HOMAJ: A Tool for Higher Order Mutation Testing in AspectJ and Java

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

***Abstract*—The availability of automated tool support is an important consideration for software developers before they can incorporate higher order mutation testing into their software development processes. This paper presents HOMAJ, a higher order mutation testing tool for AspectJ and Java. HOMAJ automates the process of generating and evaluating first order mutants (FOMs) and higher order mutants (HOMs). In particular, HOMAJ can be used to generate subtle HOMs, which are HOMs that cannot be killed by an existing test set that kills all the FOMs. Subtle HOMs can be valuable for improving test effectiveness because they can simulate complex and non-trivial faults that cannot be simulated with the use of traditional FOMs.**

HOMAJ implements a number of different techniques for generating subtle HOMs, including several search-based software engineering techniques, enumeration search, and random search. HOMAJ is designed in a modular way to make it easy to incorporate a new search strategy. In this paper we demonstrate the use of HOMAJ to evaluate the implemented techniques.

Keywords*—*Higher order mutation testing; search-based software engineering; software testing; tools;

I. Introduction

During the last few decades, a number of mutation testing tools have been developed by researchers. Most of them address traditional first order mutation, where each mutant contains one syntactic change. Several such tools exist for Java and AspectJ [1]–[9]. However, there are few tools that can generate higher order mutants, which *combine* two or more first order mutants. MILU [10] is a higher order mutation tool for C programs. To promote the use of higher order mutation testing among researchers and practitioners, automated tool support needs to be provided for a variety of languages.

The importance of higher order mutation stems from the fact that a large majority of traditional First Order Mutants (FOMs) represent trivial faults that are often easily detected [11]–[14]. Therefore, researchers have proposed the use of Higher Order Mutants (HOMs) to simulate non-trivial and complex faults that can be used to further improve the fault-detection effectiveness of test suites [11], [13], [15], [16].

Because HOMs are created by making multiple syntactic changes or by combining multiple FOMs, the number of candidate HOMs is exponentially large. Moreover, a large majority of HOMs are killed by any test suite that kills all the FOMs of the program under test; this is known as the *coupling effect* [15]–[18]. However, research has shown some HOMs can be valuable for increasing test effectiveness and reducing test cost.

Omar et al. [15] introduce the notion of subtle HOMs, which are those that are not killed by existing test suites that kill all the FOMs for the program under test. Subtle HOMs denote non-trivial and complex faults that cannot be simulated with FOMs. Jia and Harman [11] introduced the notion of strongly subsuming HOMs, which are those that are harder to kill than their constituent FOMs and can replace their constituent FOMs without loss of test effectiveness. Strongly subsuming HOMs can reduce the number of FOMs to be compiled and executed and thus, reduce the test cost. Other researchers used HOMs to detect equivalent FOMs and reduce their density [16], [19], [20].

In this paper, we describe and evaluate HOMAJ, a higher order mutation testing tool for AspectJ and Java programs, which automates the processes of (1) generating both FOMs and HOMs, and (2) compiling and executing them against the given test suite. HOMAJ can generate subtle mutants by exploring the search space of all HOMs using a variety of techniques, such as search-based software engineering, enumeration search, and random search. Software testers can improve their existing test sets by creating new test cases that kill the subtle HOMs. HOMAJ can also 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 [11] and their construction approaches [15].

The rest of the paper is organized as follows. Section II presents background information on higher order mutation testing. Section III describes the key features of HOMAJ. Section IV presents the results obtained by evaluating HOMAJ. Section V summarizes the related work in the area of mutation testing tools. Section VI concludes the paper and outlines directions for future work.

II. Background

In this section we summarize techniques that classify HOMs based on the test sets that kill the HOMs and FOMs, as well as the construction approach used to create HOMs.

Jia and Harman [11] classified HOMs in terms of their coupling relationships with FOMs. An HOM is considered to be coupled to its constituent FOMs "if a test set that kills the FOMs also contains test cases that kill the HOM" [11]. Otherwise the HOM is decoupled.

Decoupled HOMs are valuable to the testing process because they represent different faults than their constituent FOMs and can be used to improve fault detection effectiveness.

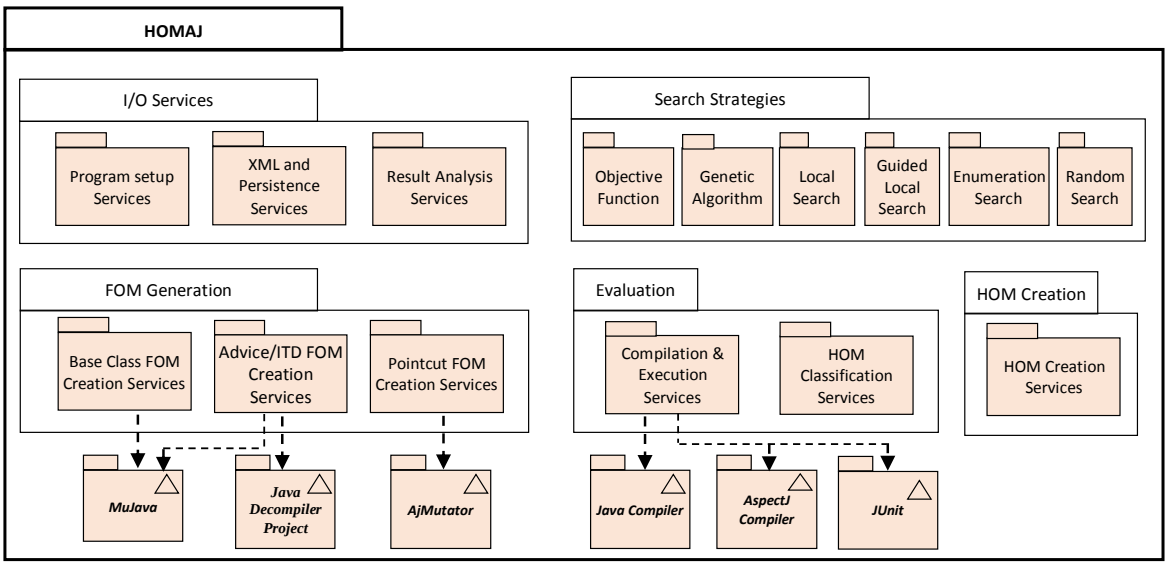

Fig. 1. Architecture of HOMAJ

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" [11]. 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.

Omar et al. [15] proposed four approaches to construct HOMs for AspectJ programs. The approaches, which are based on AspectJ fault models, specify the locations of the program where mutation faults can be introduced. The four types are as follows: (1) the 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) the HOM is constructed by inserting two or more mutation faults in two or more different base classes, (3) the HOM is constructed by inserting two or more mutation faults in two or more different aspects, and (4) the HOM is constructed by inserting at least one fault in a base class and at least one fault in an aspect.

III. Design and Implementation of HOMAJ

Figure 1 shows the architecture of HOMAJ. It consist of five components. It also uses some existing third-party components, which are marked with a triangle. Below we describe the components and their key functionality.

A. I/O Services

This component is responsible for receiving inputs from testers and presenting outputs. HOMAJ takes as inputs the program under test along with the test suite. HOMAJ also takes as inputs 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 original source code, the test suites, the generated FOMs and subtle HOMs, and a set of output files that contain the execution and classification results of HOMs.

HOMAJ uses XML records to store information about generated FOMs and 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 different types of reports.

B. FOM Generation

HOMAJ utilizes all the operators of two available traditional mutation testing tools, MuJava [2] and AjMutator [8], to generate FOMs. MuJava generates both traditional 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. [21]. It can compile the mutants and execute test cases in conjunction with an AspectJ compiler and a Java runtime environment.

To create Java base class FOMs, HOMAJ calls the public methods in the `MuJava.MutationSystem` class. It passes 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 information, such as the applied mutation operator, the line number of the mutated statement, the mutated method or operator, and the actual mutation (i.e., the original code fragment and the mutated code fragment). HOMAJ extracts the information and produces a

file that contains XML records, such that each XML record provides complete information about a generated FOM. The XML records are henceforth called mutant metadata records. Figure 2 shows a file containing three FOM metadata records.

```
<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>
```

Fig. 2. FOM Metadata Example

To create pointcut FOMs of 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 the tool 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 actual mutation is obtained from both the mutated file and the original file. After extracting this information, HOMAJ produces metadata records for the pointcut FOMs as shown in Figure 2.

To mutate aspect advice and inter-type declarations, we used a technique previously used by Wedyan and Ghosh [22]. The technique requires the use of a Java decompiler. We used *The Java Decompiler Project* [23] to decompile AspectJ files into Java files on which MuJava can be used for mutation. 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.

C. 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). Because FOMs are represented at the statement level, HOMAJ currently can only insert one mutation fault per code statement.

D. Mutant Evaluation

The "Evaluation" component is responsible for the compilation and execution of both FOMs and HOMs as well as the classification of HOMs. HOMAJ uses Java and AspectJ compilers and JUnit to compile and execute mutants. Because HOMs can contain faults that may be dispersed among different class and aspect files, HOMAJ currently maintains a complete version of the program for each HOM with all base class and aspect files for each HOM. The process of compiling and executing the mutants involves creating HOMs, one at a time, compiling and executing the HOM against the given test suite, and then adding the execution results to the corresponding metadata record of the HOM. The execution results include a list of identifiers of all the test cases that kill the HOM.

A particular HOM metadata record might be generated more than once during the search process. This would require compiling and executing the same HOM against the test suite every time it is created. To avoid the cost of compiling and executing the same HOM more than once, the tool checks for duplicate metadata records and creates each HOM only once.

To classify HOMs, HOMAJ uses results from executing both HOMs and FOMs to classify the HOMs based on the definitions of subsumption and coupling relationships proposed by Jia and Harman [11]. To determine the category of each HOM, the following steps are performed:

1) Retrieve the constituent FOM information of the HOM from the metadata files.
2) Retrieve the test set that kills each of the constituent FOMs.
3) Compare the test set that kills the constituent FOMs with the test set that kills the HOM.
4) Assign the corresponding subsuming and/or coupled classification based on the results of comparison.
5) Add the assigned classification to the metadata record of the HOM.

HOMAJ also classifies HOMs based on their construction approach proposed by Omar et al. [15]. HOMAJ can be configured to create HOMs by using a particular construction approach and evaluate those HOMs.

E. Search Strategies

The component "Search Strategies" includes the implementations of each search strategy. The design is such that a new strategy can be easily added to HOMAJ. Currently, HOMAJ implements five search techniques: Genetic Algorithm, Local Search, Guided Local Search, Random Search, and Enumeration Search, for finding subtle HOMs. Each search technique has a set of configurable parameters and require the list of FOM metadata records for the program under test in order to start the search process. Each search technique maintains a list of distinct, subtle HOM metadata records that were found during the search process, which are 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 HOMs that the search technique is allowed to explore, the time limit that the search technique is allowed to run for, and the number of subtle HOMs that the search technique is required to find. The search technique stops if any of these conditions is met. Below, we first describe the objective function used by the search techniques to evaluate the fitness of HOMs. Then we briefly describe the search techniques for finding subtle HOMs.

1) HOM Fitness Evaluation: The search techniques use an objective function to identify subtle HOMs. Each subtle HOM represents an optimal solution and we want as many distinct optimal solutions as possible. The objective function classifies HOMs based on their fitness value into four types. *Entirely Coupled HOMs*, which have a fitness value of 0 (worst value), represent the cases 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 the individual constituent FOMs. *Partially Coupled HOMs*, which have a fitness value greater than 0 but less than 1, represent the cases where there is a difference between the set of test cases that kill the HOM and the union of all sets of test cases that kill the individual constituent FOMs. *Decoupled HOMs*, which have a fitness value of 1, are those with no overlap between the set of test cases that kill the HOM and the union of all sets of test cases that kill the individual constituent FOMs. *Subtle HOMs* have a fitness value of 2 and represent new faulty behavior that has not been tested because it was not killed by any test case in the given test suite.

HOMs with higher fitness values are favored in the selection process. Partially coupled and decoupled HOMs have the potential to develop into subtle HOMs because their constituent FOMs can interact to produce different faulty behavior.

2) Genetic Algorithm: The configurable parameters for the Genetic Algorithm (GA) [15] include the number of crossover points, mutation rate, population size, degree of HOMs in the first population, and the number of elite HOMs that get carried over at each iteration. The chromosome is represented as a one dimensional array of strings such that each element in the array represents a line of code (one Java/AspectJ statement) of the program under test.

The Genetic Algorithm evolves a set of HOMs using crossover and then mutation to generate new population. The Genetic Algorithm uses *tournament selection* to implement generational replacement of the population. However, a certain number of HOMs with the highest fitness values in the current population are automatically carried over (copied) to the next generation.

3) Local Search: The maximum degree of the incumbent HOM, which represents the search starting point for the Local Search (LS) [15], can be configured. The incumbent HOM is randomly generated by selecting a random number of FOMs. After evaluating the incumbent HOM, Local Search explores all the HOMs neighboring the incumbent HOM. The neighboring HOMs are those that vary by one FOM (one step) from the incumbent HOM and they are maintained in a list.

After creating and evaluating all HOMs in the neighborhood list, Local Search looks for the best neighboring HOM that has an equal or higher fitness value than the incumbent HOM. If such an HOM exists, it becomes the next incumbent HOM, and the process starts all over again. If no better HOM exists, Local Search restarts by selecting a new incumbent HOM.

4) Guided Local Search: The Guided Local Search (GLS) technique uses the same steps as Local Search. However, Guided Local Search utilizes program structural information to help it focus on the FOM combinations that are more likely to produce subtle HOMs. Guided Local Search uses a heuristic that an HOM with data flows between its mutated statements is more likely to produce different or new faulty behavior (partially coupled, decoupled, or subtle HOMs). The mutated statements represent the program statements where the mutation faults are being inserted.

When an incumbent HOM is selected and evaluated, Guided Local Search explores a smaller set of neighboring HOMs than Local Search. Guided Local Search 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.

5) Random Search and Restricted Random Search: Random Search (RS) [15] explores the space of all candidate HOMs by randomly selecting HOMs, one at a time. Restricted Random Search (RRS) performs the same steps as the Random Search. However, we have found most of the discovered subtle HOMs were of lower order (less than six). Restricted Random Search uses a configurable parameter in the Randon Search algorithm (set to six) to allow it to control the maximum degree of generated HOMs.

6) Enumeration Search: The Enumeration Search (ES) technique explores candidate HOMs in the search space in a predefined sequence. It creates and evaluates HOMs one at a time, starting with all candidate second order mutants, then third order mutants and so on until one of the stopping conditions is met.

IV. Evaluation

We conducted a study to demonstrate and evaluate the ability of HOMAJ to produce FOMs and subtle HOMs. We used two Java programs, Roman and Coordinate (from course assignments), and two AspectJ programs, Banking [24] and Movie Rental [16]. The programs are of different sizes and implement various Java and AspectJ constructs. All mutation operators were selected while generating FOMs. However, some operators did not produce any FOMs because the subject programs did not include the constructs to which these operators can be applied. For each subject program we used a test suite that achieved statement coverage and killed all non-equivalent FOMs. Table I provides information about the subject programs.

We first used HOMAJ to generate FOMs for the four subject programs. Equivalent FOMs were manually identified and removed. Table II shows the total number of non-equivalent FOMs that were generated for each program.

Second, we evaluated the ability of HOMAJ to find subtle HOMs. For each subject program we ran each of the four techniques 10 times with the stopping condition of exploring 10,000 HOMs. Enumeration Search was run only once per

TABLE I. SUBJECT PROGRAM

Programs	Type	LOC	Classes	Methods	Aspects	Advice	Inter-type methods	Test cases
Banking	AspectJ	243	2	11	2	1	1	8
Movie Rental	AspectJ	191	3	16	1	8	0	16
Coordinate	Java	242	2	15	NA	NA	NA	13
Roman	Java	208	2	4	NA	NA	NA	11

TABLE II. NON-EQUIVALENT FOMS GENERATED FOR EACH SUBJECT PROGRAM

Programs	# Base Class FOMs	# Aspect FOMs
Banking	34	58
Movie Rental	84	232
Coordinate	121	0
Roman	191	0

subject program because it produces the same result each time. Table III shows the maximum, average, median, standard deviation, and minimum number of subtle HOMs found for the 10 runs per technique.

We used multiple 64-bit Linux machines with Intel Core™ 4x3.3G and 8Gb memory. The search techniques were configured as follows. For Genetic Algorithm, the number of crossover points was set at two, mutation rate at three, number of elite HOMs at 15, population size at 300, and first population degree at two. For both Local Search and Guided Local Search, the incumbent HOM degree was set at two. We used the restricted form of Random Search, where the maximum HOM degree allowed was set at six. Enumeration Search has no configurable parameters.

TABLE III. NUMBER OF SUBTLE HOMS FOUND BY SEARCH TECHNIQUES

Programs	Technique	# Subtle HOMs				
		Max	Average	Median	St. Dev	Min
Movie Rental	Genetic	2.0	0.5	0.0	0.8	0.0
	Local	6.0	1.9	1.0	2.3	0.0
	Random	1.0	0.4	0.0	0.5	0.0
	Guided	11.0	2.5	0.5	3.6	0.0
	Enumeration	13.0	13.0	NA	NA	NA
Coordinate	Genetic	6.0	3.7	4.0	1.8	1.0
	Local	49.0	17.7	14.5	15.4	3.0
	Random	7.0	3.5	4.0	1.8	1.0
	Guided	66.0	32.6	32.0	18.6	7.0
	Enumeration	8.0	8.0	NA	NA	NA
Banking	Genetic	12.0	8.9	9.0	2.2	4.0
	Local	20.0	14.5	16.0	4.7	7.0
	Random	13.0	7.4	6.5	2.8	5.0
	Guided	21.0	18.7	19.5	1.8	16.0
	Enumeration	20.0	20.0	NA	NA	NA
Roman	Genetic	6.0	4.9	5.0	1.1	3.0
	Local	11.0	5.6	5.0	2.5	3.0
	Random	6.0	3.0	3.0	1.4	1.0
	Guided	11.0	7.7	7.5	2.0	5.0
	Enumeration	4.0	4.0	NA	NA	NA

For each subject program, all the techniques found subtle HOMs. However, Guided Local Search was more successful in finding the highest number of subtle HOMs than the other search techniques for three out of four programs. Enumeration Search was more successful for the Movie Rental program. This indicates that using heuristics to guide the search process can be a rewarding strategy.

For most programs, Restricted Random Search found almost as many subtle HOMs as the Genetic Algorithm, which uses different sophisticated operators to find subtle HOMs. This is because Restricted Random Search was configured to limit the search to the space of HOMs of lower degrees where more subtle HOMs are expected to be found. This seems to be a good strategy because in general increasing the degree of an HOM by adding more FOMs make it easier to kill. Further, subtle HOMs can be more difficult to find in the space of higher degrees as the number of candidate HOMs grow exponentially (at least to a certain degree). Restricted Random Search might have produced different results if the maximum degree of HOMs was set at a higher number.

Last, we compared the performance of the tool with respect to time. Table IV shows the time taken by each search technique to explore 10,000 HOMs. The second column shows the time taken by each search technique to create, compile and execute 10,000 HOMs. The last column shows the time excluding the time for compilation and execution.

TABLE IV. TIME TAKEN BY SEARCH TECHNIQUES TO EXPLORE 10,000 HOMS FOR THE MOVIE RENTAL PROGRAM

Search Technique	Time Including Compilation and Execution	Time Excluding Compilation and Execution
Genetic	150 minutes	5 seconds
Local	107 minutes	3.5 seconds
Restricted Random	171 minutes	3 seconds
Guided	141 minutes	3.6 seconds
Enumeration	210 minutes	2.9 seconds

Table IV shows that the process of compilation and execution of HOMs constitutes the largest part of the computational cost. Although the current implementation checks for duplicates and does not compile and execute the same HOM twice, the number of HOMs is large and the tool compiles and executes every distinct HOM that gets generated.

We propose to extend the implementation and adopt one of the cost reduction techniques that were proposed for traditional FOMs. Researchers have applied the idea of schemas or meta-programs to encode all the FOMs in one program for several programming languages such as Java (e.g., Judy [7]). However, to the best of our knowledge, such an idea has not been applied to AspectJ pointcuts. One reason is that aspect advice code is woven based on the pointcut expression, and join points can change based on a change in the pointcut expression. A straightforward application of the schema idea will not work in the context of AspectJ because one cannot weave a schema of a pointcut with the Java base code. We propose to use utility programs, such as *Make* [25] and *Apache Ant* [26], to perform selective compilation of Java and AspectJ files.

V. RELATED WORK

Researchers developed a variety of mutation operators and mutation tools for Java and AspectJ. Moore [1] developed the first mutation tool for Java programs, Jester. The tool makes a specific syntactic modifications to the source code and JUnit test cases to help testers identify errors in their testing code.

Chevalley and Fosse [3] developed the first Java mutation tool, JavaMut, with a graphical user interface support.

Ma et al. [2], [27] 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(s), 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.

Kim et al. [5] introduced a technique and a prototype tool for mutation testing called MUGAMMA be used for regression testing.

Irvine et al. [4] developed Jumble, a bytecode level mutation testing tool. Jumble implements heuristics to speed the execution and analysis processes of FOMs.

Schuler et al. [6] developed 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 and uses techniques to reduce the number of generated FOMs and the number of test cases that need to be executed on each FOM.

Madeyski and Radyk [7] developed Judy, a tool that takes advantage of the pointcut and advice mechanisms provided by AspectJ to enhance the performance of the mutation testing process.

Delamare et al. [8] developed AjMutator, an AspectJ mutation tool that implements the pointcut mutation operators proposed by Ferrari [21]. Ferrari et al. [9] developed a tool for mutation testing of AspectJ programs called Proteum/AJ. The tool implements 24 mutation operators and allows the tester to select and manage mutation operators in several ways. Anbalagan et al. [28] developed a framework that help testers identifies equivalent pointcut FOMs.

Jia and Harman [10] developed a mutation testing tool, MILU, for both first order and higher order mutation testing for C programs. The tool implements 77 C mutation operators and provides a flexible scripting language for the customization of operators. Testers can use MILU to find strongly subsuming HOMs using 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.

VI. Conclusions and Future Work

We presented HOMAJ, a higher order mutation testing tool for AspectJ and Java programs. HOMAJ automates the processes of generating FOMs and HOMs, compiling and executing them against test suites, classifying HOMs, and analysing results. HOMAJ implements various techniques for finding subtle HOMs, which can be used to improve fault detection effectiveness of test suites.

We are extending HOMAJ to support the combination of multiple FOMs within a single statement. We are also investigating exact methods for generating subtle HOMs as well as heuristics for improving the performance of guided local search. We are working on improving the performance of HOMAJ by reducing the cost of compilation and execution.

References

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

[2] 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.

[3] 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.

[4] 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*, 2007, pp. 169–175.

[5] 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*, 2006, pp. 10–10.

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

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

[8] 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*, 2009, pp. 200–204.

[9] 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*, 2010, pp. 51–58.

[10] 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*, 2008, pp. 94–98.

[11] ——, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

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

[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] M. Harman, Y. Jia, and W. B. Langdon, "A manifesto for higher order mutation testing," in *ICST Workshop*, 2010, pp. 80–89.

[15] E. Omar, S. Ghosh, and D. Whitley, "Constructing subtle higher order mutants for Java and AspectJ programs," in *ISSRE*, 2013, pp. 340–349.

[16] E. Omar and S. Ghosh, "An exploratory study of higher order mutation testing in aspect-oriented programming," in *ISSRE*, 2012, pp. 1–10.

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

[18] 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.

[19] M. Polo, M. Piattini, and I. García-Rodríguez, "Decreasing the cost of mutation testing with second-order mutants," *Software Testing, Verification and Reliability*, 19(2):111–131, 2009.

[20] M. Kintis, M. Papadakis, and N. Malevris, "Isolating first order equivalent mutants via second order mutation," in *ICST*, 2012, pp. 701–710.

[21] F. C. Ferrari, J. C. Maldonado, and A. Rashid, "Mutation Testing for Aspect-Oriented Programs," in *ICST*, 2008, pp. 52–61.

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

[23] Source Forge, "The Java Decompiler project," http://dcompiler.sourceforge.net/, 2002.

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

[25] A. Oram, *Managing Projects with make*. O'Reilly, 1991.

[26] "Apache ant," http://ant.apache.org/, 2007.

[27] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: a mutation system for Java," in *ICSE*, 2006, pp. 827–830.

[28] P. Anbalagan and T. Xie, "Automated generation of pointcut mutants for testing pointcuts in AspectJ programs," in *ISSRE*, 2008, pp. 239–248.