The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java

René Just
University of Washington
Computer Science and Engineering
Seattle, WA, USA
rjust@cs.washington.edu

ABSTRACT

Mutation analysis seeds artificial faults (mutants) into a program and evaluates testing techniques by measuring how well they detect those mutants. Mutation analysis is well-established in software engineering research but hardly used in practice due to inherent scalability problems and the lack of proper tool support. In response to those challenges, this paper presents Major, a framework for mutation analysis and fault seeding. Major provides a compiler-integrated mutator and a mutation analyzer for JUnit tests.

Major implements a large set of optimizations to enable efficient and scalable mutation analysis of large software systems. It has already been applied to programs with more than 200,000 lines of code and 150,000 mutants. Moreover, Major features its own domain specific language and is designed to be highly configurable to support fundamental research in software engineering. Due to its efficiency and flexibility, the Major mutation framework is suitable for the application of mutation analysis in research and practice. It is publicly available at `http://mutation-testing.org`.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Testing tools

Keywords

Mutation testing, compiler-integrated mutation, weak mutation, strong mutation

1. INTRODUCTION

Mutation analysis is a well-known approach to assess the quality of test suites or testing techniques [1]. It measures a test suite's ability to distinguish a program under test (*original version*) from many small syntactic variations, called *mutants*. Quality is quantified in the *mutation score*, which is the percentage of mutants that a test suite can *kill*, i.e., distinguish from the original version. Mutants are created by systematically injecting small artificial faults into the program under test, based on *mutation operators*. Examples for such mutation operators are the deletion of program instructions, the replacement of arithmetic operators (e.g., `a+b` $\mapsto$ `a-b`), or the manipulation of branch conditions.

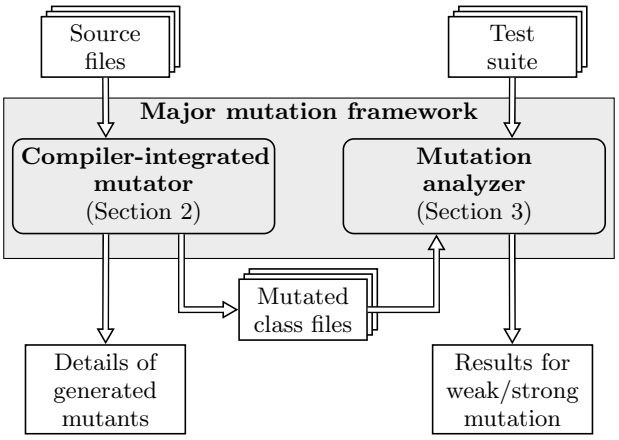

Figure 1: The main components of the Major mutation framework.

Although mutation analysis is well-established in software engineering research, it is not yet widely accepted in practice due to scalability issues and the lack of proper tool support. In response to those challenges, this paper presents Major, a full-fledged framework for mutation analysis. Major enables efficient and scalable mutation analysis, and also fundamental research in software engineering.

The Major mutation framework consists of the following two main components, as visualized in Figure 1:

- **Compiler-integrated mutator**
  In contrast to other mutation analysis tools, Major's mutator can be used standalone (e.g., for fault seeding) as it is integrated into the OpenJDK Java compiler. Moreover, Major's mutator can be employed as the mutator component for an arbitrary mutation analysis back-end.
- **Mutation analyzer**
  Major provides a default mutation analyzer for JUnit tests. This analyzer iteratively executes the JUnit tests and provides the mutation analysis results. Besides implementing a wide variety of optimizations, the mutation analyzer supports weak and strong mutation analysis.

ISSTA'14, July 21–25, 2014, San Jose, CA, USA
Copyright 2014 ACM 978-1-4503-2645-2/14/07...$15.00
http://dx.doi.org/10.1145/2610384.2628053

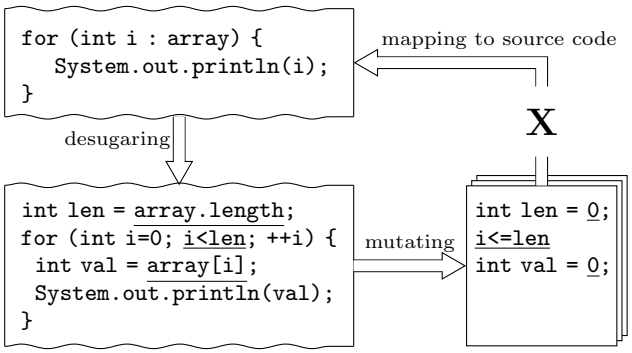

Figure 2: Byte code mutants of a desugared for-each loop that cannot be mapped back to the original source code (mutated expressions are underlined). In contrast, Major's mutator applies mutation before the desugaring step, thus avoids such mutants.

2. COMPILER-INTEGRATED MUTATOR

Developers of mutation tools for Java programs have hitherto either focused on source code mutation (e.g., Jester, MuJava[1] [8]) or byte code mutation (e.g., Javalanche [9], Jumble [2], and PIT).

Compiler-integrated mutation, which transforms the abstract syntax tree (AST), has been widely neglected — presumably due to engineering challenges and effort. Major is the only mutation framework for Java that features a compiler-integrated mutator. Major's mutator is integrated in the OpenJDK Java compiler and transforms the attributed AST. Attributed means that type information is available on the AST (cf. [7]). Major's mutator generates and embeds all mutants during the compilation of the original source code, hence prevents costly recompilation of mutants.

2.1 Why Compiler-Integrated Mutation?

This section sheds light on the advantages of Major's compiler-integrated approach compared to source code and byte code mutation.

2.1.1 Major vs. Source Code Mutation

Applying source code mutation (e.g., pattern-based replacements) is straightforward and requires little engineering effort. There are, however, two serious drawbacks to this approach. First, the lack of semantic information (e.g., type information) leads to many uncompilable mutants (e.g., invalid mutations of final fields or the control flow). Second, individually compiling all generated mutants leads to a significant compilation overhead. Major's mutator avoids those drawbacks by operating on the AST and embedding all mutants in the generated class files [7].

2.1.2 Major vs. Byte Code Mutation

Byte code mutation is the predominant approach in existing mutation tools. It is appealing for two reasons. First, it prevents recompilation and can be applied on the fly, e.g., during class loading. Second, byte code representation is simpler than source code and therefore easier to mutate. However, byte code mutation also has drawbacks and limitations. Most importantly, byte code is desugared and simplified. Desugaring refers to the process in which syntactic sugar (i.e., a language feature that improves expressiveness) is replaced with an equivalent but more complex language construct.

Figure 2 gives an example for desugaring in Java: the for-each loop, which represents syntactic sugar, is transformed into an ordinary for loop. There are two problems that arise when desugared code is mutated. First, mutating desugared code generates mutants that could have never been introduced into the source code. Second, the generated mutants cannot be mapped back to the source code, which hampers manual inspection of mutants. Since Major's mutator operates on the AST, which represents the code that a developer wrote, Major avoids generating mutants that cannot be mapped to a specific location in the original source code.

2.2 Mutation Operators

Major's mutator supports a set of commonly applied mutation operators (cf. [10]):

- Binary operator replacement: replace occurrences of binary operators such as arithmetic, logical, shift, conditional, and relational operators.
- Unary operator replacement: replace occurrences of unary operators such as negation.
- Constant value replacement: replace numerical and string constants with pre-defined constants.
- Branch condition manipulation: manipulate branch conditions without logical connectors.
- Statement deletion: delete (omit) single statements such as method calls.

Some mutation operators have been shown to generate redundant mutants [6], which have negative effects on efficiency and accuracy. Therefore, Major's mutator employs the non-redundant versions of those mutation operators by default.

2.3 Generating Mutants

Recall that Major's mutator is integrated into the Java compiler. To enable mutation in Major's mutator, the compiler option `-XMutator` has to be provided. This option takes a list of mutation operator names as an argument. For a hassle-free mutation using Major's default configuration, this option also provides a wildcard to enable all mutation operators. The following two commands show examples for (1) generating all mutants using the wildcard `ALL` and (2) generating mutants using only the `AOR` (arithmetic operator replacement) and `STD` (statement deletion) mutation operators:

(1) `javac -XMutator:ALL MyFile.java`

(2) `javac -XMutator:AOR,STD MyFile.java`

Table 1 shows run-time results for applying Major's compiler to mutate 12 open source projects using all mutation operators. The depicted run time is the total run time necessary to generate, embed, and compile all mutants. The average compilation overhead per 1,000 generated mutants is 0.14 seconds, which is negligible in consideration of the large number of mutants.

[1] MuJava is not a traditional source code mutation tool but compiles mutants individually to filter invalid mutants.

Table 1: Total run time of Major's mutator to generate, embed, and compile all mutants. Numbers in parentheses give the baseline run time of the original Java compiler.

Program	KLOC	Classes	Mutants	Compile time
Apache POI	223	2,056	157,965	75s (54s)
GNU Trove	117	1,594	71,683	31s (24s)
IText PDF	76	592	108,174	28s (12s)
JFreeChart	91	610	67,097	20s (12s)
Math[1]	40	536	55,550	14s (8s)
Joda Time	27	227	18,415	14s (10s)
Collections[1]	26	273	11,832	8s (7s)
Jaxen	21	318	7,179	6s (4s)
Lang[1]	19	147	18,887	8s (5s)
JDom	15	161	10,778	10s (8s)
IO[1]	9	104	6,756	5s (3s)
Numerics4J	4	86	5,650	3s (2s)
Overall	668	6,704	539,966	222s (149s)

[1]Apache Commons libraries

2.4 Inspecting and Exporting Mutants

Major's mutator embeds all mutants during compilation and produces a detailed summary of the generated mutants. This summary includes all necessary information (such as source code location and mutation operator) to inspect or visualize the mutants. Besides, Major's mutator provides an option to generate mutants as individual source files to support use cases that require individual source code mutants.

2.5 Major's Domain Specific Language

When conducting experiments in software testing research, more control of the mutation process might be desirable. Therefore, Major features a domain specific language (DSL) that provides a high degree of control of the mutation process without overwhelming the user with a large number of additional options. In summary, Major's DSL enables:

- Selection of specific packages, classes, or methods. This prevents changing the build infrastructure if mutation analysis should only be applied to specific parts of the code base.
- Configuration of enabled/disabled mutation operators on a per package, class, or method basis. This enables applying different sets of mutation operators to different parts of the code base.
- Configuration of built-in mutation operators (e.g., apply only certain replacements). This generally provides fine-grained selective mutation but also enables conducting or reproducing empirical studies on mutant reduction.

3. MUTATION ANALYZER

Major provides a default mutation analyzer for the mutation analysis of JUnit tests. This analyzer builds on top of Apache Ant and implements a wide variety of optimizations to ensure efficiency and scalability. Generally, Major's mutation analyzer operates in three phases:

1. Pre-pass for monitoring state infection: execute test suite and determine which test has to be executed on which mutant. This phase also measures the run time of each test case.
2. Test suite prioritization based on test run time.
3. Strong mutation analysis leveraging the information of phase 1 and the prioritized test suite of phase 2.

One unique feature of Major is that it supports weak and strong mutation analysis. In weak mutation analysis, a test kills a mutant if the test execution leads to a difference between the program state of the mutant and the program state of the original version. In contrast, strong mutation requires that this difference propagates to an observable output, i.e., an assertion failure or an exception.

3.1 Monitoring State Infection

Gathering coverage information to avoid unnecessary mutant executions is a common optimization in mutation analysis — a test that does not cover (i.e., reach and execute) a mutant does not need to be run on that mutant. Major takes this approach a step further and monitors state infection. The program state of a mutant is said to be infected if it differs from the program state of the original version after execution of the mutated expression [3]. A test is not executed on a mutant if it cannot achieve state infection on that mutant. Major efficiently monitors state infection during its pre-pass (phase 1) and only considers state-infected mutants for the strong mutation analysis (phase 3).

3.2 Test Suite Prioritization

Another common optimization in mutation analysis is to exclude mutants from the analysis once they have been killed. When performing mutation analysis to assess the quality of a test suite, it is sufficient to kill a mutant with one test case. Major also takes this optimization a step further and orders the test suite by runtime of the individual test cases. This ensures that a mutant is killed by the fastest test case that can kill that mutant [5].

3.3 Strong Mutation Analysis

Based on the information gathered in the first two phases, Major's mutation analyzer performs strong mutation analysis using the prioritized test suite. It only executes a test on a (not killed) mutant if the test achieved state infection on that mutant. Some mutants lead to an infinite loop, for instance when mutating loop conditions. Therefore, the analyzer implements a timeout heuristic to prevent the mutation analysis process from getting stuck — the timeout for a test on a certain mutant is calculated based on the test's run time on the original version.

Major's mutation analyzer reports the following results for strong mutation analysis:

- Number of generated, covered, state-infected, and killed mutants — Major also reports all ratios including the mutation score.
- Reason why a mutant was killed (test assertion, exception, or timeout).
- List of mutants not killed, for further inspection.

Table 2: Total run time of Major's mutation analyzer to perform weak and strong mutation. Run time in parentheses gives the baseline run time of all tests on the uninstrumented program version. Mutation score is the ratio of detected to generated mutants and mutation score in parentheses gives the ratio of detected to covered mutants.

Program	Mutants	Tests	Mut. score	Analysis run time		
				strong	weak	
IText PDF	108,174	92	12% (75%)	118m	21s	(16s)
GNU Trove	71,683	544	5% (64%)	25m	41s	(20s)
Scratchpad[2]	71,125	703	34% (81%)	609m	117s	(85s)
JFreeChart	67,097	2,130	28% (53%)	293m	279s	(187s)
Core[2]	64,567	1,874	9% (49%)	35m	4s	(3s)
Math[1]	55,550	2,169	72% (81%)	105m	402s	(246s)
Ooxml[2]	22,273	643	45% (60%)	439m	183s	(179s)
Lang[1]	18,887	2,047	69% (76%)	14m	41s	(32s)
Joda Time	18,415	3,855	73% (87%)	93m	417s	(146s)
Collections[1]	11,832	1,161	61% (72%)	21m	42s	(34s)
JDom	10,778	1,638	75% (83%)	19m	73s	(55s)
Jaxen	7,179	634	41% (64%)	39m	14s	(13s)
IO[1]	6,756	344	34% (76%)	17m	62s	(60s)
Numerics4J	5,650	218	65% (69%)	1m	5s	(3s)
Overall	539,966	18,052	45% (71%)	1,828m	1,701s	(1,080s)

[1]Apache Commons libraries
[2]Apache POI components

Table 2 gives the run times for performing strong mutation analysis on 14 open source programs. Tests represents the number of developer-written tests released with each program, and the mutation score of those tests is given with respect to generated and covered mutants.

3.4 Weak Mutation Analysis

Recall that the criterion to kill a mutant in weak mutation analysis is to achieve state infection for that mutant. Major monitors state infection during its pre-pass (phase 1), hence weak mutation analysis in Major requires only one execution of the test suite. In fact, if weak mutation analysis is enabled, Major's analyzer only performs the pre-pass and reports the corresponding results.

Table 2 also gives the run times for performing weak mutation analysis on the depicted 14 open source programs. Performing weak mutation analysis requires program instrumentation for the pre-pass phase, which leads to a run-time overhead. For an easy comparison with the baseline, Table 2 gives the run time of each test suite on the uninstrumented program version (numbers in parentheses).

4. CONCLUSIONS AND FUTURE WORK

This paper presents Major, a framework for mutation analysis and fault seeding. It provides a compiler-integrated mutator and a mutation analyzer for JUnit tests. Major supports weak and strong mutation analysis, and is efficient and scalable due to a large set of implemented optimizations.

Major has already been applied in large-scale experiments (e.g., [3, 4]). Moreover, Major is highly configurable and offers several features that support conducting experiments or reproducing prior studies in software engineering research. Hence, the Major mutation framework is suitable for the application of mutation analysis in research and practice.

Recent studies on the correlation between mutants and real faults have revealed that the commonly applied set of mutation operators should be augmented [4]. Those new mutation operators will be available in a future release of Major. Morever, mutation analysis offers great potential for parallelization — every mutant can be analyzed independently. Therefore, providing built-in support for parallelization is another area for improving Major.

The Major mutation framework is publicly available on its project web site:

`http://mutation-testing.org`

5. REFERENCES

[1] 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.

[2] 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 (TAIC PART)*, pages 169–175, 2007.

[3] R. Just, M. D. Ernst, and G. Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2014. To appear.

[4] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? Technical Report UW-CSE-14-02-02, University of Washington, 2014.

[5] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 11–20, 2012.

[6] R. Just and F. Schweiggert. Higher accuracy and lower runtime: Efficient mutation analysis using non-redundant mutation operators. *Software Testing, Verification and Reliability (JSTVR)*, 2014. To appear.

[7] R. Just, F. Schweiggert, and G. M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 612–615, 2011.

[8] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. MuJava: A mutation system for Java. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 827–830, 2006.

[9] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for Java. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 297–298, 2009.

[10] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 351–360, 2008.