

CSE P 504

Advanced topics in Software Systems

Fall 2022

Mutation-based Testing

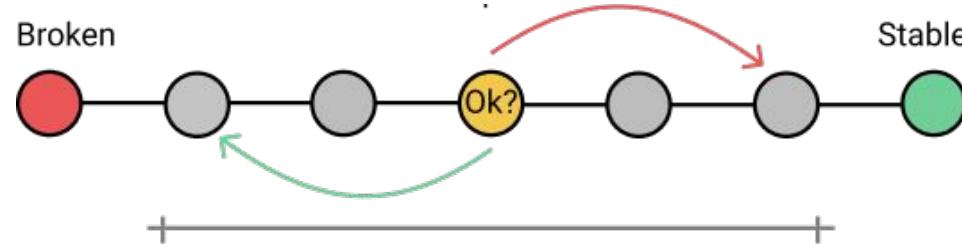
October 24, 2022

Today

- Recap: Git bisect exercise
- Mutation-based testing
 - The basics
 - Productive mutants
 - Mutant subsumption
- Coverage-based vs. mutation-based testing
- In-class exercise 3

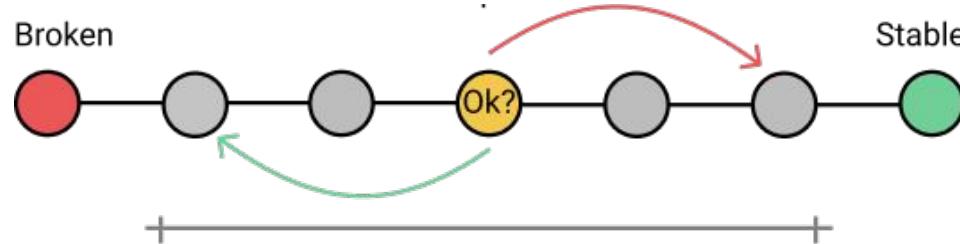
Recap: Git bisect

- **Git bisect run-time complexity is always $O(\log(n))$**

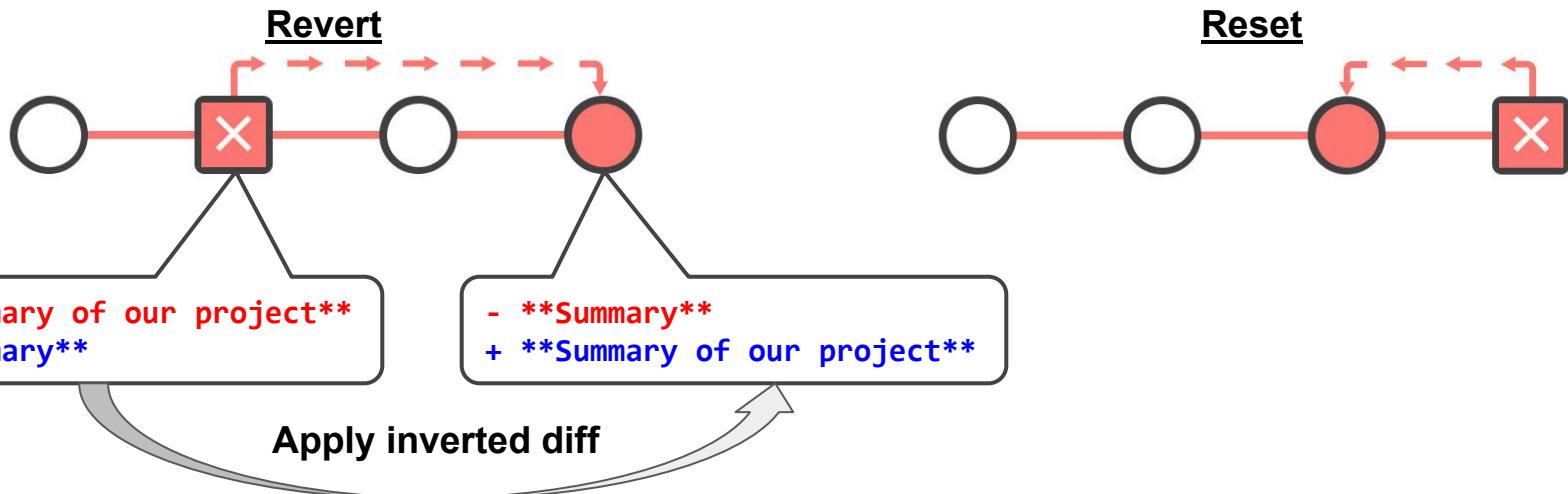


Recap: Git bisect

- **Git bisect run-time complexity is always $O(\log(n))$**

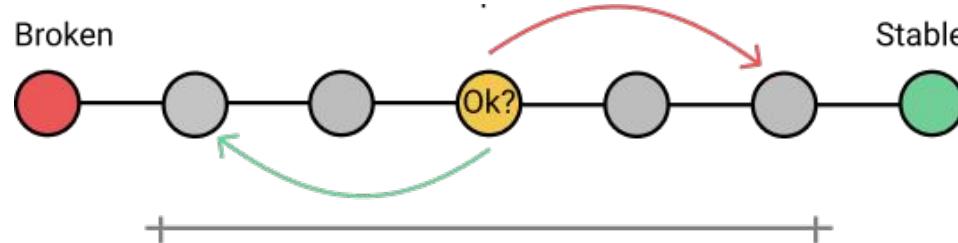


- **Git revert vs. git reset**



Recap: Git bisect

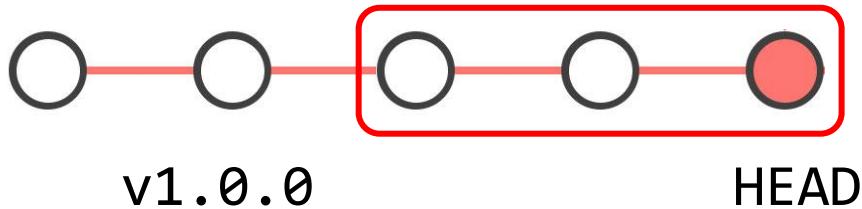
- **Git bisect run-time complexity is always $O(\log(n))$**



- **Git revert vs. git reset**



- **git rev-list v1.0.0..HEAD (or HEAD ^v1.0.0)**



Mutation-based testing: the basics

Mutation testing

```
public class GruenbergSettings extends Game {
    ...
    // (1) add state variables here
    ...
    // (2) add initialization of state variables here
    ...
    public void GruenbergSettings() {
        condition = 0;
    }
    ...
    // (3) add lattice initialization here
    ...
    public void initialize(Lattice l) {
        l.set((GruenbergSettings)l.getInitial(l));
        condition = 3;
    }
    ...
    // (4) add drawing/color definition here
    ...
    // drawing null means the cell will not be drawn
    ...
    public void drawCell(Cell cell) {
        condition = cell.getCondition();
        if (condition == 0) {
            cell.setColor(Color.CYAN);
        } else if (condition == 1) {
            cell.setColor(Color.MAGENTA);
        } else if (condition == 2) {
            cell.setColor(Color.BLUE);
        } else if (condition == 3) {
            cell.setColor(Color.GRAY);
        }
    }
    ...
    // (5) add state variables. copy code here
    ...
    public void copyState(Cell c) {
        GruenbergSettings source = (GruenbergSettings)c;
        condition = source.condition;
    }
    ...
    // (6) add logic function. code here
    ...
    // evaluating the next state of the cell
    ...
    public void transition(Cell cell) {
        if (condition == 0) {
            condition = condition;
        } else if (condition == 1) {
            int[] neighbors = cell.getNeighbors();
            for (int i=0; i<neighbors.length; i++) {
                if (neighbors[i] == 1) {
                    condition = 2;
                }
            }
        }
    }
}
```



Mutation
testing

Program

Mutation testing: mutant generation

```
public class GossenbergsString extends Game {
    ...
    // (1) add state variables here
    ...
    // (2) add initialization of state variables here
    ...
    public void gossenbergsString() {
        condition = 0;
    }
    ...
    // (3) add lattice initialization here
    ...
    public void initializeLattice() {
        ((GossenbergsString) getLattice(0)).condition = 3;
    }
    ...
    // (4) add drawing code definition here
    // (warning: null means the cell will not be drawn)
    ...
    public void drawCell(int row, int column, Object cell) {
        if (cell == null) {
            return;
        }
        ...
        // (5) add state variables: copy code here
        ...
        public void copyState() {
            gossenbergsString source = (GossenbergsString) source();
            ...
            // (6) add target function: code here
            // (warning: the set state to the cell)
            ...
            public void transition(Cell cell) {
                if (condition == condition1) {
                    ...
                } else if (condition == condition2) {
                    ...
                }
            }
        }
    }
}
```



Mutation
testing

Program



$Lhs < rhs \xrightarrow{\text{X}} Lhs \leq rhs$

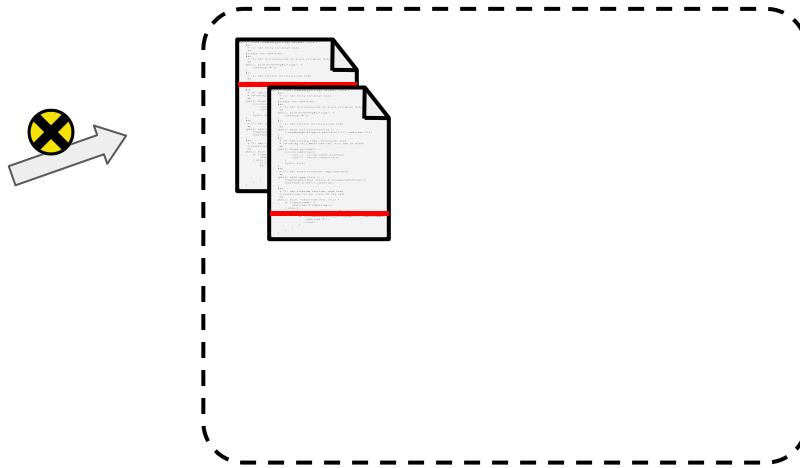
$Lhs < rhs \xrightarrow{\text{X}} Lhs \neq rhs$

$stmt \xrightarrow{\text{X}} no\text{-}op$

Mutation operators

Mutation testing: mutant generation

```
public class GridCellSettings extends GridSettings {  
    ...  
    * (1) add state variables here  
    *  
    * (2) add initialization of state variables here  
    *  
    public void drawCell(GridSettings g) {  
        if (condition == 0) {  
            ...  
            * (3) add lattice initialization here  
        }  
    }  
    ...  
    * (4) add drawing color definition here  
    * (5) overriding null means the cell will not be drawn  
    *  
    public void setCellColor(int condition) {  
        switch (condition) {  
            case 0: return Color.BLUE;  
            ...  
        }  
    }  
    ...  
    * (6) add state variables copy code here  
    *  
    public void copyState(Cell c) {  
        GridCellSettings source = (GridCellSettings)c;  
        ...  
        * (7) add logic function code here  
        * (8) overriding the set state method for cell  
        public void transaction(Cell cell) {  
            if (condition == 0) {  
                condition = condition+1;  
            } else {  
                if (condition >= 10) {  
                    condition = 0;  
                }  
            }  
        }  
    }  
}
```



Program

Mutants

$$Lhs < rhs \xrightarrow{\text{X}} Lhs \leq rhs$$

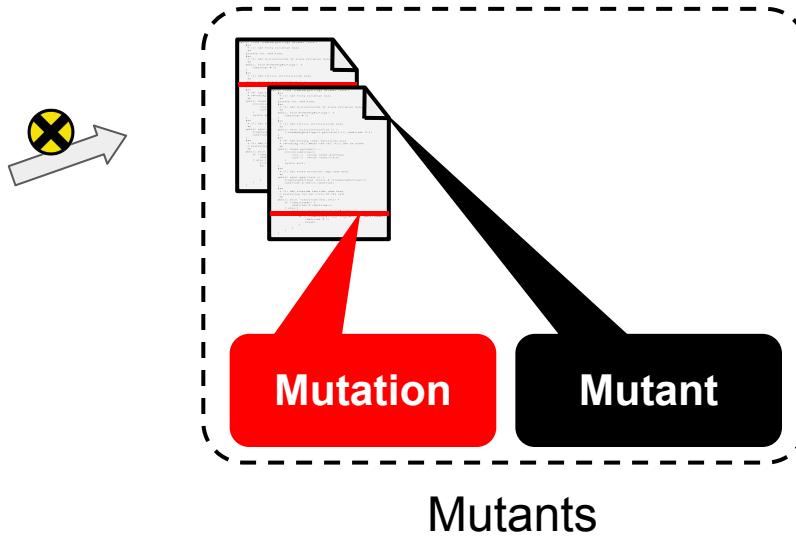
$$Lhs < rhs \xrightarrow{\text{X}} Lhs \neq rhs$$

$$\begin{matrix} stmt \\ \xrightarrow{\text{X}} \\ no-op \end{matrix}$$

Mutation operators

Mutation testing: mutant generation

```
public class GruenbergSettings extends Game {
    ...
    * (1) add state variables here
    ...
    * (2) add initialization of state variables here
    ...
    public void drawGruenbergSettings() {
        ...
        condition = 0;
    }
    ...
    * (3) add lattice initialization here
    public void initializeLattice() {
        ...
        * (4) add drawing color definition here
        // Drawing null means the cell will not be drawn
        ...
        conditionDefinition = new ConditionDefinition();
        conditionDefinition.setCellColor(Color.DARK_GRAY);
        ...
        cell = new Cell(conditionDefinition);
        ...
        switch (null);
    }
    ...
    * (5) add state variables copy code here
    ...
    public void copyState() {
        ...
        GruenbergSettings source = (GruenbergSettings) source;
        ...
        condition = source.condition;
    }
    ...
    * (6) add logic function code here
    // Evaluating the set state for the cell
    public void transition(Cell cell) {
        if (condition == 0)
            ...
        else
            ...
        if (condition == 1)
            ...
        condition = condition;
    }
}
```

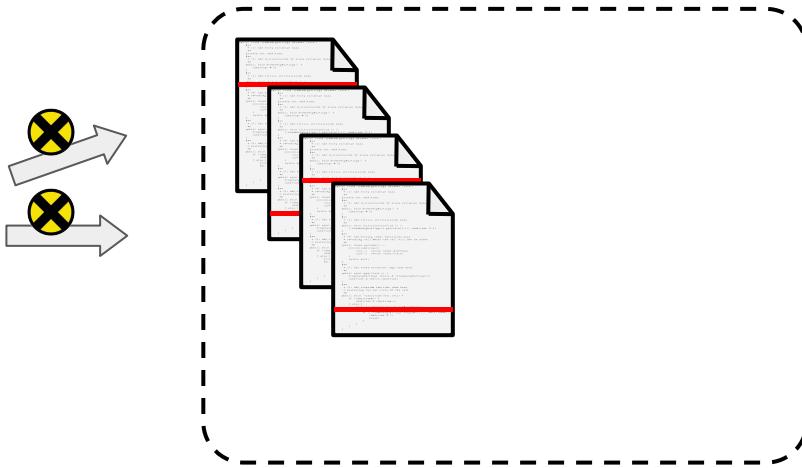


$$Lhs < rhs \xrightarrow{\text{X}} Lhs \leq rhs$$
$$Lhs < rhs \xrightarrow{\text{X}} Lhs \neq rhs$$
$$stmt \xrightarrow{\text{X}} no-op$$

Mutation operators

Mutation testing: mutant generation

```
public class GridCellSettings extends GridSettings {  
    ...  
    * (1) add state variables here  
    * (2) add initialization of state variables here  
    * (3) add lattice initialization here  
    public void initialize(Lattice l) {  
  
        /* (1) add drawing code definition here  
         * (2) drawing null means the cell will not be drawn  
         */  
        l.addCellDefinition(  
            new CellDefinition() {  
                public void drawCell(GridCell cell, Object viewer) {  
                    cell.setCellColor(Color.BLUE);  
                }  
            },  
            null);  
        ...  
        * (3) add state variables copy code here  
        * (4) add copyState code here  
        * (5) add update function code here  
        * (6) add transition function code here  
        public void copyState(Cell source, GridCell target) {  
            if (condition == 0) {  
                target.setCellColor(source.getCellColor());  
            } else {  
                if (condition == 1) {  
                    target.setCellColor(Color.RED);  
                } else {  
                    target.setCellColor(Color.GREEN);  
                }  
            }  
        }  
    }  
}
```



Program

Mutants

$$Lhs < rhs \xrightarrow{\text{X}} Lhs \leq rhs$$

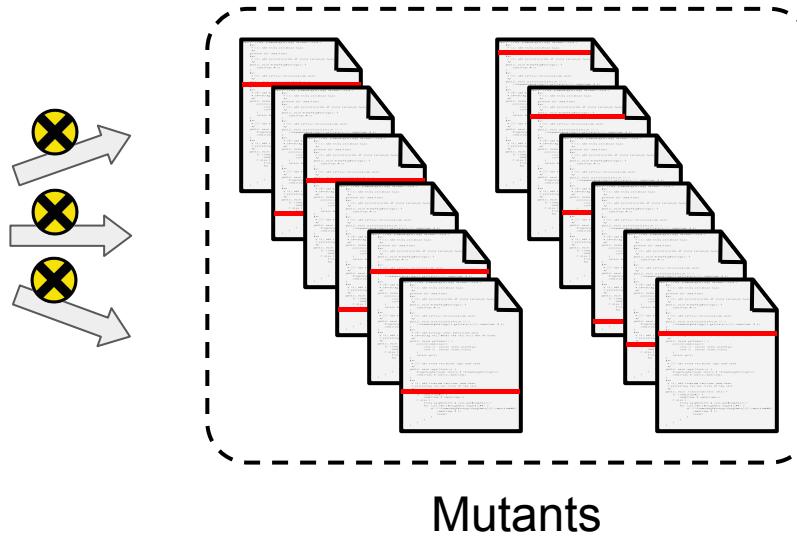
$$Lhs < rhs \xrightarrow{\text{X}} Lhs \neq rhs$$

$$\text{stmt} \xrightarrow{\text{X}} \text{no-op}$$

Mutation operators

Mutation testing: mutant generation

```
public class GridCellSettings extends GridCell {  
    ...  
    // (1) add state variables here  
    ...  
  
    // (2) add lattice initialization here  
    public void initialize(Lattice l) {  
  
        // (3) add drawing/color definition here  
        // setting null means the cell will not be drawn  
        l.setCellColor(null);  
        l.setCellCondition();  
  
        // switch null  
        ...  
        // (4) add state variables/copy code here  
        ...  
  
        // mutation - check condition here  
        ...  
  
        public void transition(GridCell cell) {  
            if (condition == 0)  
                condition = condition + 1;  
            else  
  
                if (condition == 0) condition = condition + 1;  
                else condition = condition - 1;  
            ...  
        }  
    }  
}
```

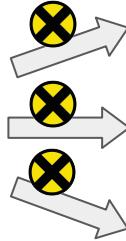


$$\begin{array}{c} Lhs < rhs \xrightarrow{\text{X}} Lhs \leq rhs \\[10pt] Lhs < rhs \xrightarrow{\text{X}} Lhs \neq rhs \\[10pt] stmt \xrightarrow{\text{X}} no-op \end{array}$$

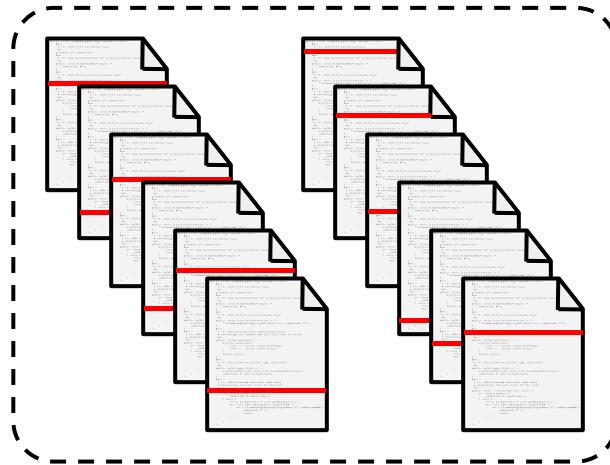
Mutation operators

Mutation testing: test creation

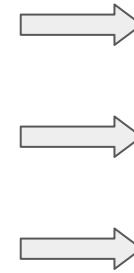
```
public class GridCellDrawing {
    ...
    public void drawCell() {
        ...
        if (condition1) {
            ...
        } else if (condition2) {
            ...
        }
        ...
    }
}
```



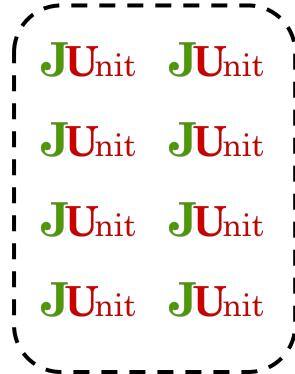
Program



Mutants



Tests



Assumptions

- Mutants are coupled to real faults
- Mutant detection is correlated with real-fault detection

https://homes.cs.washington.edu/~rjust/publ/mutants_real_faults_fse_2014.pdf

https://homes.cs.washington.edu/~rjust/publ/mutation_testing_practices_icse_2021.pdf

Mutation testing: a concrete example

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutant 1:

```
public int min(int a, int b) {  
    return a;  
}
```

Mutation testing: another example

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutant 2:

```
public int min(int a, int b) {  
    return b;  
}
```

Mutation testing: yet another example

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutant 3:

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

Mutation testing: last example (I promise)

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutant 4:

```
public int min(int a, int b) {  
    return a <= b ? a : b;  
}
```

Mutation testing: exercise



Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutants:

```
M1: return a;  
M2: return b;  
M3: return a >= b ? a : b;  
M4: return a <= b ? a : b;
```

**For each mutant, provide a test case that detects it
(i.e., passes on the original program but fails on the mutant)**

Mutation testing: exercise

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutants:

```
M1: return a;  
M2: return b;  
M3: return a >= b ? a : b;  
M4: return a <= b ? a : b;
```

M4 cannot be detected (equivalent mutant).

a	b	Original	M1	M2	M3	M4
1	2	1	1	2	2	1
1	1	1	1	1	1	1
2	1	1	2	1	2	1

Mutation testing: exercise

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutants:

```
M1: return a;  
M2: return b;  
M3: return a >= b ? a : b;  
M4: return a <= b ? a : b;
```

Which mutant(s) should we show to a developer?

a	b	Original	M1	M2	M3	M4
1	2	1	1	2	2	1
1	1	1	1	1	1	1
2	1	1	2	1	2	1

Mutation testing: summary

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutants:

- M1: return a;
- M2: return b;
- M3: return a >= b ? a : b;
- M4: return a <= b ? a : b;

Redundant

Equivalent

a	b	Original	M1	M2	M3	M4
1	2	1	1	2	2	1
1	1	1	1	1	1	1
2	1	1	2	1	2	1

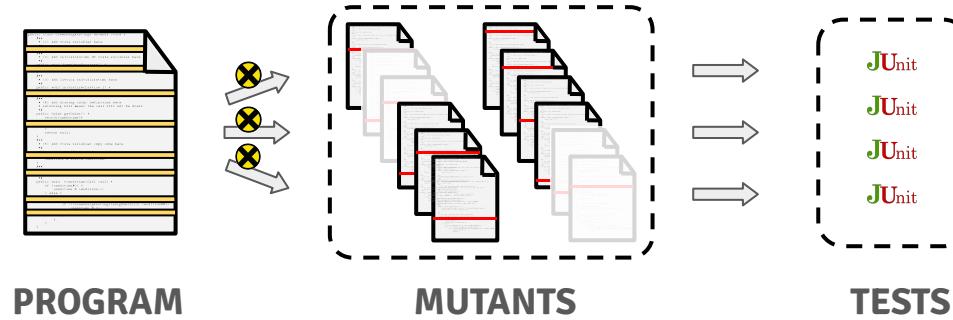
Mutation testing: challenges

- Redundant mutants
 - Inflate the mutant detection ratio
 - Hard to assess progress and remaining effort
- Equivalent mutants
 - Max mutant detection ratio != 100%
 - Waste resources (CPU and human time)

a	b	Original	M1	M2	M3	M4
1	2	1	1	2	2	1
1	1	1	1	1	1	1
2	1	1	2	1	2	1

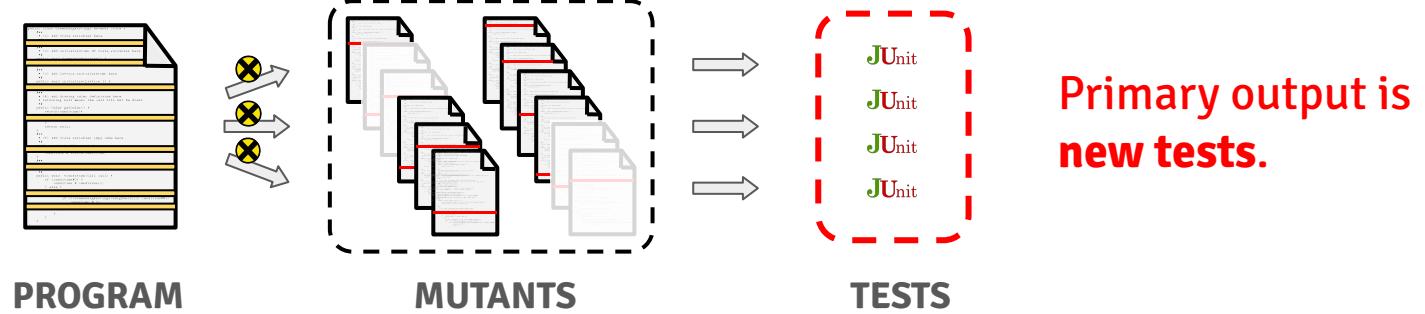
Mutation Testing vs. Mutation Analysis

Mutation
Testing



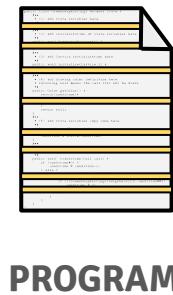
Mutation Testing vs. Mutation Analysis

Mutation
Testing



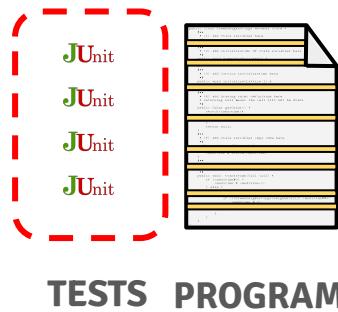
Mutation Testing vs. Mutation Analysis

Mutation
Testing



Primary output is
new tests.

Mutation
Analysis

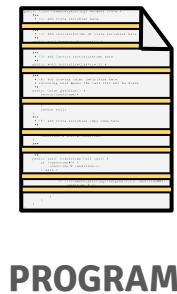


80%
ADEQUACY
SCORE

Primary output is
**adequacy score for
existing tests.**

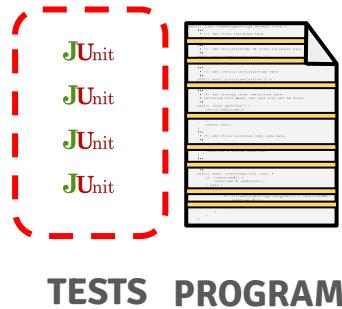
Mutation Testing vs. Mutation Analysis

Mutation
Testing



Primary output is
new tests.

Mutation
Analysis



Primary output is
**adequacy score for
existing tests.**

How expensive is mutation testing?
Is the mutation score meaningful?

Mutation-based testing: productive mutants

Detectable vs. productive mutants

Historically

- Detectable mutants are good → tests
- Equivalent mutants are bad → no tests

A more nuanced view

- Detectable vs. equivalent is too simplistic
- Productive mutants elicit effective tests, but
 - detectable mutants can be useless, and
 - equivalent mutants can be useful!

The core question here concerns test-goal utility
(applies to any adequacy criterion).

Detectable vs. productive mutants

Historically

- Detectable mutants are good → tests
- Equivalent mutants are bad → no tests

A more nuanced view

- Detectable vs. equivalent is too simplistic
- Productive mutants elicit effective tests, but
 - detectable mutants can be useless, and
 - equivalent mutants can be useful!

The notion of productive mutants is fuzzy!

A mutant is **productive** if it is

1. **detectable and elicits an effective test** or
2. **equivalent and advances code quality or knowledge**

Productive mutants: mutation testing at Google

```
int RunMe(int a, int b) {  
    if (a == b || b == 1) {
```

7

8

▼ Mutants Changing this 1 line to

14:25, 28 Mar

```
        if (a != b || b == 1) {
```

does not cause any test exercising them to fail.

Consider adding test cases that fail when the code is mutated to ensure those bugs would be caught.

Mutants ran because goranpetrovic is whitelisted

[Please fix](#)

[Not useful](#)

Practical Mutation Testing at Scale: A view from Google ([Reading 3](#))

Productive mutants: mutation testing at Google

```
int RunMe(int a, int b) {  
    if (a == b || b == 1) {
```

▼ Mutants

14:25, 28 Mar

Changing this 1 line to

```
if (a != b || b == 1) {
```

does not cause any test exercising them to fail.

Consider adding test cases that fail when the code is mutated to ensure those bugs would be caught.

Mutants ran because goranpetrovic is whitelisted

[Please fix](#)

[Not useful](#)

Practical Mutation Testing at Scale: A view from Google ([Reading 3](#))

Detectable vs. productive mutants (1)

Original program

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Mutant

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum * nums[i];  
    }  
  
    return sum / len;  
}
```

Is the mutant is **detectable**?

Detectable vs. productive mutants (1)

Original program

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Mutant

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum * nums[i];  
    }  
  
    return sum / len;  
}
```

The mutant is **detectable**, but is it **productive**?

Detectable vs. productive mutants (1)

Original program

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Mutant

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum * nums[i];  
    }  
  
    return sum / len;  
}
```

The mutant is **detectable**, but is it **productive**? Yes!

Detectable vs. productive mutants (2)

Original program

```
public double getAvg(double[] nums) {  
    int len = nums.length;  
    double sum = 0;  
    double avg = 0;  
  
    for (int i = 0; i < len; ++i) {  
        avg = avg + (nums[i] / len);  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Mutant

```
public double getAvg(double[] nums) {  
    int len = nums.length;  
    double sum = 0;  
    double avg = 0;  
  
    for (int i = 0; i < len; ++i) {  
        avg = avg * (nums[i] / len);  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Is the mutant **detectable**?

Detectable vs. productive mutants (2)

Original program

```
public double getAvg(double[] nums) {  
    int len = nums.length;  
    double sum = 0;  
    double avg = 0;  
  
    for (int i = 0; i < len; ++i) {  
        avg = avg + (nums[i] / len);  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Mutant

```
public double getAvg(double[] nums) {  
    int len = nums.length;  
    double sum = 0;  
    double avg = 0;  
  
    for (int i = 0; i < len; ++i) {  
        avg = avg * (nums[i] / len);  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

The mutant is **not detectable**, but is it **unproductive**?

Detectable vs. productive mutants (2)

Original program

```
public double getAvg(double[] nums) {  
    int len = nums.length;  
    double sum = 0;  
    double avg = 0;  
  
    for (int i = 0; i < len; ++i) {  
        avg = avg + (nums[i] / len);  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Mutant

```
public double getAvg(double[] nums) {  
    int len = nums.length;  
    double sum = 0;  
    double avg = 0;  
  
    for (int i = 0; i < len; ++i) {  
        avg = avg * (nums[i] / len);  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

The mutant is **not detectable**, but is it **unproductive**? **No!**

Detectable vs. productive mutants (3)

Original program

```
...  
Set cache = new HashSet(a * b);  
...
```

Mutant

```
...  
Set cache = new HashSet(a + b);  
...
```

Is the mutant **detectable**?

Detectable vs. productive mutants (3)

Original program

```
...  
Set cache = new HashSet(a * b);  
...
```

Mutant

```
...  
Set cache = new HashSet(a + b);  
...
```

The mutant is **detectable**, but is it **productive**?

Detectable vs. productive mutants (3)

Original program

```
...  
Set cache = new HashSet(a * b);  
...
```

Mutant

```
...  
Set cache = new HashSet(a + b);  
...
```

The mutant is **detectable**, but is it **productive**? No!

Mutation-based testing: mutant subsumption

Mutant subsumption

Mutant	MutOp	Tests			
		t_1	t_2	t_3	t_4
$m_1: < \mapsto !=$		○	○	○	○
$m_2: < \mapsto ==$		○	●	○	●
$m_3: < \mapsto <=$		★	★	★	★
$m_4: < \mapsto >$		○	●	○	○
$m_5: < \mapsto >=$		○	●	○	●
$m_6: < \mapsto \text{true}$		★	★	★	★
$m_7: < \mapsto \text{false}$		○	●	○	○
$m_8: < \mapsto !=$		●	○	○	○
$m_9: < \mapsto ==$		○	●	○	○
$m_{10}: < \mapsto <=$		○	○	○	○
$m_{11}: < \mapsto >$		●	●	○	○
$m_{12}: < \mapsto >=$		●	●	○	○
$m_{13}: < \mapsto \text{true}$		●	○	○	○
$m_{14}: < \mapsto \text{false}$		○	●	○	○

Mutant detected
(assertion)

Mutant detected
(exception)

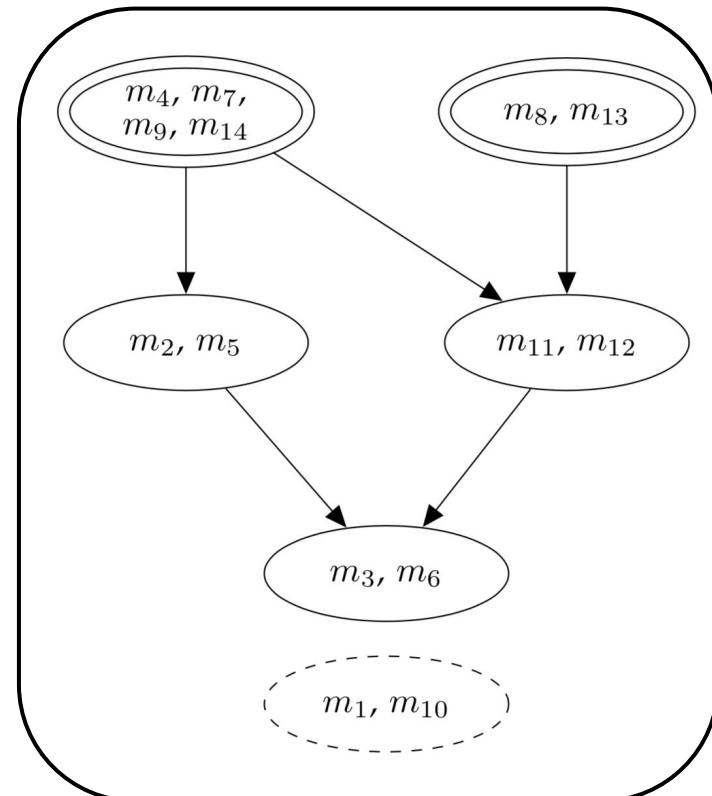
Mutant not detected

DMSG: Dynamic Mutant Subsumption Graph

Mutant	MutOp	Tests			
		t_1	t_2	t_3	t_4
$m_1: < \mapsto !=$		○	○	○	○
$m_2: < \mapsto ==$		○	●	○	●
$m_3: < \mapsto <=$		★	★	★	★
$m_4: < \mapsto >$		○	●	○	○
$m_5: < \mapsto >=$		○	●	○	●
$m_6: < \mapsto \text{true}$		★	★	★	★
$m_7: < \mapsto \text{false}$		○	●	○	○
$m_8: < \mapsto !=$		●	○	○	○
$m_9: < \mapsto ==$		○	●	○	○
$m_{10}: < \mapsto <=$		○	○	○	○
$m_{11}: < \mapsto >$		●	●	○	○
$m_{12}: < \mapsto >=$		●	●	○	○
$m_{13}: < \mapsto \text{true}$		●	○	○	○
$m_{14}: < \mapsto \text{false}$		○	●	○	○



DMSG



Prioritizing Mutants to Guide Mutation Testing ([Reading 2](#))

Coverage-based vs. mutation-based testing

See dedicated [Slides \(4 pages\)](#).