Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States

René Just\textsuperscript{1} & Michael D. Ernst\textsuperscript{1} & Gordon Fraser\textsuperscript{2}

\textsuperscript{1}University of Washington, USA
\textsuperscript{2}University of Sheffield, UK

July 25, 2014
**A rough outline**

**Motivation**

What is mutation analysis — is it useful?
A rough outline

Motivation

What is mutation analysis — is it useful?

Problem

Mutation analysis is expensive!
A rough outline

Motivation
What is mutation analysis — is it useful?

Problem
Mutation analysis is expensive!

Solution
Dynamic prepass analysis to make mutation practical!
Test suite quality

Why assess test suite quality?

- Selection: Given two test suites, which is better?
- Minimization: Are there redundant tests in a test suite?
- Prioritization: Which tests of a test suite should run first?
Test suite quality

Why assess test suite quality?
- Selection: Given two test suites, which is better?
- Minimization: Are there redundant tests in a test suite?
- Prioritization: Which tests of a test suite should run first?

How to assess test suite quality?
- A good test suite detects real faults
Test suite quality

Why assess test suite quality?

- Selection: Given two test suites, which is better?
- Minimization: Are there redundant tests in a test suite?
- Prioritization: Which tests of a test suite should run first?

How to assess test suite quality?

- A **good** test suite detects **real faults**
- **Problem:** Real faults in a program are unknown
- **Solution:** Seed **artificial faults** into the program
Test suite quality

Why assess test suite quality?
- Selection: Given two test suites, which is better?
- Minimization: Are there redundant tests in a test suite?
- Prioritization: Which tests of a test suite should run first?

How to assess test suite quality?
- A **good** test suite detects **real faults**
- **Problem**: Real faults in a program are unknown
- **Solution**: Seed **artificial faults** into the program

**Mutation analysis**: systematically seed artificial faults
Mutation analysis overview

- Program
- Test suite
Mutation analysis overview

Program → Test suite

Generate mutants → Mutants
Mutation analysis overview

```
public int max(int a, int b)
{
    return (a > b) ? a : b;
}
```

Original

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

Mutant 1

```
public int max(int a, int b)
{
    return (a != b) ? a : b;
}
```

Mutant 2

Each mutant contains a small syntactic change
Mutation analysis overview

Program

Generate mutants

Mutants

Test suite

Execute test suite

Mutation score

Ratio of detected mutants
Where is the catch?

Many mutants can be generated!

```plaintext
a > b ? a : b
```
Where is the catch?

Many mutants can be generated!

```plaintext
a > b ? a : b
a >= b ? a : b
a < b ? a : b
a <= b ? a : b
a != b ? a : b
a == b ? a : b
!(a > b) ? a : b
true ? a : b
false ? a : b
```
Where is the catch?

Many mutants can be generated!

\[
a > b \ ? \ a : b \\
a \geq b \ ? \ a : b \\
a < b \ ? \ a : b \\
a \leq b \ ? \ a : b \\
a \neq b \ ? \ a : b \\
a = b \ ? \ a : b \\
!(a > b) \ ? \ a : b \\
true \ ? \ a : b \\
false \ ? \ a : b
\]

\[
0 > b \ ? \ a : b \\
0 > 0 \ ? \ a : b \\
a > 0 \ ? \ a : b \\
a >-b \ ? \ a : b \\
a > b ? 0 : b \\
a > b ? a : 0 \\
a > b ? -a : b \\
a > b ? a : -b \\
a > b ? a : -b \\
a > b ? -a : b \\
a > b ? a : b \\
-(a > b ? a : b) \\
a > b ? b : a \\
a > b ? a : b \\
-(a > b ? a : b) \\
a \\
b
\]
Mutation analysis overview

**Program**

Generate mutants

Mutants

Test suite

Problem
Test suite has to be executed for many mutants

Execute test suite

Mutation score
Mutation analysis overview

Program

Generate mutants

Test suite

Problem
Test suite has to be executed for many mutants

Filter mutants

Mutants

Execute test suite

Mutation score
Mutation analysis overview

Program → Generate mutants → Mutants

Test suite → Execute test suite → Mutation score

Related work: Mutant sampling (Jia and Harman, TSE’11)

Problem
Test suite has to be executed for many mutants
Mutation analysis overview

Program

Generate mutants

Mutants

Test suite

Execute test suite

Related work:
Selective mutation
(Namin et al., ICSE’08)

Problem
Test suite has to be executed for many mutants

Filter mutants

Mutants

Mutation score
Mutation analysis overview

- Program
  - Generate mutants
- Test suite
- Related work: Code coverage analysis (Major, Javalanche, PIT, ...)
- Problem: Test suite has to be executed for many mutants

- Mutants
  - Filter mutants
  - Execute test suite
  - Mutation score

- Generate mutants
  - Program
  - Test suite
Mutation analysis overview

![Diagram of mutation analysis process]

- **Related work:**
  - Test suite prioritization
    - (Just et al., ISSRE’12)
    - (Zhang et al., ISSTA’13)

**Problem**
- Test suite has to be executed for many mutants
Mutation analysis overview

Program

Generate mutants

Test suite

Problem
Test suite has to be executed for many mutants

Our solution
Filter mutants with a dynamic analysis

Filter mutants

Execute test suite

Mutation score
Wasted effort in mutation analysis

Example: testing triangle classification

```java
public TriangleType classify
    (int a, int b, int c) {  
    ...
    if ( a + b <= c ) {
        return Invalid;
    }
    ...
}  
```

Original
Wasted effort in mutation analysis

Example: testing triangle classification

```java
public TriangleType classify
    (int a, int b, int c) {
    ...
    if (a + b <= c) {
        return Invalid;
    }
    ...
}
```

Original
Wasted effort in mutation analysis

Example: testing triangle classification

Original

```java
public TriangleType classify(int a, int b, int c) {
    ...
    if (a + b <= c) {
        return Invalid;
    }
    ...
}
```

Mutant 1

```java
public TriangleType classify(int a, int b, int c) {
    ...
    if (a * b <= c) {
        return Invalid;
    }
    ...
}
```
Wasted effort in mutation analysis

Example: testing triangle classification

```java
public TriangleType classify(int a, int b, int c) {
    ...
    if (a + b <= c) {
        return Invalid;
    }
    ...
}
```

```
public TriangleType classify(int a, int b, int c) {
    ...
    if (a * b <= c) {
        return Invalid;
    }
    ...
}
```

Identical code surrounding mutation
Wasted effort in mutation analysis

Example: testing triangle classification

```
public TriangleType classify
(int a, int b, int c) {
    ...
    if (a + b <= c) {
        return Invalid;
    }
    ...
}
```

```
public TriangleType classify
(int a, int b, int c) {
    ...
    if (a * b <= c) {
        return Invalid;
    }
    ...
}
```

Optimizations:
- Infection
Wasted effort in mutation analysis

Example: testing triangle classification

```java
public TriangleType classify(int a, int b, int c) {
    if (a + b <= c) {
        return Invalid;
    }
    ...
}
```

Original

```java
public TriangleType classify(int a, int b, int c) {
    if (a * b <= c) {
        return Invalid;
    }
    ...
}
```

Mutant 1

Optimizations:

- Infection
- Propagation
Wasted effort in mutation analysis

Example: testing triangle classification

```
public TriangleType classify (int a, int b, int c) {
    ...
    if (a + b <= c) {
        return Invalid;
    }
    ...
}
```

```
public TriangleType classify (int a, int b, int c) {
    ...
    if (a * b <= c) {
        return Invalid;
    }
    ...
}
```

```
public TriangleType classify (int a, int b, int c) {
    ...
    if (a - b <= c) {
        return Invalid;
    }
    ...
}
```

Optimizations:

- Infection
- Propagation
Wasted effort in mutation analysis

Example: testing triangle classification

```java
public TriangleType classify(int a, int b, int c) {
    ...
    if (a + b <= c) {
        return Invalid;
    }
    ...
}
```

Original

```java
public TriangleType classify(int a, int b, int c) {
    ...
    if (a * b <= c) {
        return Invalid;
    }
    ...
}
```

Mutant 1

```java
public TriangleType classify(int a, int b, int c) {
    ...
    if (a - b <= c) {
        return Invalid;
    }
    ...
}
```

Mutant 2

Optimizations:

- Infection
- Propagation
- Partitioning
The big picture

Set of mutants

Execute test suite on 100% of mutants
The big picture

Set of mutants

Mutation score: 56%

Undetectable mutants

Detectable mutants

Execute test suite on 100% of mutants
The big picture

Set of mutants

Infection

Execute test suite on 90% of mutants
The big picture

- Infection
- Propagation

Set of mutants

Execute test suite on 84% of mutants
The big picture

- Infection
- Propagation
- Partitioning

Set of mutants

Execute test suite on 70% of mutants
Dynamic analysis to filter mutants
Infection

A test infects the execution state of a mutant if the expression values of the mutation and the original version differ.
Infection

A test infects the execution state of a mutant if the expression values of the mutation and the original version differ.

Example: $a=2, b=2, c=0$

$$\text{if} \ (a + b > c)$$
Infection

A test infects the execution state of a mutant if the expression values of the mutation and the original version differ.

Example: \( a=2, \ b=2, \ c=0 \)

\[
\text{if} \ (a + b > c)
\]

\[
\begin{align*}
& a \times b \\
& a \div b \\
& a \mod b \\
& a - b
\end{align*}
\]
**Infection**

A test infects the execution state of a mutant if the expression values of the mutation and the original version differ.

**Example:** a=2, b=2, c=0

\[
\text{if ( } a + b > c \text{ )}
\]

\[
\begin{align*}
&\text{a \ast b} & &\text{a \div b} & &\text{a \% b} & &\text{a - b} \\
&4 & &1 & &0 & &0
\end{align*}
\]

**Optimization**

- Execute mutations and monitor infected execution states
- Filter mutants whose execution state is not infected
Infection

A test infects the execution state of a mutant if the expression values of the mutation and the original version differ.

Example: \( a=2, \ b=2, \ c=0 \)

```
if ( a + b > c )
```

- \( a \times b \) = 4
- \( a / b \) = 1
- \( a \% b \) = 0
- \( a - b \) = 0

Optimization

- Execute mutations and monitor infected execution states
- Filter mutants whose execution state is not infected
An infected execution state propagates if it leads to an infected execution state of a lexically enclosing expression.

**Example:** a=2, b=2, c=0

```plaintext
if ( a + b > c )
```

4

```plaintext
4
a * b
```

1

```plaintext
a / b
```

0

```plaintext
a % b
```

0

```plaintext
a - b
```

René Just, UW CSE

Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States
An infected execution state propagates if it leads to an infected execution state of a lexically enclosing expression.

**Example:** $a=2$, $b=2$, $c=0$

```
true
if ( (a + b) > c )
```

```
4
a * b
```

```
1
a / b
```

```
0
a % b
```

```
0
a - b
```
**Propagation**

An infected execution state propagates if it leads to an infected execution state of a lexically enclosing expression.

**Example:** \(a=2, \ b=2, \ c=0\)

- \(\text{true}\) if \((a + b > c)\)
- \(a \ast b\) evaluates to 4
- \(a / b > c\) evaluates to \(\text{false}\)
- \(a \% b > c\) evaluates to \(\text{false}\)
- \(a - b > c\) evaluates to \(\text{false}\)

**Optimization**

- Propagate infected execution states in composed expressions
- Filter mutants whose infected state does not propagate
**Propagation**

An infected execution state propagates if it leads to an infected execution state of a lexically enclosing expression.

**Example:** $a=2, \ b=2, \ c=0$

```
if (a + b > c)
true
```

```
a * b
4
```

```
a / b > c
true
```

```
a % b > c
false
```

```
a - b > c
false
```

**Optimization**

- Propagate infected execution states in composed expressions
- Filter mutants whose infected state does not propagate
Partitioning

Build partition of identically infected execution states.

Example: $a=2$, $b=2$, $c=0$

- $a + b > c$: true
- $a \times b = 4$
- $a \div b > c$: true
- $a \mod b > c$: false
- $a - b > c$: false
Partitioning

Build partition of identically infected execution states.

Example: \(a=2, \ b=2, \ c=0\)

- \(a + b > c\) \(\text{true}\)
- \(a \times b = 4\)
- \(a \div b = 1\) \(\text{true}\)
- \(a \mod b > c\) \(\text{false}\)
- \(a - b > c\) \(\text{false}\)

Optimization

- **Partition** mutants based on their expression values
- Only execute a test for one mutant per **partition cell**
### Partitioning

Build partition of identically infected execution states.

**Example:** $a=2$, $b=2$, $c=0$

```markdown
if ((a + b) > c)  # true

if (a * b > c)  # false

if (a / b > c)  # true

if (a % b > c)  # false

if (a - b > c)  # false
```

### Optimization

- **Partition** mutants based on their expression values
- Only execute a test for one mutant per **partition cell**
Implementation details

\[ a + b > c \]
Implementation details

\[ a + b > c \]

eval(5, eval(3, a, b), c)

Expressions

\[
\begin{align*}
3_e & \mapsto +, 1_e, 2_e \\
5_e & \mapsto >, 3_e, 4_e
\end{align*}
\]
Implementation details

Expressions

\[ 3_e \mapsto +, \ 1_e, \ 2_e \]
\[ 5_e \mapsto >, \ 3_e, \ 4_e \]

Mutations

\[ 1_m \mapsto 3_e, \ - \]
\[ 2_m \mapsto 3_e, \ * \]
\[ 3_m \mapsto 5_e, \ >= \]
\[ 4_m \mapsto 5_e, \ == \]

Evaluation states:

\[ \text{eval}(5, \text{eval}(3, \ a, \ b), \ c) \]

Mapping:

\[ 1_e + 2_e > 3_e \]

Instrumentation:

\[ a - b > c \]
\[ a * b > c \]
\[ a + b >= c \]
\[ a + b == c \]
Implementation details

\[
\begin{align*}
&\quad 5_e \\
&\quad 3_e \\
&\quad 1_e \\
&\quad 2_e \\
&\quad 4_e \\
\end{align*}
\]

\[
\begin{align*}
\text{Expressions:} & \quad 3_e \mapsto +, 1_e, 2_e \\
& \quad 5_e \mapsto >, 3_e, 4_e \\
\end{align*}
\]

\[
\begin{align*}
\text{Mutations:} & \quad 1_m \mapsto 3_e, - \\
& \quad 2_m \mapsto 3_e, * \\
& \quad 3_m \mapsto 5_e, >= \\
& \quad 4_m \mapsto 5_e, == \\
\end{align*}
\]

Example: \( a = 2, b = 2, c = 1 \)

InfectedValues
Implementation details

\[ a + b > c \]

**Expressions**

\[ 3_e \mapsto +, \ 1_e, \ 2_e \]
\[ 5_e \mapsto >, \ 3_e, \ 4_e \]

**Mutations**

\[ 1_m \mapsto 3_e, - \]
\[ 2_m \mapsto 3_e, * \]
\[ 3_m \mapsto 5_e, \geq \]
\[ 4_m \mapsto 5_e, == \]

**Example:** \( a=2, \ b=2, \ c=1 \)

\[ \text{InfectedValues} \]

**InfectedValues**

\[ 3_e : \ a + b = 4 \]
\[ 1_m : \ a - b = 0 \]
\[ 2_m : \ a \ast b = 4 \]
Implementation details

Expressions

\[ 3_e \mapsto +, \ 1_e, \ 2_e \]
\[ 5_e \mapsto >, \ 3_e, \ 4_e \]

Mutations

\[ 1_m \mapsto 3_e, \ - \]
\[ 2_m \mapsto 3_e, \ * \]
\[ 3_m \mapsto 5_e, \ \geq \]
\[ 4_m \mapsto 5_e, \ == \]

Example: \( a=2, \ b=2, \ c=1 \)

\[ \text{InfectedValues} \]
\[ 1_m: \ 0 \]

\[ \text{eval}(5, \ \text{eval}(3, \ a, \ b), \ c) \]

\[ \begin{align*}
3_e: & \ a + b = 4 \\
1_m: & \ a - b = 0 
\end{align*} \]
Implementation details

Expressions

\[ 3_e \mapsto +, \ 1_e, \ 2_e \]
\[ 5_e \mapsto >, \ 3_e, \ 4_e \]

Mutations

\[ 1_m \mapsto 3_e, - \]
\[ 2_m \mapsto 3_e, * \]
\[ 3_m \mapsto 5_e, >= \]
\[ 4_m \mapsto 5_e, == \]

Example: \( a=2, \ b=2, \ c=1 \)

InfectedValues

\[ 1_m : \ 0 \]

\[ 5_e : \ 4 > c = \text{true} \]
\[ 3_m : \ 4 >= c = \text{true} \]
\[ 4_m : \ 4 == c = \text{false} \]
Implementation details

\[
\begin{align*}
\text{Expressions} & : \quad 3_e \mapsto +, \ 1_e, 2_e \\
& \quad 5_e \mapsto >, \ 3_e, 4_e
\end{align*}
\]

\[
\begin{align*}
\text{Mutations} & : \quad 1_m \mapsto 3_e, - \\
& \quad 2_m \mapsto 3_e, \ast \\
& \quad 3_m \mapsto 5_e, \geq \\
& \quad 4_m \mapsto 5_e, ==
\end{align*}
\]

Example: \(a=2, b=2, c=1\)

Infected Values

\[
\begin{align*}
5_e & : \quad 4 > c = \text{true} \\
3_m & : \quad 4 \geq c = \text{true} \\
4_m & : \quad 4 == c = \text{false} \\
1_m & : \quad 0 > c = \text{false}
\end{align*}
\]
Implementation details

\[ a + b > c \]

Expressions

\[ 3_e \leftrightarrow +, 1_e, 2_e \]
\[ 5_e \leftrightarrow >, 3_e, 4_e \]

Mutations

\[ 1_m \leftrightarrow 3_e, - \]
\[ 2_m \leftrightarrow 3_e, * \]
\[ 3_m \leftrightarrow 5_e, >= \]
\[ 4_m \leftrightarrow 5_e, == \]

Example: \( a=2, b=2, c=1 \)

InfectedValues

\[ 4_m: \text{false} \]
\[ 1_m: \text{false} \]

\[ 5_e: 4 > c = \text{true} \]
\[ 4_m: 4 == c = \text{false} \]
\[ 1_m: 0 > c = \text{false} \]
Implementation details

\[
\begin{align*}
5_e & \quad 3_e \\
1_e & \quad 2_e \\
a & + & b & > & c
\end{align*}
\]

Expressions
\[
\begin{align*}
3_e & \mapsto +, 1_e, 2_e \\
5_e & \mapsto >, 3_e, 4_e
\end{align*}
\]

Mutations
\[
\begin{align*}
1_m & \mapsto 3_e, - \\
2_m & \mapsto 3_e, * \\
3_m & \mapsto 5_e, >= \\
4_m & \mapsto 5_e, ==
\end{align*}
\]

Example: \(a=2, b=2, c=1\)

\[
\begin{align*}
4_m: & \quad false \\
1_m: & \quad false
\end{align*}
\]
Implementation details

\[
\begin{align*}
\text{Expressions} & \quad \text{Mutations} \\
3_e & \mapsto +, 1_e, 2_e & 1_m & \mapsto 3_e, - \\
5_e & \mapsto >, 3_e, 4_e & 2_m & \mapsto 3_e, * \\
& & 3_m & \mapsto 5_e, >= \\
& & 4_m & \mapsto 5_e, == \\
\end{align*}
\]

**Example:** a=2, b=2, c=1

eval(5, eval(3, a, b), c)

**InfectedValues**

\[
\begin{align*}
4_m : & \quad \text{false} \\
1_m : & \quad \text{false} \\
\end{align*}
\]

**Implemented in Major**

- Compact instrumentation
- Soundly handles side effects and short-circuit operators
Experimental setup

14 subject programs

- Open-source programs from different application domains
- 670,000 lines of code
- 540,000 generated mutants
Experimental setup

14 subject programs
- Open-source programs from different application domains
- 670,000 lines of code
- 540,000 generated mutants

4 test suites for each program
- 1 developer-written test suite (released with program)
- 3 generated test suites (EvoSuite)
  - Weak-mutation
  - Branch coverage
  - Random
Experimental setup

14 subject programs
- Open-source programs from different application domains
- 670,000 lines of code
- 540,000 generated mutants

4 test suites for each program
- 1 developer-written test suite (released with program)
- 3 generated test suites (EvoSuite)
  - Weak-mutation
  - Branch coverage
  - Random

Coverage optimization is baseline
**Ratio of analyzed mutants**

![Bar chart showing the ratio of analyzed mutants across different test suites and infection stages.](chart.png)

- **Findings**:
  - Only 70% of covered mutants need to be analyzed.
  - Similar ratio for Propagation and Partitioning.
  - Partitioning is (most) effective after Propagation.
  - Lower ratio for weaker test suites (e.g., Random).
**Ratio of analyzed mutants**

<table>
<thead>
<tr>
<th>Mutant Type</th>
<th>Infect</th>
<th>Infect+Prop</th>
<th>Infect+Part</th>
<th>Infect+Prop+Part</th>
</tr>
</thead>
</table>
| Manual      | ![Bar Graph](image)
| Weak-Mut.   | ![Bar Graph](image)
| Branch      | ![Bar Graph](image)
| Random      | ![Bar Graph](image) |

**Findings**

- Only 70% of covered mutants need to be analyzed
- Similar ratio for Propagation and Partitioning
- Partitioning is (most) effective after Propagation
- Lower ratio for weaker test suites (e.g., Random)
Ratio of total runtime

- Total run time reduced by 40%
- Filtering costs are almost negligible
- Partitioning is (most) effective after Propagation
- Run-time improvements similar for all test suites
**Ratio of total runtime**

![Bar chart showing ratio of total runtime for different test suites: manual, weak-mut, branch, random.](chart)

### Findings

- Total run time reduced by 40%
- Filtering costs are almost negligible
- Partitioning is (most) effective after Propagation
- Run-time improvements similar for all test suites
Future work

Equivalent mutant detection

- Can propagation predict equivalent mutants?
- Solve constraints necessary to achieve propagation
Future work

Equivalent mutant detection

- Can propagation predict equivalent mutants?
- Solve constraints necessary to achieve propagation

Test generation

- Generate tests that achieve propagation
- Improve mutation-driven test generation
  
  (Zhang et al., ICSM’10, Fraser and Zeller, TSE’12)
Contributions

Dynamic prepass analysis
▶ Three new optimizations that significantly improve efficiency
▶ Filter mutants with single test execution on instrumented program

Empirical evaluation
▶ 14 programs and 540,000 mutants
▶ Total run time reduced by 40%
▶ Propagation and Partitioning should be combined

http://www.mutation-testing.org