
Table 7.3: Number of subtle HOMs that were found for Movie Rental

Technique	# Subtle HOMs				
	Maximum	Average	St. Deviation	Median	Minimum
Genetic	57	39.1	8.7	36.5	28
Local	90	59.8	17.6	58	35
Data-Interaction Guided	117	93.3	13.5	91.5	74
Test-Case Guided	19	15.3	4.1	17	8
Restricted Random	8	4.7	2.2	5	2
Restricted Enumeration	22	22	0	22	22

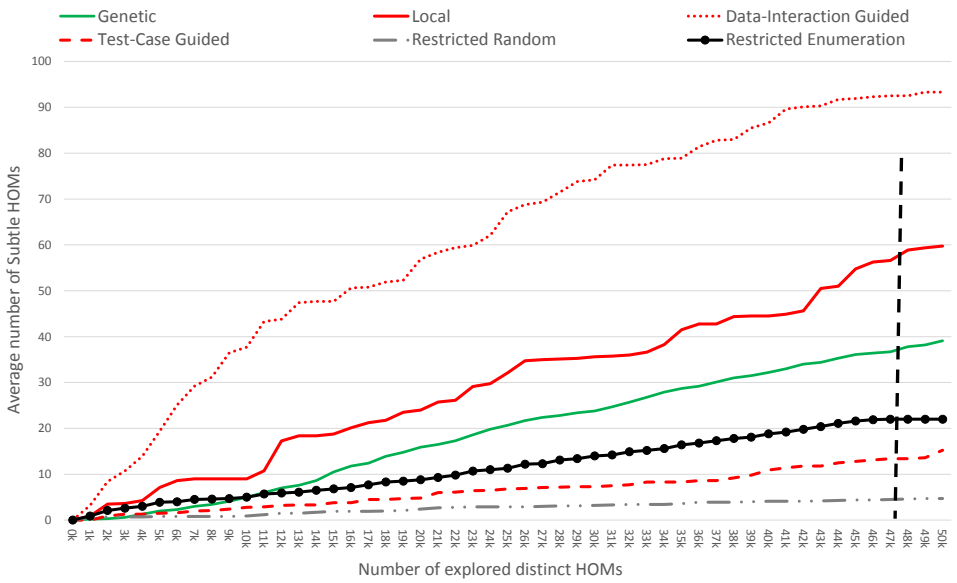
Table 7.3 and Box Plot 7.4 show that Data-Interaction Guided Local Search was more effective than the other techniques for the Movie Rental program. Line Chart 7.5 shows that Data-Interaction Guided Local Search was more effective than the other techniques during all stages of the search

ANOVA Result 7.4: Movie Rental

Source of Variation	SS	df	MS	F statistic	P-value	F critical
Between Groups	51726.7	5	10345.3	52	1.92433E-19	2.4
Within Groups	10734.3	54	198.8			
Total	62461	59				



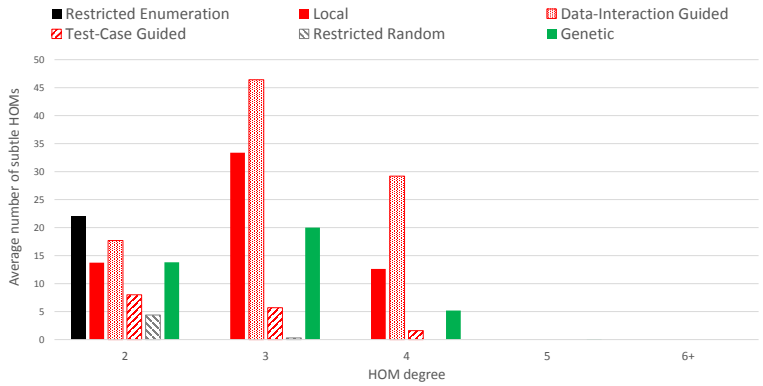
Box Plot 7.4: Distribution of the number of subtle HOMs that were found for Movie Rental



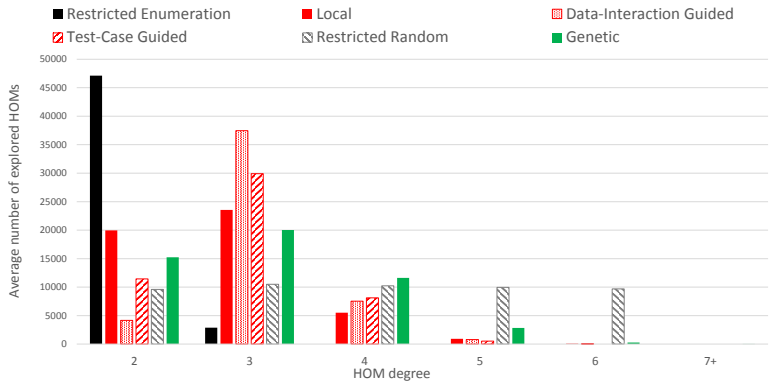
Line Chart 7.5: Growth in the average number of subtle HOMs that were found over the number of explored HOMs for Movie Rental

process. The Genetic Algorithm and Local Search were more effective than Restricted Enumeration Search.

Test-Case Guided Local Search was less effective than Restricted Enumeration Search during all stages of the search process. However, it was more effective than Restricted Random Search.



(a) Distinct Subtle HOMs



(b) All Explored HOMs

Column Chart 7.6: Number of HOMs with respect to the degree for Movie Rental

The Chi-Square test produced a *P-value* of 4.9E-12, which indicates that there is a significant difference between the number of subtle HOMs that were found by the search techniques with respect to the degree of HOMs.

Column Chart 7.6 shows that only Restricted Enumeration Search was able to find all the subtle SOMs (22 subtle SOMs). However, Restricted Enumeration Search did not find any subtle HOMs of degrees higher than two. The Movie Rental program has a large number of SOMs (47122 HOMs) and Restricted Enumeration Search explored only 2878 third order mutants.

Data-Interaction Guided Local Search, Local Search, and the Genetic Algorithm found more than 62% of the subtle SOMs that were found by Restricted Enumeration Search.

The three techniques found 169, 106, and 64 subtle HOMs of degrees three and higher, respectively, and that is why they were more effective than Restricted Enumeration Search.

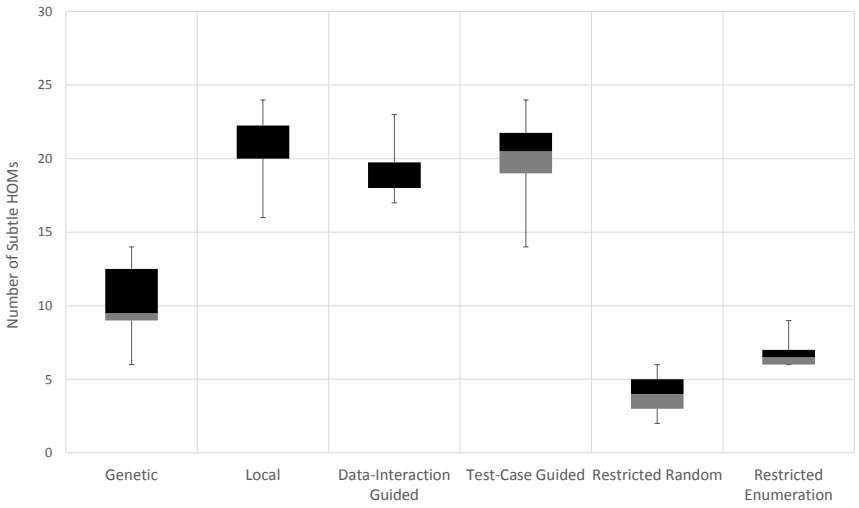
3- Telecom Program

Table 7.5: Number of subtle HOMs that were found for Telecom

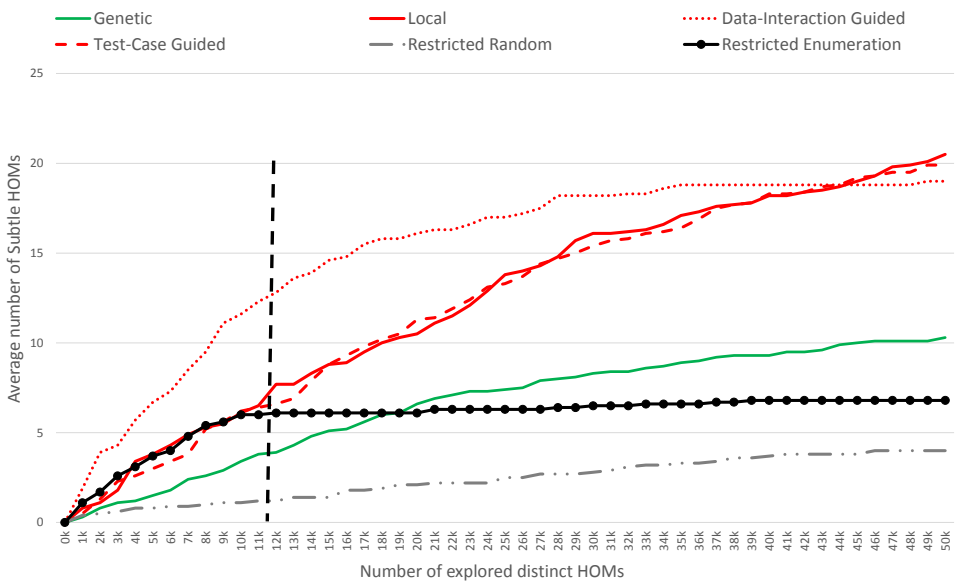
Technique	# Subtle HOMs				
	Maximum	Average	St. Deviation	Median	Minimum
Genetic	14	10.3	2.7	9.5	6
Local	24	20.5	2.3	20	16
Data-Interaction Guided	23	19	1.8	18	17
Test-Case Guided	24	19.9	2.8	20.5	14
Restricted Random	6	4	1.2	4	2
Restricted Enumeration	9	6.8	1	6.5	6

ANOVA Result 7.6: Telecom

Source of Variation	SS	df	MS	F statistic	P-value	F critical
Between Groups	2655.5	5	531.1	119.9	5.72273E-28	2.4
Within Groups	239.1	54	4.4			
Total	2894.6	59.0				



Box Plot 7.7: Distribution of the number of subtle HOMs that were found for Telecom



Line Chart 7.8: Growth in the average number of subtle HOMs that were found over the number of explored HOMs for Telecom

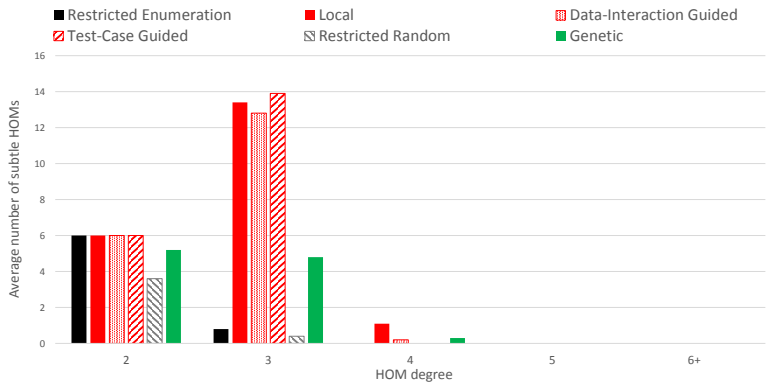
Table 7.5 and Box Plot 7.7 show that Local Search and both the Guided Local Search techniques were more effective than the other techniques for the Telecom program. The average number of subtle HOMs that were found by these techniques deviated by less than two subtle HOMs.

Line Chart 7.8 shows that Data-Interaction Guided Local Search was more effective than Restricted Enumeration Search during all stages of the search process. Although the Genetic Algorithm was less effective than Restricted Enumeration Search during the early stages of the search process, it eventually found a higher average and median numbers of subtle HOMs than Restricted Enumeration Search.

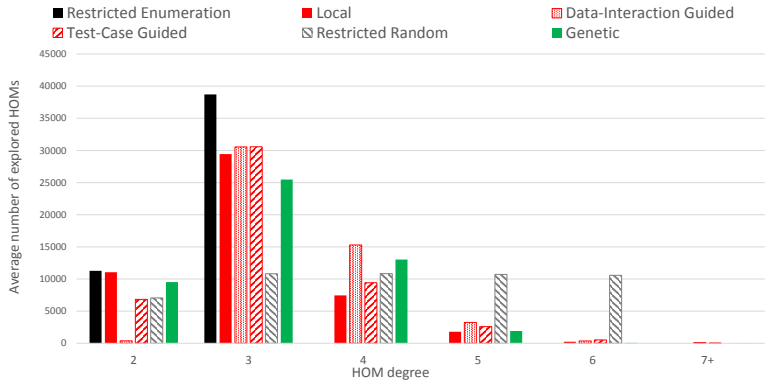
Telecom is a medium size program with respect to the other programs and it has 928 lines of code, ten classes, and three aspects. However, the number of subtle HOMs that were found for this program was lower than all other programs. Restricted Enumeration Search found six subtle SOMs.

The Chi-Square test produced a *P-value* of 0.13, which is higher than the significance level of 0.05. This indicates that there is no significant difference between the number of subtle HOMs that were found by the search techniques with respect to the degree of HOMs.

Column Chart 7.9 shows that Local Search, Data-Interaction Guided Local Search, and Test-



(a) Distinct Subtle HOMs



(b) All Explored HOMs

Column Chart 7.9: Number of HOMs with respect to the degree for Telecom

Case Guided Local Search found all subtle SOMs. The three techniques were more effective than Restricted Enumeration Search because they found more subtle HOMs of degree three and higher (15, 14, and 13 respectively), which could not be found by Restricted Enumeration Search. However, while Local Search fully explored the space of SOMs, Data-Interaction Guided Local Search explored less than 4% of the space of SOMs. Test-Case Guided Local Search explored 60% of the same space.

4- Kettle Program

Table 7.7 and Box Plot 7.10 show that both the Guided Local Search techniques and Local Search were more effective than the other techniques for the Kettle program. The average number of subtle HOMs that were found by these techniques deviated by less than three subtle HOMs.

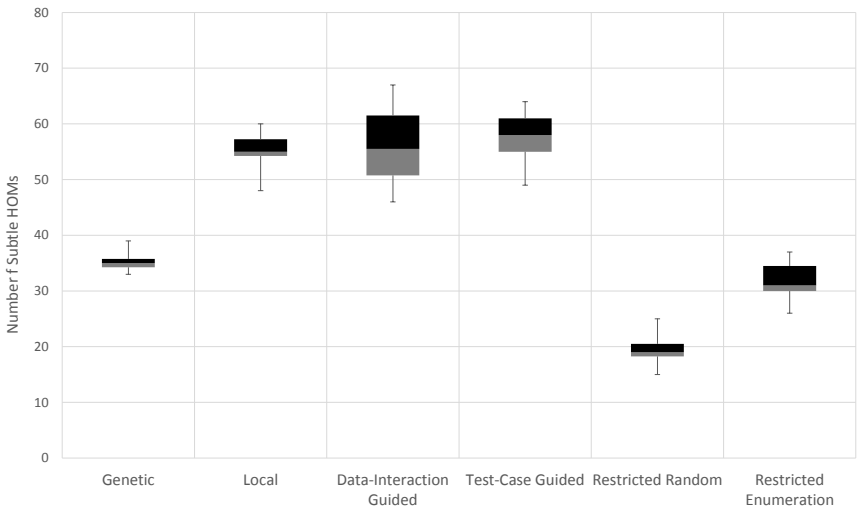
Restricted Enumeration Search was more effective than the other techniques during the early

Table 7.7: Number of subtle HOMs that were found for Kettle

Technique	# Subtle HOMs				
	Maximum	Average	St. Deviation	Median	Minimum
Genetic	39	35.3	1.9	35	33
Local	60	55.1	3.5	55	48
Data-Interaction Guided	67	56.1	7.6	55.5	46
Test-Case Guided	64	57.7	4.5	58	49
Restricted Random	25	19.7	2.9	19	15
Restricted Enumeration	37	31.5	3.6	31	26

ANOVA Result 7.8: Kettle

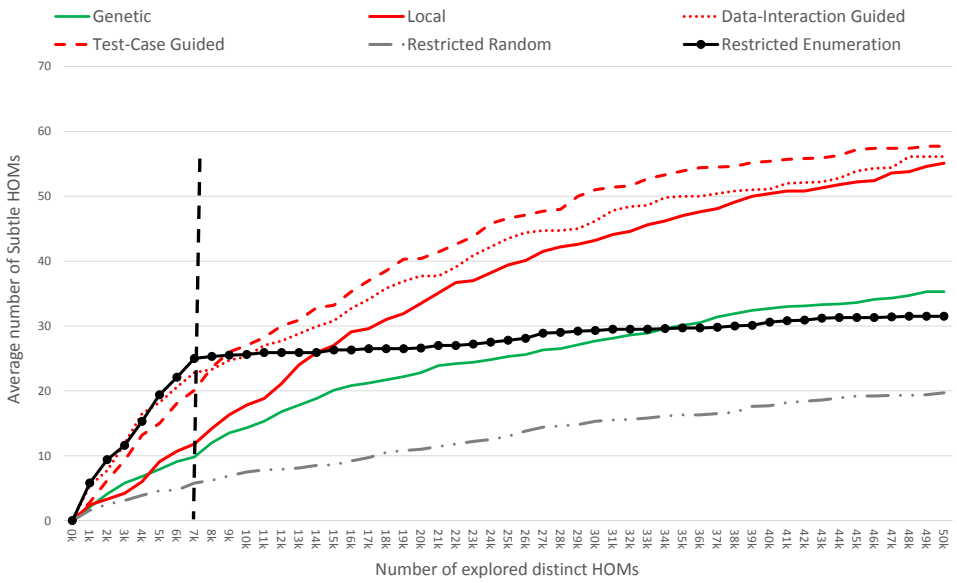
Source of Variation	SS	df	MS	F statistic	P-value	F critical
Between Groups	12674.1	5	2534.8	131.5	5.83389E-29	2.4
Within Groups	1040.6	54	19.3			
Total	13714.7	59				



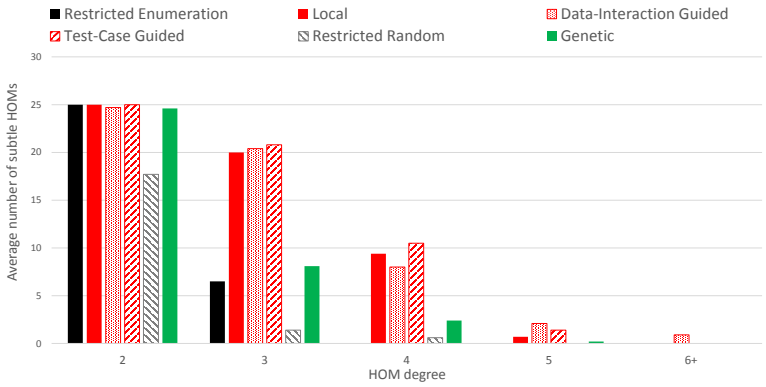
Box Plot 7.10: Distribution of the number of subtle HOMs that were found for Kettle

stages of the search process (see Line Chart 7.11) because it quickly explored the small space of SOMs (7617 SOMs) and found all subtle SOMs. Although the Genetic Algorithm was less effective than Restricted Enumeration Search during most stages of the search process, it eventually found a higher average number of subtle HOMs than Restricted Enumeration Search.

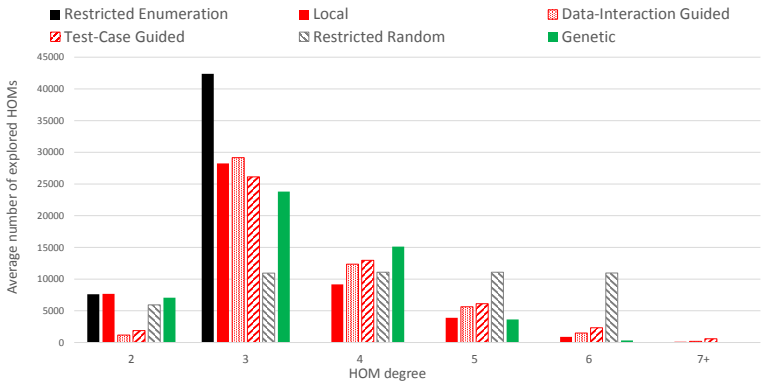
the Chi-Square test produced a *P-value* of 0.02, which indicates that there is a significant difference between the number of subtle HOMs that were found by the search techniques with respect to the degree of HOMs.



Line Chart 7.11: Growth in the average number of subtle HOMs that were found over the number of explored HOMs for Kettle



(a) Distinct Subtle HOMs



(b) All Explored HOMs

Column Chart 7.12: Number of HOMs with respect to the degree for Kettle

Column Chart 7.12 shows that Local Search and both the Guided Local Search techniques were more effective than Restricted Enumeration Search because each of the three techniques found all of the subtle SOMs that were found by Restricted Enumeration Search and more than 31 subtle HOMs of degree three and higher. The Genetic found all of the subtle SOMs and 10 more subtle HOMs of higher degrees.

5- Banking Program

Table 7.9: Number of subtle HOMs that were found for Banking

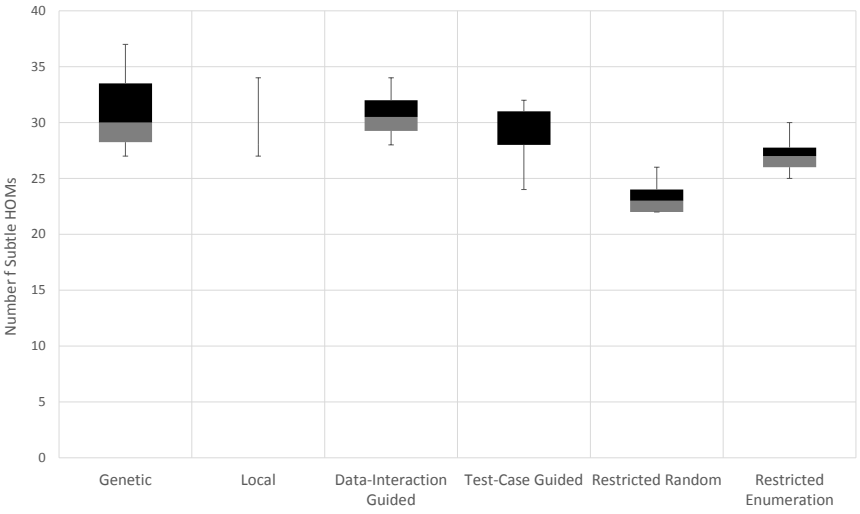
Technique	# Subtle HOMs				
	Maximum	Average	St. Deviation	Median	Minimum
Genetic	37	30.9	3.5	30	27
Local	34	29.2	2.6	28	27
Data-Interaction Guided	34	30.8	2.2	30.5	28
Test-Case Guided	32	28.9	2.6	28	24
Restricted Random	26	23.3	1.4	23	22
Restricted Enumeration	30	27.1	1.8	27	25

ANOVA Result 7.10: Banking

Source of Variation	SS	df	MS	F statistic	P-value	F critical
Between Groups	404.3	5	80.9	13.6	1.37149E-08	2.4
Within Groups	321.3	54	6			
Total	725.6	59				

Table 7.9 and Box Plot 7.13 show that subtle HOMs for the banking program were relatively easy to find by all techniques. The average number of subtle HOMs that were found by Local Search, both the Guided Local Search techniques, the Genetic Algorithm, and Restricted Enumeration Search deviated by less than four subtle HOMs. Same for the median number of subtle HOMs.

Line Chart 7.14 shows that Restricted Enumeration Search was more effective than the other techniques during the early stages of the search process. It quickly explored the small space of 4065 SOMs and found all subtle SOMs.

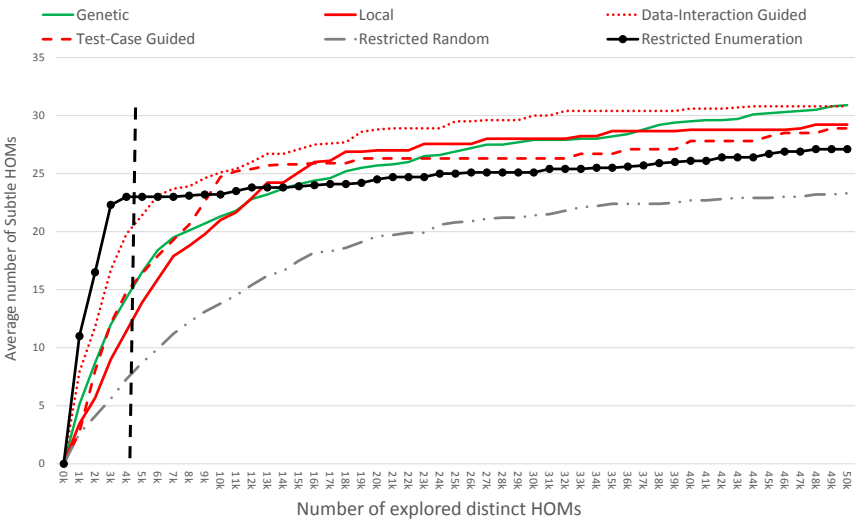


Box Plot 7.13: Distribution of the number of subtle HOMs that were found for Banking

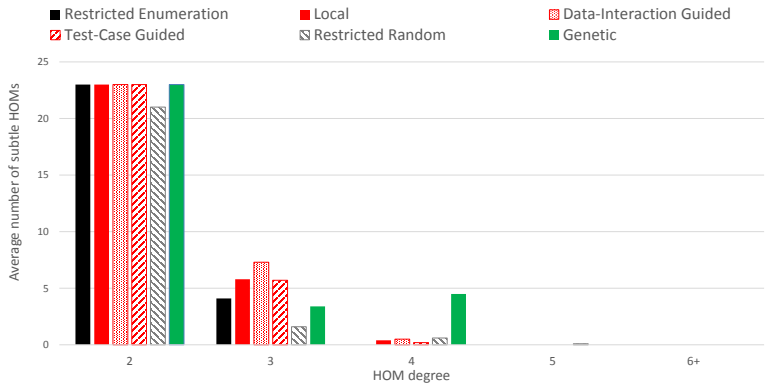
The Chi-Square test produced a *P-value* of 0.31, which indicates that there is no significant difference between the number of subtle HOMs that were found by the search techniques with respect to the degree of HOMs.

Column Chart 7.15 shows that all subtle SOMs were found by all techniques except for Restricted Random Search. Data-Interaction Guided Local search explored the lowest number of SOMs, 714 HOMs.

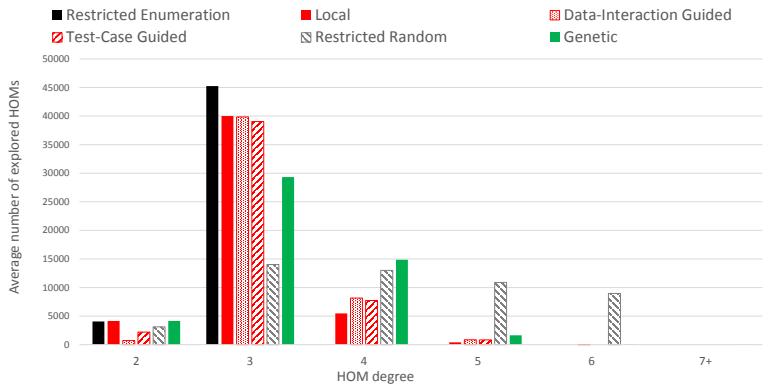
Although Restricted Enumeration Search explored more distinct HOMs of degree three than the other techniques, it found a lower number of subtle HOMs of degree three than Local Search and both the Guided Local Search techniques. The Genetic Algorithm was more effective than the other techniques at finding subtle HOMs of degree four and higher.



Line Chart 7.14: Growth in the average number of subtle HOMs that were found over the number of explored HOMs for Banking



(a) Distinct Subtle HOMs



(b) All Explored HOMs

Column Chart 7.15: Number of HOMs with respect to the degree for Banking

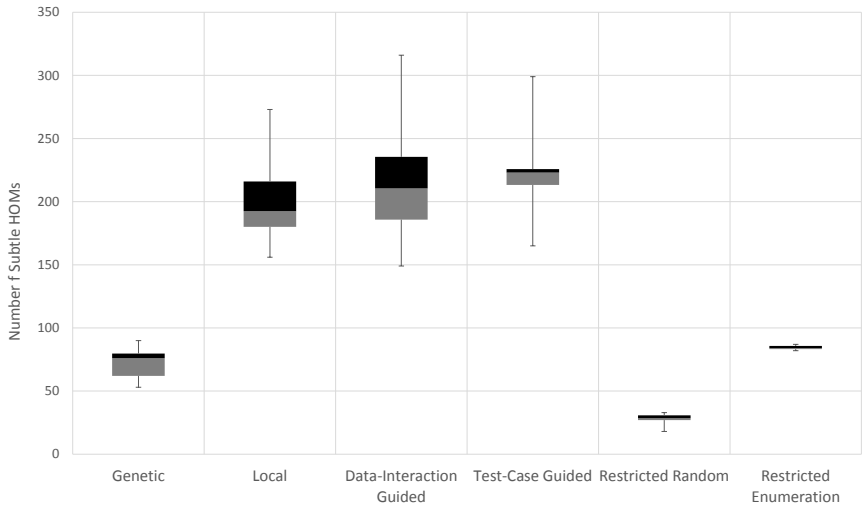
6- Coordinate Program

Table 7.11: Number of subtle HOMs that were found for Coordinate

Technique	# Subtle HOMs				
	Maximum	Average	St. Deviation	Median	Minimum
Genetic	90	72.4	13	76	53
Local	273	200.9	34.3	192.5	156
Data-Interaction Guided	316	213.8	50	210.5	149
Test-Case Guided	299	223.3	33.1	223	165
Restricted Random	33	27.5	4.9	28.5	18
Restricted Enumeration	87	84.4	1.8	84	82

ANOVA Result 7.12: Coordinate

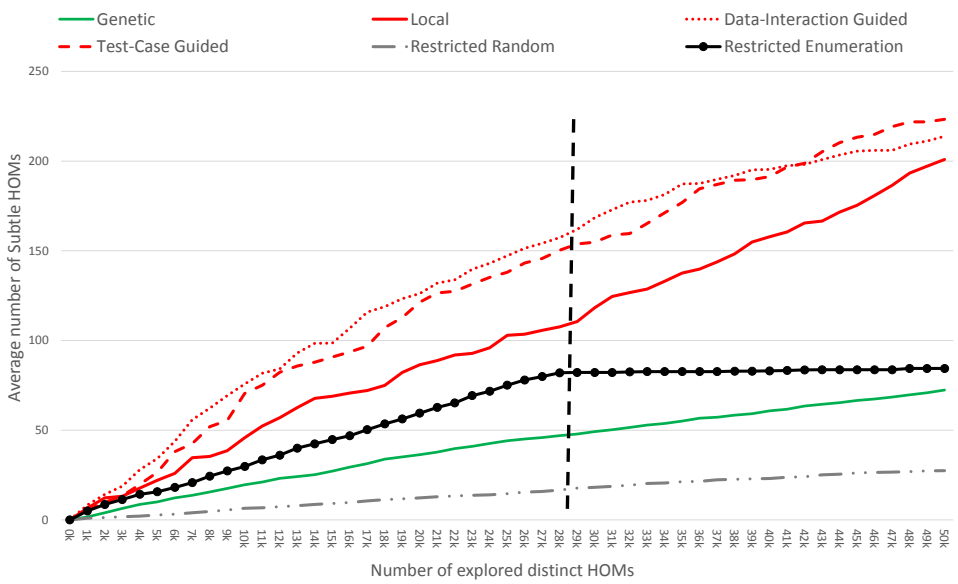
Source of Variation	SS	df	MS	F statistic	P-value	F critical
Between Groups	363593	5	72718.6	87.8	1.11749E-24	2.4
Within Groups	44725.9	54	828.3			
Total	408318.9	59				



Box Plot 7.16: Distribution of the number of subtle HOMs that were found for Coordinate

Table 7.11 and Box Plot 7.16 show that both the Guided Local Search techniques and Local Search were more effective than the other techniques for Coordinate, while both the Guided Local Search techniques were more effective than Local Search.

Although the Coordinate program is the smallest in terms of size (lines of code) with respect to the other program, the number of subtle HOMs that were found is higher than all other programs.

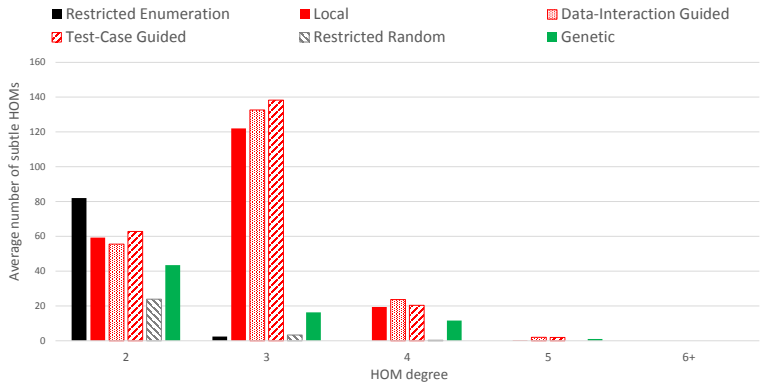


Line Chart 7.17: Growth in the average number of subtle HOMs that were found over the number of explored HOMs for Coordinate

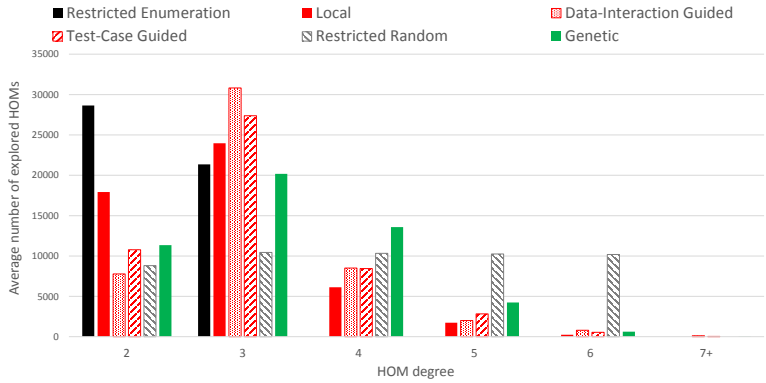
Line Chart 7.17 shows that both the Guided Local Search techniques and Local Search were more effective than Restricted Enumeration Search during all stages of the search process. The Genetic Algorithm was less effective than Restricted Enumeration Search during all stages of the search process.

The Chi-Square test produced a *P-value* of 1.3E-35, which indicates that there is a significant difference between the number of subtle HOMs that were found by the search techniques with respect to the degree of HOMs.

Column Chart 7.18 shows that none of the search techniques found all subtle SOMs that were found by Restricted Enumeration Search. However, only Restricted Enumeration Search explored all the space of SOMs. Both the Guided Local Search techniques and Local Search found a large number of third order subtle HOMs and that is why they were more effective than Restricted Enumeration Search. The average number of subtle HOMs that were found by Restricted Random Search was low compared to the other techniques.



(a) Distinct Subtle HOMs



(b) All Explored HOMs

Column Chart 7.18: Number of HOMs with respect to the degree for Coordinate

7- Elevator Program

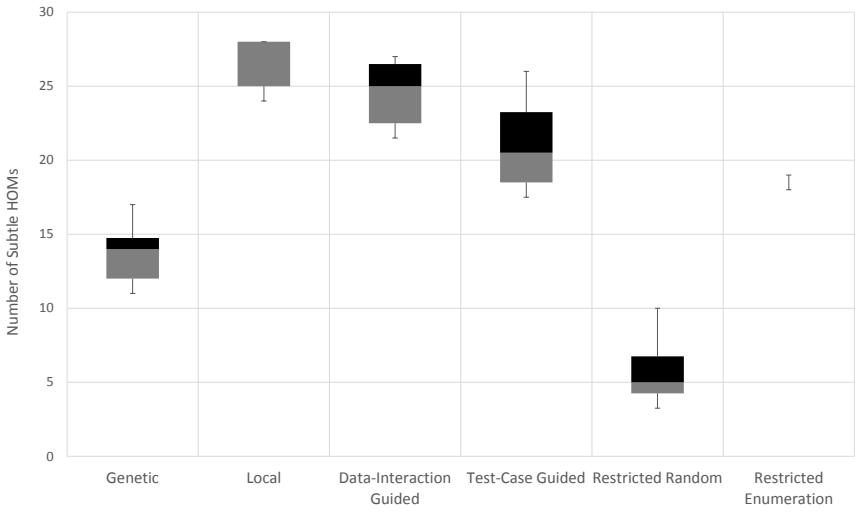
Table 7.13: Number of subtle HOMs that were found for Elevator Program

Technique	# Subtle HOMs				
	Maximum	Average	St. Deviation	Median	Minimum
Genetic	17	13.7	1.9	14	11
Local	28	26	3.5	28	22
Data-Interaction Guided	27	24.1	3	25	18
Test-Case Guided	26	20.6	3.7	20.5	15
Restricted Random	10	5.7	2.1	5	3
Restricted Enumeration	19	19	0	19	19

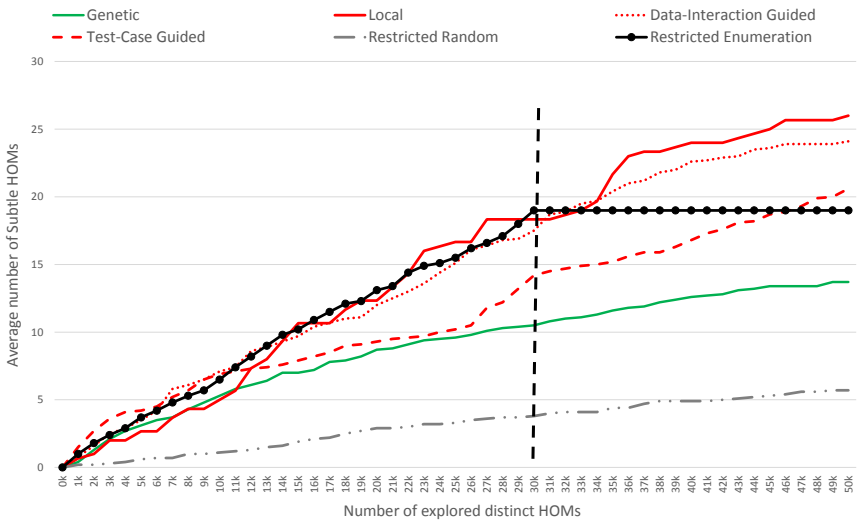
Table 7.13 and Box Plot 7.19 show that Local Search was more effective than the other techniques for the Elevator program. Line Chart 7.20 shows that Local Search, Data-Interaction Guided Local Search, and Restricted Enumeration Search found a comparable average number of subtle

ANOVA Result 7.14: Elevator

Source of Variation	SS	df	MS	F statistic	P-value	F critical
Between Groups	2301.3	5	460.3	72.7	5.89814E-21	2.4
Within Groups	297.5	47	6.3			
Total	2598.8	52				

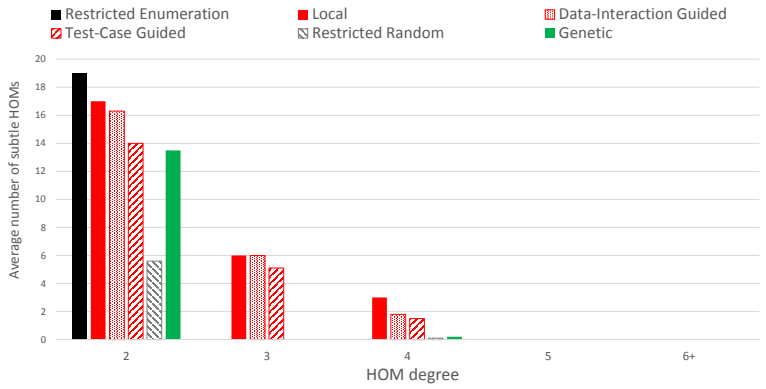


Box Plot 7.19: Distribution of the number of subtle HOMs that were found for Elevator

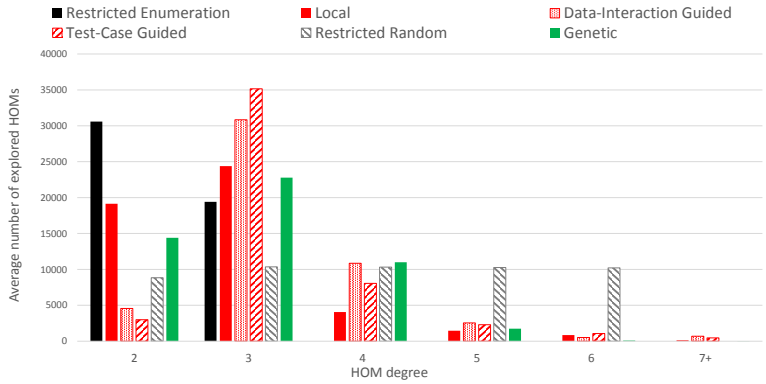


Line Chart 7.20: Growth in the average number of subtle HOMs that were found over the number of explored HOMs for Elevator

HOMs during the early stages of the search process. The Genetic Algorithm was less effective than Restricted Enumeration Search during all stages of the search process.



(a) Distinct Subtle HOMs



(b) All Explored HOMs

Column Chart 7.21: Number of HOMs with respect to the degree for Elevator

The Chi-Square test produced a *P-value* of 0.1, which indicates that there is no significant difference between the number of subtle HOMs that were found by the search techniques with respect to the degree of HOMs.

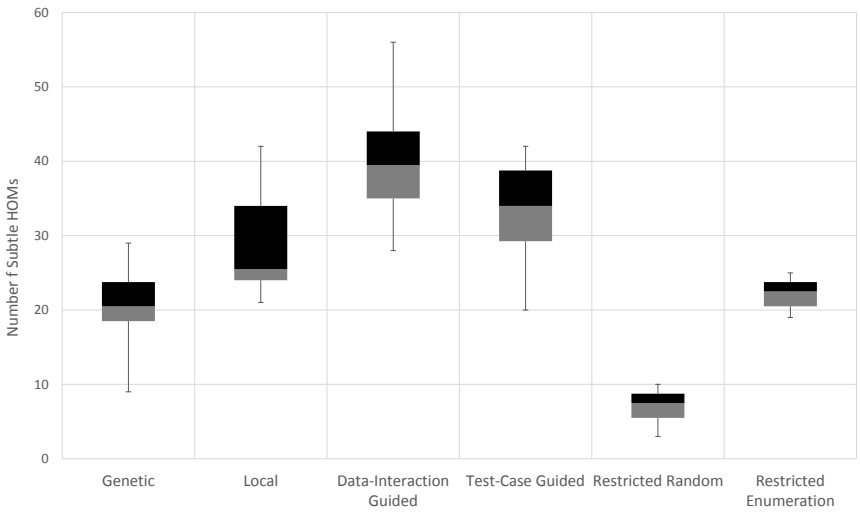
Column Chart 7.21 shows that only Restricted Enumeration Search was able to find all the 19 subtle SOMs. However, Restricted Enumeration Search did not find any subtle HOMs of degrees higher than two. Local Search, both the Guided Local Search techniques, and the Genetic Algorithm found more than half the number of subtle SOMs. However, each of the Guided Local Search techniques explored less than 15% of the space of SOMs. Local Search and both the Guided Local Search techniques were more effective than the other techniques at finding subtle HOMs of degree three and higher.

Table 7.15: Number of subtle HOMs that were found for Cruise Control (AspectJ)

Technique	# Subtle HOMs				
	Maximum	Average	St. Deviation	Median	Minimum
Genetic	29	20.3	6.2	20.5	9
Local	42	29	7	25.5	21
Data-Interaction Guided	56	39.9	8.6	39.5	28
Test-Case Guided	42	33.2	7	34	20
Restricted Random	10.0	7	2.3	7.5	3
Restricted Enumeration	25	22.1	2.1	22.5	19

ANOVA Result 7.16: Cruise Control (AspectJ)

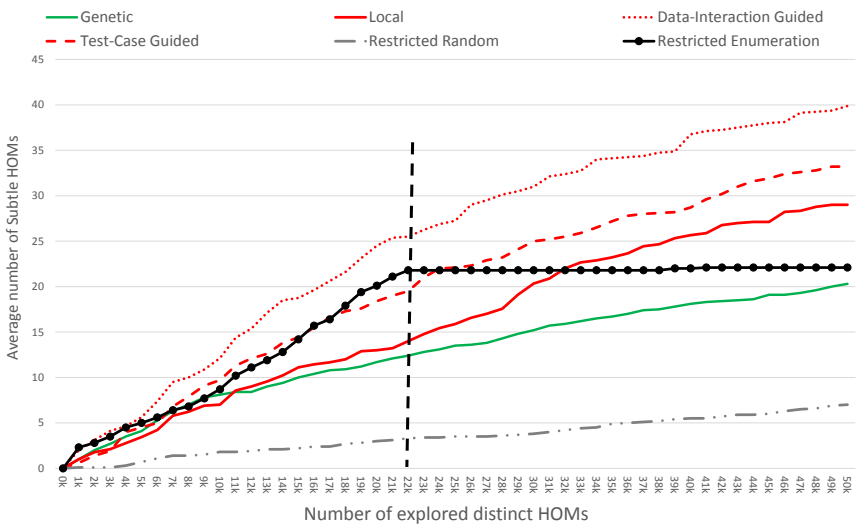
Source of Variation	SS	df	MS	F statistic	P-value	F critical
Between Groups	4753.3	5	950.7	11.7	1.05428E-07	2.4
Within Groups	4373.6	54	81			
Total	9126.9	59				



Box Plot 7.22: Distribution of the number of subtle HOMs that were found for Cruise Control (AspectJ)

8- Cruise Control (AspectJ) Program

The effectiveness of some of the search techniques for the Cruise Control (AspectJ) version varied from their effectiveness for the Cruise Control (Java) version. The Genetic Algorithm was more effective than Restricted Enumeration Search for the Java version, but it was less effective than Restricted Enumeration Search during all stages of the search process for the AspectJ version. The Genetic Algorithm was more effective than Restricted Enumeration Search during the last



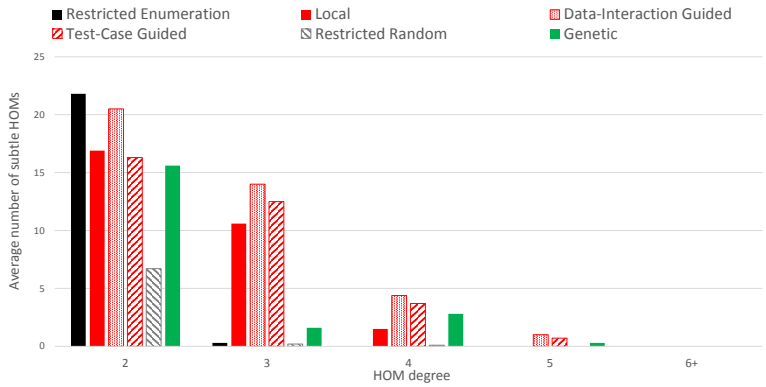
Line Chart 7.23: Growth in the average number of subtle HOMs that were found over the number of explored HOMs for Cruise Control (AspectJ)

stages of the search process for all the other AspectJ programs.

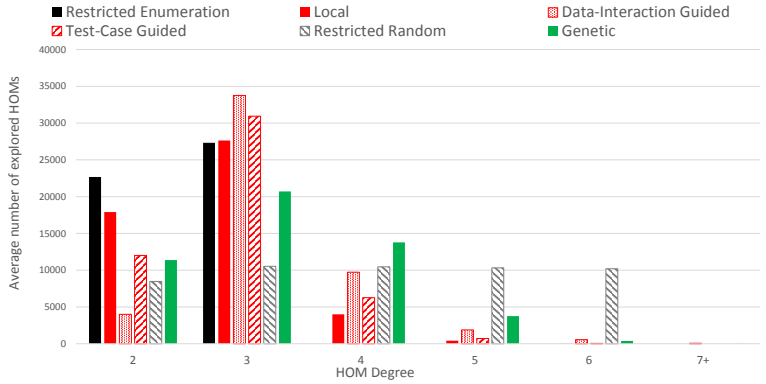
Table 7.15 and Box Plot 7.22 show that Data-Interaction Guided Local Search was more effective than the other techniques for Cruise Control (AspectJ) program. Line Chart 7.23 shows that Local Search and Test-Case Guided Local Search were more effective than Restricted Enumeration Search during the latter stages of the search process.

The Chi-Square test produced a *P-value* of 0.03, which indicates that there is a significant difference between the number of subtle HOMs that were found by the search techniques with respect to the degree of HOMs.

Column Chart 7.24 shows that none of the search techniques found all of the 22 subtle SOMs that were found by Restricted Enumeration Search. However, Data-Interaction Guided Local Search explored less than 18% of the space of SOMs and found 21 of the subtle SOMs. Local Search and both the Guided Local Search techniques were more effective at finding subtle HOMs of degree three and higher.



(a) Distinct Subtle HOMs



(b) All Explored HOMs

Column Chart 7.24: Number of HOMs with respect to the degree for Cruise Control (AspectJ)

9- Roman Program

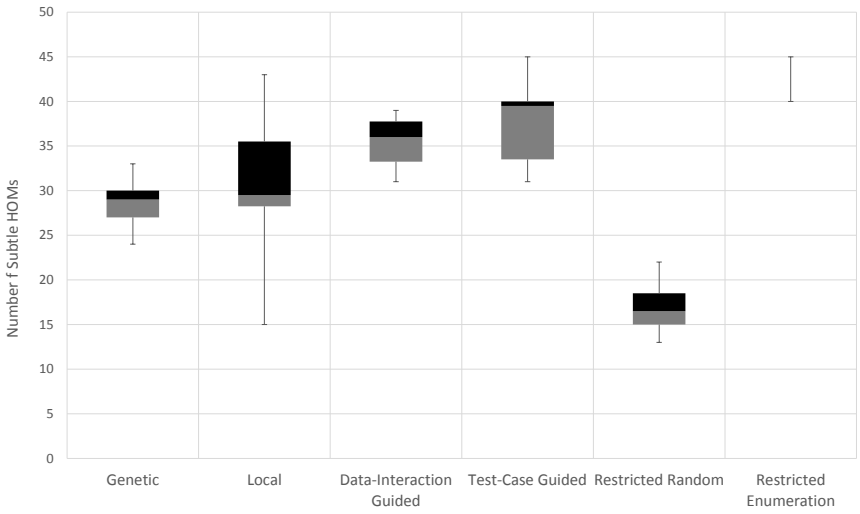
Table 7.17: Number of subtle HOMs that were found for Roman

Technique	# Subtle HOMs				
	Maximum	Average	St. Deviation	Median	Minimum
Genetic	33	28.6	2.9	29	24
Local	43	30.4	7.8	29.5	15
Data-Interaction Guided	39	35.4	3.1	36	31
Test-Case Guided	45	37.9	4.9	39.5	31
Restricted Random	22	16.7	2.7	16.5	13
Restricted Enumeration	45	41	2.1	40	40

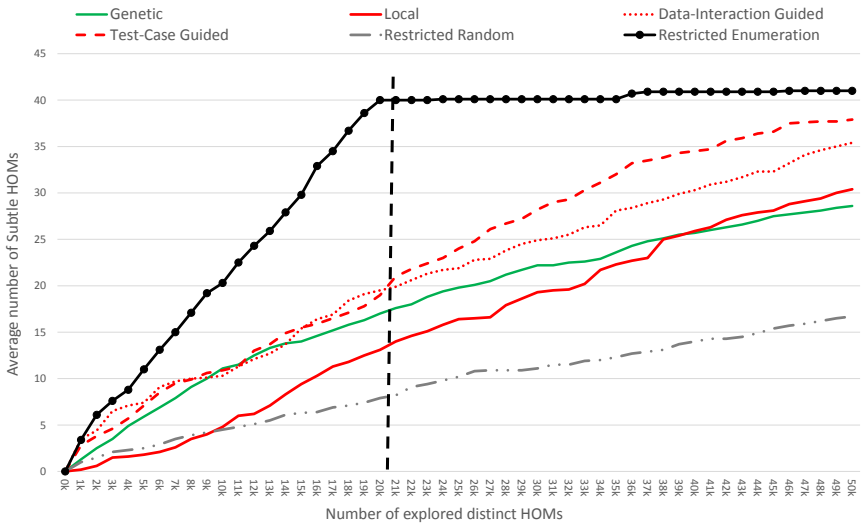
Table 7.17 and Box Plot 7.25 show that Restricted Enumeration Search was more effective than the other techniques for the Roman program. The Roman program is the only subject program where the average and median numbers of subtle HOMs that were found by Restricted Enumera-

ANOVA Result 7.18: Roman

Source of Variation	SS	df	MS	F statistic	P-value	F critical
Between Groups	3749.1	5	749.8	39.3	8.07299E-17	2.4
Within Groups	1030.2	54	19.1			
Total	4779.3	59				



Box Plot 7.25: Distribution of the number of subtle HOMs that were found for Roman

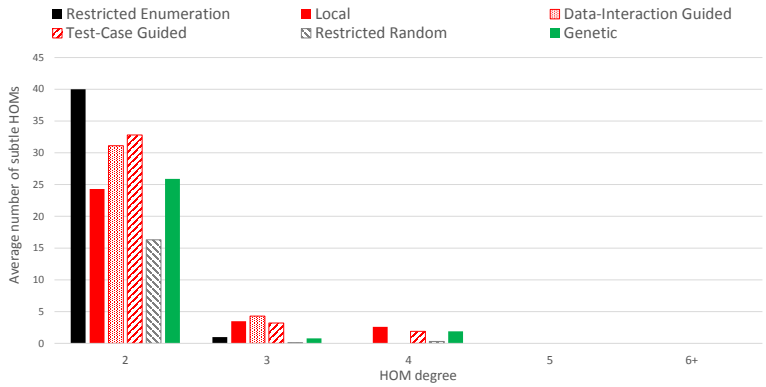


Line Chart 7.26: Growth in the average number of subtle HOMs that were found over the number of explored HOMs for Roman

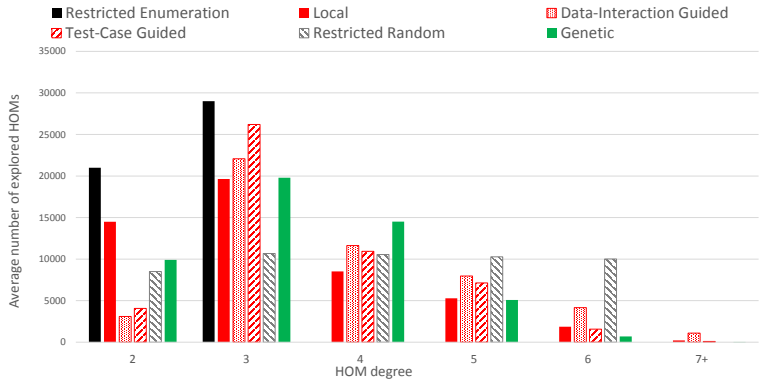
tion Search were higher than all other techniques.

Line Chart 7.26 shows that Restricted Enumeration Search was more effective than the other techniques during all stages of the search process. Local Search was less effective than Restricted

Random Search during the early stages of the search process and less effective than the Genetic Algorithm during most stages of the search process. Both the Guided Local Search techniques were more effective than Local Search during all stages of the search process.



(a) Distinct Subtle HOMs



(b) All Explored HOMs

Column Chart 7.27: Number of HOMs with respect to the degree for Roman

The Chi-Square test produced a *P-value* of 0.27, which indicates that there is no significant difference between the number of subtle HOMs that were found by the search techniques with respect to the degree of HOMs.

Column Chart 7.27 shows that none of the search techniques found all of the 40 subtle SOMs that were found by Restricted Enumeration Search. The chart also shows that none of the search techniques, except for Restricted Enumeration Search, fully explored the space of SOMs.

Local Search, both the Guided Local Search techniques, and the Genetic Algorithm were not effective at finding subtle HOMs of degree three and higher, which could explains why they were overall less effective than Restricted Enumeration Search. Each of the four techniques found less

than five subtle HOMs of degree three and higher, in addition to finding around half the number of subtle SOMs.

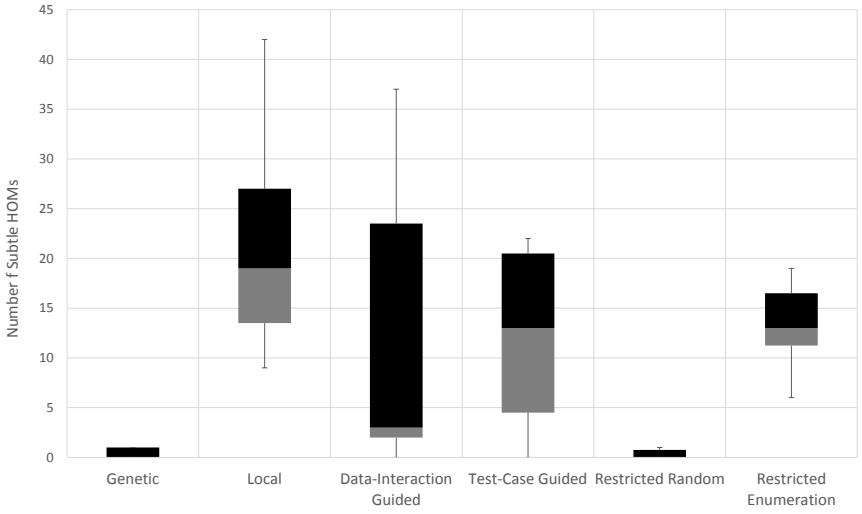
10- XStream Program

Table 7.19: Number of subtle HOMs that were found for XStream

Technique	# Subtle HOMs				
	Maximum	Average	St. Deviation	Median	Minimum
Genetic	1	0.4	0.5	0	0
Local	42	20	10.8	19	9
Data-Interaction Guided	37	11.4	14.2	3	0
Test-Case Guided	22	12	10.7	13	0
Restricted Random	1	0.3	0.5	0	0
Restricted Enumeration	19	13.4	4	13	6

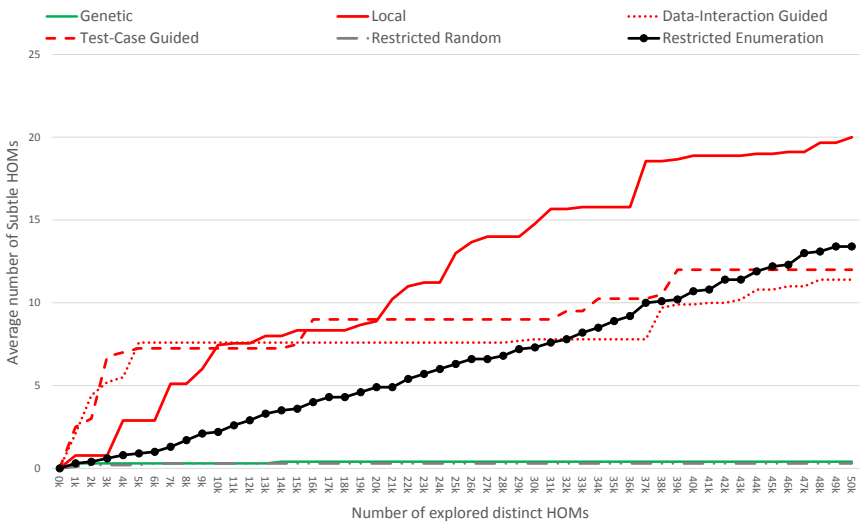
ANOVA Result 7.20: XStream

Source of Variation	SS	df	MS	F statistic	P-value	F critical
Between Groups	3427.4	5	685.5	10	8.60154E-07	2.4
Within Groups	3715	54	68.8			
Total	7142.4	59				



Box Plot 7.28: Distribution of the number of subtle HOMs that were found for XStream

Table 7.19 and Box Plot 7.28 show that Local Search was more effective than the other techniques for the XStream program. The average and median numbers of subtle HOMs that were



Line Chart 7.29: Growth in the average number of subtle HOMs that were found over the number of explored HOMs for XStream

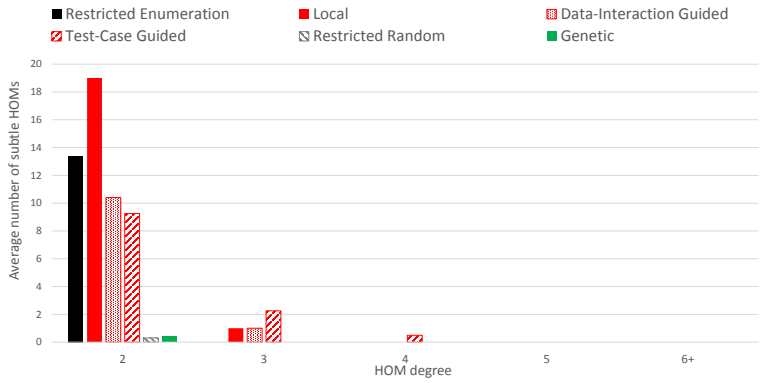
found by Local Search were higher than all other techniques. The average number of subtle HOMs that were found by the Genetic Algorithm and Restricted Random Search were low compared to the other techniques.

Line Chart 7.29 shows that Local Search was more effective than Restricted Enumeration Search during all stages of the search process. Both the Guided Local Search techniques were more effective than Restricted Enumeration Search during the early stages of the search process.

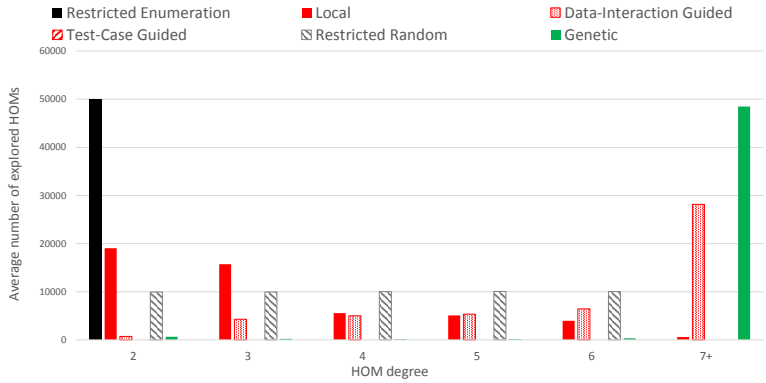
The Chi-Square test produced a *P-value* of 0.85, which indicates that there is no significant difference between the number of subtle HOMs that were found by the search techniques with respect to the degree of HOMs.

Column Chart 7.30 shows that Local Search found more subtle SOMs than Restricted Enumeration Search. Restricted Enumeration Search was not able to fully explore the large space of SOMs, which is larger than the 50,000 distinct HOMs Restricted Enumeration Search was allowed to explore. Both the Guided Local Search techniques found more than half the number of subtle SOMs that were found by Restricted Enumeration Search.

The effectiveness of the Genetic Algorithm and Restricted Random Search found was low compared with other search techniques. Figure 7.30 shows that the Genetic Algorithm did explored a low number of HOMs of lower degrees where more subtle HOMs can be found. More than 94%



(a) Distinct Subtle HOMs



(b) All Explored HOMs

Column Chart 7.30: Number of HOMs with respect to the degree for XStream

of all distinct HOMs explored by the Genetic Algorithm were of degrees higher than nine.

The crossover operator of the Genetic Algorithm kept increasing the degree of HOMs at each iteration for the XStream program. When two parents are crossed-over, the set of FOMs that need to be applied to each offspring can contains some FOMs that cannot be combined because they apply mutation to the same token in the same program statement. The crossover operator in such cases cannot apply all FOMs intended for an offspring. As a result, the both offspring could have lower degrees than their parents. There is a higher chance for this to happen for programs with a low number of statements that can be mutated, which is the case for the other nine programs. However, XStream is a large program with a large number of statements that can be mutated and thus there is a lower chance to combine FOMs that apply mutation to the same token than the other nine programs. As a result, it became easier for the crossover operator to create more offspring HOMs of higher degrees. This caused the Genetic Algorithm to spend most of the execution time

exploring the space of HOMs of high degrees where it is harder to find subtle HOMs and less number of subtle HOMs are expected to be found. The large number of HOMs for the XStream program made it even harder for Restricted Random Search to find many subtle HOMs.

7.3 Summary of findings

All the search techniques found subtle HOMs for all programs. However, Local Search and both the Guided Local Search techniques were more effective than the other techniques in terms of their ability to find subtle HOMs.

The search techniques found more subtle HOMs of second and third degrees; subtle HOMs of higher degrees were harder to find. The Genetic Algorithm, Local Search, and both the Guided Local Search techniques were more effective than Restricted Enumeration for finding subtle HOMs of degree three and higher.

Overall, Data-Interaction Guided Local Search was more effective than the other techniques for finding a higher number of distinct, subtle HOMs. It found on an average 94% of the subtle SOMs while it explored only 16% of the space of SOMs.

Chapter 8

Comparing Sets of Subtle HOMs Found by Different Search Techniques

This chapter presents a study to investigate the difference between the sets of subtle HOMs that were found by different search techniques. First, we present the research question motivating the study then we present the findings and analysis of the results.

8.1 Research Question

The study is motivated by the following research question.

RQ1: What set of subtle HOMs is found by all techniques and what set of subtle HOMs is uniquely found by each technique?

The search techniques use different operators to generate HOMs and that can have an affect what set of subtle HOMs found by each technique. We define the hardest-to-find subtle HOM as the one that can be uniquely found by only one search technique. We also define the easiest-to-find subtle HOM as the one that can be found by all the search techniques. We investigated the numbers of subtle HOMs that were the hardest-to-find and easiest-to-find by the search techniques.

In a practical setting, a tester may not have the time nor the resources to run all the search techniques to find subtle HOMs. Therefore, knowing what set of subtle HOMs can be uniquely found by each search technique and what set of subtle HOMs can be found by all techniques can help testers select and prioritize the search techniques based on the desired subtle HOMs.

For each subject program we generated six sets of subtle HOMs. Each set represents the union of the 30 sets of subtle HOMs that were found by a search technique over the 30 runs. We then used these six sets to determine the subtle HOMs that were the hardest-to-find and easiest-to-find for each program. In addition, we calculated the number of subtle HOMs that were found by two, three, four, and five search techniques. We also calculated the union of the six sets to obtain the set

of all distinct, subtle HOMs that were found by all the search techniques for each subject program. This set is called the set of all subtle HOMs.

8.2 Results and Analysis

Bar Charts 8.1 through 8.19 show the six sets of subtle HOMs for each program, which are depicted by the black bars. Each chart also shows the set of all distinct, subtle HOMs for each program, which is depicted by the striped bar.

The Column Charts shown in part (a) of Figures 8.2 through 8.20 show the number of subtle HOMs with respect to the number of search techniques that found them. For example, Figure 8.2 (a) shows that 38 subtle HOMs were the easiest-to-find as they were found by all the search techniques. The figure also shows that 223 subtle HOMs were the hardest-to-find as each one was uniquely found by one technique. The hardest-to-find subtle HOMs are depicted by the gray column.

The Pie Charts shown in part (b) of Figures 8.2 through 8.20 show the number of the hardest-to-find subtle HOMs with respect to the search technique that found them. For example, Figure 8.2 (b) shows that the 223 hardest-to-find subtle HOMs shown in part (a) were as follows: 176 subtle HOMs were uniquely found by the Genetic Algorithm, two subtle HOMs were uniquely found by Local Search, 11 subtle HOMs were uniquely found by Data-Interaction Guided Local Search, 33 subtle HOMs were uniquely found by Test-Case Guided Local Search, and one subtle HOMs was uniquely found by Restricted Random Search.

The search techniques found more of the hardest-to-find subtle HOMs than the easiest-to-find subtle HOMs. For eight out of the ten subject programs, the number of the hardest-to-find subtle HOMs was higher than the number of the easiest-to-find subtle HOMs. This shows that the different operators implemented in the search techniques can impact the type of subtle HOMs that can be found by each technique.

Our investigation showed that 94% of the hardest-to-find subtle HOMs of all programs were of degree three and higher, and more than 92% of the easiest-to-find subtle HOMs of all programs were subtle SOMs. More than 95% of the hardest-to-find subtle HOMs of all programs

resulted from the Genetic Algorithm, Data-Interaction Guided Local Search, Test-Case Guided Local Search, and Local Search. The three techniques were more effective than the other techniques at finding subtle HOMs of degree three and higher.

Restricted Enumeration Search was less effective than the other techniques at finding the hardest-to-find subtle HOMs despite Restricted Enumeration Search explored more distinct third order mutants than the four techniques for five programs.

For the XStream program, which has much larger search space than the other nine programs, Restricted Enumeration Search found a larger number of the hardest-to-find subtle HOMs than the Genetic Algorithm, both the Guided Local Search techniques, and Restricted Random Search. Restricted Enumeration Search found only three and two hardest-to-find subtle HOMs for another two programs.

Although Restricted Random Search was less effective at finding subtle HOMs than the other techniques, it found hardest-to-find subtle HOMs for seven out of the ten programs. However, Restricted Random Search found a low number of hardest-to-find subtle HOMs compared to the other techniques.

More than 38% of the hardest-to-find subtle HOMs of all programs were found by the Genetic Algorithm. For six out of the ten programs, the Genetic Algorithm found a higher number of the hardest-to-find subtle HOMs than all other techniques.

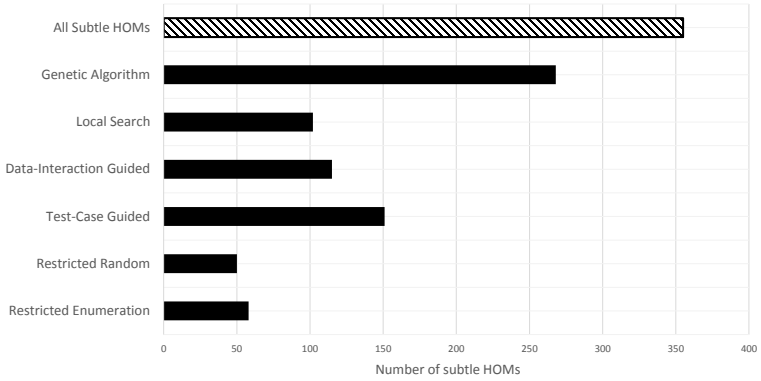
Although Local Search, Data-Interaction Guided Local Search, and Test-Case Guided Local Search used a similar mechanism to explore the search space, each of the three techniques was able to find hardest-to-find subtle. However, both the Guided Local Search Techniques found more of the hardest-to-find subtle HOMs of all programs than Local Search.

For nine out of the ten programs, the set of subtle HOMs that were found by Local Search, both the Guided Local Search techniques, and the Genetic Algorithm included more than 98% of the subtle HOMs that were found by Restricted Enumeration and Restricted Random Search. This suggest that the former four techniques need to be used in order to find a large number of subtle HOMs that can be used to improve the effectiveness of the test suites.

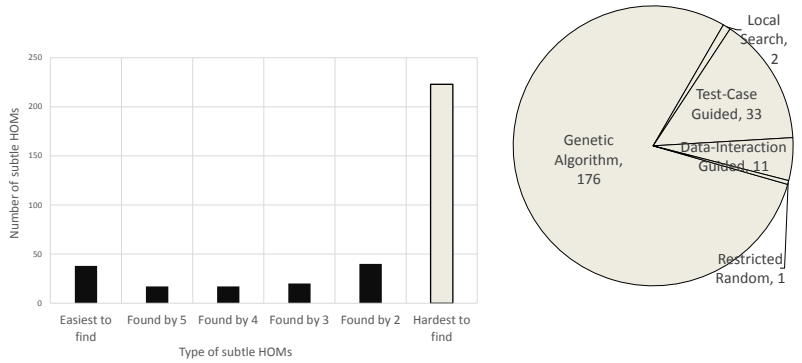
For five out of the ten subject programs, the number of all distinct, subtle HOMs that were found by all search techniques was higher than the number of non-equivalent FOMs for these programs. For another four programs, the number of all distinct, subtle HOMs was more than half the number of non-equivalent FOMs for these programs.

In the remainder of this section, we further analyze the results and highlight the findings for each subject program.

1- Cruise Control (Java) Program



Bar Chart 8.1: Number of subtle HOMs that were found by the search techniques over 30 runs for Cruise Control (Java)



(a) Easiest-to-find and hardest-to-find subtle HOMs

(b) hardest-to-find subtle HOMs by search technique

Figure 8.2: Number of subtle HOMs with respect to the number of search techniques that found them for Cruise Control (Java)

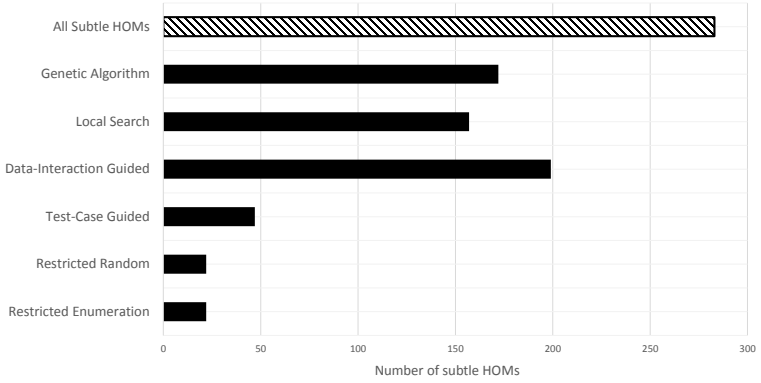
Figure 8.2 (a) shows that the number of the hardest-to-find subtle HOMs is higher than the number of the easiest-to-find subtle HOMs for the Cruise Control (Java) Program. Figure 8.2 (b)

shows that the majority of the hardest-to-find subtle HOMs were found by the Genetic Algorithm, which found the largest set of subtle HOMs over 30 runs (see Bar Chart 8.1). The majority of the hardest-to-find subtle HOMs that were found by the Genetic Algorithm were of degree three and higher that could not be found by the other techniques.

Restricted Enumeration Search did not find any the hardest-to-find subtle HOMs while Restricted Random Search found only one hardest-to-find subtle HOM. Both techniques were the least effective in terms of their ability to find subtle HOMs for the Cruise Control (Java) Program (see Chapter 7). Further, Bar Chart 8.1 shows that both techniques found smaller sets of subtle HOMs over 30 runs than the other techniques.

The set of subtle HOMs that were found by the Genetic Algorithm and both the Guided Local Search techniques included more than 99% of all the subtle HOMs that were found for the Cruise Control (Java) program.

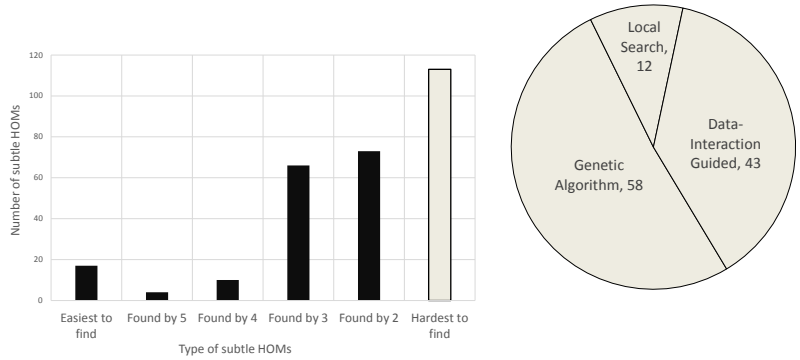
2- Movie Rental Program



Bar Chart 8.3: Number of subtle HOMs that were found by the search techniques over 30 runs for Movie Rental

Figure 8.4 (a) shows that the number of the hardest-to-find subtle HOMs was higher than the number of the easiest-to-find subtle HOMs. The hardest-to-find subtle HOMs were found by the Genetic Algorithm, Data-Interaction Guided Local Search, and Local Search.

Figure 8.4 (b) shows that the Genetic Algorithm found more of the hardest-to-find subtle HOMs than Data-Interaction Guided Local Search. This is despite the fact that the latter technique found



(a) Easiest-to-find and hardest-to-find subtle HOMs

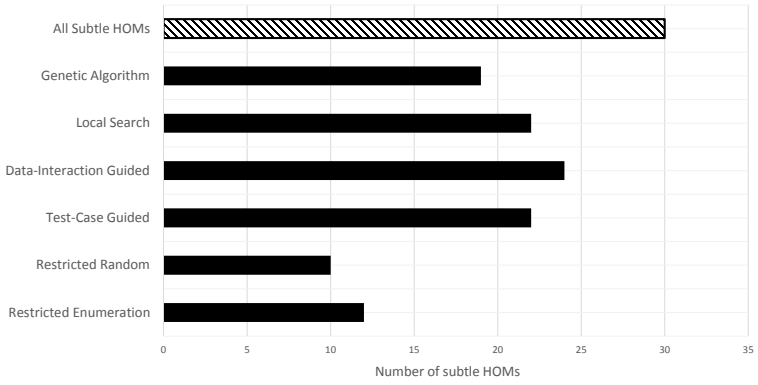
(b) hardest-to-find subtle HOMs by search technique

Figure 8.4: Number of subtle HOMs with respect to the number of search techniques that found them for Movie Rental

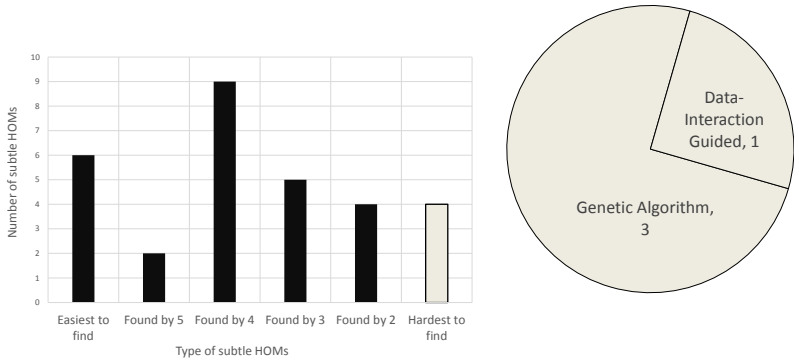
more subtle HOMs over the 30 runs than all the other techniques (see Bar Chart 8.3) and it also found the highest average number of subtle HOMs (see Chapter 7).

The set of subtle HOMs that were found using the Genetic Algorithm, Data-Interaction guided Local Search, and Local Search included all subtle HOMs that were found for the Movie Rental program. The sets of subtle HOMs that were found by the Genetic Algorithm, Data-Interaction guided Local Search, and Local Search individually included all subtle HOMs that were found by Restricted Enumeration Search.

3- Telecom Program



Bar Chart 8.5: Number of subtle HOMs that were found by the search techniques over 30 runs for Telecom



(a) Easiest-to-find and hardest-to-find subtle HOMs

(b) hardest-to-find subtle HOMs by search technique

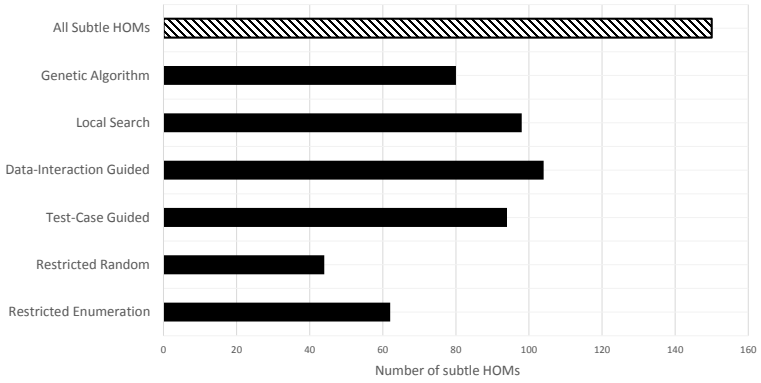
Figure 8.6: Number of subtle HOMs with respect to the number of search techniques that found them for Telecom

Bar Chart 8.5 shows that the set of all subtle HOMs that were found by all the search techniques included 30 subtle HOMs, which is the lowest number of subtle HOMs for the ten subject programs.

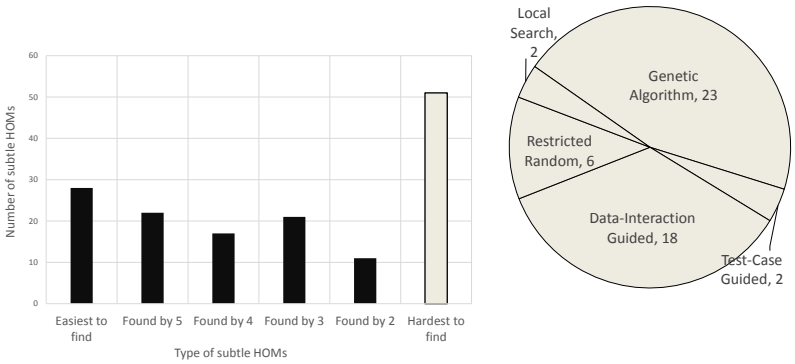
Figure 8.6 (a) shows that the number of the easiest-to-find subtle HOMs is slightly higher than the number of the hardest-to-find subtle HOMs. The number of subtle HOMs that were found by four techniques was the highest. This is because the Genetic Algorithm, both the Guided Local Search techniques, and Local Search found many of the same subtle HOMs. Each of the four techniques found more than half the number of subtle HOMs that were found by all techniques.

The Genetic Algorithm and Data-Interaction Guided Local Search were the only techniques to find the hardest-to-find subtle HOMs. Although the Genetic Algorithm was not the most effective technique for the Telecom program, it found a few more of the hardest-to-find subtle HOMs than Data-Interaction Guided Local Search, which was one of the most effective technique for the Telecom program.

The set of subtle HOMs that were found by the Genetic Algorithm, Data-Interaction guided Local Search, and Local Search included all subtle HOMs that were found by all other techniques.



Bar Chart 8.7: Number of subtle HOMs that were found by the search techniques over 30 runs for Kettle



(a) Easiest-to-find and hardest-to-find subtle HOMs

(b) hardest-to-find subtle HOMs by search technique

Figure 8.8: Number of subtle HOMs with respect to the number of search techniques that found them for Kettle

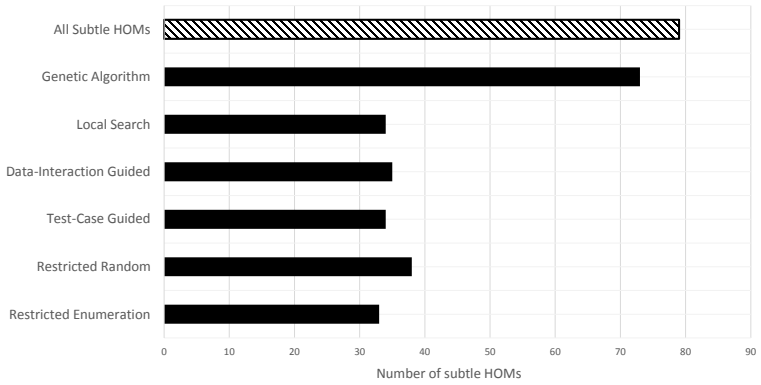
4- Kettle Program

Figure 8.8 (a) shows that the number of the hardest-to-find subtle HOMs was higher than the number of the easiest-to-find subtle HOMs. This is because five out of the six search techniques found the hardest-to-find subtle HOMs for the Kettle program. Restricted Enumeration Search did not find any of the hardest-to-find subtle HOMs.

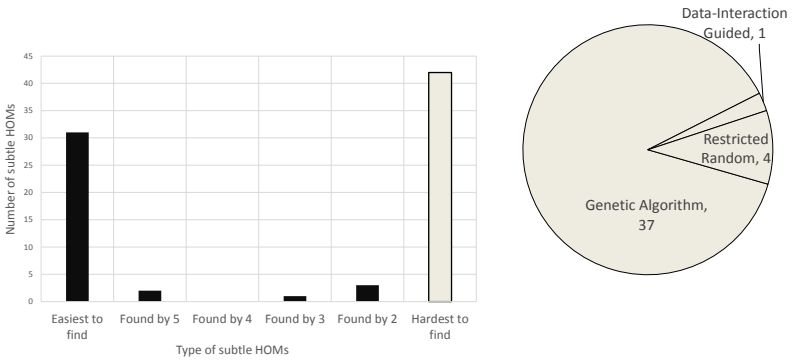
The Genetic Algorithm found a higher number of the hardest-to-find subtle HOMs than the other techniques followed by Data-Interaction Guided Local Search. Restricted Random Search found a higher number of the hardest-to-find subtle HOMs than Local Search and Test-Case Guided Local Search, which both were far more effective than Restricted Random Search in terms of their

ability to find subtle HOMs (see Chapter 7). The set of subtle HOMs that were found by Local Search included all subtle HOMs that were found by Restricted Enumeration Search.

5- Banking Program



Bar Chart 8.9: Number of subtle HOMs that were found by the search techniques over 30 runs for Banking



(a) Easiest-to-find and hardest-to-find subtle HOMs

(b) hardest-to-find subtle HOMs by search technique

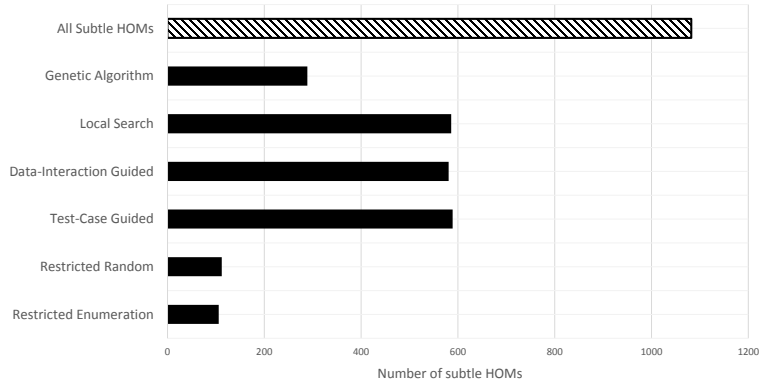
Figure 8.10: Number of subtle HOMs with respect to the number of search techniques that found them for Banking

Figure 8.10 (a) shows that the number of the hardest-to-find subtle HOMs was higher than the number of the easiest-to-find subtle HOMs. The majority of the hardest-to-find subtle HOMs were found by the Genetic Algorithm, which found the largest set of subtle HOMs over 30 runs (see Bar Chart 8.9). The majority of the hardest-to-find subtle HOMs that were found by the Genetic Algorithm were of degree three and higher and these could not be found by the other techniques.

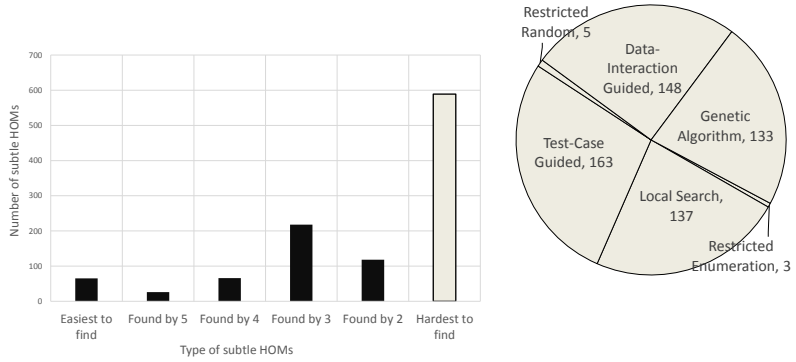
The number of the easiest-to-find subtle HOM was also high because all techniques were effective for the Banking program (see Chapter 7) and they found many of the same subtle HOMs.

The set of subtle HOMs that were found by the Genetic Algorithm, Data-Interaction Guided Local Search, and Restricted Random Search included all subtle HOMs that were found by all other techniques.

6- Coordinate Program



Bar Chart 8.11: Number of subtle HOMs that were found by the search techniques over 30 runs for Coordinate



(a) Easiest-to-find and hardest-to-find subtle HOMs

(b) hardest-to-find subtle HOMs by search technique

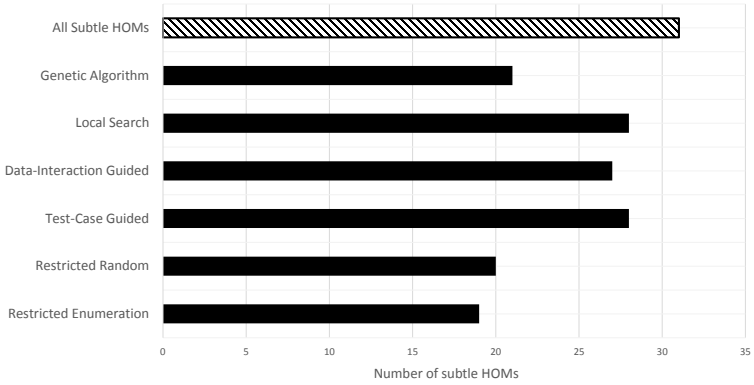
Figure 8.12: Number of subtle HOMs with respect to the number of search techniques that found them for Coordinate

Bar Chart 8.11 shows that the number of subtle HOMs that were found for the Coordinate program was higher than those found by all other programs. Although all search techniques found

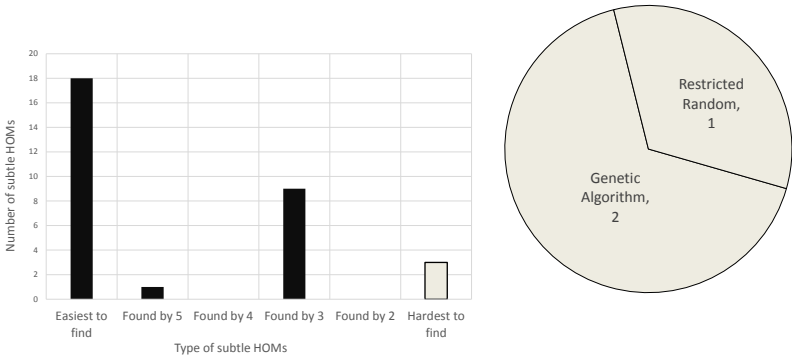
the hardest-to-find subtle HOMs, Restricted Enumeration Search and Restricted Random Search found a low number of the hardest-to-find subtle HOMs compared to the other techniques.

The number of the hardest-to-find subtle HOMs was higher than the number of the easiest-to-find subtle HOMs. Test-Case Guided Local Search, Data-Interaction Guided Local Search, Local Search, and the Genetic Algorithm found a high number of the hardest-to-find subtle HOMs of higher degrees.

7- Elevator Program



Bar Chart 8.13: Number of subtle HOMs that were found by the search techniques over 30 runs for Elevator



(a) Easiest-to-find and hardest-to-find subtle HOMs

(b) hardest-to-find subtle HOMs by search technique

Figure 8.14: Number of subtle HOMs with respect to the number of search techniques that found them for Elevator

Similar to the Telecom program, the Elevator program has a low number of subtle HOMs. Bar Chart 8.13 shows that the set of all subtle HOMs that were found by all the search techniques

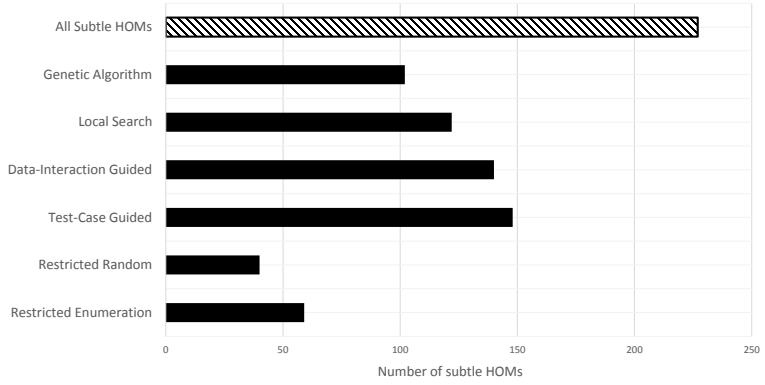
included 31 subtle HOMs, which is the second lowest number of subtle HOMs among the ten subject programs.

Figure 8.14 (a) shows that the number of the easiest-to-find subtle HOMs was higher than the number of the hardest-to-find subtle HOMs. The search techniques found only three of the hardest-to-find subtle HOMs, which is the lowest number of hardest-to-find subtle HOMs amongst the ten programs.

Figure 8.14 (a) shows that the number of subtle HOMs that were found by three techniques was high because both the Guided Local Search techniques and Local Search found many of the same subtle HOMs.

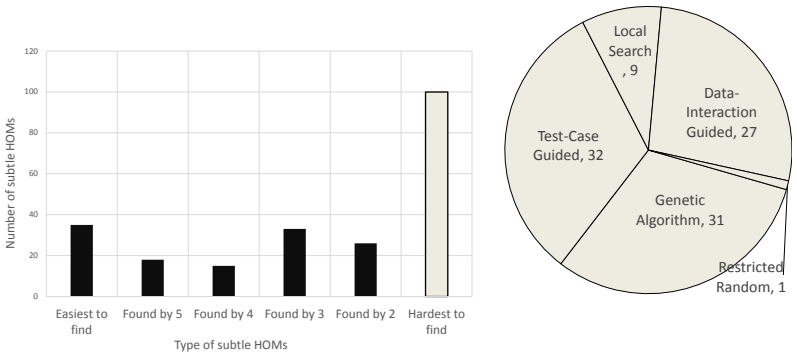
The set of subtle HOMs that were found using the Genetic Algorithm, Data-Interaction guided Local Search, and Restricted Random Search included all subtle HOMs that were found using all other techniques. The sets of subtle HOMs that were found by the Genetic Algorithm, Test-Case Guided Local Search, and Local Search individually included all subtle HOMs that were found by Restricted Enumeration Search.

8- Cruise Control (AspectJ) Program



Bar Chart 8.15: Number of subtle HOMs that were found by the search techniques over 30 runs for Cruise Control (AspectJ)

Figure 8.16 (a) shows that the number of the hardest-to-find subtle HOMs was higher than the number of the easiest-to-find subtle HOMs. This is because five out of the six search techniques found the hardest-to-find subtle HOMs for the Cruise Control (AspectJ) program. Restricted Enu-



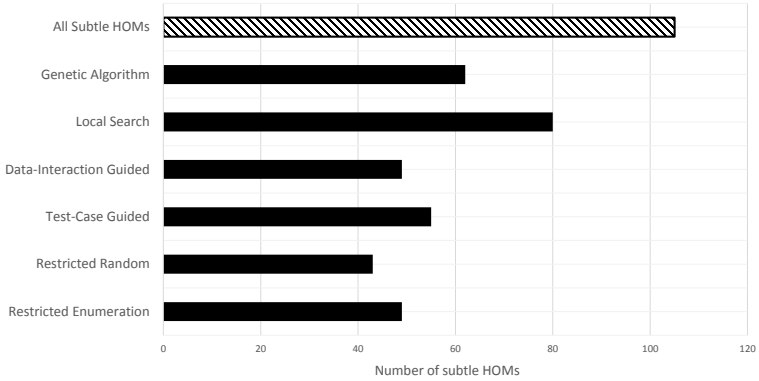
(a) Easiest-to-find and hardest-to-find subtle HOMs

(b) hardest-to-find subtle HOMs by search technique

Figure 8.16: Number of subtle HOMs with respect to the number of search techniques that found them for Cruise Control (AspectJ)

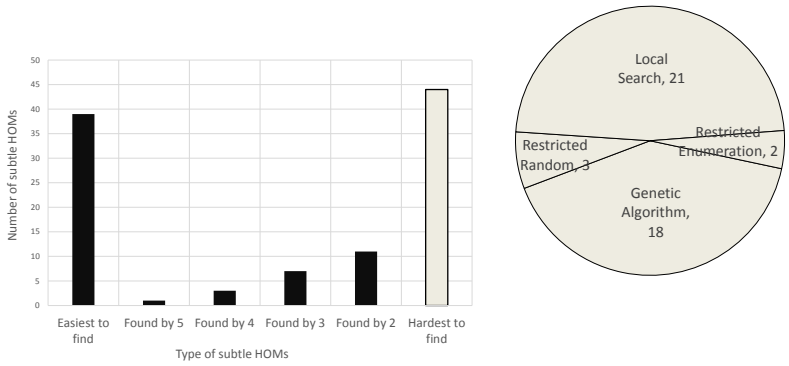
meration Search did not find any hardest-to-find subtle HOMs, while Restricted Random Search found only one hardest-to-find subtle HOMs. The Genetic Algorithm and both the Guided Local Search techniques found a higher number of hardest-to-find subtle HOMs than Local Search.

9- Roman Program



Bar Chart 8.17: Number of subtle HOMs that were found by the search techniques over 30 runs for Roman

Figure 8.18 (a) shows that the number of the hardest-to-find subtle HOMs was higher than the number of the easiest-to-find subtle HOMs. The Genetic Algorithm and Local Search found the majority of the hardest-to-find subtle HOMs, while Restricted Enumeration Search and Restricted Random Search found two and three hardest-to-find subtle HOMs, respectively.



(a) Easiest-to-find and hardest-to-find subtle HOMs

(b) hardest-to-find subtle HOMs by search technique

Figure 8.18: Number of subtle HOMs with respect to the number of search techniques that found them for Roman

None of the Guided Local Search techniques found any hardest-to-find subtle HOMs. The set of subtle HOMs that were found by Local Search included all subtle HOMs that were found by Data-Interaction Guided Local Search and Test-Case Guided Local Search.

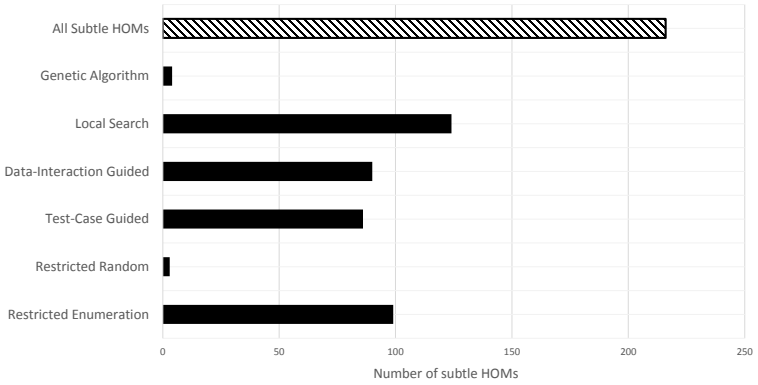
10- XStream Program

Figure 8.20 (a) shows that the search techniques did not find any of the easiest-to-find subtle HOMs for the XStream program. That is, no subtle HOM was found by all techniques. Further, only one subtle HOM was found by five techniques. This is because the Genetic Algorithm and Restricted Random Search found a low number of subtle HOMs compared to the other techniques.

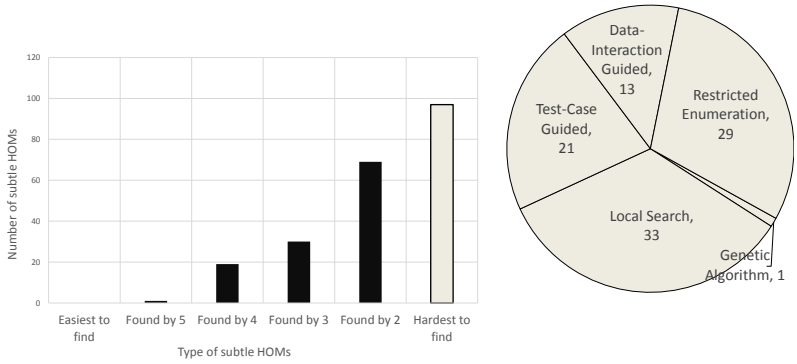
The search techniques found 97 hardest-to-find subtle HOMs and Local Search and Restricted Enumeration Search together found more than half the number of these subtle HOMs.

8.3 Summary of findings

Subtle HOMs that were uniquely found by one search technique represented 50% of all the subtle HOMs found in the ten programs. This shows that the different operators implemented in the search techniques have an impact on the set of subtle HOMs that can be uniquely found by each technique. 94% of the uniquely found subtle HOMs were of degree three and higher; and 92% of the subtle HOMs that were found by all techniques were subtle SOMs.



Bar Chart 8.19: Number of subtle HOMs that were found by the search techniques over 30 runs for XStream



(a) Easiest-to-find and hardest-to-find subtle HOMs

(b) hardest-to-find subtle HOMs by search technique

Figure 8.20: Number of subtle HOMs with respect to the number of search techniques that found them for XStream

For most programs, the set of subtle HOMs that were found by the Genetic Algorithm, both the Guided Local Search techniques, and Local Search included all subtle HOMs that were found by the other two techniques. This shows that using the four search-based software engineering techniques is more likely to produce a large number of distinct, subtle HOMs that can be used to improve the fault-detection effectiveness of test suites.

Chapter 9

Impact of Programming Language Constructs on Creating Subtle HOMs

In this chapter we present a study to investigate the impact of the different constructs provided by the programming languages, AspectJ and Java, on the creation of subtle HOMs. The goal is to determine whether or not subtle HOMs are more likely to be created by combining FOMs that apply mutation faults to a specific set of constructs of the programming language.

9.1 Research Questions

This section presents two research questions that motivated the study presented in this chapter.

RQ1: What mutation operators are more likely to create subtle HOMs?

The mutation operators for a programming language are designed based on the syntax, constructs, and structure of the language. The mutation operators that we used to create the FOMs for the subject programs can be classified into three types. The method-level mutation operators apply changes to expressions that include primitive Java operators. The method-level mutation operators apply mutations, such as replacing, deleting, and inserting a primitive operator, in the Java class methods, AspectJ advices, and AspectJ inter-type declaration methods.

The second type is class-level mutation operators and they apply changes to expressions that implement Object-Oriented features, such as encapsulation, inheritance, and polymorphism. The last type is aspect-level mutation operators and they apply changes to expressions that implement Aspect-Oriented features, such as pointcut descriptors, advice weaving kind, and precedence among aspects.

For each program, we used the FOM XML records to obtain the set of mutation operators that were used to create all the non-equivalent FOMs. We investigated what set of mutation operators

were used to create the constituent FOMs of the subtle HOMs that were found by each search technique. We also investigated whether or not the search techniques favor combining FOMs that were created by a specific set of mutation operators.

RQ2: Are subtle HOMs more likely to be created when combining mutated constructs from specific locations?

We investigated whether subtle HOMs are more likely to be created when combining mutated constructs (FOMs) from specific locations within the program, such as when combining FOMs that apply mutation to the same program statement or that apply mutation to different statements of the same program method. For each program, we investigated the number of subtle HOMs and all explored HOMs with respect to the location of their constituent FOMs. We classified HOMs based on the location of their constituent FOMs into the following four categories:

1. *HOMs of the same statement:* represent HOMs that their constituent FOMs apply mutation to different tokens of the same statement of code.

2. *HOMs of the same method/advice:* represent HOMs that their constituent FOMs apply mutation to different statements in the same method/advice of a class.

3. *HOMs of the same class/aspect:* represent HOMs that their constituent FOMs apply mutation to different statements in different methods/advices in the same class/aspect.

4. *HOMs of different classes/aspects:* represent HOMs that their constituent FOMs apply mutation to different statements in different classes/aspects.

We also investigated the number of subtle HOMs and all explored HOMs with respect to the construction approach used to create the HOMs in AspectJ programs. The goal was to determine if subtle HOMs are more likely to be created using one of the construction approaches.

The HOM construction approaches [91] specify the locations of the program where single changes can be made, where each change corresponds to an FOM. For example, an HOM may be constructed by combining only FOMs of the same class or the same aspect. The approaches are

based on Aspect-Oriented Programming fault models [11, 26, 73]. The construction approaches 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.

9.2 Results and Analysis

This section presents the results for the two research questions and analysis of the results.

9.2.1 RQ1: What mutation operators are more likely to create subtle HOMs?

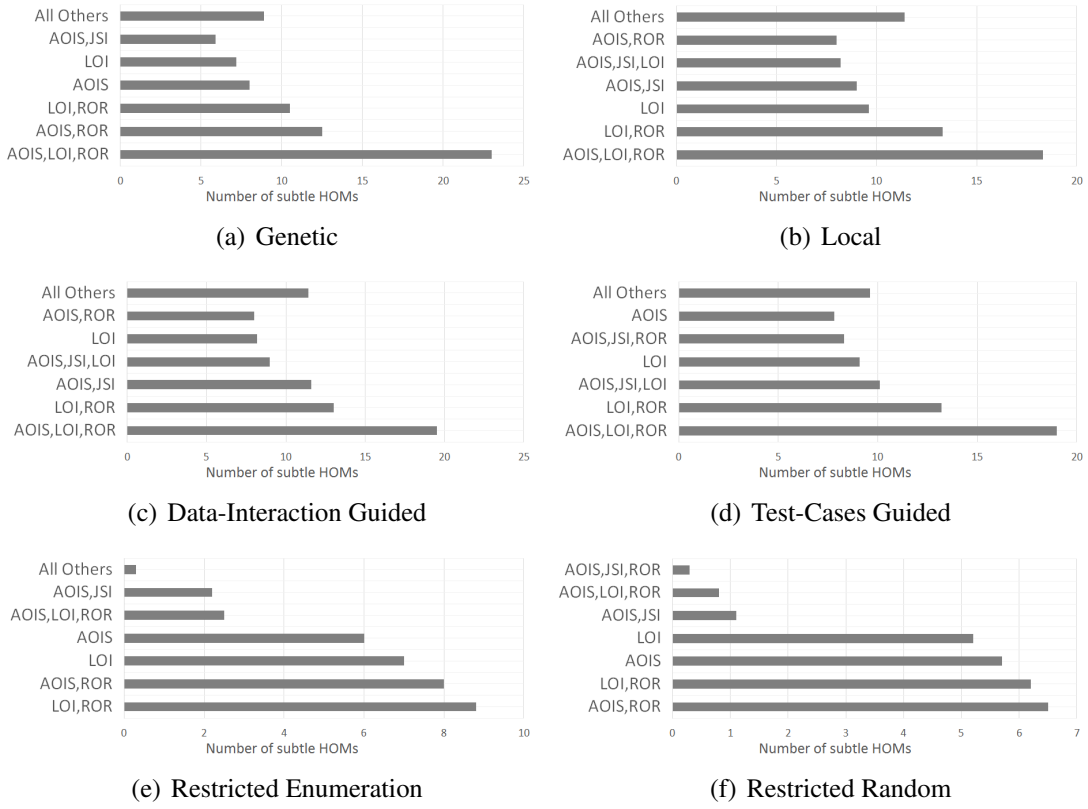
Figure 9.1 through 9.9 show the number of subtle HOMs with respect to the combination of mutation operators that created their constituent FOMs. For example, the Bar Chart in Figure 9.1 (a) shows that the Genetic Algorithm found 23 subtle HOMs, such that each of these subtle HOM was created by combining only FOMs created by the three mutation operators: Arithmetic Operator Insertion (AOIS), Logical Operator Insertion (LOI), and Relational Operator Replacement (ROR). The same Bar Chart also shows that the Genetic Algorithm found 13 subtle HOMs where each subtle HOM was created by combining only FOMs created by the two mutation operators: Arithmetic Operator Insertion (AOIS) and Relational Operator Replacement (ROR).

The results showed that more than 92% of subtle HOMs of all programs resulted from combining FOMs created by method-level mutation operators, which apply mutation faults to expressions that include Java primitive operators. The FOMs that were created by class-level and aspect-level

mutation operators produced a low number of subtle HOMs. Although no a specific set of method-level mutation operators seems to be more likely to create subtle HOMs, the mutation operators that created more subtle HOMs were those that created more FOMs for each subject programs.

In the remainder of this section, we further analyze the results and highlight the findings for each subject program.

1- Cruise Control (Java) Program



(a) Genetic

(b) Local

(c) Data-Interaction Guided

(d) Test-Cases Guided

(e) Restricted Enumeration

(f) Restricted Random

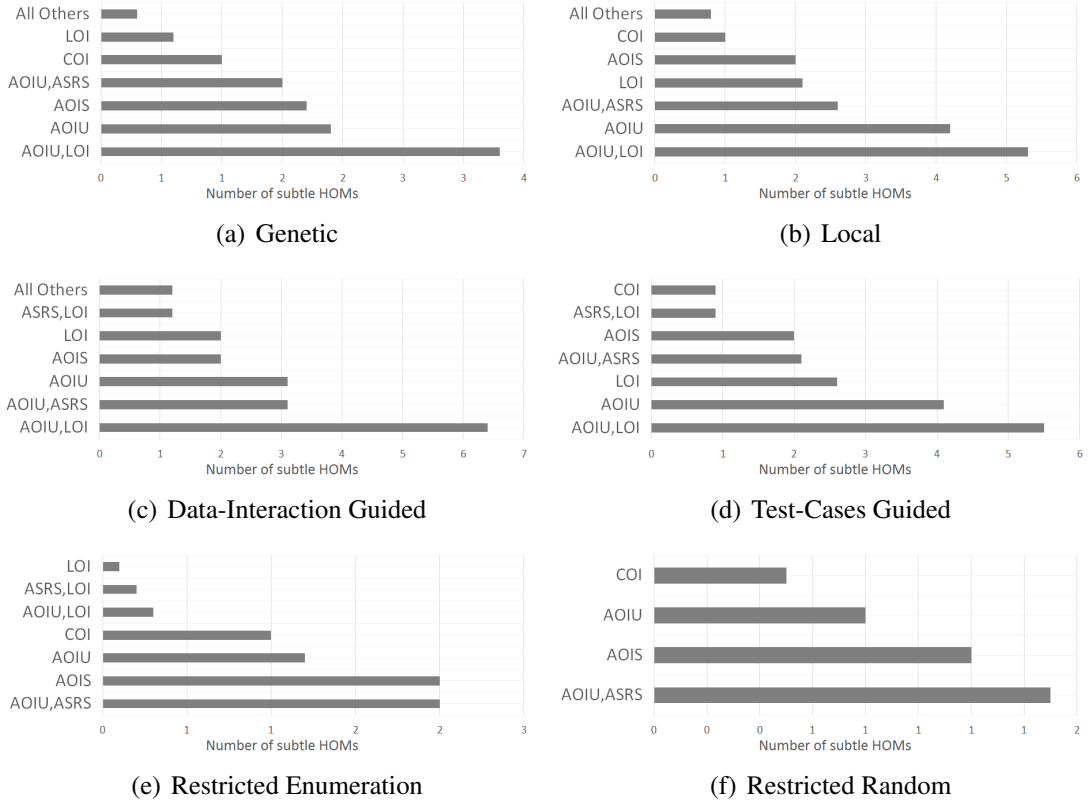
Figure 9.1: Distribution of Subtle HOMs based on their Mutation Operators for Cruise Control (Java)

The set of non-equivalent FOMs for the Cruise Control (Java) program was created by eight mutation operators. However, more than 75% of these FOMs were created using only three method-level mutation operators: Arithmetic Operator Insertion of Short-cut (AOIS), Relational Operator Replacement (ROR), and Logical Operator Insertion (LOI).

We analyzed the HOMs explored by the search techniques and found out that different combinations of these three mutation operators represented more than 60% of all HOMs explored by each search technique.

Figure 9.1 shows that the majority of subtle HOMs found by all search techniques included different combinations of FOMs created using the three mutation operators, AOIS, ROR, and LOI. However, the Genetic Algorithm, Local Search, and both the Guided Local Search techniques found a large number of subtle HOMs by combining only FOMs created by these three mutation operators. All search techniques found subtle HOMs by combining FOMs of Static Modifier Insertion (JSI), which is a class-level mutation operator, with FOMs of method-level mutation operators.

2- Telecom Program



(a) Genetic (b) Local

(c) Data-Interaction Guided (d) Test-Cases Guided

(e) Restricted Enumeration (f) Restricted Random

Figure 9.2: Distribution of Subtle HOMs based on their Mutation Operators for Telecom

The set of non-equivalent FOMs for the Telecom program was created by 18 mutation operators, which is higher than most other programs. However, more than 54% of the FOMs were created using only three method-level mutation operators: Arithmetic Operator Insertion of Short-cut (AOIS), Logical Operator Insertion (LOI), and Arithmetic Operator Insertion of basic Unary (AOIU).

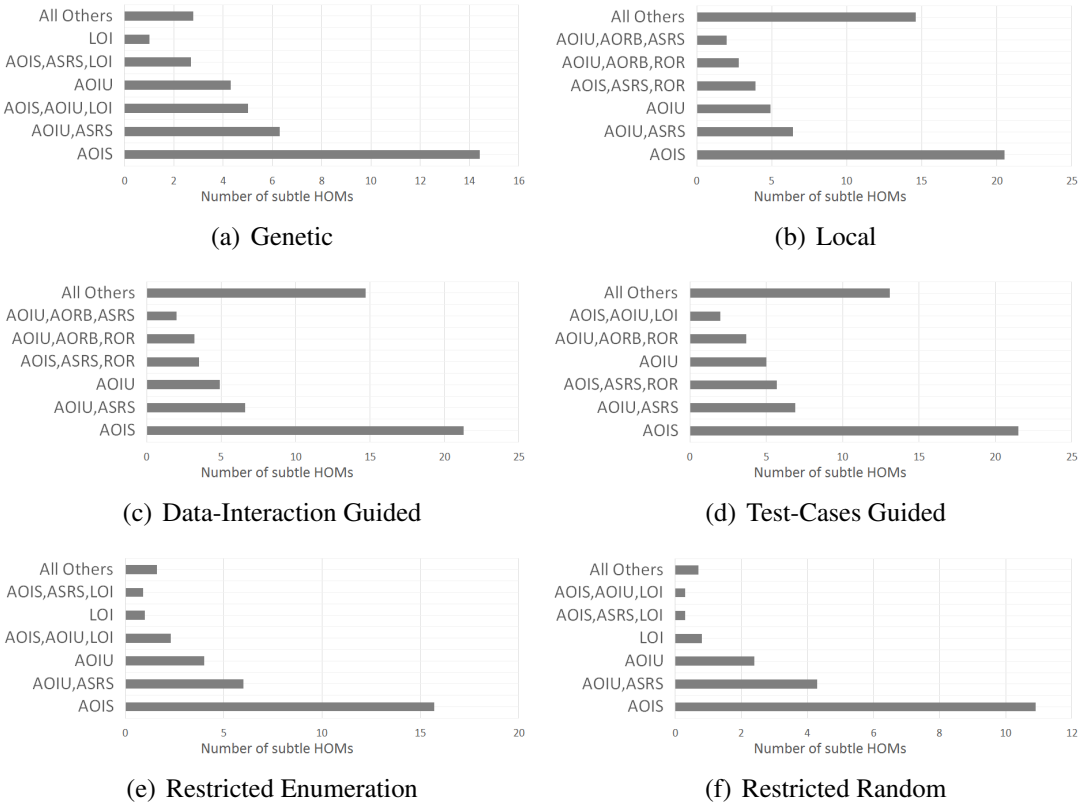
Although the three mutation operators generated more than 54% of FOMs, our analysis of the HOMs explored by the search techniques showed that the different combinations of these mutation operators represented less than 25% of all HOMs explored by each search technique. This is due to the large number of mutation operators for the Telecom program.

Figure 9.2 shows that the majority of subtle HOMs found by all search techniques included different combinations of the three mutation operators. All search techniques found subtle HOMs by combining only FOMs of AOIS.

3- Kettle Program

The set of non-equivalent FOMs for the Kettle program was created by ten mutation operators and more than 80% of these FOMs were created using the four method-level mutation operators: Arithmetic Operator Insertion of Short-cut (AOIS), Assignment Operator Replacement (ASRS), Relational Operator Replacement (ROR), and Arithmetic Operator Insertion of basic Unary (AOIU). Our analysis showed that different combinations of these mutation operators represented around 50% of all HOMs explored by each search technique.

Figure 9.3 shows that the majority of the subtle HOMs found by the search techniques represented different combinations of the four method-level mutation operators. However, all search techniques found subtle HOMs by combining only FOMs of AOIS and by combining only FOMs of AOIU.



(a) Genetic (b) Local

(c) Data-Interaction Guided (d) Test-Cases Guided

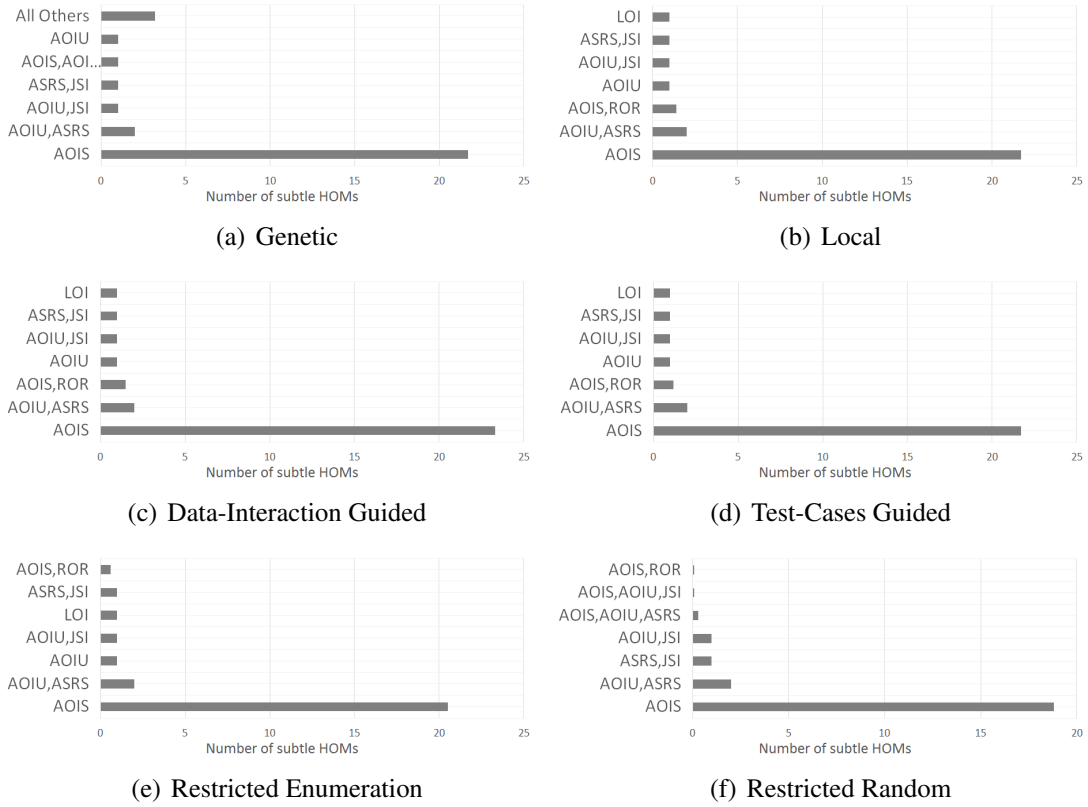
(e) Restricted Enumeration (f) Restricted Random

Figure 9.3: Distribution of Subtle HOMs based on their Mutation Operators for Kettle

4- Banking Program

The set of non-equivalent FOMs for the Banking program was created by eight mutation operators and more than 75% of these FOMs were created using three method-level mutation operators: Arithmetic Operator Insertion of Short-cut (AOIS), Assignment Operator Replacement (ASRS), and Relational Operator Replacement (ROR).

Our investigation revealed that different combinations of these three method-level mutation operators represented more than 50% of all HOMs explored by each search technique. However, the majority of the subtle HOMs found by all search techniques were created by combining only FOMs of AOIS. All search techniques found subtle HOMs by combining FOMs of Static Modifier Insertion (JSI), which is a class-level mutation operator, with FOMs of method-level mutation operators.



(a) Genetic (b) Local

(c) Data-Interaction Guided (d) Test-Cases Guided

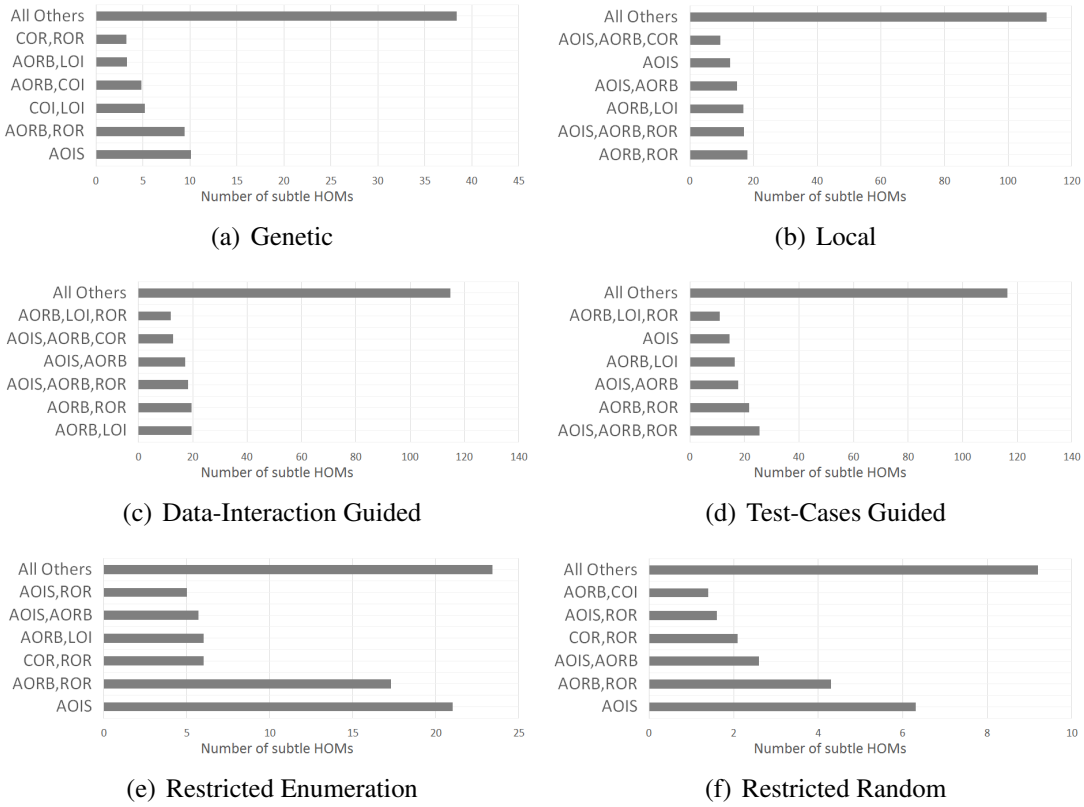
(e) Restricted Enumeration (f) Restricted Random

Figure 9.4: Distribution of Subtle HOMs based on their Mutation Operators for Banking

5- Coordinate Program

The set of non-equivalent FOMs for the Coordinate program was created using ten mutation operators and more than 71% of these FOMs were created using three method-level mutation operators: Arithmetic Operator Insertion of Short-cut (AOIS), Logical Operator Insertion (LOI), and Arithmetic Operator Replacement basic Binary (AORB).

Our investigation revealed that different combinations of the three method-level mutation operators represented less than 30% of all HOMs explored by each search technique. However, Figure 9.5 shows that different combinations of the three mutation operators represented the majority of the subtle HOMs found by each search technique. All search techniques found subtle HOMs by combining only FOMs of AOIS.



(a) Genetic (b) Local

(c) Data-Interaction Guided (d) Test-Cases Guided

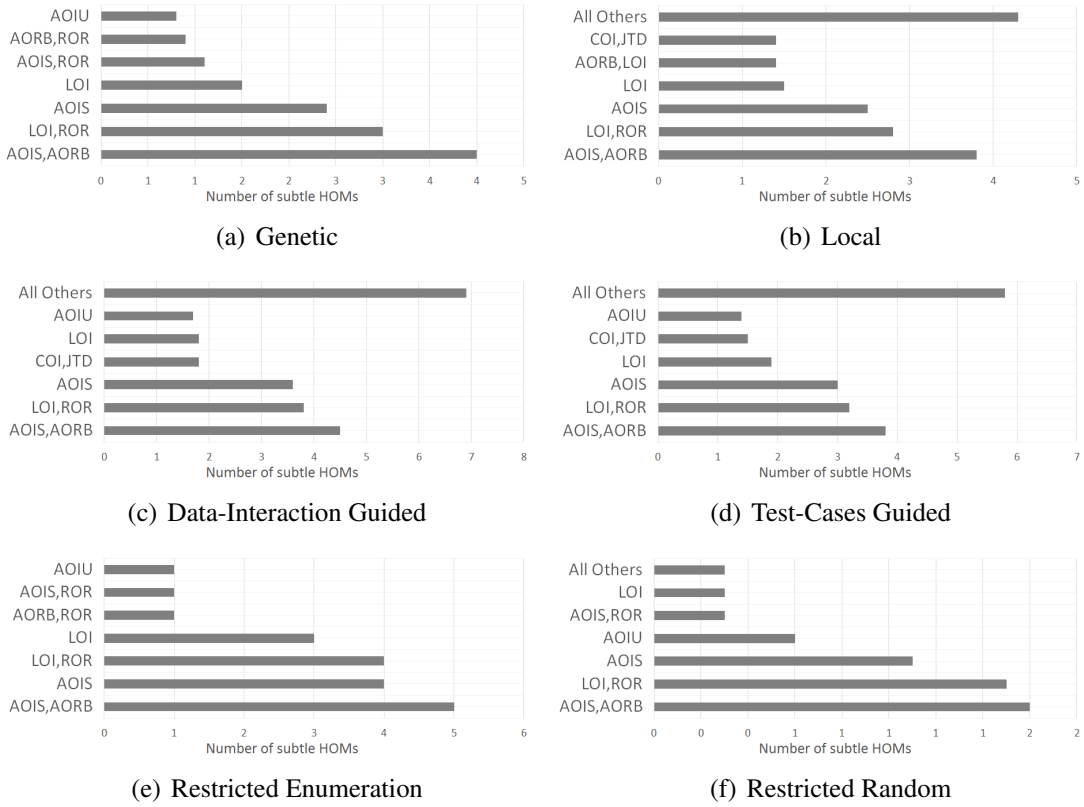
(e) Restricted Enumeration (f) Restricted Random

Figure 9.5: Distribution of Subtle HOMs based on their Mutation Operators for Coordinate

6- Elevator Program

The set of non-equivalent FOMs for the Elevator program was created by 14 mutation operators and 70% of these FOMs were created using four method-level mutation operators: Arithmetic Operator Insertion of Short-cut AOIS, Logical Operator Insertion (LOI), Relational Operator Replacement (ROR), and Conditional Operator Insertion (COI). Our investigation revealed that different combinations of these four mutation operators represented less than 25% of all HOMs explored by each search technique. This is due to the large number of mutation operators.

Different combinations of the four method-level mutation operators represented a large number of the subtle HOMs found by all search techniques. All search techniques found subtle HOMs by combining only FOMs of AOIS, by combining only FOMs of AOIU, and by combining only FOMs of LOI. Local Search and both the Guided Local Search techniques found subtle HOMs by combining only FOMs of COI.



(a) Genetic (b) Local

(c) Data-Interaction Guided (d) Test-Cases Guided

(e) Restricted Enumeration (f) Restricted Random

Figure 9.6: Distribution of Subtle HOMs based on their Mutation Operators for Elevator

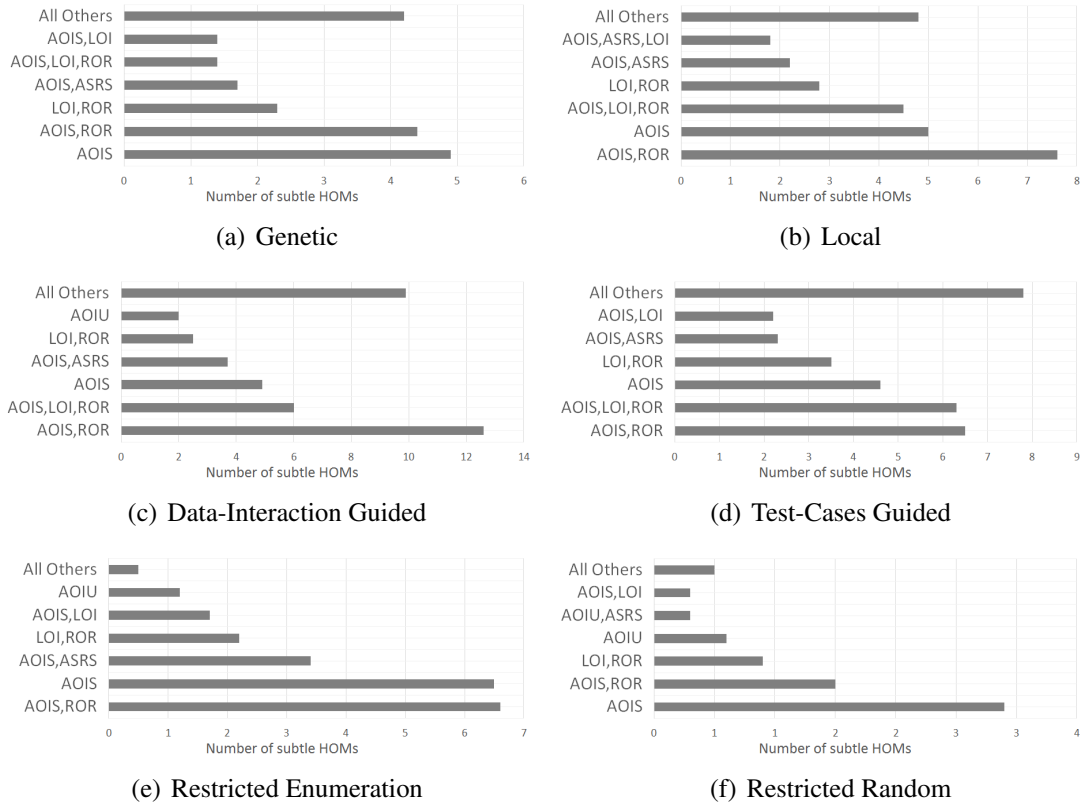
7- Cruise Control (AspectJ) Program

The set of non-equivalent FOMs for the Cruise Control (AspectJ) program was created by 12 mutation operators and 80% of these FOMs were created using four method-level mutation operators: Arithmetic Operator Insertion of Short-cut (AOIS), Relational Operator Replacement (ROR), Logical Operator Insertion (LOI), and Conditional Operator Insertion (COI).

Different combinations of the mutation operators AOIS, ROR, and LOI represented a large number of the subtle HOMs found by all search techniques. However, all search techniques found subtle HOMs by combining only FOMs of AOIS and by combining only FOMs of AOIU.

8- Roman Program

The set of non-equivalent FOMs for the Roman program was created by nine mutation operators and more than 75% of these FOMs were created using three method-level mutation operators: AOIS, ROR, and ASRS. Although different combinations of these three method-level mutation



(a) Genetic (b) Local

(c) Data-Interaction Guided (d) Test-Cases Guided

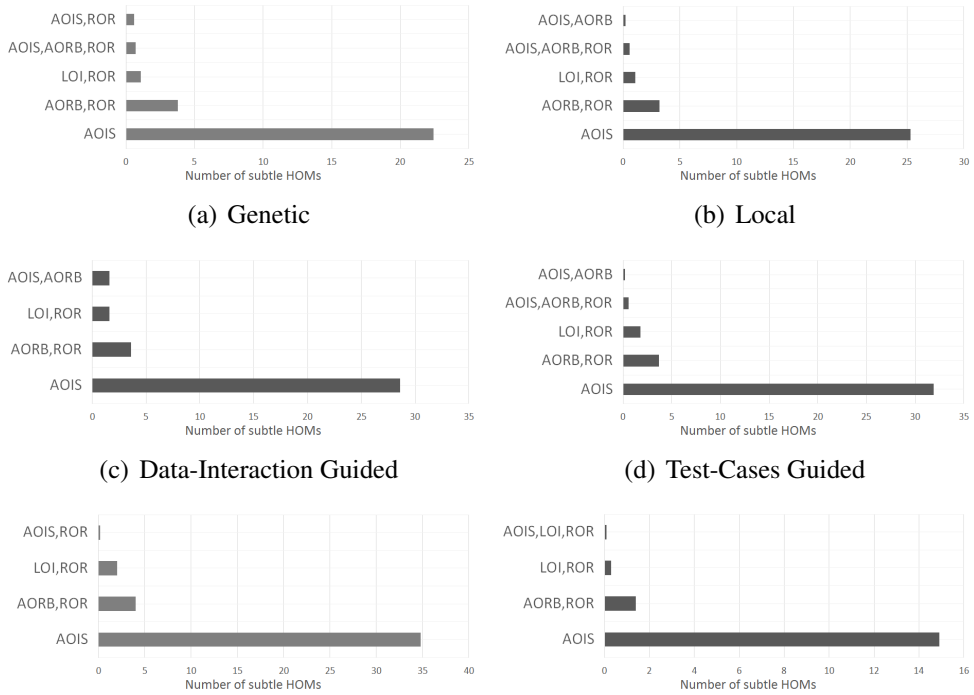
(e) Restricted Enumeration (f) Restricted Random

Figure 9.7: Distribution of Subtle HOMs based on their Mutation Operators for Cruise Control (AspectJ)

operators represented the majority of HOMs explored by all search techniques, the majority of the subtle HOMs found by all search techniques were created by combining only FOMs of AOIS.

9- XStream Program

The set of non-equivalent FOMs for the XStream program was created by 25 mutation operators and 46% of these FOMs were created using three method-level mutation operators: COI, ROR, and AOIS. The mutation operators that created a large number of FOMs were more successful in creating subtle HOMs. Similar to other programs, all search techniques found subtle HOMs by combining only FOMs of AOIS.



(a) Genetic (b) Local

(c) Data-Interaction Guided (d) Test-Cases Guided

(e) Restricted Enumeration (f) Restricted Random

Figure 9.8: Distribution of Subtle HOMs based on their Mutation Operators for Roman

(a) Genetic (b) Local

(c) Data-Interaction Guided (d) Test-Cases Guided

(e) Restricted Enumeration (f) Restricted Random

Figure 9.9: Distribution of Subtle HOMs based on their Mutation Operators for XStream

9.2.2 RQ2: Are subtle HOMs more likely to be created when combining mutated constructs from specific locations?

Part (a) in Figures 9.10 through 9.24 shows the number of subtle HOMs that were found by each search technique with respect to the location of their constituent FOMs. Part (b) in these figures shows the number of all HOMs that were explored by each search technique with respect to the location of their constituent FOMs.

Part (a) in Figs 9.12 through 9.22 shows the number of subtle HOMs that were found by each search technique with respect to their construction approach and part (b) shows the number of all explored HOMs with respect to their construction approach.

The results showed that subtle HOMs that were constructed by combining FOMs of the same statement and of the same method represented 35% and 31% respectively, of subtle HOMs that were found for all programs.

For five out of the ten programs, the majority of subtle HOMs found by all search techniques were constructed by combining FOMs of the same statement. For three programs, the majority of subtle HOMs were constructed by combining FOMs of the same method. Subtle HOMs constructed by combining FOMs of the same class represented the majority only for the Telecom program, and subtle HOMs constructed by combining FOMs from different classes represented the majority only for the Kettle program.

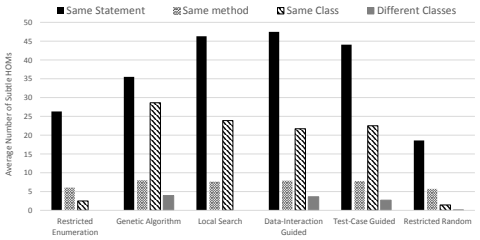
The results also showed that the Genetic Algorithm, Local Search and both the Guided Local Search techniques were able to find more subtle HOMs of the same class and subtle HOMs of different classes than Restricted Enumeration Search and Restricted Random Search. This is because the former four techniques were able to find more of subtle HOMs of higher degrees than the latter two techniques. Subtle HOMs of higher degrees have more FOMs than HOMs of lower degrees and that create higher chances to include FOMs from different methods and classes.

Because the majority of subtle HOMs for most AspectJ programs were constructed by combining FOMs of the same statement or same method, the majority of subtle HOMs for these programs were counted for Single Base Class or Aspect Approach (SCA). However, the search techniques found few subtle HOMs for all other construction approach. For one program, Kettle, the majority

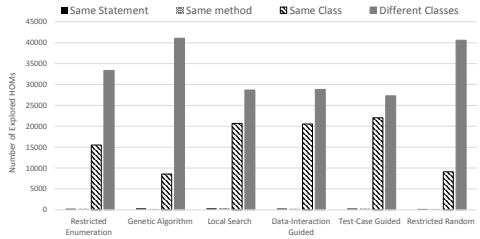
of subtle HOMs were constructed by Dispersed Base Class and Aspect Approach (BC&AS).

In the remainder of this section, we further analyze the results and discuss the findings for each subject program.

1- Cruise Control (Java) Program



(a) Distinct Subtle HOMs



(b) Explored HOMs

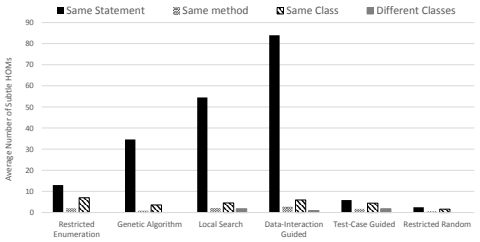
Figure 9.10: Distribution of HOMs based on the location of their constituent FOMs for Cruise Control (Java)

Figure 9.10 shows that the majority of HOMs explored by the search techniques were constructed by combining FOMs from different classes. The Cruise Control (Java) program has six Java classes, which is a quite large number of classes with respect to other program and that increased the chance of combining FOMs from different classes. However, because 93% of the non-equivalent FOMs for the Cruise Control (Java) program were created from only two classes, the number of HOMs constructed by combining FOMs of the same class is quite high as well.

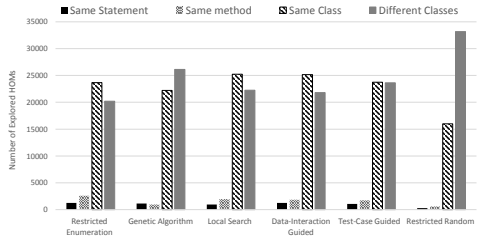
The majority of subtle HOMs found by all search techniques were constructed by combining FOMs of the same statement. However, the search techniques found subtle HOMs constructed by combining FOMs of the same method, same class, and different classes. The Genetic Algorithm, Local Search and both the Guided Local Search techniques found a higher number of subtle HOMs constructed by combining FOMs of the same class than the other techniques. The four techniques

also found few subtle HOMs constructed by combining FOMs from different classes. This is because the four techniques found more subtle HOMs of higher degrees than Restricted Enumeration Search and Restricted Random Search.

2- Movie Rental Program



(a) Distinct Subtle HOMs

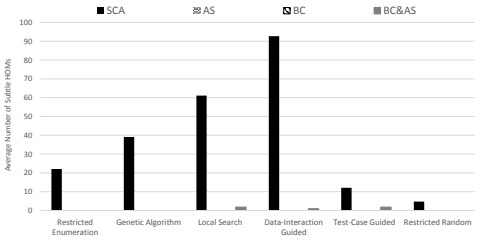


(b) Explored HOMs

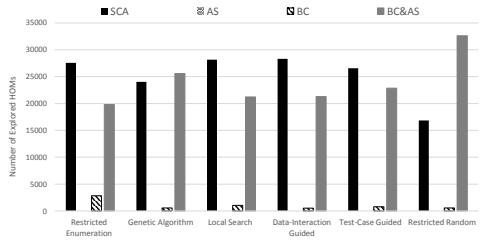
Figure 9.11: Distribution of HOMs based on the location of their constituent FOMs for Movie Rental

The number of explored HOMs constructed by combining FOMs from different classes is high because Movie Rental has four classes, which increased the chance of combining FOMs from different classes. The Movie Rental has three Java classes and one aspect and 74% of the non-equivalent FOMs were created from the aspect. For this reason the number of explored HOMs constructed by combining FOMs of the same class, which corresponds to Single Base Class or Aspect Approach (SCA), is also high. The number of explored HOMs corresponding to Dispersed Base Class and Aspect Approach (BC&AS) is high because this number represent all cases where at least on FOM of the aspect is combined with any number of FOMs from the other three Java classes.

The majority of subtle HOMs found by all search techniques were constructed by combining FOMs of the same statement, which corresponds to Single Base Class or Aspect Approach (SCA).



(a) Distinct Subtle HOMs



(b) Explored HOMs

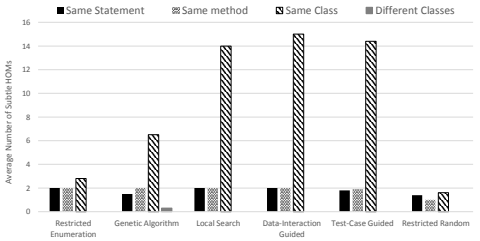
Fig 9.12: Distribution of HOMs based on their construction approach for Movie Rental

However, the search techniques found few subtle HOMs by combining FOMs of the same method and same class. Local Search and both the Guided Local Search techniques found few subtle HOMs by combining FOMs from different classes. The three techniques also found few subtle HOMs corresponding to Dispersed Base Class and Aspect Approach (BC&AS).

3- Telecom Program

The number of explored HOMs constructed by combining FOMs from different classes is the highest because the Telecom program has ten classes, three aspects and seven Java classes. Figure 9.14 shows that the number of explored HOMs that correspond to Dispersed Base Class and Aspect Approach (BC&AS) represent the majority of explored HOMs. This is because 44% of the non-equivalent FOMs for Telecom were created from the three aspects.

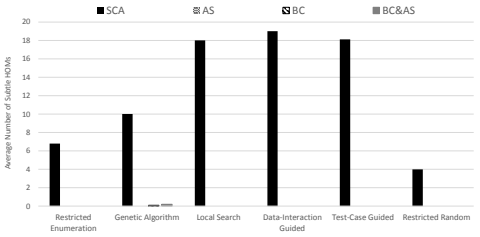
The majority of subtle HOMs found by all search techniques were constructed by combining FOMs of the same class. However, all search techniques found few subtle HOMs by combining FOMs of the same statement and same method. These three types of HOMs correspond to Single Base Class or Aspect Approach (SCA), which is why it has higher number of subtle HOMs than the other construction approaches.



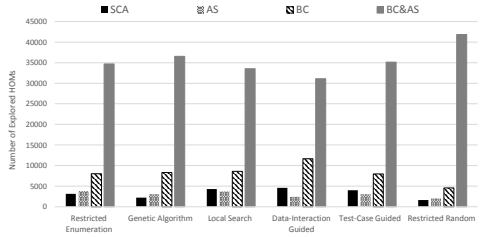
(a) Distinct Subtle HOMs

(b) Explored HOMs

Figure 9.13: Distribution of HOMs based on the location of their constituent FOMs for Telecom



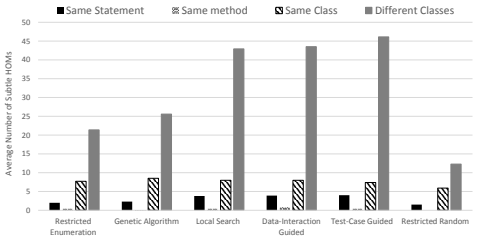
(a) Distinct Subtle HOMs



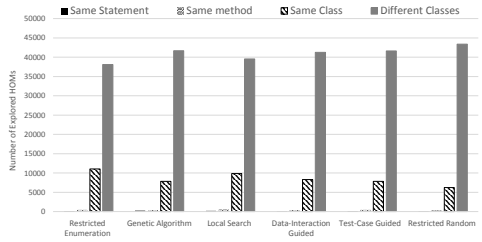
(b) Explored HOMs

Fig 9.14: Distribution of HOMs based on their construction approach for Telecom

The Genetic Algorithm found few subtle HOMs that were constructed by combining FOMs from different classes. Most of these were high degree subtle HOMs that correspond to Dispersed Base Class Approach (BC) or Dispersed Base Class and Aspect Approach (BC&AS).



(a) Distinct Subtle HOMs



(b) Explored HOMs

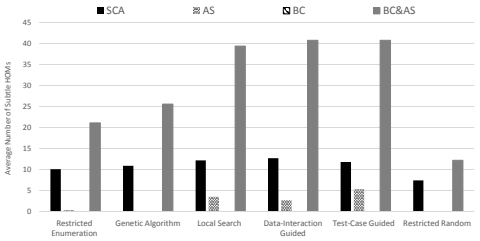
Figure 9.15: Distribution of HOMs based on the location of their constituent FOMs for Kettle

4- Kettle Program

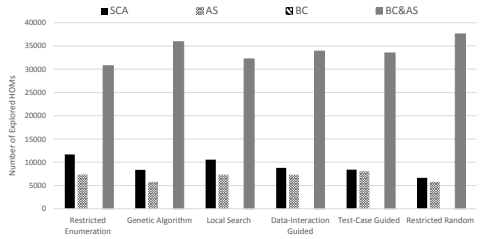
The majority of subtle HOMs found by all search techniques for Kettle were constructed by combining FOMs from different classes and the majority of those subtle HOMs correspond to Dispersed Base Class and Aspect Approach (BC&AS). The Kettle program is one of the smallest programs and it has two aspects and one Java class. The two aspects contain five advices that are executed after every method in the Java class. This results in high interaction between the methods and advices of this program and any mutation applied to these advices affects every method and functionality in that program. However, all search techniques found subtle HOMs by combining FOMs of the same class and same statement. Local Search and both the Guided Local Search techniques found few subtle HOMs by combining FOMs of the same method. Local Search and both the Guided Local Search techniques also found few subtle HOMs constructed by Dispersed Aspect Approach (AS).

5- Banking Program

The majority of the explored HOMs were constructed by combining FOMs from different classes and the majority of these HOMs correspond to Dispersed Base Class and Aspect Approach



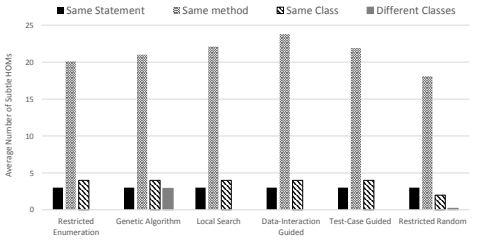
(a) Distinct Subtle HOMs



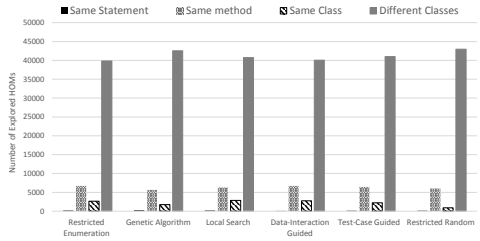
(b) Explored HOMs

Fig 9.16: Distribution of HOMs based on their construction approach for Kettle

(BC&AS). The Banking program has four classes, two Java classes and two aspects, and the numbers of non-equivalent FOMs are evenly distributed between the aspects and base classes.



(a) Distinct Subtle HOMs

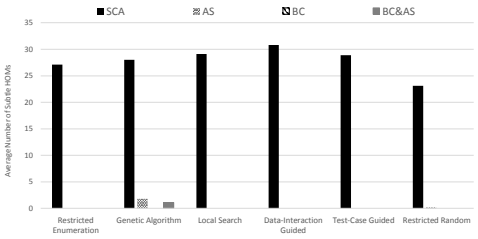


(b) Explored HOMs

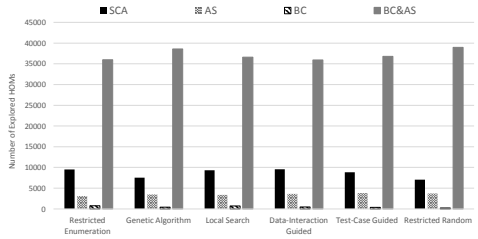
Figure 9.17: Distribution of HOMs based on the location of their constituent FOMs for Banking

The majority of subtle HOMs found by all search techniques were constructed by combining FOMs of the same method. However, all search techniques found subtle HOMs constructed by

combining FOMs of the same class and same statement. The Genetic Algorithm found subtle HOMs constructed by combining FOMs from different classes.



(a) Distinct Subtle HOMs



(b) Explored HOMs

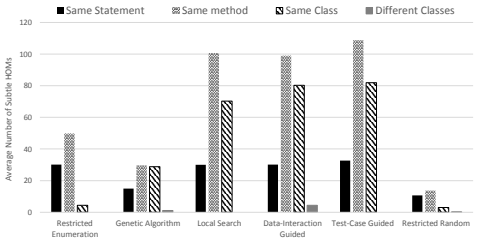
Fig 9.18: Distribution of HOMs based on their construction approach for Banking

6- Coordinate Program

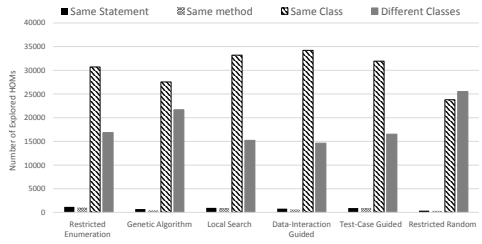
Figure 9.19 shows that the number of explored HOMs constructed by combining FOMs of the same class is higher than the number of explored HOMs constructed by combining FOMs from different classes. This is because the Coordinate program has two classes and 84% of the non-equivalent FOMs were generated only from one class.

All search techniques found subtle HOMs by combining FOMs of the same statement, same method, and same class. However, subtle HOMs constructed by combining FOMs of the same method represented the majority of subtle HOMs found by all search techniques.

Restricted Enumeration Search and Restricted Random Search found a low number of subtle HOMs constructed by combining FOMs of the same class compared to the other techniques. This is because Restricted Enumeration Search and Restricted Random Search found a low number of subtle HOMs of higher degrees compared to the other search techniques. The Genetic Algorithm, Data-Interaction Guided Local Search, and Restricted Random Search found few subtle HOMs by



(a) Distinct Subtle HOMs

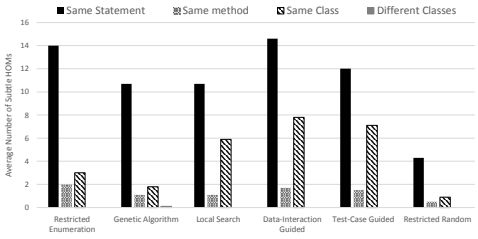


(b) Explored HOMs

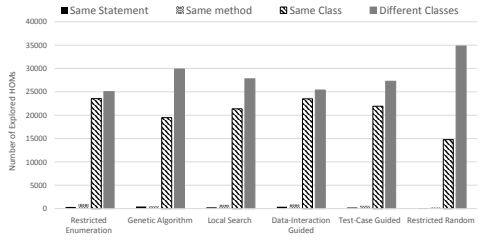
Figure 9.19: Distribution of HOMs based on the location of their constituent FOMs for Coordinate

combining FOMs from different classes.

7- Elevator Program



(a) Distinct Subtle HOMs



(b) Explored HOMs

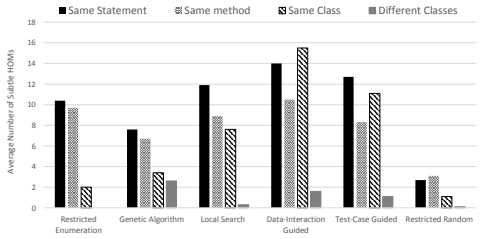
Figure 9.20: Distribution of HOMs based on the location of their constituent FOMs for Elevator

The majority of HOMs explored by all search techniques were constructed by combining FOMs from different classes. Elevator program has 17 Java classes, which increased the chance of com-

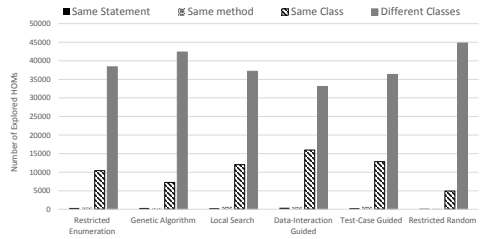
bining FOMs from different classes. However, because 73% of the non-equivalent FOMs for the Elevator program were generated from only one class, the number of explored HOMs constructed by combining FOMs of the same class is quite high as well.

The majority of subtle HOMs found by all search techniques were constructed by combining FOMs of the same statement. However, all search techniques found subtle HOMs constructed by combining FOMs of the same method and same class. Local Search and both the Guided Local Search techniques found more subtle HOMs of the same class than the other techniques. The Genetic Algorithm found few subtle HOMs constructed by combining FOMs from different classes.

8- Cruise Control (AspectJ) Program



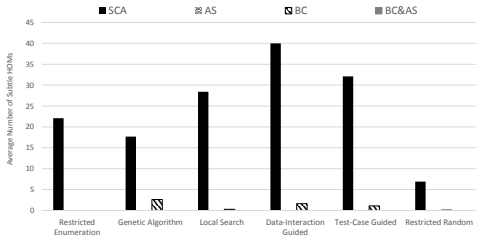
(a) Distinct Subtle HOMs



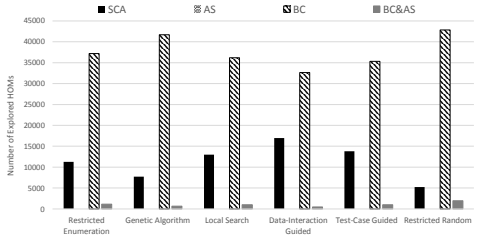
(b) Explored HOMs

Figure 9.21: Distribution of HOMs based on the location of their constituent FOMs for Cruise Control (AspectJ)

The number of explored HOMs constructed by combining FOMs from different classes is the highest because the Cruise Control (AspectJ) program has 12 classes, three aspects and nine Java classes, and that increased the chance of combining FOMs from different classes. The number of explored HOMs that correspond to Dispersed Base Class and Aspect Approach (BC&AS) is low because the FOMs created from the aspects represent less than 2% of all the non-equivalent FOMs.



(a) Distinct Subtle HOMs



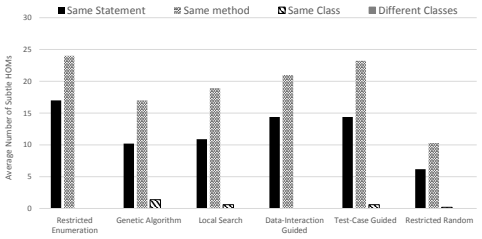
(b) Explored HOMs

Fig 9.22: Distribution of HOMs based on their construction approach for Cruise Control (AspectJ)

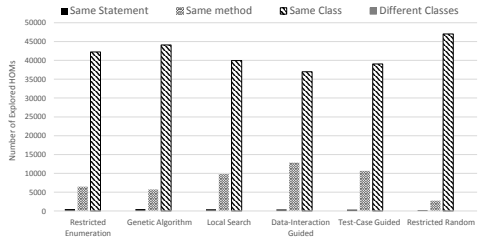
The search techniques found subtle HOMs by combining FOMs of the same statement, same method, and same class. The three types of HOMs correspond to Single Base Class or Aspect Approach (SCA). The Genetic Algorithm found few subtle that were constructed by combining FOMs from different classes. Most of these subtle HOMs are of high degrees and correspond to Dispersed Base Class Approach (BC) or Dispersed Base Class and Aspect Approach (BC&AS).

9- Roman Program

Although the Roman program has two classes, all non-equivalent FOMs were created from one class. That is why no HOMs were created by combining FOMs from different classes. The majority of subtle HOMs found by the search techniques were created by combining FOMs of the same method and same statement. The Genetic Algorithm, Local Search, Test-Case Guided Local Search, and Restricted Random Search found few subtle that were constructed by combining FOMs of the same class, witch are mostly HOMs of higher degrees.



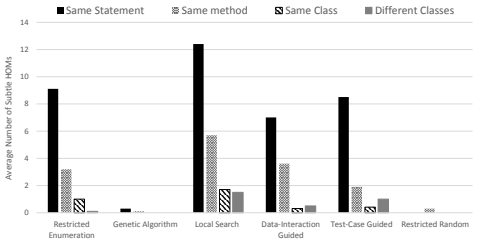
(a) Distinct Subtle HOMs



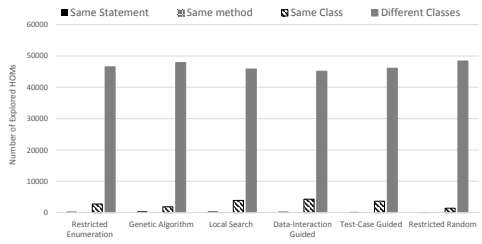
(b) Explored HOMs

Figure 9.23: Distribution of HOMs based on the location of their constituent FOMs for Roman

10- XStream Program



(a) Distinct Subtle HOMs



(b) Explored HOMs

Figure 9.24: Distribution of HOMs based on the location of their constituent FOMs for XStream

The XStream program has the largest number of classes amongst the ten subject programs, 318 Java classes, which explain why the large majority of explored HOMs were created by combining FOMs from different classes.

The majority of subtle HOMs found by search techniques were constructed by combining FOMs of the same statement. However, the search techniques found subtle HOMs by combining FOMs of the same method and same class. Local Search and both the Guided Local Search techniques found few subtle HOMs by combining FOMs from different classes.

9.3 Summary of findings

More than 92% of the subtle HOMs that were found resulted from combining FOMs that applied mutation to primitive Java operators, such as replacing, deleting, and inserting a primitive operator. FOMs that applied mutation to OOP and AOP constructs and expressions resulted in a low number of subtle HOMs.

More than 67% of the subtle HOMs that were found were constructed by combining FOMs that applied mutation to different tokens within the same statement of code or different statements of code within the same method of a class/aspect.

Chapter 10

Cost of Finding Subtle HOMs

This chapter presents a study to measure the cost of finding subtle HOMs. We present the research questions motivating the study then we present the results and analysis.

10.1 Research Questions

The study is motivated by the following research questions.

RQ1: What is the computational cost of finding subtle HOMs using the search techniques?

We measured the computational cost of finding subtle HOMs in terms of the time taken to find subtle HOMs. We separately ran each technique once per subject program on an isolated machine, and measured the time taken by each technique to explore, create, compile, and execute 50,000 distinct HOMs.

The exploration time refer to the time taken by the different operators of the search techniques to create and maintain the XML records that correspond to an HOM. The creation time refer to the time taken to write the program classes that correspond to the HOM in the physical drive to be compiled and executed. The compilation time refer to the time to generate the binary code from the created classes. The execution time refers to the time taken by the Java virtual machine to execute the Junit test suites on the binary code that corresponds to the HOMs.

First, we measured the cost with respect to the overall time, which includes the time to explore, create, compile, and execute 50,000 distinct HOMs. For each technique, we calculated the average time over the ten programs.

Second, we investigated the cost of the different operators of the search techniques and we measured the exploration time for each technique. For each technique, we calculated the average exploration time over the ten programs.

Last, we investigated the reduction in the computational cost that resulted from optimizing the compilation process. We configured HOMAJ to rely only on Java and AspectJ virtual machines to perform the compilation. This means that HOMAJ recompiles all class and aspect files for each HOM. We then run HOMAJ over the ten programs and calculated the overall time for each technique to explore, create, compile, and execute 50,000 distinct HOMs.

RQ2: What proportion of subtle HOMs that were found constitutes equivalent mutants?

The presence of equivalent mutants is a major obstacle for the practical use of mutation testing. HOMAJ does not identify equivalent HOMs and it treats them as subtle HOMs because they are not killed by the test suites. The presence of equivalent HOMs amongst the subtle HOMs can increase the cost of finding subtle HOMs because of the additional human effort needed to identify equivalent HOMs. A high proportion of equivalent HOMs can degrade the practical value of subtle HOMs because it increases the cost.

For each program, we identified the non-equivalent subtle HOMs and measured the proportion of equivalent HOMs. In addition, we investigated the difficulty of killing non-equivalent subtle HOMs using new randomly generated test cases. We added 8963 randomly generated test cases to our original test suites of the ten programs, and measured the number of subtle HOMs that were killed by the new test suites.

Because subtle HOMs are harder to kill than the FOMs, the new test cases were more exhaustive than the test cases that were generated to kill all the FOMs. The new test cases randomly varied in size from 20 to 100 method calls per test case, while the test cases that were generated to kill all the FOMs randomly varied from two to 20 method calls per test case. We did not generate new test suites for the XStream program because we used all the test cases from the 60 test suites that were available with the program. Equivalent HOMs for the XStream program were manually identified.

Table 10.1 shows the number of test cases that killed all the FOMs and the number of new test cases that were generated to kill subtle HOMs.

Table 10.1: Number of test cases that killed all FOMs and some of the subtle HOMs

Subject program	# of test cases that killed all FOMs	# of test cases that were generated to kill subtle HOMs.
Coordinate	14	1290
Roman	11	876
Cruise (Java)	18	818
Elevator	14	1017
XStream	96	0
Kettle	12	912
Movie Rental	15	1015
Banking	9	1012
Telecom	10	1115
Cruise (AspectJ)	26	908

For each program, we calculated the set of all distinct, subtle HOMs that were found. This set represent the union of all sets of subtle HOMs that were found by all techniques over the 30 runs. We then evaluated this set of subtle HOMs against the new test set and calculated the number of non-equivalent subtle HOMs that were killed by the new test set. Finally, we manually inspected the remaining subtle HOMs that were not killed and identified equivalent HOMs.

10.2 Results and Analysis

This section presents answers for the research questions and highlights the general findings.

10.2.1 RQ1: What is the computational cost of finding subtle HOMs using the search techniques?

Table 10.2 shows the average times taken by the search techniques over the ten subject programs. The time is presented in terms of hours:minutes:second (hh:mm:ss). The second column shows the average exploration time for each technique. The third column shows the time taken to explore, create, compile without optimization, and execute 50,000 distinct HOMs. The last column shows the same time in the second column when the compilation process is optimized using a makefile.

The major contributor to the cost of finding subtle HOMs is the compilation and execution process of HOMs. The exploration time represented a small fraction of the total time taken by

Table 10.2: Average time for finding subtle HOMs

Search Technique	Exploration Time	Overall time	Overall time when compilation is optimized
Restricted Enumeration	0:00:46	18:52:05	11:35:5
Local	0:10:23	19:15:17	13:02:6
Test-Case Guided	0:08:55	19:51:01	11:44:0
Data-Interaction Guided	0:08:46	19:06.0	11:35:07
Restricted Random	0:00:36	18:03:04	13:58:5
Genetic	0:11:49	19:38:6	11:01:25

each search technique. However, Restricted Random Search and Restricted Enumeration Search have the lowest average exploration times. This is because the other four techniques use more operators to create and manipulate the XML records of the HOMs. For example, the Genetic Algorithm uses selection, crossover, and then mutation to create a population of HOMs, which explains why the Genetic Algorithm had the highest average exploration time.

Optimizing the compilation process reduced the computational cost of finding subtle HOMs by each technique by more than 32%.

10.2.2 RQ2: What proportion of subtle HOMs that were found constitutes equivalent mutants?

Figure 10.1: Subtle HOMs and Equivalent HOMs

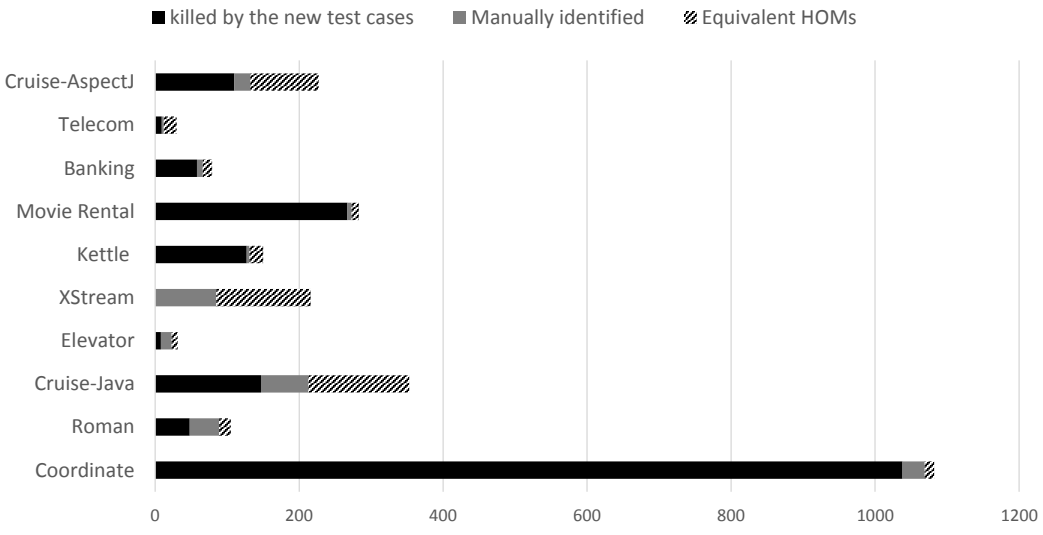


Table 10.3: Subtle HOMs and Equivalent HOMs

Subject Program	All subtle HOMs	# Non-equivalent Subtle HOMs		Equivalent HOMs
		Killed by new test cases	Manually identified	
Coordinate	1082	1038	31	13 (1%)
Roman	105	48	41	16 (15%)
Cruise (Java)	353	147	66	140 (40%)
Elevator	31	8	15	8 (26%)
XStream	216	0	85	131 (61%)
Kettle	150	127	4	19 (13%)
Movie Rental	283	267	6	10 (4%)
Banking	79	58	8	13 (17%)
Telecom	30	9	3	18 (60%)
Cruise (AspectJ)	227	110	22	95 (42%)
Total	2556	1812	281	463 (18%)

In Table 10.3, the second column shows the number of all distinct, subtle HOMs that were found by the search techniques. The third column shows the number of non-equivalent subtle HOMs that were killed by the new test suites. The fourth column shows the number of non-equivalent subtle HOMs that were not killed by the new test suites and manually identified. The last column shows the number of subtle HOMs that are considered to be equivalent HOMs and their percentages out of all subtle HOMs.

Although the list of FOMs for each program did not include any equivalent FOMs, the search techniques found equivalent 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 resulted second order mutant is equivalent to the original program.

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

The set of all subtle HOMs for all ten programs contained 2556 subtle HOMs. Our investigation revealed that 463 of these subtle HOMs, which represent around 18%, are considered to be equivalent HOMs. The remaining subtle HOMs were indeed subtle (i.e., not equivalent), and they exposed weaknesses in the fault-detection effectiveness of the test suites that killed all the FOMs for the ten programs.

The new test suites that were generated to kill subtle HOMs were much larger in size and more exhaustive than the test suites that killed all the FOMs. However, the new test suites still have weaknesses in their fault-detection effectiveness because they did not kill all the subtle HOMs. We manually identified 281 subtle HOMs that were not equivalent and not killed by the new test suites. These subtle HOMs required a specific set of input combinations that were not generated by the test input generator tool. These subtle HOMs represented 14% of the set of all non-equivalent subtle HOMs.

For eight out of the ten programs, the search techniques found more non-equivalent subtle HOMs than equivalent HOMs. For the XStream and Telecom programs, the search techniques found more equivalent HOMs than non-equivalent subtle HOMs.

10.3 Summary of findings

More than 98% of the cost of finding subtle HOMs was related to compiling and executing HOMs. Restricted Random Search and Restricted Enumeration Search have the lowest average exploration times because they use less operators to create and manipulate the XML records of the HOMs. Optimizing the compilation process of HOMs reduced the computational cost of finding subtle HOMs by 32%.

For all subject programs, 82% of the subtle HOMs were non-equivalent subtle HOMs. The equivalent HOMs were treated as subtle in this work because they cannot be killed by test suites. Of the non-equivalent subtle HOMs, 14% were not killed by test suites that were much larger in size and much more exhaustive than the test suites that killed all the FOMs for the subject programs.

Chapter 11

Composition and Decomposition Relationships Between Subtle HOMs

This chapter presents a study to investigate different factors that affect the discovery of subtle HOMs of higher degrees. Subtle HOMs of degree four and higher were harder to find than subtle HOMs of lower degrees and that is due to two reasons. First, increasing the degree of an HOM by adding more FOMs makes it easier to kill in most cases and that can cause the number of subtle HOMs of higher degrees to be low.

Second, the exponential growth in the number of HOMs makes it even harder to find subtle HOMs of higher degrees. For example, the number of HOMs for the Banking program, which has the smallest search space, grew exponentially until degree 13 resulting in over 38 million HOMs. Finding subtle HOMs of higher degrees in such a large search space is not easy even for search-based software engineering techniques, especially when the search techniques were allowed to explore only 50,000 distinct HOMs due to the high computational cost. Increasing the number of explored HOMs might give the search techniques the chance to find more subtle HOMs of higher degree but the computational cost will increase as well. For example, exploring and evaluating one million distinct HOMs for the Banking program can take up to 10 days, which might not be practical from a tester's point of view, and exploring and evaluating 38 million HOMs is not feasible.

We investigated alternative techniques that can be more effective for finding subtle HOMs of higher degrees. We investigated composing subtle HOMs that were found by the Restricted Enumeration Search to create new subtle HOMs of higher degrees. In addition, we analyzed the subtle HOMs of higher degrees that were found by the search techniques to determine to what extent subtle HOMs of higher degrees represent a composition of subtle HOMs of lower degrees.

In the remainder of this chapter, we present the research questions motivating the study and analysis of the results. Then we summarize the general findings.

11.1 Research Questions

The study aimed to answer the following research questions.

RQ1: Can subtle HOMs be composed to create new subtle HOMs of higher degrees?

For each program, we selected the set of subtle HOMs found by the median run of Restricted Enumeration Search and composed these subtle HOMs in two different ways to create new HOMs. First, we created new HOMs by composing all pairs of subtle HOMs. Second, we created new HOMs by using all combinations of three subtle HOMs.

Some of the newly created HOMs cannot include all the constituent FOMs of the composed subtle HOMs. Such HOMs result from cases where the composed subtle HOMs contain FOMs that cannot be composed in one HOM because they apply mutation to the same token in the same statement of code. Such HOMs were ignored. There are also cases when combining two Second Order Mutants (SOMs) result in a third order mutant instead of a fourth order mutant because both of the SOMs have a common FOM. Such HOMs were considered. We then evaluated the newly created HOMs to determine how many of them represent new subtle HOMs.

RQ2: To what extent do subtle HOMs of higher degrees represent a composition of subtle HOMs of lower degrees?

We classified subtle HOMs based on how they can be decomposed into three types: (1) fully decomposable, (2) partially decomposable, and (3) not decomposable into other subtle HOMs. A subtle HOM is considered to be fully decomposable if all of its constituent FOMs can be composed to form other subtle HOMs, partially decomposable if some of its constituent FOMs can be composed to form other subtle HOMs, and not decomposable if none of its constituent FOMs can be composed to form other subtle HOMs. We investigated the number of subtle HOMs that were found by each search technique with respect to their decomposition type.

For each program, we selected the sets of subtle HOMs that were found by the best run out of the 30 runs for each search technique. The best run represents the case where the search technique found the highest number of subtle HOMs. From each set, we selected the subtle HOMs of degrees higher than two because they can be decomposed into other HOMs. Each subtle HOM was decomposed into a number of HOMs that represent all subcombinations of the constituent FOMs. In other words, each subtle HOM was treated as a set of FOMs and we created all possible subsets that represent HOMs.

We evaluated the newly created HOMs and calculated the number of subtle HOMs that were fully decomposable, partially decomposable, and not decomposable into other subtle HOMs.

RQ3: How often subtle HOMs of higher degrees strongly subsume their decomposed subtle HOMs of lower degrees?

We conducted an initial investigation of the implications of HOM subsumption relationships [8] on subtle HOMs. The subsumption relationships were introduced by Jia and Harman [8] are defined in terms of an HOM and its constituent FOMs. Jia and Harman defined a strongly subsuming HOM as one that can replace its constituent FOMs without loss of test effectiveness. That is, if a test case kills a strongly subsuming HOM, it also kills all its constituent FOMs. As a result, strongly subsuming HOMs reduce the number of FOMs that need to be evaluated and the number of test cases that need to be executed.

We redefined the subsumption relationships in terms of a subtle HOM of higher degree and its decomposed subtle HOMs of lower degrees. That is, a subtle HOM of higher degree strongly subsumes its decomposed subtle HOMs of lower degrees if the set of test cases that kills the subtle HOM of higher degree will also kill all its decomposed subtle HOMs of lower degrees. We performed a preliminary investigation on how often a set of test cases that kills a subtle HOM of higher degrees will also kill all its decomposed subtle HOMs of lower degrees. The strongly subsuming subtle HOMs of higher degree can reduce the total number of test cases that need to be developed and the number of subtle HOMs that need to be evaluated.

From a tester's point of view, developing fewer test cases that kill a high degree subtle HOM and its decomposed subtle HOMs is cost effective.

11.2 Results and Analysis

This section presents the results for the two research questions.

11.2.1 RQ1: Can subtle HOMs be composed to create new subtle HOMs of higher degrees?

Table 11.1 shows the results of composing subtle HOMs. For example, Table 11.1 shows that the median run of the Restricted Enumeration Search for the Elevator program produced 19 subtle HOMs (second column). We obtained a set of 162 HOMs (fourth column and second row) when we composed two subtle HOMs at a time, and a set of 818 HOMs (fourth column and third row) when we composed three subtle HOMs at a time. Both new sets of HOMs do not contain any HOMs that existed in the set of the 19 subtle HOMs.

Evaluating HOMs showed that the set of 162 HOMs contained 161 new subtle HOMs, such that all of them were of degree four (column six). The set of 818 HOMs contained 801 new subtle HOMs, such that all of them were of degree six. The last column shows that 99.4% of the HOMs created by composing two subtle HOMs at a time were subtle; 97.9% of the HOMs created by composing three subtle HOMs at a time were subtle.

Table 11.2 shows the number of all distinct, subtle HOMs that were found by the search techniques (second column) over the 30 runs and the number of all distinct, subtle HOMs that were found by composing subtle HOMs that were found by Restricted Enumeration Search (third column).

Composing subtle HOMs is an effective way for finding new subtle HOMs of higher degrees. For all ten programs, the number of subtle HOMs that were found by composing subtle HOMs that were found by Restricted Enumeration Search was higher than the number of all subtle HOMs that were found by all search techniques, see Table 11.2. Overall, 88% of all HOMs created by composing subtle HOMs represented new subtle HOMs.

Table 11.1: Composing subtle HOMs to create new subtle HOMs

Program	# of subtle HOMs	# of composed subtle HOMs	# of generated HOMs	# of new subtle HOMs by degree						Percentage of new subtle HOMs
				3	4	5	6	7	8+	
Elevator	19	2	162	0	161	0	0	0	0	99.4
		3	818	0	0	0	801	0	0	97.9
Cruise (Java)	34	2	465	9	346	92	3	0	0	96.8
		3	3418	0	2	276	1988	785	60	91
Roman	40	2	673	4	659	0	0	0	0	98.5
		3	6332	0	0	108	5946	0	0	95.6
XStream	14	2	73	1	70	0	0	0	0	97.2
		3	237	0	1	8	212	0	0	93.2
Coordinate	83	2	2944	38	2713	74	0	0	0	96
		3	11987	0	10156	610	60	13	0	90.4
Telecom	7	2	20	0	14	5	0	0	0	95
		3	30	0	0	0	16	9	0	83.3
Banking	27	2	233	0	139	40	0	0	0	76.8
		3	967	0	18	25	344	127	0	53.2
Kettle	30	2	275	0	128	71	7	0	0	74.9
		3	1105	0	7	8	215	237	64	48.1
Cruise (AspectJ)	22	2	199	0	94	18	0	0	0	56.3
		3	983	0	0	6	362	78	0	45.4
Movie Rental	22	2	190	12	160	0	0	0	0	90.5
		3	833	0	14	144	448	0	0	72.7

Table 11.2: Comparing the number of subtle HOMs that were found by the search techniques and by composing subtle HOMs that were found by Restricted Enumeration Search

Program	# of all subtle HOMs that were found by the search techniques	# of all subtle HOMs that were found by composing subtle HOMs
Elevator	31	962
Cruise (Java)	355	3464
Roman	105	6717
XStream	216	291
Telecom	30	44
Banking	79	650
Kettle	150	724
Cruise (AspectJ)	227	558
Movie Rental	283	764

Composing two subtle HOMs is more likely to produce new subtle HOMs than composing three subtle HOMs. 93% of all HOMs created by composing two subtle HOMs represented new subtle HOMs; while 87% of all HOMs created by composing three subtle HOMs represented new subtle HOMs.

Composing subtle HOMs for Java programs produced a higher percentage of new subtle HOMs than for AspectJ programs. Our investigation revealed that 97% of all HOMs created by composing two subtle HOMs for all Java programs were subtle, while 75% of all HOMs created by composing two subtle HOMs for all AspectJ programs were subtle. This is probably because some of the FOMs in the AspectJ programs are technically HOMs and that is due to the nature of aspect weaving in AspectJ programs. Since an aspect advice is potentially woven in multiple locations of the base code, a single fault in an advice gets woven in multiple places, resulting in an HOM. This can increase the degree of the composed subtle HOMs making them less likely to be subtle. Composing less number of subtle HOMs is more likely to produce new subtle HOMs.

The Telecom program was the only AspectJ program for which the percentage of new subtle HOMs was comparable to that generated from the Java programs. This is explained by the observation that the majority of FOMs that constitute the subtle HOMs in the Telecom program are base class FOMs (i.e., FOMs created from the Java classes in the Telecom program).

11.2.2 RQ2: To what extent do subtle HOMs of higher degrees represent a composition of subtle HOMs of lower degrees?

Figure 11.1 shows that subtle HOMs of higher degrees that were found by the search techniques are more likely to be decomposable into other subtle HOMs. The higher the degree of the subtle HOM, the higher the chance it was decomposable. For example, all of the subtle HOMs of degree five and higher were either fully or partially decomposable, 95% of those with degree four were either fully or partially decomposable, and 74% of those with degree three were either fully or partially decomposable.

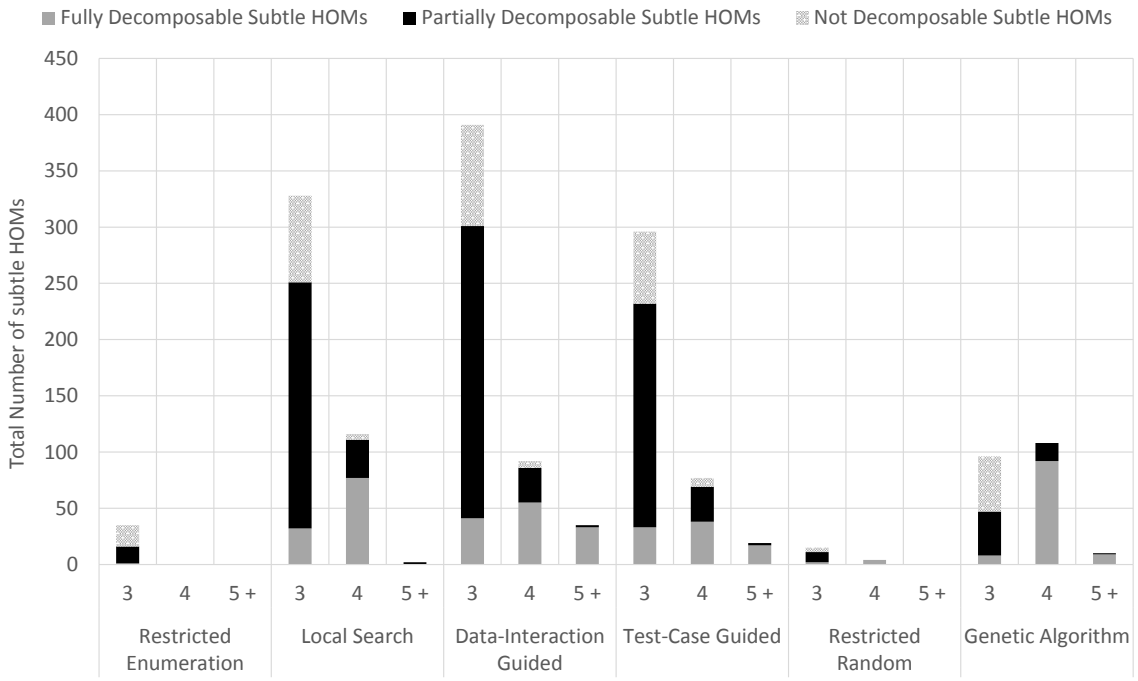


Figure 11.1: Number of subtle HOMs that were found by the search techniques with respect to their decomposition type for all subject programs
The set of all subtle HOMs that were found by the best run for Local Search for all programs contained 382 subtle HOMs of degree three, 116 subtle HOMs of degree four, and two subtle HOMs of degree five. The 382 subtle HOMs of degree three were such that, 32 fully decomposable, 219 partially decomposable, and 77 not decomposable into other subtle HOMs.

This observation raises an important question of whether or not subtle HOMs of higher degrees can only exist as a composition of other subtle HOMs. If this were true, then it would be more effective and practical for a tester to use Restricted Enumeration Search to enumerate all Second Order Mutants (SOMs) and then explore different combinations of these SOMs to create new subtle

HOMs of higher degrees. This approach can produce a higher number of subtle HOMs than any of the six search techniques with less cost. However, for large programs, such as the XStream, enumerating all SOMs can be expensive and the search-based software engineering techniques can be more effective for such programs.

To further analyze the results, we investigated whether the search-based software engineering techniques can find subtle HOMs of higher degrees that cannot be found when composing subtle HOMs that were found by Restricted Enumeration Search. We investigated the overlap between the set of subtle HOMs of higher degrees found by composing subtle HOMs found by Restricted Enumeration Search and the set of subtle HOMs of higher degrees that were found by the search-based software engineering techniques. We used the best runs out of the 30 runs for each search technique.

Table 11.3 shows the number of subtle HOMs of degree three and higher that were found by the best run. Table 11.4 shows the number of subtle HOMs that overlap between the sets of subtle HOMs in Table 11.3 and the sets of the new subtle HOMs of higher degrees that were found by composing subtle HOMs. For example, Table 11.3 shows that the best run of Local Search for the Coordinate program found 217 subtle HOMs of degree three and higher and Table 11.4 shows that 20 out of the 217 subtle HOMs were also found by composing subtle HOMs that were found by Restricted Enumeration Search.

Table 11.3: Number of subtle HOMs of degree three and higher that were found by the best run for each technique

Program	Local	Data-Interaction Guided	Test-Case Guided	Genetic
Elevator	7	9	9	0
Cruise (Java)	51	66	66	62
Roman	13	2	9	4
XStream	5	3	2	0
Coordinate	217	252	222	52
Telecom	14	17	16	9
Banking	11	11	9	14
Kettle	35	42	39	15
Cruise (AspectJ)	21	31	26	14
Movie Rental	77	97	0	49
Total	451	530	398	219

Table 11.4: Overlap between the sets of subtle HOMs of higher degrees

Program	Local	Data-Interaction Guided	Test-Case Guided	Genetic
Elevator	0	0	0	0
Cruise (Java)	7	8	13	36
Roman	2	0	2	3
XStream	0	0	0	0
Coordinate	20	38	15	19
Telecom	0	0	0	0
Banking	0	0	0	7
Kettle	0	0	0	4
Cruise (AspectJ)	0	0	0	0
Movie Rental	6	6	0	0
Total	35	52	30	69

Although composing subtle HOMs produced a large number of subtle HOMs of higher degrees, the search-based software engineering techniques found many subtle HOMs of higher degrees that could not be found by composing subtle HOMs that were found by Restricted Enumeration Search. The set of subtle HOMs of higher degrees found by composing subtle HOMs that were found by Restricted Enumeration Search contained 32%, 10%, 8%, and 8% of the subtle HOMs of higher degrees that were found by the best runs of the Genetic Algorithm, Data-Guided Local Search, Test-Case Guided Local Search, and Local Search for all programs, respectively.

Figure 11.1 shows that the different operators implemented in the search techniques affected the type of subtle HOMs that were found by each technique. The Genetic Algorithm found a higher proportion of fully decomposable subtle HOMs than the other techniques; and Local Search and both the Guided Local Search techniques found a higher proportion of partially decomposable subtle HOMs than the other techniques.

The combination of elite selection and crossover probably caused the Genetic Algorithm to find a higher proportion of fully decomposable subtle HOMs. The selection of subtle HOMs as elite HOMs and crossing them over (i.e., combining them) creates a higher chance for creating subtle HOMs that are fully decomposable into other subtle HOMs. This trend can also be seen from the results in Table 11.4, where 32% of the subtle HOMs of higher degrees that were found by the Genetic Algorithm were also found by composing subtle HOMs that were found by Restricted

Enumeration Search. The selection of the best neighbor and the neighborhood graph probably caused Local Search and both the Guided Local Search techniques to find a higher proportion of partially decomposable subtle HOMs. When a subtle HOM is selected as the incumbent HOM, the neighborhood graph will create neighboring HOMs that include the subtle HOM. This process creates a higher chance for creating subtle HOMs that are partially decomposable into other subtle HOMs.

11.2.3 RQ3: How often subtle HOMs of higher degrees strongly subsume their decomposed subtle HOMs of lower degrees?

We selected three programs where we evaluated the set of all composed subtle HOMs that were found against the new set of test cases that were generated to kill subtle HOMs. We then selected the set of composed subtle HOMs that were killed by the new test cases and evaluated how many of these composed subtle HOMs strongly subsume their decomposed subtle HOMs of lower degrees. The strongly subsuming subtle HOMs represent the subtle HOMs of higher degrees that can replace their decomposed subtle HOMs of lower degree without loss of test effectiveness.

In Table 11.5, the second column shows the number of all distinct, composed, subtle HOMs that were killed by the new set of test cases that were developed to kill subtle HOMs. The third column shows how many of these composed subtle HOMs strongly subsumed their decomposed lower degrees subtle HOMs.

Table 11.5: Strongly subsuming subtle HOMs of higher degrees

Program	# of subtle HOMs that were killed by the test cases	# of strongly subsuming subtle HOMs of higher degrees
Elevator	387	0
Cruise (Java)	348	0
Coordinate	3891	1327

For the first two programs, none of the composed subtle HOMs that were killed by the new set of test cases strongly subsumed their decomposed subtle HOMs. This means that these composed subtle HOMs represent different faults than their decomposed subtle HOMs. Thus, the tester needs to develop new test cases to kill the decomposed lower degree subtle HOMs and that can increase

test effectiveness. For the Coordinate program, 34% of the composed subtle HOMs that were killed by the new set of test cases strongly subsumed their decomposed subtle HOMs. These strongly subsuming subtle HOMs reduce test effort.

11.3 Summary of findings

Subtle HOMs of higher degrees can be effectively found by composing subtle HOMs. Composing subtle HOMs that were found by Restricted Enumeration Search produced a high number of subtle HOMs while exploring a relatively small number of HOMs. However, the search-based software engineering techniques can find many subtle HOMs of higher degrees that cannot be found by composing subtle HOMs that were found by Restricted Enumeration Search.

Subtle HOMs of higher degrees are more likely to be decomposable into other subtle HOMs of lower degrees. All subtle HOMs of degree four and higher that were found by the search techniques were either fully or partially decomposable into other subtle HOMs. However, not all subtle HOMs of higher degrees can substitute their decomposed subtle HOMs of lower degrees without loss of test effectiveness.

The results suggest that we perform more empirical studies to further investigate the implication of subsumption relationships amongst subtle HOMs. The strongly subsuming subtle HOMs can reduce test effort and the non-strongly subsuming subtle HOMs can increase test effectiveness. However, we need to investigate what characteristics of subtle HOMs cause them to be strongly subsuming subtle HOMs and to what extent can strongly subsuming subtle HOMs reduce test effort.

The results also suggest that we perform further empirical studies to investigation to determine whether high degree subtle HOMs that are composed of other subtle HOMs can further improve the fault-detection effectiveness of the test suite when we already added test cases that kill the decomposed subtle HOMs of lower degrees.

Chapter 12

Threats to Validity

We identified four types of threats to the validity of our empirical studies: external validity, internal validity, construct validity, and conclusion validity [118, 119]. External validity refers to how well the results can be generalized outside the scope of the study. Internal validity is concerned with cause and effect relationships: the extent to which we can state that the changes in dependent variables are caused by changes in independent variables. Construct validity refers to the meaningfulness of measurements. Conclusion validity refers to whether the conclusions about the relationship among variables based on the data are correct. Below we summarize the threats to validity of our empirical studies.

12.1 External Validity

One threat to external validity is that the selected subject programs may not be representative of AspectJ and Java programs in general, and thus, the results of the study may not be generalizable to all AspectJ and Java programs. However, the selected programs differ in size and complexity and contain various OOP and AOP constructs. Furthermore, many of these programs have been used in various empirical studies by the mutation testing research community.

Another threat to external validity stems from the mutation operators that were used to generate FOMs. We used both MuJava [30] and AjMutator [33] to generate FOMs. Using different tools that implement different mutation operators to generate FOMs can lead to different results. However, MuJava is the most commonly used mutation testing tool for Java programs and it covers most of the mutation operators defined in the mutation testing literature. AjMutator on the other hand, implements only pointcut mutation operators and it is the only mutation tool that is publicly available for AspectJ programs. We used MuJava to apply mutations to the inter-type declaration methods and advices. This technique was previously used by Wedyan and Ghosh [11].

12.2 Internal Validity

A threat to internal validity stems from the quality of the test suites and the number of test cases. There is a possibility of getting different results if we used test suites that are created with different test objectives. However, the used test suites kill all the FOMs, which is the precondition for finding subtle HOMs. Furthermore, the used test suites achieved statement coverage.

The configuration of the parameters of the search techniques represents a threat to internal validity. An optimal configuration that leads to best results can be hard to identify. However, in this dissertation we used experimental evaluation to determine the configuration of the parameters that gave the best results.

12.3 Construct validity

Some of the FOMs are technically HOMs because of the nature of aspect weaving in AspectJ programs. Since an aspect advice is potentially woven in multiple locations of the base code, a single fault in an advice gets woven in multiple places, resulting in an HOM. However, the traditional practice is to defined FOMs and HOMs with respect to the number of syntactic changes to the source code. Nevertheless, because of aspect weaving, the order of an HOM may have been higher than the number of its constituent FOMs.

The use of the number of explored distinct HOMs as a quasi-representation of time instead of actual time represents a threat to construct validity. Using actual time is a better way to compare the relative effectiveness of the search techniques. This is because the search techniques use different operators to generate and manipulate the XML records of HOMs, which can cause the search techniques to explore different numbers of HOMs in the same amount of time. The search techniques might also explore the same HOM more than once although such HOMs are counted only one time. Running each technique 30 times per subject program on isolated machines requires a long time or a large number of isolated machines. However, because the time taken by the different operators of the search techniques represented a fraction (less than 2%) of the total time, the number of explored HOMs can be used as a quasi-representation of time.

A threat to construct validity stems from the ability to measure the workload on the machines used to measure the computational cost. The machines were isolated and had identical specifications, and we controlled the processes running on these machines during the experiments.

A threat to construct validity stems from the correctness of the implementation of MuJava, AjMutator, and HOMAJ. The results presented in this dissertation depend on the three tools working correctly. For each tool, we inspected randomly selected outputs and manually verified their correctness. The manual identification of equivalent mutants is another threat to construct validity.

12.4 Conclusion validity

A threat to conclusion validity stems from the stochastic nature of the proposed search techniques. However, we ran each technique 30 times per subject program and drew the conclusions based on statistical measures. We also used the Analysis of Variance (ANOVA) test and Chi-Square test to demonstrate the significance of the results.

Chapter 13

Conclusions

We developed a set of search techniques for finding subtle HOMs in the context of Java and AspectJ programming languages. We developed four search-based software engineering techniques: (1) the Genetic Algorithm, (2) Local Search, (3) Test-Case Guided Local Search, and (4) Data-Interaction Guided Local Search. We also developed a Restricted Random Search technique and a Restricted Enumeration Search technique.

We developed an objective function that provides a metric to measure the fitness of HOMs. The search techniques used the objective function to identify subtle HOMs as well as HOMs that have the potential to develop into subtle HOMs when the right FOMs are added or removed.

We developed HOMAJ, a Higher Order Mutation Testing tool for AspectJ and Java programs for finding subtle HOMs. HOMAJ implements the developed search techniques and automates the process of creating, compiling, and executing both FOMs and HOMs.

We performed a set of empirical studies to evaluate the effectiveness of the search techniques in term of their ability to find subtle HOMs. We investigated different factors that affect the creation of subtle HOMs. Below we summarize the findings of the empirical studies.

1- Measuring the relative effectiveness of the search techniques

All the search techniques were able to find subtle HOMs. However, Local Search and both the Guided Local Search techniques were more effective than the other techniques in terms of their ability to find subtle HOMs. The combination of the fitness evaluation and the neighborhood graph resulted in a better strategy for finding subtle HOMs.

Data-Interaction Guided Local Search was more effective than Test-Case Guided Local Search. The Genetic Algorithm was more effective for AspectJ programs than for Java programs.

The majority of subtle HOMs that were found by all search techniques were of second and third degrees. Subtle HOMs of higher degrees were harder to find because increasing the degree

of an HOM by adding more FOMs makes it easier to kill in most cases, and that can cause the number of subtle HOMs of higher degrees to be low.

Although Restricted Random Search was less effective than the other techniques, it found a higher average number of subtle HOMs than a pure random search technique for all ten subject programs. This shows that limiting the search to the space of lower degree HOMs is a good strategy.

The Genetic Algorithm, Local Search, and both the Guided Local Search techniques were more effective than Restricted Enumeration for finding subtle HOMs of higher degrees (third and higher). For smaller programs, Restricted Enumeration Search was more effective than the other techniques at finding subtle SOMs.

Data-Interaction Guided Local Search was more effective than the other techniques for finding a higher number of distinct, subtle HOMs. For nine programs, Data-Interaction Guided Local Search found on an average 94% of the subtle SOMs while it explored only 16% of the space of SOMs.

2- Comparing the sets of subtle HOMs found by different search techniques

The different operators implemented in the search techniques affect what set of subtle HOMs can be uniquely found by each technique. The different operators allowed the search techniques to find more of the uniquely found (hardest-to-find) subtle HOMs than the commonly found (easiest-to-find) subtle HOMs. More than 94% of the hardest-to-find subtle HOMs were of degree three and higher, and more than 92% of the easiest-to-find subtle HOMs were subtle SOMs.

Using the four search-based software engineering techniques is more likely to produce a large number of distinct, subtle HOMs that can be used to improve the fault-detection effectiveness of test suites.

3- Impact of programming language constructs on creating subtle HOMs

Combining mutation faults of Java primitive operators is more likely to create subtle HOMs. Mutation faults of OOP and AOP constructs and expressions produced a low number of subtle

HOMs. Subtle HOMs that were constructed by combining FOMs of the same statement and of the same method represented 66% of subtle HOMs that were found for all programs.

4- Cost of finding subtle HOMs

The compilation and execution process of HOMs is the major contributor to the cost of finding subtle HOMs. The computational cost of the operators of the search technique represented less than 2% of the overall cost of finding subtle HOMs.

Equivalent HOMs represented 18% of subtle HOMs found for all ten programs. To be killed, non-equivalent subtle HOMs required test suites that were much larger in size and more exhaustive than the test suites that killed all the FOMs. Furthermore, 14% of these non-equivalent subtle HOMs required a specific set of input combinations that needed to be manually identified.

5- Composition and decomposition relationships between subtle HOMs

Subtle HOMs of higher degrees are more likely to exist as compositions of multiple subtle HOMs of lower degrees. However, not all subtle HOMs of higher degrees can substitute their decomposed subtle HOMs of lower degrees without loss of test effectiveness. Composing subtle HOMs of lower degrees that were found by Restricted Enumeration Search is an effective way for finding new subtle HOMs of higher degrees. This approach produced a large number of subtle HOMs of higher degrees while exploring a relatively small number of HOMs. However, the search-based software engineering techniques were able to find subtle HOMs of higher degrees that could not be found by composing subtle HOMs of lower degrees.

Chapter 14

Future Work

This chapter presents various opportunities for improving the work presented in this dissertation and discusses new research directions in the area of higher order mutation testing.

1- Impact of subtle HOMs on test input generation techniques

There is a lack of test input generation techniques that target HOMs. Current test generation techniques generate test inputs that allow the program execution to reach a mutated statement. However, test inputs that kill subtle HOMs need to allow the program execution to reach multiple mutated statements, and these mutated statements might need to be executed in a specific sequence in some cases. Such requirements bring new challenges and thus, further research is needed to investigate the impact of subtle HOMs on test input generation techniques.

2- Finding subtle HOMs of higher degrees

Subtle HOMs of higher degrees were harder to find and a large number of the subtle HOMs that were found were partially or fully decomposable into other subtle HOMs. The search techniques did not find any subtle HOMs of degree five or higher that is not decomposable into other subtle HOMs for any of the ten subject programs. Such subtle HOMs can be beneficial for improving the fault-detection effectiveness of test suites because they can reveal unexpected behavior of the program under test.

3- Reducing the computational cost of finding subtle HOMs

Researchers have proposed various techniques to reduce the cost associated with the compilation and execution of FOMs [52, 27, 83, 84, 85, 86] but not HOMs. HOMAJ performs selective compilation to reduce the cost of compiling HOMs but there is a chance to further reduce the compilation and execution cost of HOMs. Other FOMs cost reduction techniques can be de-

veloped for HOMs by extending compiler-integrated techniques [83], mutant schema generation techniques [84, 85], and modified bytecode techniques [86].

5- Impact of equivalent HOMs

There is a lack of techniques for identifying equivalent HOMs. Researchers proposed techniques for eliminating some equivalent FOMs but not HOMs. We propose to investigate extending some of the existing techniques to identify some equivalent HOMs and thus reduce the manual effort to identify non-equivalent subtle HOMs.

6- Generalization of the empirical results

The used subject programs may not be representative of Java and AspectJ programs in general, and thus, the results of may not be generalizable to all Java and AspectJ programs. The empirical studies need to be replicated using larger programs and using test suites that are created with different test objectives. Increasing the fault-detection effectiveness of the test suites can impact the number of subtle HOMs that were found.

7- Configuring the parameters of the search techniques

Further research is needed to investigate the implications of the configurations on how the search techniques explore the search space and what subtle HOMs can be found.

References

[1] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.

[2] R. Lipton, "Fault diagnosis of computer programs," tech. rep., Carnegie Mellon University, 1971.

[3] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[4] T. A. Budd, *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, 1980.

[5] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, vol. 3, pp. 279–290, July 1977.

[6] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[7] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in *International Conference on Software Engineering*, pp. 402–411, 2005.

[8] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

[9] 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.

[10] R. Gopinath, C. Jensen, and A. Groce, "Mutations: How close are they to real faults?," in *International Symposium on Software Reliability Engineering*, 2014.

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

[12] The AspectJ Team, "AspectJ Compiler 1.6.12." http://www.eclipse.org/aspectj/. 2011.

[13] 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.

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

[15] M. Kintis, M. Papadakis, and N. Malevris, "Isolating first order equivalent mutants via second order mutation," in *International Conference on Testing, Verification and Validation*, pp. 701–710, 2012.

[16] E. Omar, S. Ghosh, and D. Whitley, "Constructing subtle higher order mutants for Java and AspectJ programs," in *International Symposium on Software Reliability Engineering*, pp. 340–349, 2013.

[17] E. Omar, S. Ghosh, and D. Whitley, "HOMAJ: A tool for higher order mutation testing in AspectJ and Java," in *Workshop on Mutation Analysis*, pp. 165–170, 2014.

[18] E. Omar, S. Ghosh, and D. Whitley, "Comparing search techniques for finding subtle higher order mutants," in *Genetic and evolutionary computation conference*, pp. 1271–1278, 2014.

[19] P. Gong, R. Zhao, and Z. Li, "Faster mutation-based fault localization with a novel mutation execution strategy," in *Software Testing, Verification and Validation, Mutation Workshops*, pp. 1–10, IEEE, 2015.

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

[21] K. H. T. Wah, "An analysis of the coupling effect I: single test data," *Science of Computer Programming*, vol. 48, no. 23, pp. 119 – 161, 2003.

[22] M. Harman, A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys*, vol. 45, no. 1, p. 11, 2012.

[23] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833 – 839, 2001.

[24] P. R. Srivastava and T.-h. Kim, "Application of genetic algorithm in software testing," *International Journal of software Engineering and its Applications*, vol. 3, no. 4, pp. 87–96, 2009.

[25] A. Arcuri, "On the automation of fixing software bugs," in *International Conference on Software Engineering, Doctoral Symposium*, pp. 1003–1006, 2008.

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

[27] J. Offutt and K. N. King, "A Fortran 77 interpreter for mutation analysis," *Symposium on Interpreters and Interpretive Techniques*, vol. 22, pp. 177–188, 1987.

[28] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, "Design of mutant operators for the C programming language," tech. rep., Software Engineering Research Center, Purdue University, 1989.

[29] L. Madeyski and N. Radyk, "Judy-a mutation testing tool for Java," *IET Software*, vol. 4, no. 1, pp. 32–42, 2010.

[30] 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.

[31] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Informatica*, vol. 18, no. 1, pp. 31–45, 1982.

[32] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 919–930, ACM, 2014.

[33] 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, Workshop*, pp. 200–204, 2009.

[34] R. Laddad, *AspectJ in action*. Manning Publications Co, 2003.

[35] G. Kiczales1, E. Hilsdale2, J. Hugunin2, M. Kersten2, J. Palm2, and W. G. Griswold3, "An overview of AspectJ," in *European Conference on Object-Oriented Programming*, pp. 327–354, 2001.

[36] M. Harman, P. McMinn, J. T. de Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical Software Engineering and Verification*, pp. 1–59, 2012.

[37] J. Clarke, J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd, "Reformulating software engineering as a search problem," *IEE Proceedings - Software*, vol. 150, no. 3, pp. 161–175, 2003.

[38] W. Miller and D. L. Spooner, "Automatic generation of floating-point test data," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 223–226, 1976.

[39] L. Davis, *Handbook of genetic algorithms*. Van Nostrand Reinhold, 1991.

[40] K. A. De Jong, *Analysis of the behavior of a class of genetic adaptive systems*. PhD thesis, University of Michigan, Ann Arbor, 1975.

[41] J. H. Holland, *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[42] M. Mitchell, "An introduction to genetic algorithms," 1998.

[43] M. Musnjak and M. Golub, "Using a set of elite individuals in a genetic algorithm," in *International Conference on Information Technology Interfaces*, pp. 531–535, 2004.

[44] L. M. Schmitt, "Theory of genetic algorithms," *Theoretical Computer Science*, vol. 259, no. 1, pp. 1–61, 2001.

[45] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller, "A greedy algorithm for aligning dna sequences," *Journal of Computational biology*, vol. 7, no. 1-2, pp. 203–214, 2000.

[46] V. Laarhoven, P. JM, and E. H. Aarts, *Simulated annealing*. Springer, 1987.

[47] E. Aarts and J. Korst, *Simulated annealing and Boltzmann machines: a stochastic approach to combinatorial optimization and neural computing*. John Wiley & Sons, Inc., 1989.

[48] W. L. Price, "A controlled random search procedure for global optimisation," *The Computer Journal*, vol. 20, no. 4, pp. 367–370, 1977.

[49] Z. B. Zabinsky, "Random search algorithms," *Wiley Encyclopedia of Operations Research and Management Science*, 2009.

[50] R. Hamlet, *Random Testing*. John Wiley & Sons, Inc., 2002.

[51] T. A. Budd, F. Sayward, R. J. Lipton, and R. DeMillo, "The design of a prototype mutation system for program testing," in *National Computer Conference*, pp. 623–127, 1978.

[52] K. N. King and J. Offutt, "A Fortran language system for mutation-based software testing," *Software - Practice and Experience*, vol. 21, pp. 685–718, 1991.

[53] B. Choi, R. A. DeMillo, E. W. Krauser, R. Martin, A. Mathur, J. Offutt, H. Pan, and E. H. Spafford, "The mothra tool set (software testing)," in *Hawaii International Conference on System Sciences*, vol. 2, pp. 275–284, 1989.

[54] M. E. Delamaro and J. C. Maldonado, "Proteum-a tool for the assessment of test adequacy for C programs," tech. rep., Software Engineering Research Center, Purdue University, 1996.

[55] I. Moore, "Jester-a Junit test tester," *eXtreme Programming and Flexible Processes in Software Engineering*, pp. 84–87, 2000.

[56] S.-W. Kim, J. A. Clark, and J. A. McDermid, "The rigorous generation of java mutation operators using hazop," in *International Conference Software and Systems Engineering and their Applications*, 1999.

[57] S.-W. Kim, J. A. Clark, and J. A. McDermid, "Investigating the effectiveness of object-oriented testing strategies using the mutation method," in *Workshop on Mutation Analysis*, 2001.

[58] S.-W. Kim, J. A. Clark, and J. A. McDermid, "Assessing test set adequacy for object-oriented programs using class mutation," in *Symposium on Software Technology*, pp. 72–83, 1999.

[59] Y.-S. Ma, Y.-R. Kwon, and J. Offutt, "Inter class mutation operators for Java," in *International Symposium on Software Reliability Engineering*, no. 12, pp. 352–366, 2002.

[60] D. Firesmith, "Testing object-oriented software," in *Technology of Object-Oriented Languages and Systems*, pp. 407–426, 1993.

[61] R. V. Binder, "Testing object-oriented software: a survey," *Software Testing, Verification and Reliability*, vol. 6, no. 3-4, pp. 125–252, 1996.

[62] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson, "A fault model for subtype inheritance and polymorphism," in *International Symposium on Software Reliability Engineering*, pp. 84–93, 2001.

[63] S.-W. Kim, J. A. Clark, and J. A. McDermid, "Class mutation: Mutation testing for object-oriented programs," in *Object-Oriented Software Systems*, pp. 9–12, 2000.

[64] P. Chevalley, "Applying mutation analysis for object-oriented programs using a reflective approach," in *Asia-Pacific Software Engineering Conference*, pp. 267–270, 2001.

[65] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: a mutation system for Java," in *International conference on Software engineering*, pp. 827–830, 2006.

[66] Y. S. Ma, M. J. Harrold, and Y. R. Kwon, "Evaluation of mutation testing for object-oriented programs," in *International conference on Software engineering*, pp. 869–872, 2006.

[67] J. Offutt, Y.-S. Ma, and Y.-R. Kwon, "The class-level mutants of MuJava," in *International workshop on Automation of software test*, pp. 78–84, 2006.

[68] P. Chevalley and P. Thévenod-Fosse, "A mutation analysis tool for Java programs," *International journal on software tools for technology transfer*, vol. 5, no. 1, pp. 90–103, 2003.

[69] S.-W. Kim, M. J. Harrold, and Y.-R. Kwon, "Mugamma: Mutation analysis of deployed software to increase confidence and assist evolution," in *Workshop on Mutation Analysis*, pp. 10–10, 2006.

[70] S. A. Irvine, T. Pavlinic, L. Trigg, J. G. Cleary, S. Inglis, and M. Utting, "Jumble Java byte code to measure the effectiveness of unit tests," in *Testing: Academic and Industrial Conference Practice and Research Techniques*, pp. 169–175, 2007.

[71] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *International symposium on Software testing and analysis*, pp. 69–80, 2009.

[72] "PIT, mutation testing system." http://pitest.org/, 2011.

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

[74] R. T. Alexander, J. M. Bieman, and A. A. Andrews, "Towards the Systematic Testing of Aspect-Oriented Programs," tech. rep., Department of Computer Science, Colorado State University, 2004.

[75] M. Ceccato and P. T. F. Ricca, "Is AOP code easier or harder to test than OOP code?," in *International Conference on Aspect-Oriented Software Development, Workshop*, pp. 123–127, 2005.

[76] A. V. Deursen, M. Marin, and L. Moonen, "A Systematic Aspect-Oriented Refactoring and Testing Strategy, and its Application to JHotDraw," tech. rep., Delft University of Technology, 2005.

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

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

[79] A. Acree, *On Mutation*. PhD thesis, Georgia Institute of Technology, Atlanta, Georgia, 1980.

[80] S. Hussain, *Mutation clustering*. PhD thesis, Kings College London, London, 2008.

[81] J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *International conference on Software Engineering*, pp. 100–107, 1993.

[82] A. P. Mathur, "Performance, effectiveness, and reliability issues in software testing," in *International Computer Software and Applications Conference*, pp. 604–605, 1991.

[83] R. A. Demillo, E. W. Krauser, and A. P. Mathur, "Compiler-integrated program mutation," in *International Computer Software and Applications Conference*, pp. 351 – 356, 1991.

[84] R. H. Untch, "Mutation-based software testing using program schemata," in *ACM Southeast Regional Conference*, pp. 285–291, 1992.

[85] R. H. Untch, J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," *SIGSOFT Software Engineering Notes*, vol. 18, no. 3, pp. 139–148, 1993.

[86] J. Offutt, Y.-S. Ma, and Y.-R. Kwon, "An experimental mutation system for Java," *SIGSOFT Software Engineering Notes*, vol. 29, pp. 1–4, 2004.

[87] Y. Jia and M. Harman, "Milu: A customizable, runtime-optimized higher order mutation testing tool for the full C language," in *Testing: Academic and Industrial Conference - Practice and Research Techniques*, pp. 94–98, 2008.

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

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

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

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

[92] O. Räihä, K. Koskimies, E. Mäkinen, and T. Systa, "Pattern-based genetic model refinements in MDA," *Nordic Journal of Computing*, vol. 14, no. 4, pp. 338–355, 2008.

[93] M. Pereira and S. R. Vergilio, "Gptest: A testing tool based on genetic programming," in *Genetic and Evolutionary Computation Conference*, pp. 1343–1350, 2002.

[94] L. Shan and H. Zhu, "Testing software modelling tools using data mutation," in *International Conference on Software Engineering*, pp. 43–49, 2006.

[95] M. M. Masud, A. Nayak, M. Zaman, and N. Bansal, "Strategy for mutation testing using genetic algorithms," in *Canadian Conference on Electrical and Computer Engineering*, pp. 1049–1052, 2005.

[96] K. Adamopoulos and R. M. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Genetic and Evolutionary Computation Conference*, pp. 1338–1349, 2004.

[97] E. Davies, J. McMaster, and M. Stark, "The use of genetic algorithms for flight test and evaluation of artificial intelligence and complex software systems," in *The role of intelligent systems in defence*, pp. 36–47, 1995.

[98] N. Tracey, J. A. Clark, and K. Mander, "Automated program flaw finding using simulated annealing," *ACM Sigsoft Software Engineering Notes*, vol. 23, pp. 73–81, 1998.

[99] L. C. Briand, Y. Labiche, and M. Shousha, "Stress testing real-time systems with genetic algorithms," in *Genetic and Evolutionary Computation Conference*, pp. 1021–1028, 2005.

[100] R. Ferguson and B. Korel, "The chaining approach for software test data generation," *ACM Transactions on Software Engineering and Methodology*, vol. 5, pp. 63–86, 1996.

[101] K. Lakhotia, M. Harman, and P. Mcminn, "A multi-objective approach to search-based test data generation," in *Genetic and Evolutionary Computation Conference*, pp. 1098–1105, 2007.

[102] P. Tonella, "Evolutionary testing of classes," *ACM Sigsoft Software Engineering Notes*, vol. 29, pp. 119–128, 2004.

[103] M. Harman, F. Islam, T. Xie, and S. Wappler, "Automated test data generation for aspect-oriented programs," in *Aspect-Oriented Software Development*, pp. 185–196, 2009.

[104] T. Xie and J. Zhao, "A framework and tool supports for generating test inputs of AspectJ programs," in *International conference on Aspect-oriented software development*, pp. 190–201, 2006.

[105] T. Xie, J. Zhao, D. Marinov, and D. Notkin, "Automated test generation for AspectJ programs," in *International conference on aspect-oriented software development, workshop on Testing Aspect-Oriented Programs*, pp. 102–111, 2005.

[106] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[107] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *International Symposium on Software Testing and Analysis*, pp. 102–112, 2000.

[108] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[109] S. Forrest, T. Nguyen, W. Weimer, and C. L. Goues, "A genetic programming approach to automated software repair," in *Genetic and Evolutionary Computation Conference*, pp. 947–954, 2009.

[110] C. L. Goues, W. Weimer, and S. Forrest, "Representations and operators for improving evolutionary software repair," in *Genetic and evolutionary computation conference*, pp. 959–966, 2012.

[111] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, pp. 54–72, 2012.

[112] SourceForge, "The Java Decompiler project." `http://dcompiler.sourceforge.net/`, 2002.

[113] A. Oram, *Managing Projects with make*. O'Reilly, 1991.

[114] "Software-artifact Infrastructure Repository." http://sir.unl.edu/portal/index.php.

[115] S. Mouchawrab, L. C. Briand, Y. Labiche, and M. Di Penta, "Assessing, comparing, and combining state machine-based testing and structural testing: a series of experiments," *Software Engineering, IEEE Transactions on*, vol. 37, no. 2, pp. 161–187, 2011.

[116] J. Walnes, "XStream for object serializations." http://xstream.codehaus.org/. 2011.

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

[118] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[119] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. Boston, MA, USA: PWS Publishing Co., 2nd ed., 1998.

Appendices

Appendix 1: Coordinate Program

```
1  public class Coordinate
2  {
3      private int x;
4
5      private int y;
6
7      protected World world;
8
9      public Coordinate( World w, int x, int y )
10     {
11         if (helper( x ) && helper( y )) {
12             world = w;
13             this.x = x % world.board.length;
14             this.y = y % world.board[0].length;
15         }
16     }
17
18     public java.lang.Object get()
19     {
20         return world.get( x, y );
21     }
22
23     public void put( java.lang.Object o )
24     {
25         if (o == null) {
26
27         } else {
28             world.put( x, y, o );
29         }
30     }
31
32     public boolean equals( java.lang.Object o )
33     {
34         if (o instanceof Coordinate) {
35             return ((Coordinate) o).world == world &&
                   ((Coordinate) o).x == x && ((Coordinate) o).y
                   == y;
36         } else {
37             return false;
38         }
39     }
40
```

```
41      public int hashCode()
42      {
43          return x + y + world.hashCode();
44      }
45
46      public Coordinate north()
47      {
48          return new Coordinate( world, x, y + 1 );
49      }
50
51      public Coordinate south()
52      {
53          if (y == 0) {
54              return new Coordinate( world, x,
                    world.board[0].length - 1 );
55          } else {
56              return new Coordinate( world, x, (y - 1) %
                    world.board[0].length );
57          }
58      }
59
60      public Coordinate east()
61      {
62          return new Coordinate( world, x + 1, y );
63      }
64
65      public Coordinate west()
66      {
67          if (x == 0) {
68              return new Coordinate( world, world.board.length
                    - 1, y );
69          }
70          return new Coordinate( world, (x - 1) %
                world.board.length, y );
71      }
72
73      private boolean helper( int d )
74      {
75          boolean result = true;
76          if (d < 0) {
77              result = false;
78
79          }
80          return result;
81      }
```

```
82
83      public java.lang.String toString()
84      {
85          return "Coordinate(" + x + "," + y + ") in " + world;
86      }
87
88  }
89
90  public class World
91  {
92
93      protected java.lang.Object[][] board;
94
95      public World( int n, int m )
96      {
97          if (n < 1 || m < 1) {
98
99          } else {
100             board = new java.lang.Object[n][m];
101         }
102     }
103
104     public World( int n )
105     {
106         this( n, n );
107     }
108
109     protected java.lang.Object get( int x, int y )
110     {
111         return board[x][y];
112     }
113
114     protected void put( int x, int y, java.lang.Object o )
115     {
116         board[x][y] = o;
117     }
118
119     protected java.lang.Object get( Coordinate c )
120     {
121         if (this == c.world) {
122             return c.get();
123         } else {
124
125             return null;
126         }
```

```
127     }
128
129     protected void put( Coordinate c, java.lang.Object o )
130     {
131         if (this == c.world) {
132             c.put( o );
133         } else {
134
135         }
136     }
137
138     public java.lang.String toString()
139     {
140         return "World(" + board.length + "," +
                board[0].length + ")";
141     }
142 }
```

Appendix 2: Roman Program

```
1  public class Roman
2  {
3      static java.lang.String roman;
4
5      static int decimal;
6
7      static char romanChar;
8
9      public static void main( java.lang.String[] args )
10     {
11         roman = "IIX";
12         roman = roman.toUpperCase();
13         decimal = convertToDecimal( roman );
14         if (decimal != -1) {
15             System.out.println( "The Roman number  " + roman
                   + "  is equal to the Decimal number  " +
                   decimal );
16         } else {
17             System.out.println( "The roman number " + roman
                   + " is not valid " );
18         }
19     }
20
21     public static int convertToDecimal( java.lang.String
           roman )
22     {
23         roman = roman.toUpperCase();
24         decimal = 0;
25         if (isValid( roman )) {
26             int x = 0;
27             while (x < roman.length()) {
28                 romanChar = roman.charAt( x );
29                 switch (romanChar) {
30                 case 'M' :
31                     decimal += 1000;
32                     break;
33
34                 case 'D' :
35                     decimal += 500;
36                     break;
37
38                 case 'C' :
```

```
39                      decimal += 100;
40                      break;
41
42                  case 'L' :
43                      decimal += 50;
44                      break;
45
46                  case 'X' :
47                      decimal += 10;
48                      break;
49
50                  case 'V' :
51                      decimal += 5;
52                      break;
53
54                  case 'I' :
55                      decimal += 1;
56                      break;
57
58                  }
59                  x++;
60              }
61              return decimal;
62          } else {
63              return -1;
64          }
65      }
66
67      public static int pos( char chr )
68      {
69          if (chr == 'M') {
70              return 7;
71          } else {
72              if (chr == 'D') {
73                  return 6;
74              } else {
75                  if (chr == 'C') {
76                      return 5;
77                  } else {
78                      if (chr == 'L') {
79                          return 4;
80                      } else {
81                          if (chr == 'X') {
82                              return 3;
83                          } else {
```

```
84                              if (chr == 'V') {
85                                  return 2;
86                              } else {
87                                  if (chr == 'I') {
88                                      return 1;
89                                  } else {
90                                      return 10;
91                                  }
92                              }
93                          }
94                      }
95                  }
96              }
97          }
98      }
99
100     public static boolean isValid( java.lang.String roman )
101     {
102         roman = roman.toUpperCase();
103         int x = 0;
104         int y = 0;
105         try {
106             if (roman == null) {
107             }
108         } catch ( java.lang.Exception e ) {
109             return false;
110         }
111         if (roman.length() <= 0) {
112             return false;
113         }
114         while (x < roman.length()) {
115             y = x;
116             while (y <= roman.length() - 1) {
117                 if (pos( roman.charAt( x ) ) == 10 || pos(
                        roman.charAt( x ) ) < pos( roman.charAt(
                        y ) )) {
118                     return false;
119                 }
120                 y++;
121             }
122             x++;
123         }
124         return true;
125     }
126
```

```
127     public static java.lang.String convertToRoman( int
            number )
128     {
129         java.lang.String roman = "";
130         if (number <= 0 || number > 3999) {
131             return "error";
132         }
133         while (number >= 1000) {
134             roman += "M";
135             number -= 1000;
136         }
137         while (number >= 900) {
138             roman += "CM";
139             number -= 900;
140         }
141         while (number >= 500) {
142             roman += "D";
143             number -= 500;
144         }
145         while (number >= 400) {
146             roman += "CD";
147             number -= 400;
148         }
149         while (number >= 100) {
150             roman += "C";
151             number -= 100;
152         }
153         while (number >= 90) {
154             roman += "XC";
155             number -= 90;
156         }
157         while (number >= 50) {
158             roman += "L";
159             number -= 50;
160         }
161         while (number >= 40) {
162             roman += "XL";
163             number -= 40;
164         }
165         while (number >= 10) {
166             roman += "X";
167             number -= 10;
168         }
169         while (number >= 9) {
170             roman += "IX";
```

```
171             number -= 9;
172         }
173         while (number >= 5) {
174             roman += "V";
175             number -= 5;
176         }
177         while (number >= 4) {
178             roman += "IV";
179             number -= 4;
180         }
181         while (number >= 1) {
182             roman += "I";
183             number -= 1;
184         }
185         return roman;
186     }
187 
188 }
```

Appendix 3: Movie Rental Program

```
1  public class Movie
2  {
3      public static java.lang.String movieTitle;
4
5      public static char movieType;
6
7      public Movie( java.lang.String title, char movieType2 )
8      {
9          movieTitle = title;
10         movieType = movieType2;
11     }
12
13     public static double getPrice()
14     {
15         switch (movieType) {
16         case 'R' :
17             return 1.5;
18
19         case 'C' :
20             return 1;
21
22         case 'N' :
23             return 2.5;
24
25         default  :
26             return 10;
27
28         }
29     }
30
31     public static java.lang.String getTitle()
32     {
33         return movieTitle;
34     }
35
36     public static char getType()
37     {
38         return movieType;
39     }
40
41     public static void setTitle( java.lang.String tit )
42     {
```

```
43          movieTitle = tit;
44      }
45
46      public static void setType( char type )
47      {
48          movieType = type;
49      }
50
51  }
52
53  public class Customer
54  {
55
56      public static java.lang.String customerName;
57
58      public static double customerAccount;
59
60      public static double customerPayments;
61
62      public static char customerType;
63
64      public Customer( java.lang.String name, double account,
           char type )
65      {
66          customerName = name;
67          customerAccount = account;
68          customerType = type;
69          customerPayments = 0;
70      }
71
72      public static java.lang.String getCustomerName()
73      {
74          return customerName;
75      }
76
77      public static void setCustomerName( java.lang.String
           name )
78      {
79          customerName = name;
80      }
81
82      public static double getCustomerAccount()
83      {
84          return customerAccount;
85      }
```

```
86
87      public static char getCustomerType()
88      {
89          return customerType;
90      }
91
92      public static void setCustomerType( char type )
93      {
94          customerType = type;
95      }
96
97      public static void addToCustomerAccount( double charges )
98      {
99          customerAccount += charges;
100     }
101
102     public static void addCustomerPayment( double payment )
103     {
104         customerAccount -= payment;
105         customerPayments += payment;
106     }
107
108     public static double getCustomerPaymentsTotal()
109     {
110         return customerPayments;
111     }
112
113 }
114
115 public class Rental
116 {
117
118     public static Movie movie;
119
120     public static Customer customer;
121
122     public static java.util.Date startDate;
123
124     public static java.util.Date returnDate;
125
126     public Rental( java.lang.String movieName, char
            movieType, java.lang.String customerName, double
            customerBalance, char customerType, java.util.Date
            mStartDate, java.util.Date mReturnDate )
127     {
```

```
128         movie = new Movie( movieName, movieType );
129         customer = new Customer( customerName,
               customerBalance, customerType );
130         startDate = mStartDate;
131         returnDate = mReturnDate;
132     }
133
134     public static double getDaysRented()
135     {
136         return (returnDate.getTime() - startDate.getTime())
               / (1000 * 60 * 60 * 24);
137     }
138
139     public static double getCharges()
140     {
141         return movie.getPrice() * getDaysRented();
142     }
143
144     public static void setReturnDate( java.util.Date
           mNewReturnDate )
145     {
146         customer.addCustomerPayment( Rental.getCharges() );
147         returnDate = mNewReturnDate;
148         customer.addToCustomerAccount( Rental.getCharges() );
149     }
150
151 }
152
153 public aspect Updates {
154         pointcut newMovie(): execution( Movie.new(
               String,char)) ;
155         pointcut updateMovie(): execution( * Movie.setType(
               char));
156         after() returning:  newMovie()|| updateMovie() {
157                 Movie.movieType = Character.toUpperCase(
                       Movie.movieType);
158                 if ( ( Movie.movieType! = 'R') && (
                       Movie.movieType! = 'C') && (
                       Movie.movieType! = 'N') && (
                       Movie.movieType! = 'A') && (
                       Movie.movieType! = 'B') && (
                       Movie.movieType! = 'D') && (
                       Movie.movieType! = 'E'))   {
159                         Movie.movieType  = '0';
160                 }
```

```
161         }
162         pointcut newCustomer(): execution( Customer.new(
               ..)) || execution( * Customer.setCustomerType(
               ..));
163         after() returning:  newCustomer() {
164                 Customer.customerType =
                       Character.toUpperCase(
                       Customer.customerType);
165                 if ( ( Customer.customerType! = 'A') && (
                       Customer.customerType! = 'B') )       {
166                       Customer.customerType  = 'C';
167                 }
168         }
169         pointcut newRental(): execution( Rental.new( ..));
170         after() returning: newRental(){
171                 if ( Rental.startDate.after(
                       Rental.returnDate) ){
172                       Date temDate = Rental.startDate;
173                       Rental.startDate = Rental.returnDate;
174                       Rental.returnDate = temDate;
175                 }
176                 else {
177                 }
178                 Rental.customer.addToCustomerAccount(
                       Rental.getCharges());
179         }
180         after() returning:  execution( *
               Rental.setReturnDate( ..)){
181                 if ( Rental.startDate.after(
                       Rental.returnDate) ){
182                       Date temDate = Rental.startDate;
183                       Rental.startDate = Rental.returnDate;
184                       Rental.returnDate = temDate;
185                 }
186                 else {
187                 }
188         }
189         pointcut moviePriceUpdate(): call ( double
               Movie.getPrice());
190         double around(): moviePriceUpdate(){
191                 if ( Movie.movieType =  = 'A'){return 4;}
192                 else if ( Movie.movieType =  = 'B'){return
                       5;}
193                 else  if ( Movie.movieType =  = 'D'){return
                       4.5;}
```

```
194                 else if ( Movie.movieType =  = 'E'){ return
                       5.2;}
195                 else    return proceed();
196         }
197         void around( double charges): call( *
               Customer.addToCustomerAccount( double)) && args (
               charges) {
198                 if ( Customer.customerType =  = 'A'){
199                         proceed( charges-( charges*1/4));
200                 } else
201                         if ( Customer.customerType =  = 'B'){
202                                 proceed( charges-(
                                       charges*1/3));
203                         }else{
204                                 proceed( charges);
205                         }
206         }
207         void around( double payment): call( *
               Customer.addCustomerPayment( ..)) && args (
               payment) {
208                 if ( Customer.customerType =  = 'A'){
209                         proceed( payment*1.2);
210                 } else
211                         if ( Customer.customerType =  = 'B'){
212                                 proceed( payment*1.3);
213                         }else{
214                                 proceed( payment);
215                         }
216         }
217         double around (): call( * Rental.getDaysRented( ..))
               {
218                 double daysRented =  proceed();
219                 if ( ( daysRented>3) && ( daysRented<10)) {
220                         daysRented = 3;
221                 }  if ( ( daysRented> = 10) && (
                       daysRented<100)) {
222                         daysRented = 10;
223                 }  if ( ( daysRented> = 100) && (
                       daysRented<365)) {
224                         daysRented = 100;
225                 }
226                 return ( daysRented);
227         }
228 }//aspect
```