MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler

René Just and Franz Schweiggert
Department of Applied Information Processing
Ulm University
{rene.just,franz.schweiggert}@uni-ulm.de

Gregory M. Kapfhammer
Department of Computer Science
Allegheny College
gkapfham@allegheny.edu

***Abstract*—Mutation analysis is an effective, yet often time-consuming and difficult-to-use method for the evaluation of testing strategies. In response to these and other challenges, this paper presents MAJOR, a fault seeding and mutation analysis tool that is integrated into the Java Standard Edition compiler as a non-invasive enhancement for use in any Java-based development environment. MAJOR reduces the mutant generation time and enables efficient mutation analysis. It has already been successfully applied to large applications with up to 373,000 lines of code and 406,000 mutants. Moreover, MAJOR's domain specific language for specifying and adapting mutation operators also makes it extensible. Due to its ease-of-use, efficiency, and extensibility, MAJOR is an ideal platform for the study and application of mutation analysis.**

I. Introduction

Originally introduced in [2], [3], mutation analysis is a well-known technique for assessing testing strategies by injecting artificial faults. These seeded faults, or mutants, can be used to evaluate different software testing techniques, such as fault finding methods, input value generation models, and oracle solutions. Since mutation analysis is an expensive technique, in comparison to schemes like code coverage analysis, several approaches have been proposed to decrease its costs (cf. [5]). Furthermore, various tools and frameworks for different programming languages have been developed, which differ with respect to their efficiency, flexibility, available mutation operators, and the degree of automation.

This paper presents MAJOR, a compiler-integrated and non-invasive tool that provides fast fault seeding for arbitrary purposes, thus enabling efficient mutation analysis. The name MAJOR is an acronym reflecting that it is a tool for **m**utation **a**nalysis in a **J**ava c**o**mpile**r**. In contrast to existing tools such as Jumble [4], MuJava [7], or Javalanche [9], MAJOR is integrated into the Java compiler and does not require a specific mutation analysis framework. Hence, it can be used in any Java-based environment. The main contributions of MAJOR can be summarized as follows:

- Integrated into the Java Standard Edition compiler
- Non-invasive and compiler-integrated implementation
- Fast and flexible fault seeding with built-in operators
- Extensibility through a domain specific language (DSL)
- Efficient mutation analysis by means of embedded mutants and mutation coverage information

II. Conditional Mutation

The conditional mutation approach generates mutants by transforming the abstract syntax tree (AST) [6]. It uses conditional expressions and statements to encapsulate all of the mutants and the original version of the program in the same basic block. An example method and its corresponding mutated version are shown in Listings 1 and 2, where the binary expression of the framed assignment is mutated. A concrete mutant can be triggered by setting the mutant identifier `M_NO` to the mutant's number at runtime.

A mutant that is not reached and executed cannot be detected under any circumstance. In order to avoid evaluating these uncovered mutants, conditional mutation supports the collection of additional information about the coverage of the mutations at runtime. Listing 3 shows the extension of the mutated binary expression, which provides the mutation coverage information. For efficiency reasons, the covered mutants are reported as ranges within the `COVERED` method.

```
public int eval(int x){
    int a = 3, b = 1, y;

    y = a * x;

    y += b;
    return y;
}
```

Listing 1. Example with an expression as right hand side of a statement.

```
public int eval(int x){
    int a = 3, b = 1, y;

    y = (M_NO==1)? a + x:
        (M_NO==2)? a / x:
        (M_NO==3)? a % x:
                   a * x; // original
    y += b;
    return y;
}
```

Listing 2. Example of an expression mutated with conditional mutation.

```
public int eval(int x){
    int a = 3, b = 1, y;

    y = (M_NO==1)? a + x:
        (M_NO==2)? a / x:
        (M_NO==3)? a % x:
        (M_NO==0 && COVERED(1,3))?
           a * x : a * x; // original
    y += b;
    return y;
}
```

Listing 3. Gathering coverage information with conditional mutation.

978-1-4577-1639-3/11/$26.00 © 2011 IEEE ASE 2011, Lawrence, KS, USA

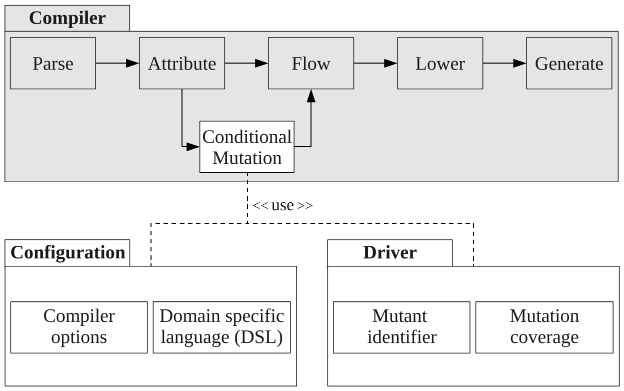

Fig. 1. Compiler-integrated mutation with configuration and driver.

III. Implementation Details

As depicted in Figure 1, a compiler contains several phases, which can be briefly described as follows:

- Parse: Build the AST by parsing the source code
- Attribute: Enhance the AST with semantic information
- Flow: Perform semantic and data flow analyses
- Lower: Simplify the AST and remove syntactic sugar
- Generate: Generate assembled or intermediate code

In MAJOR, conditional mutation is implemented as an optional transformation of the compiler's AST, which can be enabled by means of common compiler options. If the conditional mutation step is not chosen, then the compiler works exactly as if it were unmodified. The compile-time configuration of conditional mutation and the necessary runtime driver are stored externally in order to avoid dependencies and to provide a non-invasive tool. MAJOR's implementation was driven by the following considerations:

- The default behavior of the compiler must not be changed. This criterion is obligatory in order to have one compiler applicable within the build environment.
- The necessary changes within existing compiler classes should be kept to a minimum to ensure that conditional mutation can be implemented in future releases of the Java compiler with little or no additional effort.
- All configuration possibilities have to be externalized to forestall the need to recompile MAJOR for different purposes. Moreover, conditional mutation within the compiler must support the default command-line interface so that it can be used standalone, in integrated development environments (IDEs), and with frequently used build systems such as Apache Ant.
- MAJOR must provide a sufficient set of mutation operators in order to provide meaningful results that are comparable to prior empirical studies [8], [10].

A. *Configuration*

In order to use the mutation capabilities of MAJOR, the conditional mutation step has to be generally enabled at compile-time using the corresponding compiler option. MAJOR also provides either additional compiler options or mutation scripts to support the compile-time configuration of the mutation

```
begin myOP:
    // Define own replacement list for AOR
    BIN(*) -> {/,%};
    BIN(/) -> {*,%};
    BIN(%) -> {*,/};

    // Define own replacement list for ROR
    BIN(>)  -> {<=,!=,==};
    BIN(==) -> {<,!=,>};

    // Enable and invoke mutation operators
    AOR;
    ROR;
    LVR;
end
```

Listing 4. Script to define the mutation operators and process in detail.

process. The compiler options can be used to choose from the various built-in mutation operators that apply predefined replacements. The following general operator groups are currently available within MAJOR via compiler options:

- **ORB** (Operator Replacement Binary): Replace a binary arithmetic (AOR), logical (LOR), relational (ROR), or shift (SOR) operator with valid alternatives. That is, ORB is the union of AOR, LOR, ROR, and SOR.
- **ORU** (Operator Replacement Unary): Replace a unary operator with valid alternatives.
- **LVR** (Literal Value Replacement): Replace a literal value by a positive value, a negative value, and zero.

In order to avoid potential conflicts with future releases of the Java compiler, MAJOR extends the non-standardized `-X` options of the compiler. The conditional mutation step can generally be enabled with `-XMutator`. Furthermore, this option provides a wildcard and a list of valid sub-options, which correspond to the names of the mutation operators. For instance, the following two commands enable all operators by means of the wildcard `ALL` (1) and a specified subset of the available operators `AOR`, `ROR`, and `ORU` (2):

```
(1) javac -XMutator:ALL ...
(2) javac -XMutator:AOR,ROR,ORU ...
```

Instead of using compiler options, MAJOR can parse mutation scripts written in a domain specific language. These scripts enable a detailed definition and a flexible application of mutation operators. For example, the replacement list for every operator in an operator group can be specified and mutations can be enabled or disabled for certain packages, classes, or methods. The scripting language enhances the predefined mutation options while using the compiler options' keywords for the operators. Listing 4 shows an example of a mutation script that includes the following tasks:

- Define specific replacement lists for AOR and ROR
- Invoke the AOR and ROR operators on reduced lists
- Invoke the LVR operator without restrictions

B. *Driver Class*

The conditional mutation components reference the external driver to gain access to the mutant identifier `M_NO`. Additionally, the driver has to furnish the mutation coverage method `COVERED` if mutation coverage has been enabled within the compiler. Listing 5 shows an example of a driver class that

TABLE I
MEMORY CONSUMPTION OF MAJOR AND RUNTIME STATISTICS FOR THE INVESTIGATED APPLICATIONS.

Application	*LOC**	*Files*	*Mutants*			*Memory consumption**		*Runtime of test suite in seconds*	
			generated	*covered*	*killed*	*original*	*instrumented*	*original*	*instrumented*
aspectj	372,751	1,975	406,382	20,144	10,361	559	813 (45.44%)	4.3	4.8 (11.63%)
apache ant	103,679	764	60,258	28,118	21,084	237	293 (23.63%)	331.0	335.0 (1.21%)
jfreechart	91,174	585	68,782	29,485	12,788	220	303 (37.73%)	15.0	18.0 (20.00%)
itext	74,318	395	124,184	12,793	4,546	217	325 (49.77%)	5.1	5.6 (9.80%)
java pathfinder	47,951	543	37,331	8,918	4,434	182	217 (19.23%)	17.0	22.0 (29.41%)
commons math	39,991	408	67,895	54,326	44,084	153	225 (47.06%)	67.0	83.0 (23.88%)
commons lang	19,394	85	25,783	21,144	16,153	104	149 (43.27%)	10.3	11.8 (14.56%)
numerics4j	3,647	73	5,869	4,900	401	73	90 (23.29%)	1.2	1.3 (8.33%)

*Physical lines of code as reported by sloccount (non-comment and non-blank lines) and memory consumption of the compiler in MB.

```java
package major.mutation;

public class Driver{
    public static final int MAX_NO=100000;
    public static int[] COV = new int[MAX_NO];

    public static int M_NO=0;

    public static boolean COVERED(int from, int to){
        for(int i=from; i<=to; ++i){
            COV[i]++;
        }
        return false;
    }
}
```

Listing 5. Simple driver class with mutant identifier and coverage method.

provides both the mutant identifier and the mutation coverage method that gathers the coverage information at runtime. The identifier and the coverage method must be implemented in a static context to avoid any overhead caused by polymorphism and instantiation. Nevertheless, the fully qualified name of the driver class itself can be configured.

In order to keep MAJOR non-invasive, the driver class does not have to be available on the classpath during compilation. That means that MAJOR does not try to resolve the driver class at compile-time but instead assumes that the mutant identifier and the coverage method will be available in this class at runtime. Thus, the mutants can be generated without having a driver class available during compilation.

IV. PERFORMANCE EVALUATION

In order to evaluate the implementation of conditional mutation, we applied MAJOR to eight open source projects, which differ in size and application domain. Two aspects are of particular interest, namely the runtime and memory overhead of the modified compiler compared with the original version. Additionally, we considered the runtime of the compiled and mutated classes to estimate the overhead caused by injecting the conditional expressions and statements. The graph in Figure 2 shows the necessary compiler runtime for generating and compiling the mutants compared with the original compile time. The time needed is by far less than compiling a project twice and is negligible in consideration of the number of mutants. Since the mutants are generated and compiled on-the-fly by exploiting the compiler's AST, MAJOR avoids any additional overhead and, in particular, precludes recompilation. Therefore, the overhead for generating and compiling the mutants is reduced to a minimum. As shown in Table I, the

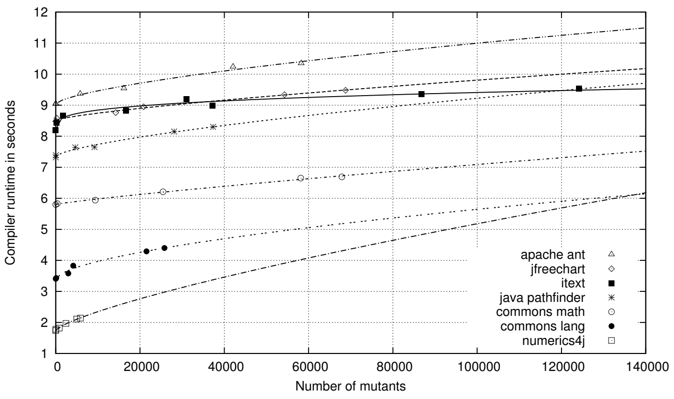

Fig. 2. Compiler runtime for generating and compiling the mutants for all projects except *aspectj*, whose high time overhead, with and without mutation analysis, obscured the visualization of the relevant trends.

space overhead in terms of memory consumption is at most 50%. This overhead includes the necessary memory for all of the transformation phases, including the expanded AST that holds all of the generated mutants. We judge that MAJOR can easily be applied on commodity workstations since it can generate 406,000 mutants with a space overhead of only 45%.

The last column in Table I gives the runtime of the test suites and reveals that the overhead associated with the inserted conditional expressions and statements is application dependent. While it is large for *commons math* (24%) and *java pathfinder* (29%), it is really small for *apache ant* (1%). Overall, the average runtime overhead of 15% is not significant. Further details from the empirical evaluation of conditional mutation's performance can be found in [6].

Evaluating all generated mutants for all tests is an expensive task and generally infeasible for large applications with a significant number of mutants. However, if the goal is to assess a complete test suite, it is sufficient to kill a mutant with one test case. Thus, ordering the test suite according to the runtime of the individual test cases leads to a reduced runtime since a mutant, which can be revealed, is always killed just once, in fact by the fastest test that can detect it.

Nevertheless, a test case usually covers a certain scenario or functionality and hence can only kill mutants corresponding to the covered methods and blocks. Moreover, mutants may not be covered by any test case. Thus, the runtime optimized analysis still leads to a significant overhead since the whole test suite is executed for every uncovered mutant.

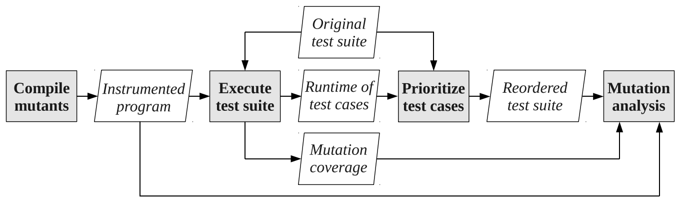

Fig. 3. Improved workflow for mutation analysis, enabled by MAJOR.

In order to avoid such unnecessary evaluations, a mutation analysis process should always take coverage information into account. Since MAJOR easily provides coverage information, it not only reduces the runtime for fault seeding but it also enables an efficient workflow for mutation analysis.

Due to the conditional mutation approach, all mutants and the original version are embedded in the compiled classes. Thus, the complete workflow that is shown in Figure 3 can be performed without any recompilation. In order to determine the runtime of the individual test cases of the corresponding test suite, this workflow first runs the complete test suite on the original program and captures the runtime of every test case. Additionally, it gathers the mutation coverage information on a per-test case basis (i.e., it knows which test case covers which mutants after executing the complete test suite). Now, it prioritizes all tests according to their runtime so that the fastest test case will run first. Excluding the unreachable mutants, every test case is run for every covered and yet not killed mutant. Thus, the runtime of the complete mutation analysis process is reduced since a mutant is always killed by the fastest test case that can detect it.

The following example, derived from the *commons math* application described in Table I, quantitatively demonstrates the benefits of the improved mutation analysis workflow enabled by MAJOR. For illustrative purposes, we focus on the 13 tests for the genetic algorithm package in *commons math*. With runtimes that vary between 4 ms and 520 ms, these tests can reach 268 mutants and ultimately kill 180 mutants. Figure 4 shows the different runtimes of the mutation analysis process, with and without employing mutation coverage information. Additionally, the optimized order of the test cases is compared with the original and random orderings. A curve with a steeper gradient in the graph kills more mutants in less time, finishes the entire process earlier, and hence indicates a better method. Since the runtimes of the individual tests vary by two orders of magnitude, ordering the tests and employing the coverage information significantly reduces the runtime by 60% from 98 sec to 39 sec.

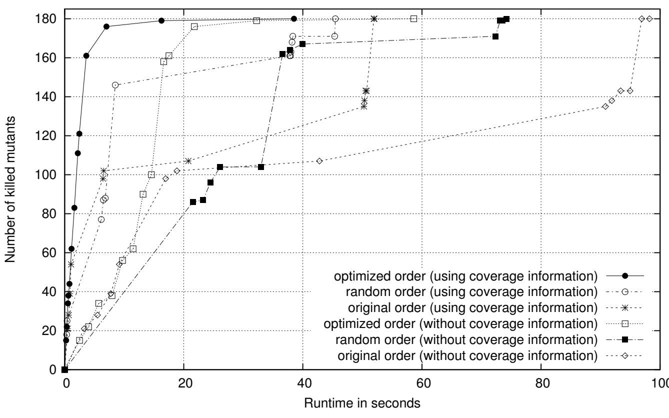

Fig. 4. Runtime of the mutation analysis process.

V. Conclusion and Future Work

This paper describes MAJOR, a fault seeding and mutation analysis system integrated into the Java Standard Edition compiler. Designed as a non-invasive tool, it is applicable in every Java-based environment and customizable with common compiler options and its own domain specific language. So far, MAJOR is implemented in the Java 6 Standard Edition compiler and it has been applied to real-world programs consisting of up to 373,000 lines of source code.

In order to further improve MAJOR, we plan to implement new mutation operators and enhance the domain specific language. We will also conduct a comprehensive study comparing MAJOR with the related tools that were mentioned in Section I. Moreover, MAJOR's ease-of-use, efficiency, and extensibility will also enable, for the first time ever, the replication of prior experiments that primarily focused on relatively simple programs written in the C programming language (e.g., [1], [10]). These same characteristics of MAJOR suggest that the tool is ready for transfer into industrial practice. After enhancing and completely evaluating both the domain specific language and the implementation for the new Java 7 compiler, MAJOR will be available for download on its project web site:

http://www.mathematik.uni-ulm.de/sai/major

References

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 402–411, 2005.
[2] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, 1980.
[3] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.
[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 *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 169–175, 2007.
[5] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. Report TR-09-06, CREST Centre, King's College London, UK, 2009.
[6] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using conditional mutation to increase the efficiency of mutation analysis. In *Proceedings of the 6th Workshop on Automation of Software Test*, AST '11, pages 50–56, 2011.
[7] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. MuJava: A mutation system for Java. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 827–830, 2006.
[8] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996.
[9] D. Schuler and A. Zeller. (Un-)covering equivalent mutants. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, ICST '10, pages 45–54, 2010.
[10] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 351–360, 2008.