Using Non-Redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis

René Just¹,² & Gregory M. Kapfhammer³ & Franz Schweiggert²

¹University of Washington, USA
²Ulm University, Germany
³Allegheny College, USA

23rd International Symposium on Software Reliability Engineering

November 28, 2012
Mutation Analysis Background

Mutation analysis assesses the quality of a test suite with artificial faults (mutants)
Mutation Analysis Background

Mutation analysis assesses the quality of a test suite with artificial faults (mutants)
Mutation Analysis Background

Mutation analysis assesses the quality of a test suite with artificial faults (mutants)

Diagram:
- Program
- Generate mutants
- Mutants
- Test suite
Mutation Analysis Background

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

Mutation analysis assesses the quality of a test suite with artificial faults (mutants)

Program

Test suite

Generate mutants

Mutants
Mutation Analysis Background

Mutation analysis assesses the quality of a test suite with artificial faults (mutants)

Program

Test suite

Generate mutants

Mutants

public int max(int a, int b){
    int max = a;
    if (b>a){
        max=b;
    }
    return max;
}
Mutation Analysis Background

Mutation analysis assesses the quality of a test suite with artificial faults (mutants).

**Program**

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

**Test suite**

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

**Generate mutants**

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

**Mutants**

Contains a small syntactic change
Mutation Analysis Background

Mutation analysis assesses the quality of a test suite with artificial faults (mutants)

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

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

Program → Test suite → Generate mutants → Execute mutants → Mutation score

Mutation score contains a small syntactic change
Mutation Analysis is Expensive

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

Original
Mutation Analysis is Expensive

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

Original

```
if (b < a)
if (b <= a)
if (b >= a)
if (b != a)
if (b == a)
```
Mutation Analysis is Expensive

Many mutants can be generated for large programs
Mutation Analysis is Expensive

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

Large programs include comprehensive test suites

Many mutants can be generated for large programs

Original

if (b < a)
if (b <= a)
if (b >= a)
if (b != a)
if (b == a)
Mutation Analysis is Expensive

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

Original

- Many mutants can be generated for large programs
- Large programs include comprehensive test suites
- Executing the entire test suite for all mutants in large programs is prohibitive!
### Overview: Efficient Mutation Analysis

Execute fewer mutants fewer times
Overview: Efficient Mutation Analysis

Execute fewer mutants fewer times

Mutant reduction

Generate fewer mutants

Execute fewer mutants
Overview: Efficient Mutation Analysis

- **Mutant reduction**
  - Generate fewer mutants
  - Execute fewer mutants

- **Test suite prioritization**
  - Test suite characteristics
  - Reordering and splitting

Execute fewer mutants fewer times
Overview: Efficient Mutation Analysis

Execute fewer mutants fewer times

Mutant reduction
- Generate fewer mutants
- Execute fewer mutants
- 27%

Test suite prioritization
- Test suite characteristics
- Reordering and splitting
- 29%

Empirical evaluation of 10 open-source projects with 560,000 mutants
Reduction of Mutants

Execute fewer mutants fewer times

Mutant reduction

Generate fewer mutants

Execute fewer mutants

27%

Empirical evaluation of 10 open-source projects with 560,000 mutants
Reduce Number of Generated Mutants

**Mutation operators may introduce redundancy:**
- Redundant mutants are subsumed by other mutants
  - $a + b \mapsto a - b$ (replace binary operator)
  - $a + b \mapsto a + (-b)$ (insert unary operator)
- Use only non-redundant mutation operators
  - Avoid the generation of such subsumed mutants
Reduce Number of Generated Mutants

**Mutation operators may introduce redundancy:**

- Redundant mutants are subsumed by other mutants
  - \( a + b \mapsto a - b \) (replace binary operator)
  - \( a + b \mapsto a + (-b) \) (insert unary operator)
- Use only non-redundant mutation operators
  - Avoid the generation of such subsumed mutants

Number of generated mutants reduced by 27%
Reduce Number of Generated Mutants

**Mutation operators may introduce redundancy:**

- Redundant mutants are subsumed by other mutants
  - \( a + b \mapsto a - b \) (replace binary operator)
  - \( a + b \mapsto a + (-b) \) (insert unary operator)
- Use only non-redundant mutation operators
- Avoid the generation of such subsumed mutants

Number of generated mutants reduced by 27%

More than 410,000 generated mutants remaining
Reduce Number of Generated Mutants

Mutation operators may introduce redundancy:
- Redundant mutants are subsumed by other mutants
  - \(a + b \mapsto a - b\) (replace binary operator)
  - \(a + b \mapsto a + (-b)\) (insert unary operator)
- Use only non-redundant mutation operators
  - Avoid the generation of such subsumed mutants

Number of generated mutants reduced by 27%
More than 410,000 generated mutants remaining

Executing all non-redundant mutants is still prohibitive!
Reduce Number of Executed Mutants

Exploit necessary conditions:

- Mutants not covered (reached) cannot be detected
- Determine covered mutants for the test suite
- Only execute the covered mutants
Reduce Number of Executed Mutants

Exploit necessary conditions:
- Mutants not covered (reached) cannot be detected
- Determine covered mutants for the test suite
- Only execute the covered mutants

Total reduction of executed mutants of more than 50%
Reduce Number of Executed Mutants

Exploit necessary conditions:
- Mutants not covered (reached) cannot be detected
- Determine covered mutants for the test suite
- Only execute the covered mutants

Total reduction of executed mutants of more than 50%
Mutation analysis runtime still up to 13 hours
Reduce Number of Executed Mutants

Exploit necessary conditions:
- Mutants not covered (reached) cannot be detected
- Determine covered mutants for the test suite
- Only execute the covered mutants

Total reduction of executed mutants of more than 50%  
Mutation analysis runtime still up to 13 hours

Further optimizations beyond the reduction of mutants are necessary!
Optimized Workflow for Mutation Analysis

Execute fewer mutants fewer times

Test suite prioritization

Test suite characteristics
Reordering and splitting

Empirical evaluation of 10 open-source projects with 560,000 mutants

29%
Motivating Example for Reordering

Mutants:
1, 2, 3, 4, 5
Motivating Example for Reordering

Mutants: 1, 2, 3, 4, 5

Test case $t_1$: 5 seconds

Test case $t_2$: 2 seconds

Test case $t_3$: 1 second
Motivating Example for Reordering

<table>
<thead>
<tr>
<th>Mutants:</th>
<th>Test case ( t_1 ): 5 seconds</th>
<th>Test case ( t_2 ): 2 seconds</th>
<th>Test case ( t_3 ): 1 second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4, 5</td>
<td>1, 2, 3, 4, 5</td>
<td>1, 3, 4, 5</td>
<td>1, 2, 3</td>
</tr>
</tbody>
</table>

Once a mutant is detected, it is not executed again!
## Motivating Example for Reordering

<table>
<thead>
<tr>
<th>Mutants: 1, 2, 3, 4, 5</th>
<th>Test case $t_1$: 5 seconds</th>
<th>Test case $t_2$: 2 seconds</th>
<th>Test case $t_3$: 1 second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Covered: 1, 2, 3, 4, 5</td>
<td></td>
<td>1, 3, 4, 5</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>Detected: 1, 2, 5</td>
<td></td>
<td>1, 4</td>
<td>3</td>
</tr>
</tbody>
</table>

Once a mutant is detected, it is not executed again!
Motivating Example for Reordering

<table>
<thead>
<tr>
<th>Mutants:</th>
<th>Test case $t_1$: 5 seconds</th>
<th>Test case $t_2$: 2 seconds</th>
<th>Test case $t_3$: 1 second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4, 5</td>
<td>1, 2, 3, 4, 5</td>
<td>1, 3, 4, 5</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>Covered:</td>
<td>1, 2, 5</td>
<td>1, 4</td>
<td>3</td>
</tr>
</tbody>
</table>

- Once a mutant is detected, it is not executed again!

**Executed mutants and total runtime:**

$t_1, t_2, t_3$
### Motivating Example for Reordering

<table>
<thead>
<tr>
<th>Mutants:</th>
<th>Test case $t_1$: 5 seconds</th>
<th>Test case $t_2$: 2 seconds</th>
<th>Test case $t_3$: 1 second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4, 5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Covered:</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4, 5</td>
<td></td>
<td>1, 3, 4, 5</td>
<td>1, 2, 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Detected:</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 5</td>
<td></td>
<td>1, 4</td>
<td>3</td>
</tr>
</tbody>
</table>

- Once a mutant is detected, it is not executed again!

**Executed mutants and total runtime:**

$t_1$  | $t_2$  | $t_3$  |
---|---|---|
1   | 2   | 3   |
4   | 5   |     |
## Motivating Example for Reordering

<table>
<thead>
<tr>
<th>Mutants:</th>
<th>Test case $t_1$: 5 seconds</th>
<th>Test case $t_2$: 2 seconds</th>
<th>Test case $t_3$: 1 second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4, 5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Covered:</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4, 5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Detected:</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Once a mutant is detected, it is not executed again!

### Executed mutants and total runtime:

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
<th>$t_5$</th>
<th>$t_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>
Motivating Example for Reordering

<table>
<thead>
<tr>
<th>Mutants: 1, 2, 3, 4, 5</th>
<th>Test case $t_1$: 5 seconds</th>
<th>Test case $t_2$: 2 seconds</th>
<th>Test case $t_3$: 1 second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Covered: 1, 2, 3, 4, 5</td>
<td>Detected: 1, 2, 5</td>
<td>Detected: 1, 4</td>
<td>Detected: 3</td>
</tr>
</tbody>
</table>

Once a mutant is detected, it is not executed again!

**Executed mutants and total runtime:**

| $t_1$ | $t_2$ | $t_3$ | 1 | 2 | 3 | 4 | 5 | 3 | 4 | 3 |
Motivating Example for Reordering

Mutants: 1, 2, 3, 4, 5

Test case $t_1$: 5 seconds

Test case $t_2$: 2 seconds

Test case $t_3$: 1 second

Covered: 1, 2, 3, 4, 5

1, 3, 4, 5

1, 2, 3

Detected: 1, 2, 5

1, 4

3

Once a mutant is detected, it is not executed again!

Executed mutants and total runtime:

$t_1 t_2 t_3$ : 1 2 3 4 5 3 4 3

$t_3 t_2 t_1$ :
Motivating Example for Reordering

Mutants: 1, 2, 3, 4, 5

Test case $t_1$: 5 seconds

Test case $t_2$: 2 seconds

Test case $t_3$: 1 second

Covered: 1, 2, 3, 4, 5

Detected: 1, 2, 5

1, 4

3

Once a mutant is detected, it is not executed again!

Executed mutants and total runtime:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>3</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1 t_2 t_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_3 t_2 t_1$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Motivating Example for Reordering

<table>
<thead>
<tr>
<th>Mutants:</th>
<th>Test case $t_1$:</th>
<th>Test case $t_2$:</th>
<th>Test case $t_3$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4, 5</td>
<td>5 seconds</td>
<td>2 seconds</td>
<td>1 second</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Covered:</th>
<th>Detected:</th>
<th>Executed mutants and total runtime:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4, 5</td>
<td>1, 2, 5</td>
<td>$t_1</td>
</tr>
<tr>
<td>1, 3, 4, 5</td>
<td>1, 4</td>
<td>1</td>
</tr>
<tr>
<td>1, 2, 3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Once a mutant is detected, it is not executed again!
Motivating Example for Reordering

Mutants: 1, 2, 3, 4, 5

Test case $t_1$: 5 seconds
Test case $t_2$: 2 seconds
Test case $t_3$: 1 second

Covered: 1, 2, 3, 4, 5
1, 3, 4, 5
1, 2, 3

Detected: 1, 2, 5
1, 4
3

Once a mutant is detected, it is not executed again!

Executed mutants and total runtime:

$t_1\, t_2\, t_3$:
\[
\begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 3 & 4 & 3 \\
\end{array}
\]

$t_3\, t_2\, t_1$:
\[
\begin{array}{ccccccc}
1 & 2 & 3 & 1 & 4 & 5 & 2 & 5 \\
\end{array}
\]
# Motivating Example for Splitting

<table>
<thead>
<tr>
<th>Mutants:</th>
<th>Test case $t_1$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4, 5</td>
<td>5 seconds</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Covered:</th>
<th>Detected:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4, 5</td>
<td>1, 2, 5</td>
</tr>
</tbody>
</table>

Once a mutant is detected, it is not executed again!
Motivating Example for Splitting

<table>
<thead>
<tr>
<th>Mutants:</th>
<th>Test case $t_1$: 5 seconds</th>
<th>Test case $t'_1$: 3 seconds</th>
<th>Test case $t''_1$: 2 seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4, 5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Covered:</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4, 5</td>
<td>1, 2, 3, 4</td>
<td>2, 3, 4, 5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Detected:</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 5</td>
<td>1, 2</td>
<td>2, 5</td>
<td></td>
</tr>
</tbody>
</table>

Once a mutant is detected, it is not executed again!
Motivating Example for Splitting

<table>
<thead>
<tr>
<th>Mutants: 1, 2, 3, 4, 5</th>
<th>Test case $t_1$: 5 seconds</th>
<th>Test case $t'_1$: 3 seconds</th>
<th>Test case $t''_1$: 2 seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Covered: 1, 2, 3, 4, 5</td>
<td>1, 2, 3, 4</td>
<td>2, 3, 4, 5</td>
<td></td>
</tr>
<tr>
<td>Detected: 1, 2, 5</td>
<td>1, 2</td>
<td>2, 5</td>
<td></td>
</tr>
</tbody>
</table>

- Once a mutant is detected, it is not executed again!

**Executed mutants and total runtime:**

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
</table>
## Motivating Example for Splitting

**Mutants:** 1, 2, 3, 4, 5

**Test case \( t_1 \):** 5 seconds

**Test case \( t'_1 \):** 3 seconds

**Test case \( t''_1 \):** 2 seconds

**Covered:** 1, 2, 3, 4, 5

**Detected:** 1, 2, 5

1, 2

2, 5

- Once a mutant is detected, it is not executed again!

**Executed mutants and total runtime:**

\[
t_1 : \begin{array}{c} 1 \end{array} \begin{array}{c} 2 \end{array} \begin{array}{c} 3 \end{array} \begin{array}{c} 4 \end{array} \begin{array}{c} 5 \end{array}
\]

\[
t'_1 \quad t''_1 : \]

Motivating Example for Splitting

Mutants: 1, 2, 3, 4, 5

Test case $t_1$: 5 seconds
Test case $t'_1$: 3 seconds
Test case $t''_1$: 2 seconds

Covered: 1, 2, 3, 4, 5
1, 2, 3, 4
2, 3, 4, 5

Detected: 1, 2, 5
1, 2
2, 5

Once a mutant is detected, it is not executed again!

Executed mutants and total runtime:

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t'_1$</th>
<th>$t''_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Motivating Example for Splitting

<table>
<thead>
<tr>
<th>Mutants:</th>
<th>Test case ( t_1 ): 5 seconds</th>
<th>Test case ( t'_1 ): 3 seconds</th>
<th>Test case ( t''_1 ): 2 seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4, 5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Covered:</th>
<th>Detected:</th>
<th>Executed mutants and total runtime:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4, 5</td>
<td>1, 2, 5</td>
<td>( t_1 ): 1 2 3 4 5</td>
</tr>
<tr>
<td>1, 2, 3, 4</td>
<td>1, 2</td>
<td>( t'_1 ): 1 2 3 4 3 4 5</td>
</tr>
<tr>
<td>2, 3, 4, 5</td>
<td>2, 5</td>
<td>( t''_1 ): 1 2 3 4 3 4 5</td>
</tr>
</tbody>
</table>

Once a mutant is detected, it is not executed again!
Runtime Distribution of Tests within Test Suites

The diagram shows the test runtime distribution for various test suites. The x-axis represents the test suites, and the y-axis represents the test runtime in seconds. The data points indicate the runtime distribution across different test suites, with some tests taking significantly longer to run than others.
Runtime Distribution of Tests within Test Suites

Most tests have short runtime
Runtime Distribution of Tests within Test Suites

Most tests have short runtime

A few long-running outliers
Runtime Distribution of Tests within Test Suites

A few tests constitute most of the total runtime: Reduce number of executions for these tests.
## Mutation Coverage Overlap

- Overlap measures the similarity of a test case with its enclosing test suite
- Pair-wise comparison of test cases is infeasible
Mutation Coverage Overlap

- Overlap measures the similarity of a test case with its enclosing test suite
- Pair-wise comparison of test cases is infeasible

**Definition:** \( \text{Overlap } O(t_i, T), \ t_i \in T \)

\[
O(t_i, T) := \begin{cases} 
1, & |Cov(t_i)| = 0 \\
\frac{|Cov(t_i) \cap Cov(T \setminus t_i)|}{|Cov(t_i)|}, & |Cov(t_i)| > 0 
\end{cases}
\]
Mutation Coverage Overlap

- Overlap measures the similarity of a test case with its enclosing test suite
- Pair-wise comparison of test cases is infeasible

**Definition:** 
Overlap $O(t_i, T)$, $t_i \in T$

$$O(t_i, T) := \begin{cases} 1, & |Cov(t_i)| = 0 \\ \frac{|Cov(t_i) \cap Cov(T \setminus t_i)|}{|Cov(t_i)|}, & |Cov(t_i)| > 0 \end{cases}$$

Most of the test cases exhibit high overlap:
Does test runtime correlate with overlap?
Correlation of Test Runtime and Mutation Coverage
Correlation of Test Runtime and Mutation Coverage

Test case with longest runtime
Correlation of Test Runtime and Mutation Coverage

- Overlapping test cases
- Test case with longest runtime
Correlation of Test Runtime and Mutation Coverage

Reorder to exploit mutation coverage overlap
Correlation of Test Runtime and Mutation Coverage

- Large mutation coverage
- Overlapping test cases
- Test case with longest runtime

Reorder to exploit mutation coverage overlap
Correlation of Test Runtime and Mutation Coverage

- Large mutation coverage
- Overlapping test cases
- Test case with longest runtime

Reorder to exploit mutation coverage overlap
Split test cases to increase coverage precision
Mutation Coverage of Test suites

test suite

class\textsubscript{#1} \quad \ldots \quad \ldots \quad class\textsubscript{#m}

method\textsubscript{#1} \quad \ldots \quad \ldots \quad method\textsubscript{#n}
Mutation Coverage of Test suites

- Test suite
  - class#1
    - method#1
  - ...
  - ...
  - class#m
    - method#n

Higher precision
Mutation Coverage of Test suites
Mutation Coverage of Test suites

Only split long-running test classes
Splitting Test Classes

Two splitting strategies
## Splitting Test Classes

### Two splitting strategies

- **Split entire long-running test class**
  - High overhead and coverage precision
Splitting Test Classes

Two splitting strategies

- **Split entire long-running test class**
  - High overhead and coverage precision

- **Extract only long-running test methods**
  - Lower overhead and coverage precision
Splitting Test Classes

Two splitting strategies

- Split entire long-running test class
  - High overhead and coverage precision

- Extract only long-running test methods
  - Lower overhead and coverage precision

Trade-off between overhead and precision: Splitting based on threshold for test runtime
Optimized workflow

1. Original program
2. Generate mutants
3. Set of non-redundant mutants
Optimized workflow

Original program

Generate mutants

Set of non-redundant mutants

Original test suite

Execute test suite
Optimized workflow

1. Original program
2. Generate mutants
3. Set of non-redundant mutants
4. Original test suite
5. Execute test suite
6. Runtime of test cases
7. Mutation coverage
Optimized workflow

Original program

Generate mutants

Set of non-redundant mutants

Execute test suite

Original test suite

Runtime of test cases

Order/split test cases

Mutation coverage

Prioritized test suite
Optimized workflow

1. **Original program**
2. **Generate mutants**
3. **Execute test suite**
4. **Runtime of test cases**
5. **Order/split test cases**
6. **Prioritized test suite**
7. **Set of non-redundant mutants**
8. **Mutation coverage**
9. **Mutation analysis**
Example with Original Test Suite
Example with Original Test Suite

Original test suite

Total runtime of test executing all covered, yet not killed, mutants
Example with Original Test Suite

Total runtime of test executing all covered, yet not killed, mutants
Example with Original Test Suite

Original test suite

Reorder

Total runtime of test executing all covered, yet not killed, mutants

800
Example with Original Test Suite

Split

Original test suite

Reorder

Total runtime of test executing all covered, yet not killed, mutants
Example with Prioritized Test Suite
Empirical Results

Reordering:
- Reordering decreases the runtime by 20%

Splitting strategies:
- Extracting long test methods reduces the runtime by 29%
- Splitting entire test classes increases the runtime by 27%

Splitting may increase runtime if:
- Test suite has a very low mutation detection rate
- Test methods exhibit huge mutation coverage overlap
Empirical Results

**Reordering:**
- Reordering decreases the runtime by 20%

**Splitting strategies:**
- Extracting long test methods reduces the runtime by 29%
- Splitting entire test classes increases the runtime by 27%

**Splitting may increase runtime if:**
- Test suite has a very low mutation detection rate
- Test methods exhibit huge mutation coverage overlap

Prioritizing test suites improves the efficiency of mutation analysis by 29% on average!
Related Work

**Reduction of generated mutants:**
- Sufficient mutation operators
  - Offutt et al., TOSEM’96
  - Namin et al., ICSE’08
- Non-redundant mutation operators
  - Kaminski et al., AST’11
  - Just et al., Mutation’12

**Mutation-based test suite optimization:**
- Test case prioritization
  - Elbaum et al. TSE’02
  - Do and Rothermel, TSE’06
## Related Work

### Reduction of generated mutants:
- Sufficient mutation operators
  - Offutt et al., TOSEM’96
  - Namin et al., ICSE’08
- Non-redundant mutation operators
  - Kaminski et al., AST’11
  - Just et al., Mutation’12

### Mutation-based test suite optimization:
- Test case prioritization
  - Elbaum et al. TSE’02
  - Do and Rothermel, TSE’06

Still contain redundancies
Related Work

**Reduction of generated mutants:**
- Sufficient mutation operators
  - Offutt et al., TOSEM’96
  - Namin et al., ICSE’08
- Non-redundant mutation operators
  - Kaminski et al., AST’11
  - Just et al., Mutation’12

**Mutation-based test suite optimization:**
- Test case prioritization
  - Elbaum et al. TSE’02
  - Do and Rothermel, TSE’06

Still contain redundancies
Used in empirical study
## Related Work

### Reduction of generated mutants:
- **Sufficient mutation operators**
  - Offutt et al., TOSEM’96
  - Namin et al., ICSE’08
- **Non-redundant mutation operators**
  - Kaminski et al., AST’11
  - Just et al., Mutation’12

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Still contain redundancies</td>
<td></td>
</tr>
<tr>
<td>Used in empirical study</td>
<td></td>
</tr>
<tr>
<td>Do not address efficiency</td>
<td></td>
</tr>
</tbody>
</table>

### Mutation-based test suite optimization:
- **Test case prioritization**
  - Elbaum et al. TSE’02
  - Do and Rothermel, TSE’06
Conclusions

**Reduction of mutants:**
- Non-redundant operators reduce number of mutants by 27%

**Test suite characteristics:**
- Most of the tests exhibit mutation coverage overlap
- Notable difference in runtime of tests

**Optimized workflow:**
- Exploits mutation coverage overlap and runtime differences
- Further reduces total runtime of mutation analysis by 29%
Conclusions

**Reduction of mutants:**
- Non-redundant operators reduce number of mutants by 27%

**Test suite characteristics:**
- Most of the tests exhibit mutation coverage overlap
- Notable difference in runtime of tests

**Optimized workflow:**
- Exploits mutation coverage overlap and runtime differences
- Further reduces total runtime of mutation analysis by 29%

Non-redundant operators and optimized workflow implemented in the MAJOR mutation system