Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs

Gregory M. Kapfhammer†

Department of Computer Science
Allegheny College
http://www.cs.allegheny.edu/~gkapfham/

University of Sheffield – February 3, 2012

† Joint with René Just and Franz Schweiggert (University of Ulm) and Jonathan Miller Kauffman (Allegheny College)
Accessing the Presentation

Scan this QR Code with your smartphone!

... or, visit this Web site:

http://is.gd/rekiwo

... or, ask me for a USB drive!
Presenter Introduction: Gregory M. Kapfhammer
What is a Test Case?

Method
Under Test
What is a Test Case?

Input → Method Under Test
What is a Test Case?

Input → Method Under Test → Output
What is a Test Case?

Diagram:
- Test Set Up
- Method Under Test
What is a Test Case?

1. Input
2. Method Under Test
3. Test Set Up
What is a Test Case?

Input ➔ Method Under Test ➔ Output

Test
Set Up
What is a Test Case?

Input → Method Under Test → Output

Test Set Up → Method Under Test

Test Clean Up
What is a Test Case?

- **Input**
- **Method Under Test**
- **Output**
- **Test Oracle**
- **Test Clean Up**
- **Test Set Up**
- **Clean Up**
What is a Test Case?

Input → Method Under Test → Output → Expected Output

Test Set Up → Method Under Test → Output → Test Oracle

Test Clean Up
What is a Test Case?

- **Input**
- Method Under Test
- **Test Set Up**
- Test Clean Up
- **Output**
- Expected Output
- **Test Oracle**
- **Test Verdict**
What is a Test Case?

- Input
- Method Under Test
- Test Clean Up
- Expected Output
- Test Oracle
- Test Verdict

Diagram:
- Input
- Method Under Test
- Output
- Expected Output
- Test Oracle
- Test Verdict

Steps:
1. Input
2. Method Under Test
3. Output
4. Expected Output
5. Test Oracle
6. Test Verdict

Flow:
- Input → Method Under Test → Output → Expected Output → Test Oracle → Test Verdict
- Input → Method Under Test → Output → Expected Output → Test Oracle → Test Verdict

Kapfhammer Allegheny College
Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs
What is a Test Case?

- **Input**
- **Method Under Test**
- **Output**
- **Expected Output**
- **Test Oracle**
- **Test Verdict**

**Process:**
1. **Set Up**
2. **Test**
3. **Clean Up**
4. **Expected Output**
5. **Test Verdict**

The test case passes and the code is correct!
What is a Test Case?

Input → Method Under Test → Output → Test Oracle → Test Verdict

- Test Set Up
- Expected Output
- Test Clean Up
- Output
- Test Verdict
- Input
What is a Test Case?

The test case fails and a defect is found!
What is a Test Suite?

$T_1$  $T_2$
## What is a Test Suite?

| $T_1$ | $T_2$ | $T_3$ | $T_4$ |
What is a Test Suite?

\[ T_1 \quad T_2 \quad T_3 \quad T_4 \quad T_5 \quad T_6 \]
What is a Test Suite?

$T_1$  $T_2$  $T_3$  $T_4$  $T_5$  $T_6$  $T_7$  $T_8$
What is a Test Suite?

\[ T_1 \quad T_2 \quad T_3 \quad T_4 \quad T_5 \quad T_6 \quad T_7 \quad T_8 \quad T_9 \quad T_{10} \]
What is a Test Suite?

Test Suite $T = \langle T_1, T_2, \ldots, T_9, T_{10} \rangle$
What is a Test Suite?

Test Suite \( T = \langle T_1, T_2, \ldots, T_9, T_{10} \rangle \)
What is a Test Suite?

Test Suite $T = \langle T_1, T_2, \ldots, T_9, T_{10} \rangle$
What is a Test Suite?

Test Suite $T = \langle T_1, T_2, \ldots, T_9, T_{10} \rangle$
What is a Test Suite?

Test Suite $T = \langle T_1, T_2, \ldots, T_9, T_{10} \rangle$
What is a Test Suite?

Test Suite \( T = \langle T_1, T_2, \ldots, T_9, T_{10} \rangle \)
What is a Test Suite?

Test Suite $T = \langle T_1, T_2, \ldots, T_9, T_{10} \rangle$
What is a Test Suite?

Test Suite \( T = \langle T_1, T_2, \ldots, T_9, T_{10} \rangle \)

Requirements \( R = \{ R_1, \ldots, R_6 \} \), Features \( F = \{ F_1, \ldots, F_4 \} \), Bug Fixes \( B = \{ B_1, B_2 \} \)
What is a Test Suite?

Test Suite $T = \langle T_1, T_2, \ldots, T_9, T_{10} \rangle$

Requirements $R = \{ R_1, \ldots, R_6 \}$, Features $F = \{ F_1, \ldots, F_4 \}$, Bug Fixes $B = \{ B_1, B_2 \}$
What is a Test Suite?

Test Suite $T = \langle T_1, T_2, \ldots, T_9, T_{10} \rangle$

Requirements $R = \{R_1, \ldots, R_6\}$, Features $F = \{F_1, \ldots, F_4\}$, Bug Fixes $B = \{B_1, \B_2\}$
What is a Test Suite?

Test Suite $T = \langle T_1, T_2, \ldots, T_9, T_{10}\rangle$

Requirements $R = \{R_1, \ldots, R_6\}$, Features $F = \{F_1, \ldots, F_4\}$, Bug Fixes $B = \{B_1, B_2\}$
What is a Test Suite?

Test Suite $T = \langle T_1, T_2, \ldots, T_9, T_{10} \rangle$

Requirements $R = \{R_1, \ldots, R_6\}$, Features $F = \{F_1, \ldots, F_4\}$, Bug Fixes $B = \{B_1, B_2\}$
What is a Test Suite?

Test Suite $T = \langle T_1, T_2, \ldots, T_9, T_{10} \rangle$

How Good is Test Suite $T$?

Requirements $R = \{ R_1, \ldots, R_6 \}$, Features $F = \{ F_1, \ldots, F_4 \}$, Bug Fixes $B = \{ B_1, B_2 \}$
What is a Test Suite?

Test Suite \( T = \langle T_1, T_2, \ldots, T_9, T_{10} \rangle \)

How Good is Test Suite \( T \)?

Coverage Analysis

Requirements \( R = \{R_1, \ldots, R_6\} \), Features \( F = \{F_1, \ldots, F_4\} \), Bug Fixes \( B = \{B_1, B_2\} \)
What is a Test Suite?

Test Suite $T = \langle T_1, T_2, \ldots, T_9, T_{10} \rangle$

How Good is Test Suite $T$?

Coverage Analysis

Mutation Analysis

Requirements $R = \{ R_1, \ldots, R_6 \}$, Features $F = \{ F_1, \ldots, F_4 \}$, Bug Fixes $B = \{ B_1, B_2 \}$
Overview of Mutation Analysis

Mutation Operator
Overview of Mutation Analysis

Mutation Operator

Mutation Operator
Overview of Mutation Analysis

Mutation Operator

Mutation Operator

Mutation Operator

Mutation Operator

Kapfhammer
Allegheny College

Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs
Overview of Mutation Analysis

Mutation Operator

Mutation Operator

Mutation Operator

Mutation Operator

Kapfhammer
Allegheny College

Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs
Overview of Mutation Analysis

Mutation Operator

Mutation Operator

Mutation Operator

Mutation Operator

Methodically inject small syntactical faults into the program under test

Kapfhammer

Allegheny College

Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs
Overview of Mutation Analysis

Mutation Operator

Mutation Operator

Mutation Operator

Mutation Operator

Methodically inject small syntactical faults into the program under test.
Overview of Mutation Analysis

Methodically inject small syntactical faults into the program under test
Overview of Mutation Analysis

Mutation Operator

Methodically inject small syntactical faults into the program under test

Kapfhammer Allegheny College
Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs
Overview of Mutation Analysis

Mutation Operator

Mutation Operator

Mutation Operator

Mutation Operator
Overview of Mutation Analysis

Execute the test suite after enabling a single mutant in the program under test.
Overview of Mutation Analysis

Execute the test suite after enabling a single mutant in the program under test.
Overview of Mutation Analysis

Execute the test suite after enabling a single mutant in the program under test
Overview of Mutation Analysis

Test Case $T_1$  Test Case $T_2$  Test Case $T_3$  Test Case $T_4$

Execute the test suite after enabling a single mutant in the program under test.
Overview of Mutation Analysis

- Test Case $T_1$
- Test Case $T_2$
- Test Case $T_3$
- Test Case $T_4$

Execute the test suite after enabling a single mutant in the program under test.
Overview of Mutation Analysis

Execute the test suite after enabling a single mutant in the program under test
Overview of Mutation Analysis

Execute the test suite after enabling a single mutant in the program under test.
Overview of Mutation Analysis

Test Case $T_1$  Test Case $T_2$  Test Case $T_3$  Test Case $T_4$

Execute the test suite after enabling a single mutant in the program under test.
Overview of Mutation Analysis

Execute the test suite after enabling a single mutant in the program under test.
## Overview of Mutation Analysis

<table>
<thead>
<tr>
<th>Test Case $T_1$</th>
<th>Test Case $T_2$</th>
<th>Test Case $T_3$</th>
<th>Test Case $T_4$</th>
</tr>
</thead>
</table>

The test suite *cannot* kill the mutant – either a test suite weakness or an equivalent mutant!
## Overview of Mutation Analysis

<table>
<thead>
<tr>
<th>Test Case $T_1$</th>
<th>Test Case $T_2$</th>
<th>Test Case $T_3$</th>
<th>Test Case $T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Repeat this process for *all* of the test cases and mutants – calculate mutation score when finished.

Kapfhammer Allegheny College

Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs
Contributions of this Presentation

Efficient Mutation Analysis
Contributions of this Presentation

Efficient Mutation Analysis

Challenges
Contributions of this Presentation

- Efficient Mutation Analysis
- Challenges
- Solutions
Contributions of this Presentation

Efficient Mutation Analysis
Challenges
Solutions
Conditional Mutation
Contributions of this Presentation

- Efficient Mutation Analysis
- Conditional Mutation
- Syntax Tree Transformation
- Challenges
- Solutions
Contributions of this Presentation

- Efficient Mutation Analysis
- Challenges
- Solutions
- Conditional Mutation
- Syntax Tree Transformation
- Expressions and Statements

Fundamental Concepts
Contributions of this Presentation

- Efficient Mutation Analysis
- Challenges
- Solutions
- Conditional Mutation
- Syntax Tree Transformation
- Compiler Integrated
- Expressions and Statements
Contributions of this Presentation

Comprehensive Empirical Study

Efficient Mutation Analysis

Conditional Mutation

Syntax Tree Transformation

Compiler Integrated

Expressions and Statements

Challenges

Solutions
Contributions of this Presentation

- Efficient Technique - Fully Integrated into the Java 6 SE Compiler
- Comprehensive Empirical Study
- Efficient Mutation Analysis
- Challenges
- Solutions
- Conditional Mutation
- Syntax Tree Transformation
- Compiler Integrated
- Expressions and Statements

Kapfhammer
Allegheny College
Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs
Understanding 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;
}
```
Understanding 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;
}
```

Methodically inject small syntactical faults into the program under test
Understanding 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;
}
```
Understanding 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;
}
```

- \( y = a - x; \)
- \( y = a + x; \)
- \( y = a / x; \)
- if \( (b < a) \)
- if \( (b \neq a) \)
- if \( (b == a) \)
Understanding 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;
}
```

Unbiased and powerful method for assessing oracles and input values
Understanding 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;
}
```

Unbiased and powerful method for assessing oracles and input values

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

Mutant Generation
Mutation Analysis Challenges

Mutation Operators → Mutant Generation
Mutation Analysis Challenges

- **Program**
- **Mutation Operators** → **Mutant Generation**
Mutation Analysis Challenges

Diagram:
- Program
  - Mutation Operators
  - Mutant Generation
  - Mutants

Fundamental Concepts
## Mutation Analysis Challenges

A program undergoes mutation generation through mutation operators. This process often yields a substantial number of mutants.
Mutation Analysis Challenges

- **Program**
- **Mutation Operators** → **Mutant Generation** → **Mutants**

Often Yields a Substantial Number of Mutants

High Time Overhead for Generation
Mutation Analysis Challenges

- Program
- Mutant Generation
- Mutants
- Mutation Analysis
- Often Yields a Substantial Number of Mutants
- High Time Overhead for Generation

Fundamental Concepts
Mutation Analysis Challenges

Program \rightarrow Mutant Generation \rightarrow Mutants

High Time Overhead for Generation

Often Yields a Substantial Number of Mutants
Mutation Analysis Challenges

Program

Mutant Generation

Mutants

Tests

Mutation Analysis

Results

Often Yields a Substantial Number of Mutants

High Time Overhead for Generation

Mutation Analysis Challenges

Introduction

Mutation Analysis

Empirical Evaluation

Conclusion

Fundamental Concepts
Mutation Analysis Challenges

High Time Overhead for Generation

Individually Executing the Mutants is Too Expensive

Program

Mutant Generation

Mutants

Mutation Analysis

Results

Often Yields a Substantial Number of Mutants

Mutation Operators

Tests
Mutation Analysis Challenges

- Prior Solutions?
- Program
- Mutation Operators
- Mutant Generation
- Mutants
- Tests
- Mutation Analysis
- Results

- Often Yields a Substantial Number of Mutants
- High Time Overhead for Generation
- Individually Executing the Mutants is Too Expensive
## Prior Work in Mutation Analysis

### Improving Mutation Analysis
Prior Work in Mutation Analysis

Improving Mutation Analysis

Offutt and Untch

Kapfhammer
Allegheny College

Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs
Prior Work in Mutation Analysis

Improving Mutation Analysis

Offutt and Untch

Do Fewer
Prior Work in Mutation Analysis

Improving Mutation Analysis

Offutt and Untch

Do Fewer

Sampling

Selection

Fundamental Concepts

Kapfhammer

Allegheny College

Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs
Prior Work in Mutation Analysis

Improving Mutation Analysis

Offutt and Untch

Do Fewer

Do Smarter
Prior Work in Mutation Analysis

Improving Mutation Analysis

- Do Fewer
- Distributed

- Offutt and Untch

Do Smarter

- Weak Mutation
# Prior Work in Mutation Analysis

Improving Mutation Analysis

- Do Fewer
- Do Smarter
- Do Faster

Offutt and Untch
Prior Work in Mutation Analysis

Offutt and Untch

Do Fewer

Improving Mutation Analysis

Do Smarter

Do Faster

Compiler Integrated

Bytecode Transformation

Mutant Schemata
## Prior Work in Mutation Analysis

### Improving Mutation Analysis

- **Do Fewer**
  - Jia and Harman

- **Do Smarter**
  - Higher Order Mutation

- **Do Faster**

### Offutt and Untch
Practical Mutation Analysis

Practical (adjective):

1. Of or concerned with the actual doing or use of something rather than with theory and ideas
2. (of an idea, plan, or method) Likely to succeed or be effective in real circumstances; feasible
3. Suitable for a particular purpose
Practical Mutation Analysis

Practical (adjective):

1. Of or concerned with the actual doing or use of something rather than with theory and ideas.
2. (of an idea, plan, or method) Likely to succeed or be effective in real circumstances; feasible.
3. Suitable for a particular purpose.

What are the practical techniques that MAJOR employs for improving the efficiency and usability of mutation analysis?
Conditional Mutation

Conditional Mutation
Conditional Mutation

Encapsulates all mutants within the same block
Conditional Mutation

Conditional Mutation

- Encapsulates all mutants within the same block
- Can be integrated within the compiler
Conditional Mutation

- Encapsulates all mutants within the same block
- Transforms the abstract syntax tree (AST)
- Can be integrated within the compiler
Conditional Mutation

- Encapsulates all mutants within the same block
- Transforms the abstract syntax tree (AST)
- Can be integrated within the compiler

Stmt → Conditional Stmt (if-then-else, switch)

Expr → Conditional Expr (conditional operator ?:)
Transforming the AST

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

    y = a * x;

    y += b;

    return y;
}
```
Transforming the AST

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

    y = a * x;
    y += b;
    return y;
}
```

ASSIGN
IDENT y
BINARY *
IDENT a
IDENT x

⇒

ASSIGN
IDENT y
BINARY *
IDENT a
IDENT x
Transforming the AST

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

ASSIGN
IDENT y
BINARY *
a x
Transforming the AST

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

```
ASSIGN
  IDENT y
  BINARY * a x
```

```
COND-EXPR
  THEN
  BINARY + a x
  COND (M_NO ==2)
  ELSE
  BINARY - a x
  COND (M_NO ==1)
  ELSE
  BINARY * a x
```
Source Code View of Inserting Mutants

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

1. Define mutation operators $MOP(x * y) = \{x - y, x + y, x/y\}$
2. Determine whether current expression or statement is affected by mutation
3. Apply mutation operators
Source Code View of Inserting Mutants

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

    y = a * x;

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

1. **Define mutation operators** $MOP(x \ast y) = \{x - y, x + y, x/y\}$
2. Determine whether current expression or statement is affected by mutation
3. Apply mutation operators
Source Code View of Inserting Mutants

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

    y = a * x;

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

1. Define mutation operators $MOP(x * y) = \{x - y, x + y, x/y\}$
2. Determine whether current expression or statement is affected by mutation
3. Apply mutation operators
**Source Code View of Inserting Mutants**

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

    y = (M_NO==1)? a - x: a * x;
    y += b;
    return y;
}
```

1. **Define mutation operators**  
   \( MOP(x \times y) = \{ x - y, x + y, x/y \} \)

2. **Determine whether current expression or statement is affected by mutation**

3. **Apply mutation operators**

---

Kapfhammer

Allegheny College

Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs
Mutation Analysis with MAJOR

Source Code View of Inserting Mutants

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

    y = (M_NO == 2) ? a + x :
       (M_NO == 1) ? a - x :
                  a * x;

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

1. Define mutation operators $MOP(x \ast y) = \{x - y, x + y, x/y \}$
2. Determine whether current expression or statement is affected by mutation
3. Apply mutation operators
Source Code View of Inserting Mutants

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

    y = (M_NO==3)? a / x :
        (M_NO==2)? a + x :
        (M_NO==1)? a - x :
                   a * x;

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

1. Define mutation operators \( MOP(x \times y) = \{ x - y, x + y, x/y \} \)
2. Determine whether current expression or statement is affected by mutation
3. Apply mutation operators
Source Code View of Inserting Mutants

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

    y = (M_NO == 3) ? a / x :
        (M_NO == 2) ? a + x :
        (M_NO == 1) ? a - x :
                     a * x;

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

1. Define mutation operators  \( MOP(x \ast y) = \{x - y, x + y, x/y\} \)
2. Determine whether current expression or statement is affected by mutation
3. Apply mutation operators

Mutants that are not executed cannot be killed
Collecting and Using Mutation Coverage

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

    y = (M_NO == 3) ? a / x : 
        (M_NO == 2) ? a + x : 
        (M_NO == 1) ? a - x : 
                      a * x;

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

Mutants that are not executed cannot be killed.
Collecting and Using Mutation Coverage

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

    y = (M_NO==3)? a / x :
       (M_NO==2)? a + x :
       (M_NO==1)? a - x :
       (M_NO==0 && COVERED(1,3))? a * x :
                        a * x ;

    y += b;

    return y;
}
Mutants that are not executed cannot be killed

Determine covered mutants with additional instrumentation

Only execute and investigate the covered mutants

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

    y = (M_NO==3)? a / x :
        (M_NO==2)? a + x :
        (M_NO==1)? a - x :
        (M_NO==0 && COVERED(1,3))?
            a * x :
            a * x;

    y += b;

    return y;
}
MAJOR’s Compiler
MAJOR’s Compiler

Enhanced Standard 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

Kapfhammer Allegheny College

Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs
MAJOR’s Compiler

- Common Compiler Options
- Domain Specific Language
- Source Files
- MAJOR’s Compiler
- Enhanced Standard Java Compiler
MAJOR’s Compiler

Common Compiler Options

Domain Specific Language

Source Files → MAJOR’s Compiler → Bytecode with Embedded Mutants

Enhanced Standard Java Compiler

Introduction

Mutation Analysis

Empirical Evaluation

Conclusion

Mutation Analysis with MAJOR

Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs
Integration into the Java Compiler
Integration into the Java Compiler

Kapfhammer
Allegheny College
Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs
Integration into the Java Compiler

Mutation Analysis with MAJOR

Kapfhammer Allegheny College

Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs
Integration into the 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

// 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">;
```
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">;
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
Compilation Efficiency

Mutant Generation and Compilation

Overhead for generating and compiling mutants is negligible
Mutation Analysis

Empirical Evaluation

Compilation Efficiency

Mutant Generation and Compilation

![Graph showing compiler runtime in seconds vs. number of mutants for different projects.](image)

Overhead for generating and compiling mutants is negligible

Kapfhammer Allegheny College

Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs
Time and Space Overhead

<table>
<thead>
<tr>
<th>Application</th>
<th>Mutants</th>
<th>Runtime of test suite</th>
<th>Memory consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>original</td>
<td>instrumented WCS</td>
</tr>
<tr>
<td>aspectj</td>
<td>406,382</td>
<td>4.3</td>
<td>4.8</td>
</tr>
<tr>
<td>apache ant</td>
<td>60,258</td>
<td>331.0</td>
<td>335.0</td>
</tr>
<tr>
<td>jfreechart</td>
<td>68,782</td>
<td>15.0</td>
<td>18.0</td>
</tr>
<tr>
<td>itext</td>
<td>124,184</td>
<td>5.1</td>
<td>5.6</td>
</tr>
<tr>
<td>java pathfinder</td>
<td>37,331</td>
<td>17.0</td>
<td>22.0</td>
</tr>
<tr>
<td>commons math</td>
<td>67,895</td>
<td>67.0</td>
<td>83.0</td>
</tr>
<tr>
<td>commons lang</td>
<td>25,783</td>
<td>10.3</td>
<td>11.8</td>
</tr>
<tr>
<td>numerics4j</td>
<td>5,869</td>
<td>1.2</td>
<td>1.3</td>
</tr>
</tbody>
</table>

- Runtime overhead is application dependent
  - Larger for CPU-bound applications
  - Small for I/O-bound applications
  - Even for large projects, applicable on commodity workstations
## Time and Space Overhead

<table>
<thead>
<tr>
<th>Application</th>
<th>Mutants</th>
<th>Runtime of test suite</th>
<th>Memory consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>original</td>
<td>instrumented</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wcs</td>
<td>wcs+cov</td>
</tr>
<tr>
<td>aspectj</td>
<td>406,382</td>
<td>4.3</td>
<td>4.8</td>
</tr>
<tr>
<td>apache ant</td>
<td>60,258</td>
<td>331.0</td>
<td>335.0</td>
</tr>
<tr>
<td>jfreechart</td>
<td>68,782</td>
<td>15.0</td>
<td>18.0</td>
</tr>
<tr>
<td>itext</td>
<td>124,184</td>
<td>5.1</td>
<td>5.6</td>
</tr>
<tr>
<td>java pathfinder</td>
<td>37,331</td>
<td>17.0</td>
<td>22.0</td>
</tr>
<tr>
<td>commons math</td>
<td>67,895</td>
<td>67.0</td>
<td>83.0</td>
</tr>
<tr>
<td>commons lang</td>
<td>25,783</td>
<td>10.3</td>
<td>11.8</td>
</tr>
<tr>
<td>numerics4j</td>
<td>5,869</td>
<td>1.2</td>
<td>1.3</td>
</tr>
</tbody>
</table>

- Runtime overhead is application dependent
  - Larger for CPU-bound applications
  - Small for I/O-bound applications
  - Even for large projects, applicable on commodity workstations
## Time and Space Overhead

<table>
<thead>
<tr>
<th>Application</th>
<th>Mutants</th>
<th>Runtime of test suite</th>
<th>Memory consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>original</td>
<td>instrumented wcs</td>
</tr>
<tr>
<td>aspectj</td>
<td>406,382</td>
<td>4.3</td>
<td>4.8</td>
</tr>
<tr>
<td>apache ant</td>
<td>60,258</td>
<td>331.0</td>
<td>335.0</td>
</tr>
<tr>
<td>jfreechart</td>
<td>68,782</td>
<td>15.0</td>
<td>18.0</td>
</tr>
<tr>
<td>itext</td>
<td>124,184</td>
<td>5.1</td>
<td>5.6</td>
</tr>
<tr>
<td>java pathfinder</td>
<td>37,331</td>
<td>17.0</td>
<td>22.0</td>
</tr>
<tr>
<td>commons math</td>
<td>67,895</td>
<td>67.0</td>
<td>83.0</td>
</tr>
<tr>
<td>commons lang</td>
<td>25,783</td>
<td>10.3</td>
<td>11.8</td>
</tr>
<tr>
<td>numerics4j</td>
<td>5,869</td>
<td>1.2</td>
<td>1.3</td>
</tr>
</tbody>
</table>

- Runtime overhead is application dependent
  - Larger for CPU-bound applications
  - Small for I/O-bound applications
- Even for large projects, applicable on commodity workstations
Time and Space Overhead

<table>
<thead>
<tr>
<th>Application</th>
<th>Mutants</th>
<th>Runtime of test suite</th>
<th>Memory consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>original</td>
<td>instrumented</td>
</tr>
<tr>
<td>wcs</td>
<td>wcs+cov</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aspectj</td>
<td>406,382</td>
<td>4.3</td>
<td>4.8</td>
</tr>
<tr>
<td>apache ant</td>
<td>60,258</td>
<td>331.0</td>
<td>335.0</td>
</tr>
<tr>
<td>jfreechart</td>
<td>68,782</td>
<td>15.0</td>
<td>18.0</td>
</tr>
<tr>
<td>itext</td>
<td>124,184</td>
<td>5.1</td>
<td>5.6</td>
</tr>
<tr>
<td>java pathfinder</td>
<td>37,331</td>
<td>17.0</td>
<td>22.0</td>
</tr>
<tr>
<td>commons math</td>
<td>67,895</td>
<td>67.0</td>
<td>83.0</td>
</tr>
<tr>
<td>commons lang</td>
<td>25,783</td>
<td>10.3</td>
<td>11.8</td>
</tr>
<tr>
<td>numerics4j</td>
<td>5,869</td>
<td>1.2</td>
<td>1.3</td>
</tr>
</tbody>
</table>

- Runtime overhead is application dependent
  - Larger for CPU-bound applications
  - Small for I/O-bound applications
- Even for large projects, applicable on commodity workstations
## Time and Space Overhead

<table>
<thead>
<tr>
<th>Application</th>
<th>Mutants</th>
<th>Runtime of test suite</th>
<th>Memory consumption</th>
<th>Application</th>
<th>Mutants</th>
<th>Runtime of test suite</th>
<th>Memory consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>original</td>
<td>instrumented</td>
<td></td>
<td></td>
<td>wcs</td>
<td>wcs+cov</td>
</tr>
<tr>
<td>aspectj</td>
<td>406,382</td>
<td>4.3</td>
<td>4.8</td>
<td>5.0</td>
<td>559</td>
<td>813</td>
<td></td>
</tr>
<tr>
<td>apache ant</td>
<td>60,258</td>
<td>331.0</td>
<td>335.0</td>
<td>346.0</td>
<td>237</td>
<td>293</td>
<td></td>
</tr>
<tr>
<td>jfreechart</td>
<td>68,782</td>
<td>15.0</td>
<td>18.0</td>
<td>23.0</td>
<td>220</td>
<td>303</td>
<td></td>
</tr>
<tr>
<td>itext</td>
<td>124,184</td>
<td>5.1</td>
<td>5.6</td>
<td>6.3</td>
<td>217</td>
<td>325</td>
<td></td>
</tr>
<tr>
<td>java pathfinder</td>
<td>37,331</td>
<td>17.0</td>
<td>22.0</td>
<td>29.0</td>
<td>182</td>
<td>217</td>
<td></td>
</tr>
<tr>
<td>commons math</td>
<td>67,895</td>
<td>67.0</td>
<td>83.0</td>
<td>98.0</td>
<td>153</td>
<td>225</td>
<td></td>
</tr>
<tr>
<td>commons lang</td>
<td>25,783</td>
<td>10.3</td>
<td>11.8</td>
<td>14.8</td>
<td>104</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>numerics4j</td>
<td>5,869</td>
<td>1.2</td>
<td>1.3</td>
<td>1.6</td>
<td>73</td>
<td>90</td>
<td></td>
</tr>
</tbody>
</table>

- Runtime overhead is application dependent
  - Larger for CPU-bound applications
  - Small for I/O-bound applications
  - Even for large projects, applicable on commodity workstations
Mutation Analysis Efficiency

Evaluating and Improving Mutation Analysis

- Mutation analysis is not feasible without coverage information
- Reordering the test suite significantly speeds up the process, especially if runtimes of tests differ by orders of magnitude

Kapfhammer
Allegheny College
Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs
Mutation analysis is not feasible without coverage information.

Reordering the test suite significantly speeds up the process, especially if runtimes of tests differ by orders of magnitude.
• Mutation analysis is not feasible without coverage information
• Reordering the test suite significantly speeds up the process, especially if runtimes of tests differ by orders of magnitude
# Revisiting Practical Mutation Analysis

## Practical (adjective):

1. Of or concerned with the actual doing or use of something rather than with theory and ideas
2. (of an idea, plan, or method) Likely to succeed or be effective in real circumstances; feasible
3. Suitable for a particular purpose
Revisiting Practical Mutation Analysis

Practical (adjective):

1. Of or concerned with the actual doing or use of something rather than with theory and ideas
2. (of an idea, plan, or method) Likely to succeed or be effective in real circumstances; feasible
3. Suitable for a particular purpose

The evidence suggests that MAJOR is “likely to succeed or be effective” in real-world software testing circumstances
Reviewing MAJOR’s Contributions

Mutation Analysis
Reviewing MAJOR’s Contributions

**Efficiency**: MAJOR has acceptable time and space overheads and scales to large, real-world programs.
Reviewing MAJOR’s Contributions

**Efficiency:** MAJOR has acceptable time and space overheads and scales to large, real-world programs

**Usability:** MAJOR’s integration into the Java SE compiler makes it a no-hassle, drop-in tool
Reviewing MAJOR’s Contributions

**Efficiency:** MAJOR has acceptable time and space overheads and scales to large, real-world programs

**Usability:** MAJOR’s integration into the Java SE compiler makes it a no-hassle, drop-in tool

We will release MAJOR as free and open source software
Conclusion

Key Concepts and Features:

- Compiler-integrated solution
- Conditional mutation with the abstract syntax tree
- Furnishes its own domain specific language
- Collects and leverages mutation coverage information
Conclusion

Key Concepts and Features:

- Compiler-integrated solution
- Conditional mutation with the abstract syntax tree
- Furnishes its own domain specific language
- Collects and leverages mutation coverage information

Characteristics of MAJOR:

- Fast and scalable technique
- Configurable and extensible mutation tool
- Enables an optimized workflow for mutation analysis
Practical Techniques for Improving the Efficiency and Usability of Mutation Analysis for Java Programs

Gregory M. Kapfhammer

Department of Computer Science
Allegheny College
http://www.cs.allegheny.edu/~gkapfham/

Thank you for your attention!
I welcome your questions and comments.