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

René Just\textsuperscript{1}, Franz Schweiggert\textsuperscript{1}, and Gregory M. Kapfhammer\textsuperscript{2}

\textsuperscript{1}Ulm University, Germany
\textsuperscript{2}Allegheny College, USA

26th International Conference on Automated Software Engineering
Lawrence, Kansas, USA
November 6 - 12, 2011
Overview of MAJOR

A Tool for Mutation Analysis

Just, Kapfhammer, and Schweiggert

Ulm University, Allegheny College

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

Compiler-Integrated

A Tool for Mutation Analysis
Overview of MAJOR

- Compiler-Integrated
- A Tool for Mutation Analysis
- Fast and Scalable
Overview of MAJOR

A Tool for Mutation Analysis

Compiler-Integrated

Domain Specific Language

Fast and Scalable

Just, Kapfhammer, and Schweiggert

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

A Tool for Mutation Analysis

- Compiler-Integrated
- Domain Specific Language
- Fast and Scalable
- Configurable and Extensible
Overview of MAJOR

- Compiler-Integrated
- Mutation Coverage Information
- Domain Specific Language

A Tool for Mutation Analysis

- Fast and Scalable
- Configurable and Extensible
Overview of MAJOR

- Compiler-Integrated
- Mutation Coverage Information
- Domain Specific Language

- A Tool for Mutation Analysis
- Fast and Scalable
- Enables Optimized Workflow
- Configurable and Extensible
Overview of Mutation Analysis
Overview of Mutation Analysis

Methodically inject small syntactical faults into the program under test

Mutation Analysis
Overview of Mutation Analysis

Methodically inject small syntactical faults into the program under test

Mutation Analysis

Unbiased and powerful method for assessing test oracles and input values
Overview of Mutation Analysis

Methodically inject small syntactical faults into the program under test

Mutation Analysis

Unbiased and powerful method for assessing test oracles and input values

Useful method for fault seeding during the empirical study of testing techniques

Just, Kapfhammer, and Schweiggert
Ulm University, Allegheny College
MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler
Overview of Mutation Analysis

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

    y = a * x;

    y += b;
    return y;
}

public int max(int a, int b) {
    int max = a;

    if (b > a) {
        max = b;
    }

    return max;
}
```

Just, Kapfhammer, and Schweiggert

Ulm University, Allegheny College

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

```java
public int eval(int x) {
    int a = 3, b = 1, y;
    y = a * x;
    y += b;
    return y;
}

public int max(int a, int b) {
    int max = a;
    if (b > a) {
        max = b;
    }
    return max;
}
```

Just, Kapfhammer, and Schweiggert

Ulm University, Allegheny College

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

```java
public int eval(int x) {
    int a=3, b=1, y;
    y = a * x;
    y += b;
    return y;
}
```

```java
public int max(int a, int b) {
    int max = a;
    if(b > a) {
        max=b;
    }
    return max;
}
```

- $y = a - x$
- $y = a + x$
- $y = a / x$
- if(b < a)
- if(b != a)
- if(b == a)
MAJOR’s Compiler
MAJOR’s Compiler

Enhanced Standard Java Compiler

Just, Kapfhammer, and Schweiggert
Ulm University, Allegheny College

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

Source Files → MAJOR's Compiler → Enhanced Standard Java Compiler
MAJOR’s Compiler

Common Compiler Options

Source Files

MAJOR’s Compiler

Enhanced Standard Java Compiler

Just, Kapfhammer, and Schweiggert

Ulm University, Allegheny College

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

Source Files → MAJOR’s Compiler → Enhanced Standard Java Compiler

Common Compiler Options

Domain Specific Language
MAJOR’s Compiler

Common Compiler Options

Domain Specific Language

Source Files

MAJOR’s Compiler

Bytecode with Embedded Mutants

Enhanced Standard Java Compiler

Just, Kapfhammer, and Schweiggert

Ulm University, Allegheny College

MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler
MAJOR’s Domain Specific Language

// variable declaration
listCOR={&&, ||, ==, !=};
// Define replacement list
BIN(+)<'org'> -> {-, *};
BIN(*)<'org'> -> {/, %};
// Define own operator
myOp{
    BIN(&&) -> listCOR;
    BIN(||) -> listCOR;
    COR;
    LVR;
}
// Enable built-in operator AOR
AOR<'org'>;
// Enable operator myOp
myOp<'java.lang.System@println'>;
MAJOR’s Domain Specific Language

```java
// variable declaration
listCOR={&&, ||, ==, !=};

// Define replacement list
BIN(+)<"org"> -> {-, *};
BIN(*)<"org"> -> {/, %};

// Define own operator
myOp{
    BIN(&&) -> listCOR;
    BIN(||) -> listCOR;
    COR;
    LVR;
}

// Enable built-in operator AOR
AOR<"org">;

// Enable operator myOp
myOp<"java.lang.System@println">;
```

Specify mutation operators in detail
MAJOR's Domain Specific Language

// variable declaration
listCOR={&&, ||, ==, !=};

// Define replacement list
BIN(+)<"org"> -> {-, *};
BIN(*)<"org"> -> {/, %};

// Define own operator
myOp{
    BIN(&&) -> listCOR;
    BIN(||) -> listCOR;
    COR;
    LVR;
}

// Enable built-in operator AOR
AOR<"org">;

// Enable operator myOp
myOp<"java.lang.System@println">;
MAJOR’s Domain Specific Language

```java
// variable declaration
listCOR={&&, ||, ==, !=};

// Define replacement list
BIN(+)<"org"> -> {−, *};
BIN(*)<"org"> -> {/, %};

// Define own operator
myOp{
  BIN(&&) -> listCOR;
  BIN(||) -> listCOR;
  COR;
  LVR;
}

// Enable built-in operator AOR
AOR<"org">;

// Enable operator myOp
myOp<"java.lang.System@println">;
```

Specify mutation operators in detail

Define own mutation operator groups

Enable operators for a specific package, class, or method
Optimized Mutation Analysis Process

1. Embed and compile all mutants
2. Run test suite on instrumented program
3. Sort tests according to their runtime
4. Perform mutation analysis with reordered test suite
Optimized Mutation Analysis Process

1. Embed and compile all mutants
2. Run test suite on instrumented program
3. Sort tests according to their runtime
4. Perform mutation analysis with reordered test suite
Optimized Mutation Analysis Process

1. Embed and compile all mutants
2. Run test suite on instrumented program
3. Sort tests according to their runtime
4. Perform mutation analysis with reordered test suite
Optimized Mutation Analysis Process

1. Embed and compile all mutants
2. Run test suite on instrumented program
3. Sort tests according to their runtime
4. Perform mutation analysis with reordered test suite
Conclusion

Key Concepts and Features:

- Compiler-integrated solution
- Furnishes its own domain specific language
- Provides mutation coverage information
Conclusion

Key Concepts and Features:

- Compiler-integrated solution
- Furnishes its own domain specific language
- Provides mutation coverage information

Characteristics of MAJOR:

- Fast and scalable technique
- Configurable and extensible mutation tool
- Enables an optimized workflow for mutation analysis
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

Do you want to learn more details about MAJOR?

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

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

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

OPTIMIZED MUTATION ANALYSIS PROCESS
1. Embed and compile all mutants into the original program
2. Run tests on original program to gather runtime and coverage
3. Sort tests in ascending order according to their runtime
4. Perform mutation analysis while employing the reordered test suite and the mutation coverage information

EVALUATION OF MUTATION ANALYSIS PROCESSES
- Minimizing the runtime of mutation analysis by means of test prioritization and mutation coverage
- Negligible overhead for generating and compiling the mutants
- Applicable on commodity workstations, even for large projects

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