MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler
René Just¹ and Franz Schweiggert¹ and Gregory M. Kapfhammer²

¹Department of Applied Information Processing, Ulm University
²Department of Computer Science, Allegheny College

IMPORTANT CONTRIBUTIONS
- Enhances the Java 6 Standard Edition compiler
- Provides its own domain specific language (DSL)
- Easily applicable in all Java development environments
- Effectively reduces mutant generation time to a minimum

CONDITIONAL MUTATION
- Transforms the program’s abstract syntax tree (AST)
- Encapsulates the mutations within conditional statements
  - \( y = (M_{NO} == 1) ? a - x : (M_{NO} == 2) ? a + x : a * x \)

COMPILATION DETAILS

Configuration
- Compiler options
- Domain specific language
- Mutant identifier
- Mutation coverage

Runtime Details
- \( y = (M_{NO} == 1) ? a - x : (M_{NO} == 2) ? a + x : a * x \)

Supported Features
- Simple compiler options enable the mutation analysis
- Configurable mutation operators by means of a DSL
- Determination of mutation coverage by running the original code

IMPLEMENTATION DETAILS
- \( y = (M_{NO} == 1) ? a - x : (M_{NO} == 2) ? a + x : a * x \)

Evaluation of Mutation Analysis Processes
- Optimized order (using coverage information)
- Random order (using coverage information)
- Original order (without coverage information)
- Original order (without coverage information)

FUTURE WORK
- Implementing new mutation operators at the semantic level
- Extending the domain specific language to support new operators
- Integrating conditional mutation into the new Java 7 compiler

Figure: Complier runtime to generate and compile the mutants for all of the projects.

Figure: Compiler runtime in seconds.

Figure: Runtime of MAJOR’s Compiler.

Figure: Optimization of mutation analysis processes.

Figure: Simple driver class implementation.

Figure: DSL script to define the mutation process.

Figure: Integration of the conditional mutation approach into the compilation process.

Figure: Multiple mutated binary expression as the right hand side of an assignment statement.

Figure: Collecting coverage information.

Figure: Minimizing the runtime of mutation analysis by means of test prioritization and mutation coverage.

Important Contributions
- Enhances the Java 6 Standard Edition compiler
- Provides its own domain specific language (DSL)
- Easily applicable in all Java development environments
- Effectively reduces mutant generation time to a minimum

Conditional Mutation
- Transforms the program’s abstract syntax tree (AST)
- Encapsulates the mutations within conditional statements

Mutation Coverage
- It is impossible to kill a mutant if it is not reached and executed
- Additional instrumentation determines the covered mutations
- Mutation coverage is only examined if the tests execute the original code
- An external driver efficiently records the covered mutations as ranges
- Only those mutants covered by a test case are executed

Mutation Coverage Process
- Compile mutants
- Execute test suite
- Mutation coverage
- Prioritize test cases
- Rendered test suite
- Mutation analysis

Figure: Runtime of the mutation analysis processes.