Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States

René Just[1] & Michael D. Ernst[1] & Gordon Fraser[2]

[1]University of Washington, USA
[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

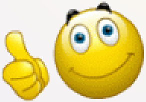
Motivation

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

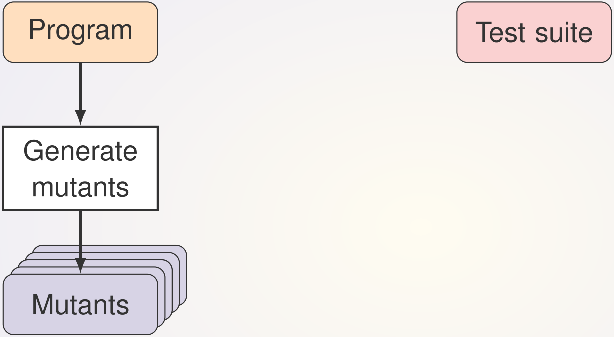
Mutation analysis: systematically seed artificial faults

Mutation analysis overview

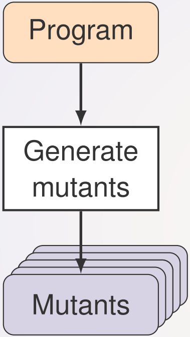
Program

Test suite

Mutation analysis overview



Mutation analysis overview



Program

Generate mutants

Mutants

```
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;
}
```

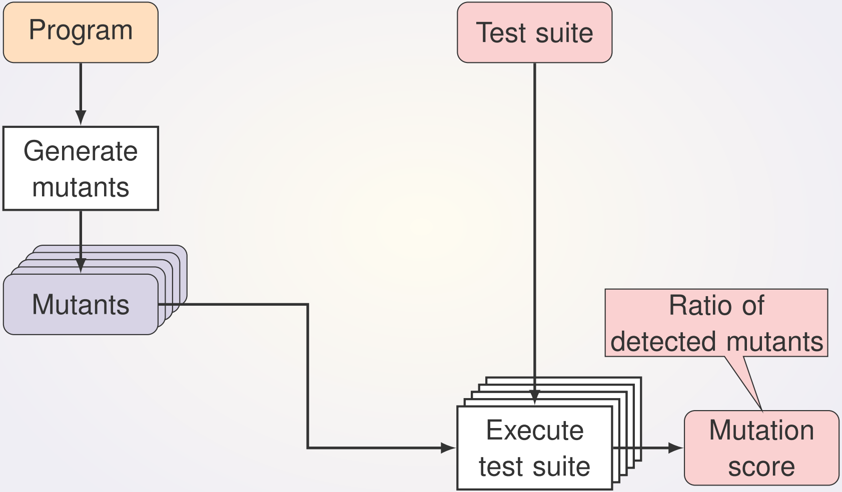
Mutant 1

```
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

Ratio of detected mutants

Mutation score

Where is the catch?

Many mutants can be generated!

```
a > b ?  a :  b
```

Where is the catch?

Many mutants can be generated!

```
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!

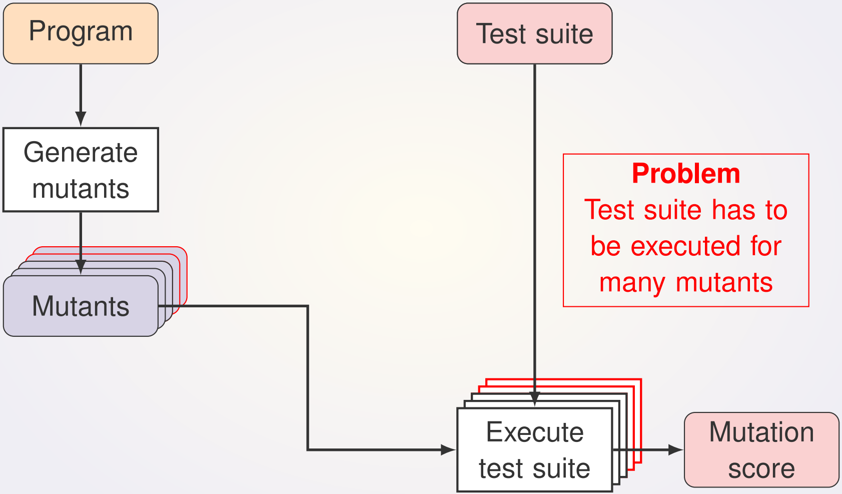
```
a > b ? a : b
```

```
a >= 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
```

```
0 > b ? a : b
a > 0 ? a : b
a > b ? 0 : b
a > b ? a : 0
0
b > a ? a : b
a > b ? b : a
```

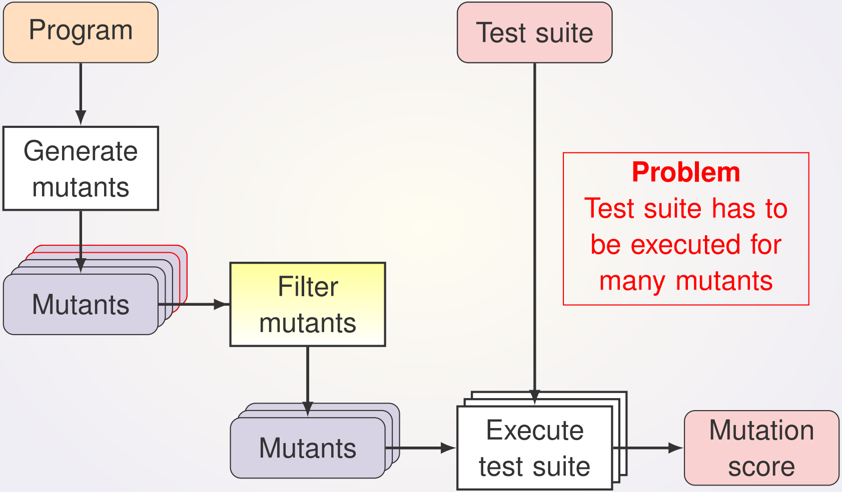
```
-a > b ?  a : b
 a >-b ?  a : b
 a > b ? -a : b
 a > b ?  a :-b
 a > b ?  a :-b
-(a > b ?  a : b)
 a
 b
```

Mutation analysis overview



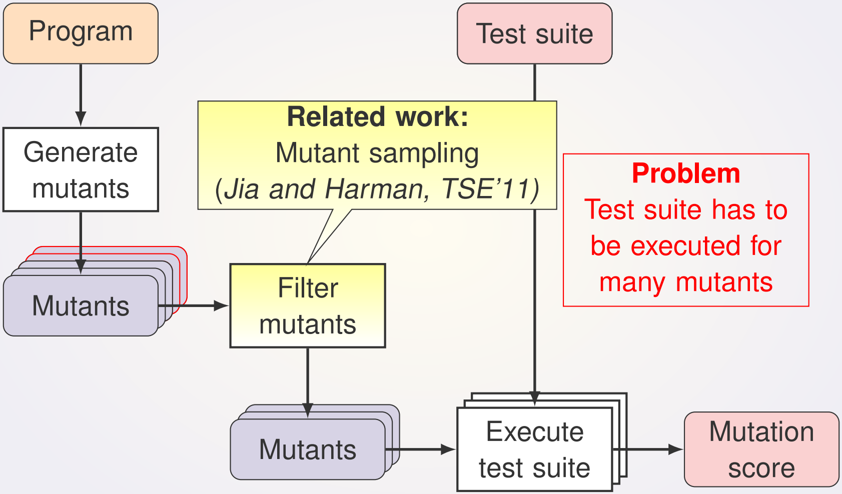
Problem
Test suite has to be executed for many mutants

Mutation analysis overview



Problem
Test suite has to be executed for many mutants

Mutation analysis overview



Program

Generate mutants

Mutants

Test suite

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

Filter mutants

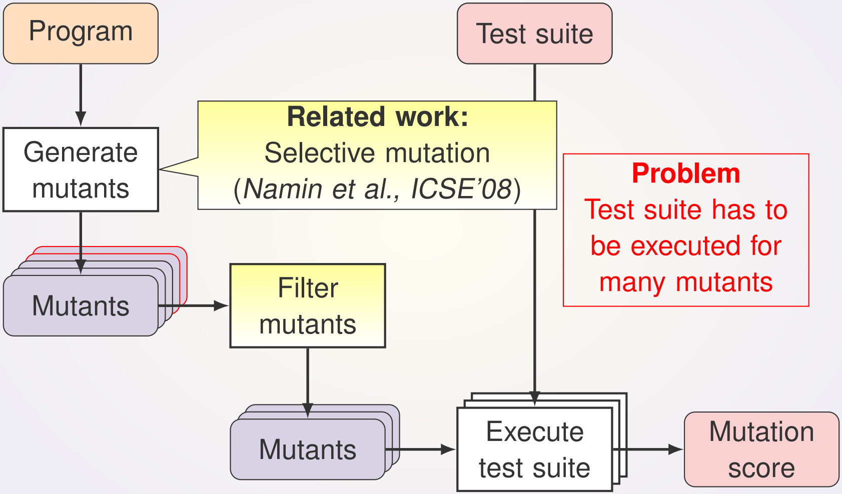
Mutants

Execute test suite

Mutation score

Problem
Test suite has to be executed for many mutants

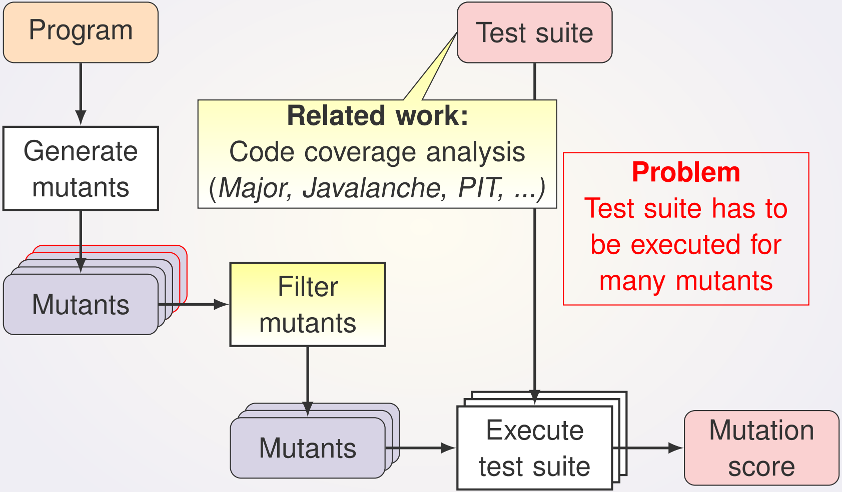
Mutation analysis overview



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

Problem
Test suite has to be executed for many mutants

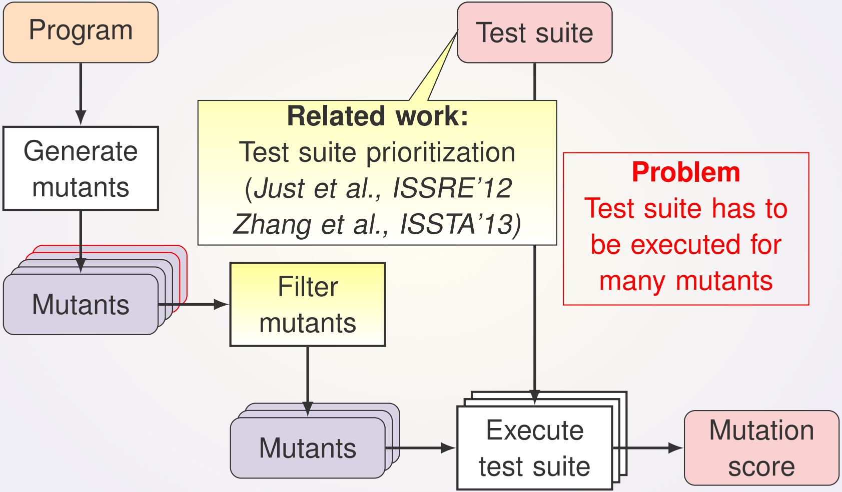
Mutation analysis overview



Related work:
Code coverage analysis
(*Major, Javalanche, PIT, ...*)

Problem
Test suite has to be executed for many mutants

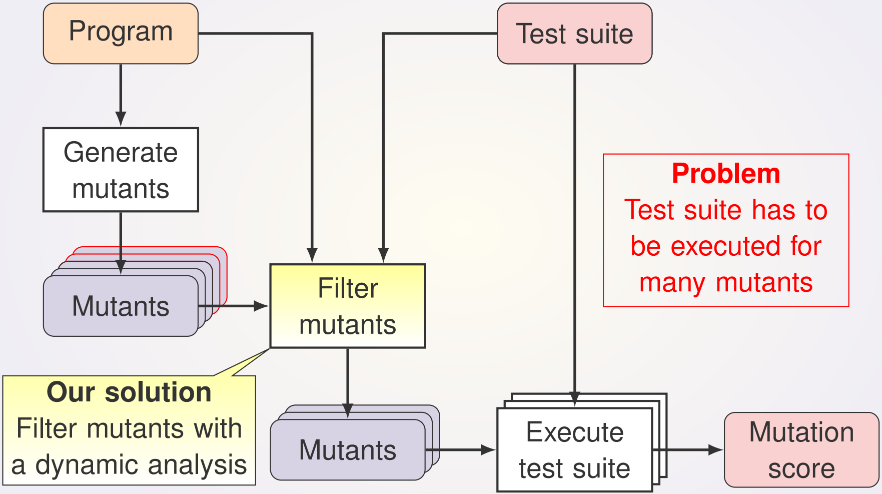
Mutation analysis overview



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



Problem
Test suite has to be executed for many mutants

Our solution
Filter mutants with a dynamic analysis

Wasted effort in mutation analysis

Example: testing triangle classification

```
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

```
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

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

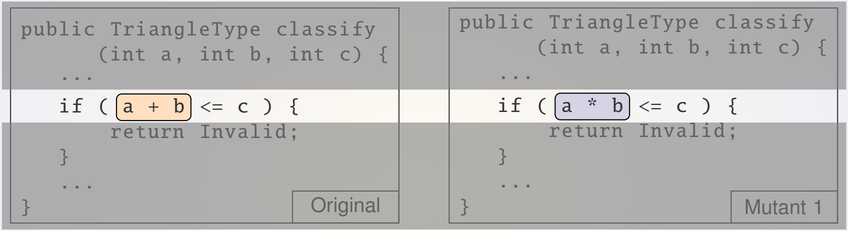
Original

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

Mutant 1

Wasted effort in mutation analysis

Example: testing triangle classification



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

Original

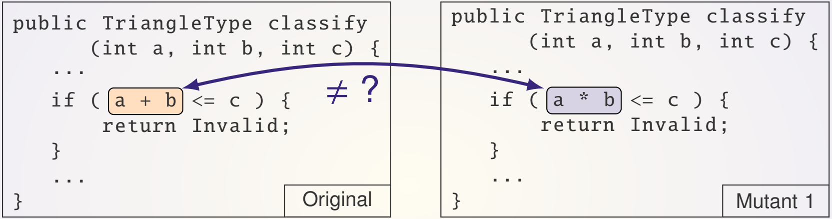
```
public TriangleType classify
      (int a, int b, int c) {
   ...
   if ( a * b <= c ) {
       return Invalid;
   }
   ...
}
```

Mutant 1

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

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

Original

```
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;
    }
    ...
}
```

Original

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

Mutant 1

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

Mutant 2

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;
   }
   ...
}
```

Original

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

Mutant 1

= ?

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

Mutant 2

Optimizations:

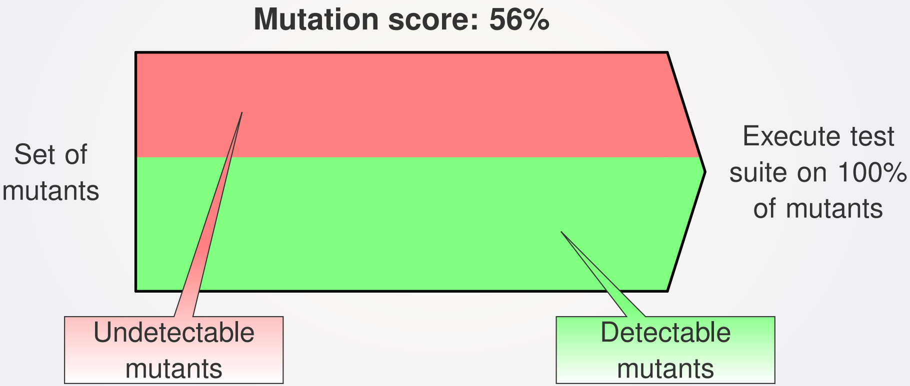
- Infection
- Propagation
- Partitioning

The big picture

Set of mutants

Execute test suite on 100% of mutants

The big picture



Mutation score: 56%

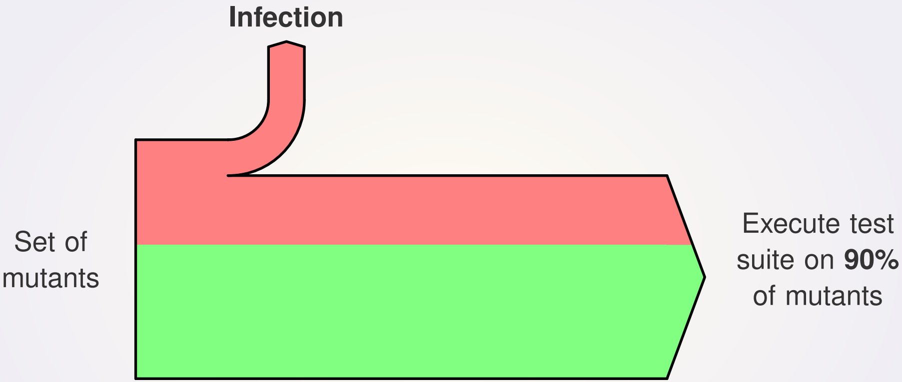
Set of mutants

Execute test suite on 100% of mutants

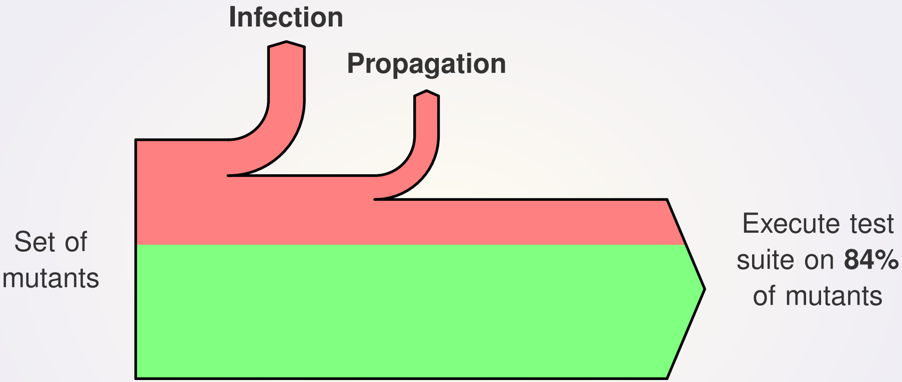
Undetectable mutants

Detectable mutants

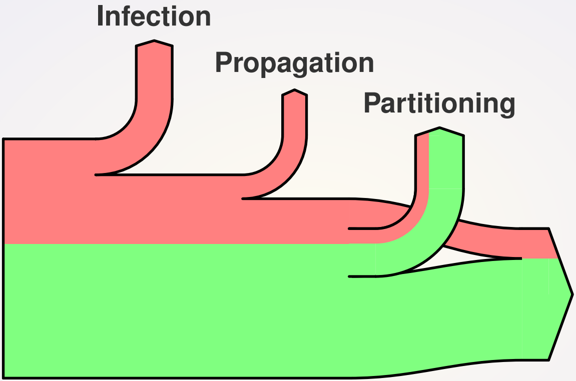
The big picture



The big picture



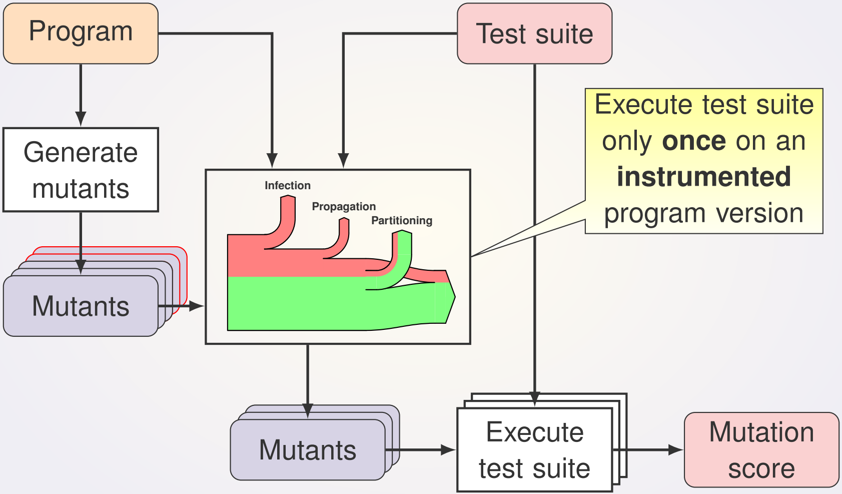
The big picture



Set of mutants

Execute test suite on **70%** of mutants

Dynamic analysis to filter mutants



Execute test suite only **once** on an **instrumented** program version

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.

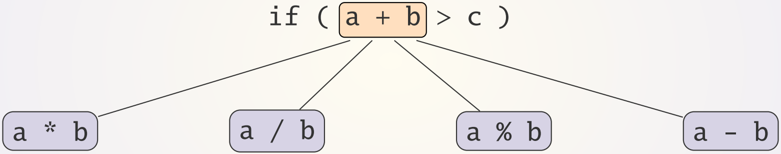
Example: `a=2, b=2, c=0`

```
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.

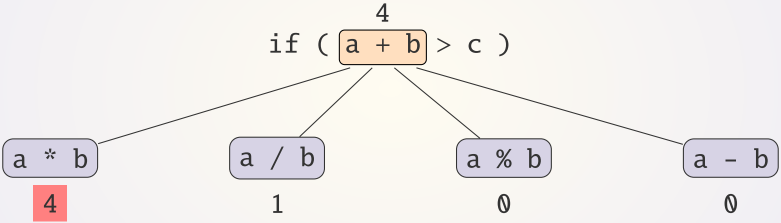
Example: `a=2, b=2, c=0`



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`



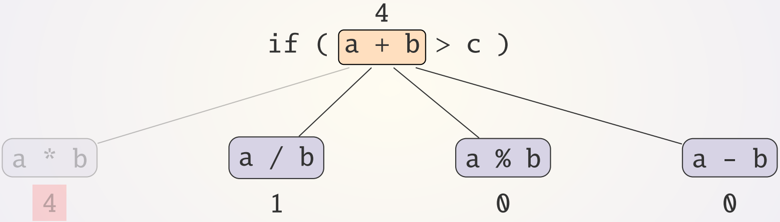
Optimization

- 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`



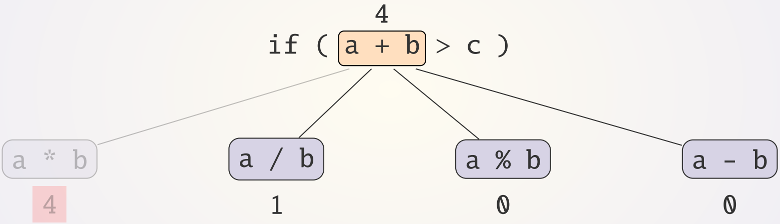
Optimization

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

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`



4

`if ( a + b > c )`

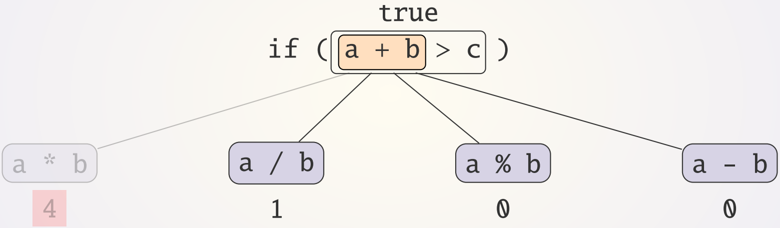
`a * b` `a / b` `a % b` `a - b`

4 1 0 0

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`



`true`

`if ( a + b > c )`

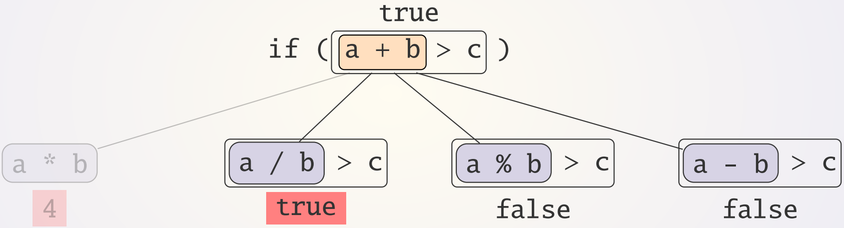
`a * b` `a / b` `a % b` `a - b`

`4` `1` `0` `0`

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`



`true`

`if ( a + b > c )`

`a * b`

`4`

`a / b > c`

`true`

`a % b > c`

`false`

`a - b > c`

`false`

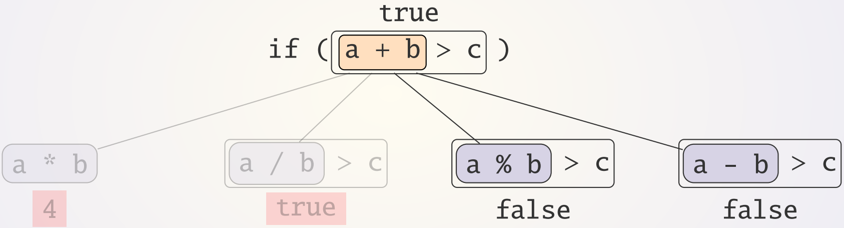
Optimization

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

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`



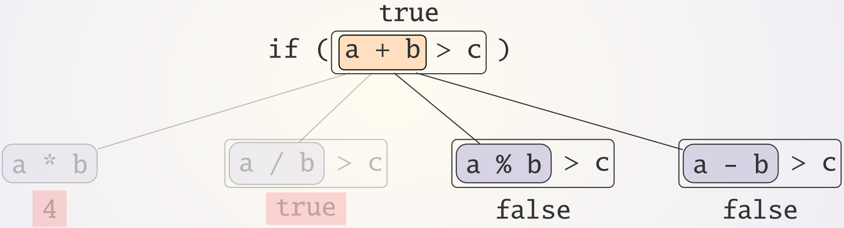
Optimization

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

Partitioning

Build partition of identically infected execution states.

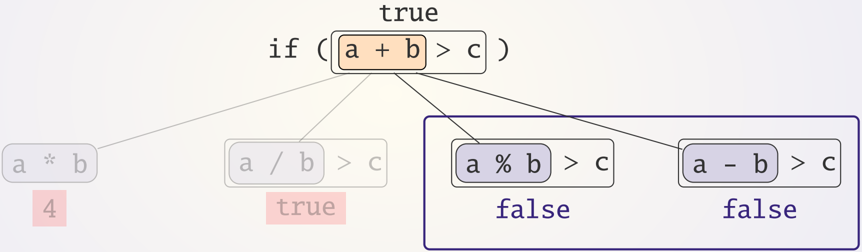
Example: `a=2, b=2, c=0`



Partitioning

Build partition of identically infected execution states.

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



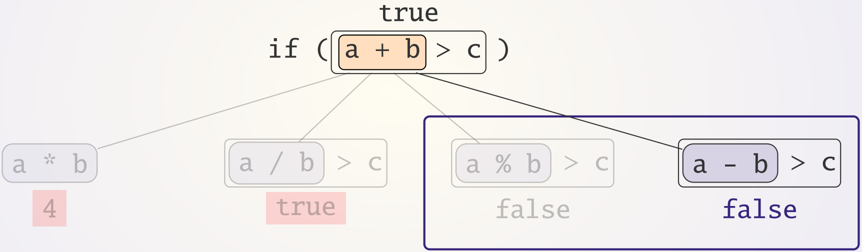
Optimization

- **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`



true

`if ( a + b > c )`

`a * b`

4

`a / b > c`

true

`a % b > c`

false

`a - b > c`

false

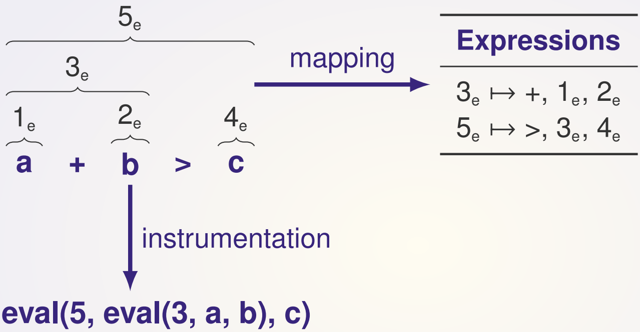
Optimization

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

Implementation details

a + b > c

Implementation details

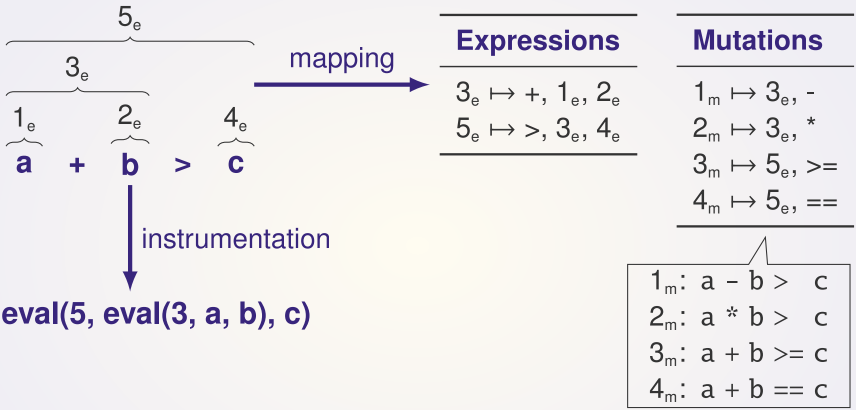


$$\overbrace{\overbrace{\overbrace{\mathbf{a}}^{1_e} + \overbrace{\mathbf{b}}^{2_e}}^{3_e} > \overbrace{\mathbf{c}}^{4_e}}^{5_e} \xrightarrow{\text{mapping}}$$

Expressions
$3_e \mapsto +, 1_e, 2_e$
$5_e \mapsto >, 3_e, 4_e$

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

Implementation details



$$\overbrace{\overbrace{\overbrace{\mathbf{a}}^{1_e} + \overbrace{\mathbf{b}}^{2_e}}^{3_e} > \overbrace{\mathbf{c}}^{4_e}}^{5_e}$$

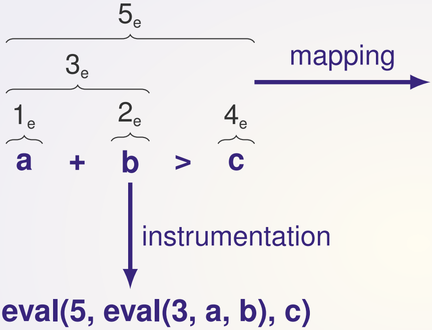
eval(5, eval(3, a, b), c)

Expressions
$3_e \mapsto +, 1_e, 2_e$
$5_e \mapsto >, 3_e, 4_e$

Mutations
$1_m \mapsto 3_e$, -
$2_m \mapsto 3_e$, *
$3_m \mapsto 5_e$, >=
$4_m \mapsto 5_e$, ==

1_m: a – b > c
2_m: a * b > c
3_m: a + b >= c
4_m: a + b == c

Implementation details



$$\overbrace{\overbrace{\overbrace{a}^{1_e} + \overbrace{b}^{2_e}}^{3_e} > \overbrace{c}^{4_e}}^{5_e}$$

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

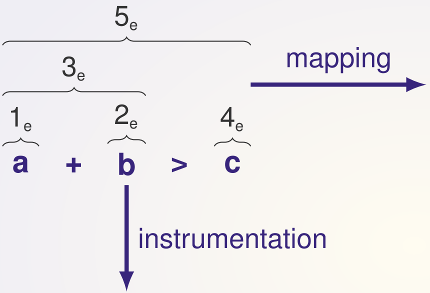
Expressions
$3_e \mapsto +, 1_e, 2_e$
$5_e \mapsto >, 3_e, 4_e$

Mutations
$1_m \mapsto 3_e$, -
$2_m \mapsto 3_e$, *
$3_m \mapsto 5_e$, >=
$4_m \mapsto 5_e$, ==

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

InfectedValues

Implementation details



$$\overbrace{\overbrace{\overbrace{a}^{1_e} + \overbrace{b}^{2_e}}^{3_e} > \overbrace{c}^{4_e}}^{5_e}$$

Expressions
$3_e \mapsto +, 1_e, 2_e$
$5_e \mapsto >, 3_e, 4_e$

Mutations
$1_m \mapsto 3_e$, -
$2_m \mapsto 3_e$, *
$3_m \mapsto 5_e$, >=
$4_m \mapsto 5_e$, ==

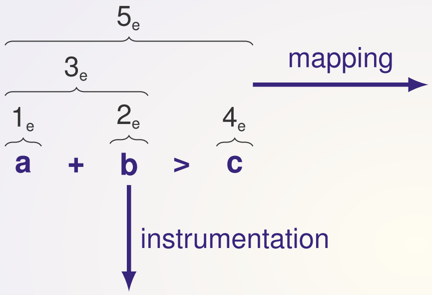
eval(5, eval(3, a, b), c)

3_e: a + b = 4
1_m: a - b = 0
2_m: a * b = 4

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

InfectedValues

Implementation details



$$\overbrace{\overbrace{\overbrace{\mathbf{a}}^{1_e} + \overbrace{\mathbf{b}}^{2_e}}^{3_e} > \overbrace{\mathbf{c}}^{4_e}}^{5_e}$$

Expressions
$3_e \mapsto +, 1_e, 2_e$
$5_e \mapsto >, 3_e, 4_e$

Mutations
$1_m \mapsto 3_e$, -
$2_m \mapsto 3_e$, *
$3_m \mapsto 5_e$, >=
$4_m \mapsto 5_e$, ==

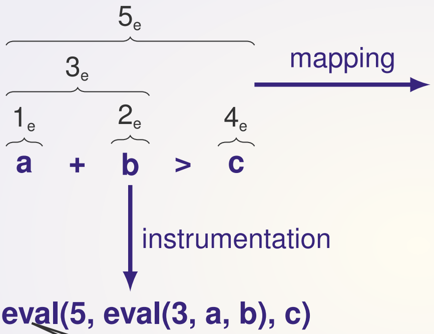
eval(5, eval(3, a, b), c)

3_e: a + b = 4
1_m: a - b = 0

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

InfectedValues
1_m: 0

Implementation details



$$\overbrace{\overbrace{\overbrace{\mathbf{a}}^{1_e} + \overbrace{\mathbf{b}}^{2_e}}^{3_e} > \overbrace{\mathbf{c}}^{4_e}}^{5_e}$$

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

5_e: 4 > c = true

3_m: 4 >= c = true

4_m: 4 == c = false

Expressions

$3_e \mapsto +, 1_e, 2_e$

$5_e \mapsto >, 3_e, 4_e$

Mutations

$1_m \mapsto 3_e$, -

$2_m \mapsto 3_e$, *

$3_m \mapsto 5_e$, >=

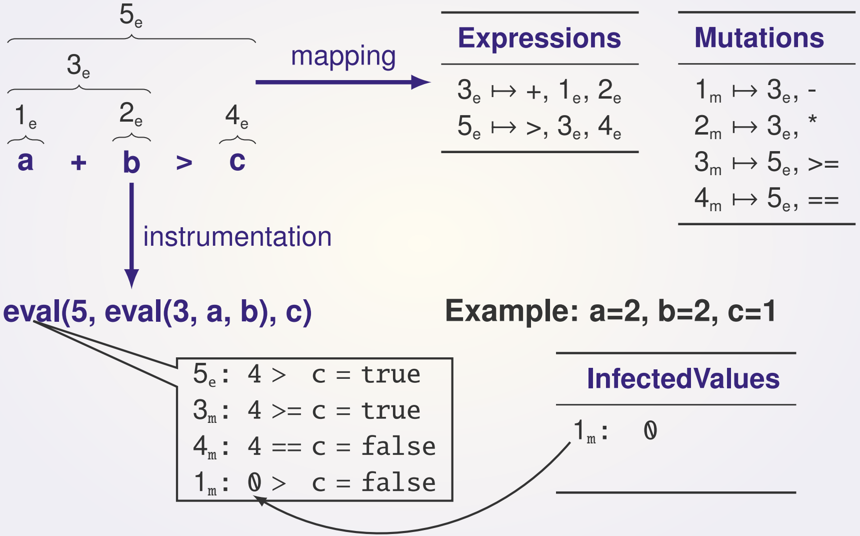
$4_m \mapsto 5_e$, ==

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

InfectedValues

1_m: 0

Implementation details



$\overbrace{\overbrace{\overbrace{\mathbf{a}}^{1_e} + \overbrace{\mathbf{b}}^{2_e}}^{3_e} > \overbrace{\mathbf{c}}^{4_e}}^{5_e}$

mapping

Expressions
$3_e \mapsto +, 1_e, 2_e$
$5_e \mapsto >, 3_e, 4_e$

Mutations
$1_m \mapsto 3_e$, -
$2_m \mapsto 3_e$, *
$3_m \mapsto 5_e$, >=
$4_m \mapsto 5_e$, ==

instrumentation

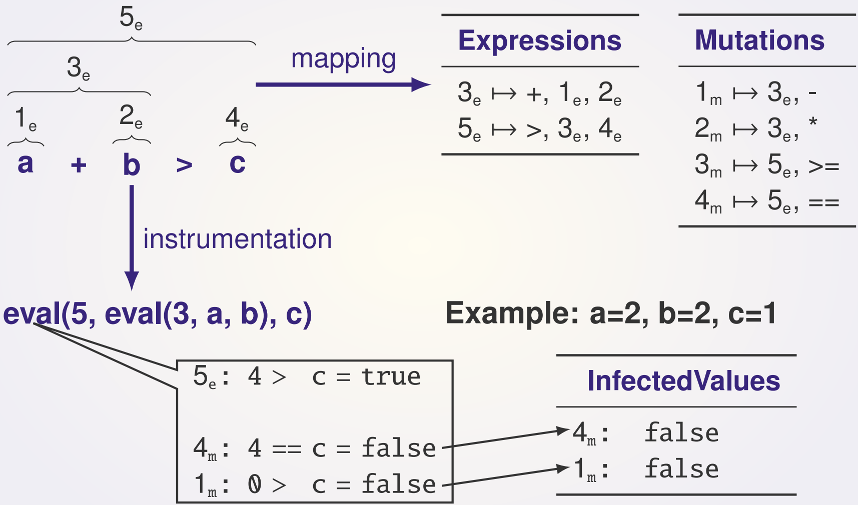
eval(5, eval(3, a, b), c)

```
5e: 4 >  c = true
3m: 4 >= c = true
4m: 4 == c = false
1m: 0 >  c = false
```

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

InfectedValues
`1m:  0`

Implementation details



$$\overbrace{\overbrace{\overbrace{\mathbf{a}}^{1_e} + \overbrace{\mathbf{b}}^{2_e}}^{3_e} > \overbrace{\mathbf{c}}^{4_e}}^{5_e} \xrightarrow{\text{mapping}}$$

Expressions
$3_e \mapsto +, 1_e, 2_e$
$5_e \mapsto >, 3_e, 4_e$

Mutations
$1_m \mapsto 3_e$, -
$2_m \mapsto 3_e$, *
$3_m \mapsto 5_e$, >=
$4_m \mapsto 5_e$, ==

instrumentation

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

5_e: 4 > c = true

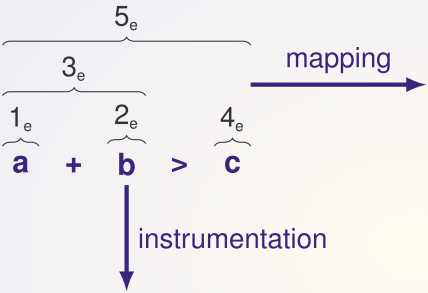
4_m: 4 == c = false

1_m: 0 > c = false

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

InfectedValues
4_m: false
1_m: false

Implementation details



$$\overbrace{\overbrace{\overbrace{a}^{1_e} + \overbrace{b}^{2_e}}^{3_e} > \overbrace{c}^{4_e}}^{5_e}$$

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

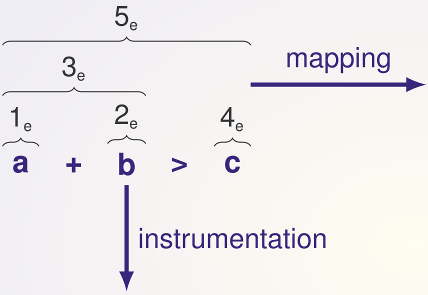
Expressions
$3_e \mapsto +, 1_e, 2_e$
$5_e \mapsto >, 3_e, 4_e$

Mutations
$1_m \mapsto 3_e$, -
$2_m \mapsto 3_e$, *
$3_m \mapsto 5_e$, >=
$4_m \mapsto 5_e$, ==

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

InfectedValues	
4_m:	`false`
1_m:	`false`

Implementation details



$$\overbrace{\overbrace{\overbrace{a}^{1_e} + \overbrace{b}^{2_e}}^{3_e} > \overbrace{c}^{4_e}}^{5_e}$$

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

Expressions
$3_e \mapsto +, 1_e, 2_e$
$5_e \mapsto >, 3_e, 4_e$

Mutations
$1_m \mapsto 3_e$, -
$2_m \mapsto 3_e$, *
$3_m \mapsto 5_e$, >=
$4_m \mapsto 5_e$, ==

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

InfectedValues
4_m: `false`
1_m: `false`

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

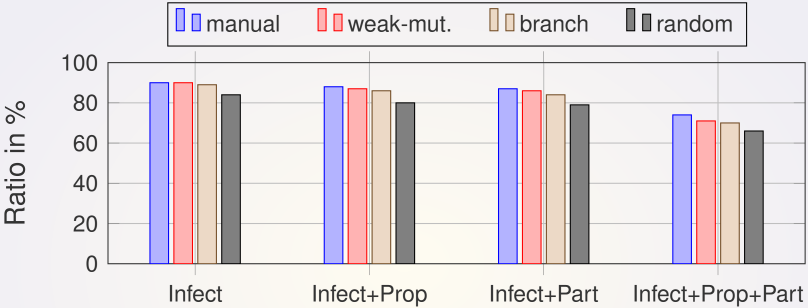
- Open-source programs from different application domains
- 670,000 lines of code
- 540,000 generated mutants

4 test suites for each program

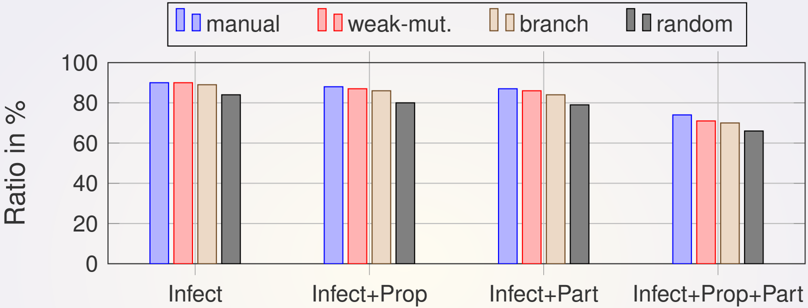
- 1 developer-written test suite (released with program)
- 3 generated test suites (EvoSuite)
  - Weak-mutation
  - Branch coverage
  - Random

Coverage optimization is baseline

Ratio of analyzed mutants



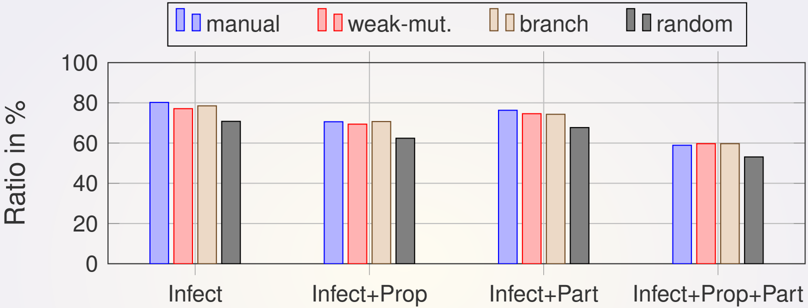
Ratio of analyzed mutants



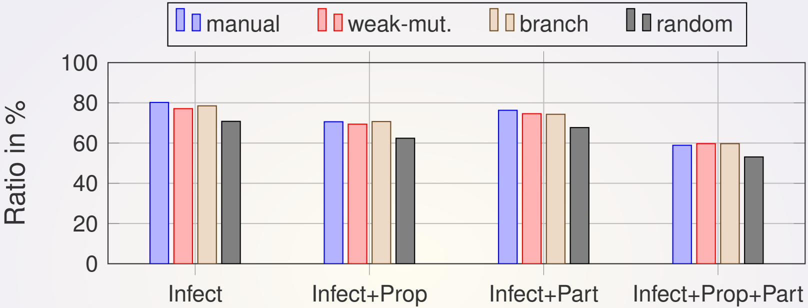
Findings

- 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



Ratio of total runtime



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

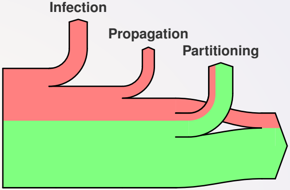
Test generation

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

Contributions

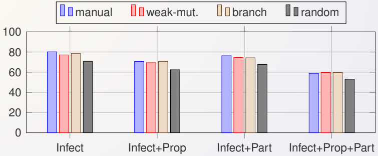
Dynamic prepass analysis

- 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