CSE P 590
Beyond Coverage: Modern Testing and Debugging
Spring 2019

Mutation-based testing

April 30, 2019
Today

- Mid-term assessment
- Recap: coverage-based testing
- Introduction to mutation-based testing
- Project presentation/discussion
Recap: structural code coverage

Entry point

\[ a == \text{null} \lor a.\text{length} == 0 \]

\[ \text{sum} = 0 \]
\[ i = 0 \]

\[ i < a.\text{length} \]

\[ \text{num} = a[i] \]

\[ \text{num} < 0 \]

\[ \text{sum} += \text{num} \]

\[ \text{sum} -= \text{num} \]

\[ ++i \]

return \( \text{sum} / a.\text{length} \)

throw new IllegalArgumentException("Array a must not be null or empty!")

Exceptional exit

Normal exit
Recap: statement coverage

Entry point

\[ a == \text{null} \text{||} a.\text{length} == 0 \]

false

sum = 0
i = 0

\[ i < a.\text{length} \]

false

return sum/a.\text{length}

true

num = a[i]

\[ \text{num} < 0 \]

false

sum += num

true

sum -= num

++i

throw new IllegalArgumentException("Array a must not be null or empty!")

Exceptional exit

Normal exit
Recap: decision coverage

Entry point

a==null || a.length==0

true

throw new IllegalArgumentException("Array a must not be null or empty!")

Exceptional exit

false

sum = 0
i = 0

false

Normal exit

true

i<a.length

false

return sum/a.length

true

num = a[i]

false

num < 0

false

sum += num

true

sum -= num

++i

false

sum += num
Recap: condition coverage

Entry point

\( a == \text{null} \lor a.\text{length} == 0 \)

true

throw new IllegalArgumentException("Array a must not be null or empty!")

Exceptional exit

false

sum = 0
i = 0

false

\( i < a.\text{length} \)

true

num = a[i]

false

num < 0

true

sum -= num

false

sum += num

true

return sum/a.\text{length}

Normal exit
Recap: subsumption relationships

Given two coverage criteria A and B,
A subsumes B iff satisfying A implies satisfying B

- Subsumption relationships:
  - Statement coverage does not subsume decision coverage
  - Decision coverage subsumes statement coverage
  - Decision coverage does not subsume condition coverage
  - Condition coverage does not subsume decision coverage
Recap: subsumption relationships

Given two coverage criteria A and B, A subsumes B iff satisfying A implies satisfying B

Subsumption relationships:
- Statement coverage does not subsume decision coverage
- Decision coverage subsumes statement coverage
- Decision coverage does not subsume condition coverage
- Condition coverage does not subsume decision coverage

Does decision coverage subsume statement coverage in a sequential method (i.e., one without any decisions)?
Code coverage: advantages

- Code coverage is easy to compute and visualize.
- Code coverage is a simple objective function.
- Code coverage has an intuitive interpretation.
Code coverage: advantages

- Code coverage is easy to compute and visualize.
- Code coverage is a simple objective function.
- Code coverage has an intuitive interpretation.

But, does coverage ensure effective testing?
Code coverage: drawbacks

- Code coverage does not require test assertions.
- Not all statements etc. are equally important.

Are there any alternatives?
Mutation testing: overview

Program

Test suite
Mutation testing: overview

Program

Generate mutants

Mutants

Test suite
Mutation testing: overview

```
public float avg(float[] data) {
    float sum = 0;
    for (float num : data) {
        sum += num;
    }
    return sum / data.length;
}
```

```
public float avg(float[] data) {
    float sum = 1;
    for (float num : data) {
        sum += num;
    }
    return sum / data.length;
}
```

Each mutant contains one small syntactic change
Mutation testing: overview

```
public float avg(float[] data) {
    float sum = 0;
    for (float num : data) {
        sum += num;
    }
    return sum / data.length;
}
```

```
public float avg(float[] data) {
    float sum = 0;
    for (float num : data) {
    }
    return sum / data.length;
}
```
Mutation testing: overview

```java
public float avg(float[] data) {
    float sum = 0;
    for (float num : data) {
        sum += num;
    }
    return sum / data.length;
}
```

```java
public float avg(float[] data) {
    public float avg(float[] data) {
    public float avg(float[] data) {
    float sum = 0;
    for (float num : data) {
        sum += num;
    }
    } return sum * data.length;
    }
    }
    }
```
Mutation testing: overview
**Assumption**: Mutant detection rate is a useful proxy for software testing. 
Mutation testing: example

Find a test case that detects the following mutant (i.e., passes on the original program but fails on the mutant)

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

Mutant:
```java
public int min(int a, int b) {
    return a;
}
```
Mutation testing: example

Find a test case that detects the following mutant (i.e., passes on the original program but fails on the mutant)

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

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Original</th>
<th>Mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Mutation testing: another example

Find a test case that detects the following mutant (i.e., passes on the original program but fails on the mutant)

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

Mutant:
```java
public int min(int a, int b) {
    return b;
}
```
Mutation testing: another example

Find a test case that detects the following mutant (i.e., passes on the original program but fails on the mutant)

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

Mutant:
public int min(int a, int b) {
    return b;
}
Mutation testing: yet another example

Find a test case that detects the following mutant (i.e., passes on the original program but fails on the mutant)

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

Mutant:
public int min(int a, int b) {
    return a >= b ? a : b;
}
Mutation testing: yet another example

Find a test case that detects the following mutant (i.e., passes on the original program but fails on the mutant)

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

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>Original</th>
<th>Mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

**Mutant:**
```java
public int min(int a, int b) {
    return a >= b ? a : b;
}
```
Mutation testing: last example (I promise)

Find a test case that detects the following mutant (i.e., passes on the original program but fails on the mutant)

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

Mutant:
public int min(int a, int b) {
    return a <= b ? a : b;
}
Mutation testing: last example (I promise)

Find a test case that detects the following mutant (i.e., passes on the original program but fails on the mutant)

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

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

There is no such test that can detect the mutant...

The mutant is undetectable: it’s semantically equivalent to the original program!
Mutation testing: summary

Original program:
```java
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;`

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Original</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
# Mutation testing: summary

## Original program:
```java
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;`

### Table:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Original</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Mutation testing: challenges

- Redundant mutants
  - Inflate the mutant detection ratio
  - Hard to assess progress and remaining effort

- Equivalent mutants
  - Max mutant detection ratio $\neq 100\%$
  - Wastes resources (CPU and human time)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Original</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Mutation testing: live example

Let’s start with a coverage-adequate test suite

```java
package avg;

public class Avg {
    /*
     * Compute the average of the absolute values of an array of doubles
     */
    public double avgAbs(double ... numbers) {
        // We expect the array to be non-null and non-empty
        if (numbers == null || numbers.length == 0) {
            throw new IllegalArgumentException("Array numbers must not be null or empty!");
        }
        double sum = 0;
        for (int i=0; i<numbers.length; ++i) {
            double d = numbers[i];
            if (d < 0) {
                sum -= d;
            } else {
                sum += d;
            }
        }
        return sum/numbers.length;
    }
}
```