Program-based Mutation Testing

CS 4501 / 6501 Software Testing

[Ammann and Offutt, “Introduction to Software Testing,” Ch. 9.2]
Applying Syntax-Based Testing to Programs

- Test requirements are derived from the syntax of software artifacts
- Syntax-based criteria originated with programs and have been used mostly with program source code
- BNF criteria are most commonly used to test compilers
  - Use BNF criteria to generate programs to test all language features that compilers must process
- Mutation testing criteria are most commonly used for unit testing and integration testing
Instantiating Grammar-Based Testing

Grammar-Based Testing

Program-based

- Grammar
- String mutation
  - Program mutation
  - Valid strings
  - Mutants are not tests
  - Must kill mutants

- Compiler testing
- Valid and invalid strings

Integration

- String mutation
  - Test how classes interact
  - Valid strings
  - Mutants are not tests
  - Must kill mutants

  - Includes OO

Model-Based

- String mutation
  - FSMs
  - Model checking
  - Valid strings
  - Traces are tests

Input-Based

- Grammar
- String mutation
  - Input validation testing
  - XML and others
  - Invalid strings
  - No ground strings
  - Mutants are tests

- Input validation testing
- XML and others
- Valid strings
Mutation Testing

- Inject changes into programs
- Strongest testing criterion
- Effective criterion for designing and evaluating tests
- Applied to C, C++, Java, JavaScript, Java EE, PHP, Angular, SQL, Android, spreadsheet, policy, ...

**Premise:**
If the software has a fault, there usually are some mutants that can only be **killed** by a **test** that also detects that fault.

**Kill:**
The test makes the output of the original program **different** from the output of the mutant
Mutation Testing

**Effective tests**

Original Program

```java
int lastZero (int[] x) {
    for (int i = x.length - 1; i >= 0; i--) {
        if (x[i] == 0)
            return i;
    }
    return -1;
}
```

**Find last index of zero**

Test \( x = \{0, 1, 2\} \):

- Input: \( x = \{1, 1, 2\} \);
- Output original: -1
- Output mutant: -1

Test \( x = \{0, 1, 2\} \):

- Input: \( x = \{0, 1, 2\} \);
- Output original: 0
- Output mutant: -1

**Ineffective tests**

- Test \( x = \{0, 1, 2\} \);
- Very effective at exploring the boundary case

[Thanks to Professor Lin Deng, Towson University]
Mutation Testing

Mutation operators
- Rules that specify how to modify the code (mutate)
- Well designed operators result in power tests

Mutation operators do one of two tasks
- Mimic typical programmer mistakes
- Encourage common test heuristics

We use mutation testing to
- Help testers design high quality tests
- Evaluate the quality of existing tests

Mutation scores \(= \frac{\# \text{ mutants killed}}{\# \text{ non-equivalent mutants}}\)

Mutants are not tests, but used to find tests

Mutants are not tests, but used to find tests

Must be valid strings (compilable)
Killing Mutants

Given a mutant \( m \in M \) for a ground string program \( P \) and a test \( t \), \( t \) is said to kill \( m \) if and only if the output of \( t \) on \( P \) is different from the output of \( t \) on \( m \).

- The quality of tests depends on mutation operators
- Different operators must be defined for different goals (and possibly for different programming languages)
- Testers can keep adding tests until all mutants have been killed
  - A mutant is killed if there is a test case for which the test results are different from the original program

Killing mutants \( \approx \) exposing faults
Categories of Mutants

• Dead mutant
  • A test case has killed it
  • The fault that a dead mutant represents will be detected by the same test that killed it

• Uncompilable mutant
  • Syntactically illegal
  • Should not be generated or should be immediately discarded

• Trivial mutant
  • Almost every test can kill it

• (Functionally) equivalent mutant
  • No test can kill it (same behavior or output as original, for all inputs)
  • Infeasible test requirements
Example: Program Mutation

Original method

```java
public static int numZero(int[] x) {
    int count = 0;
    for (int i=0; i<x.length; i++)
    {
        if (x[i] == 0)
            count++;
    }
    return count;
}
```

A fault is introduced by mutating the code

Mutant

```java
public static int numZero(int[] x) {
    int count = 0;
    \[\text{△} \text{for (int } i=1; i<x.length; i++) \]
    {
        if (x[i] == 0)
            count++;
    }
    return count;
}
```
Example: Program Mutation

```java
public static int numZero(int[] x) {
    int count = 0;
    for (int i=1; i<x.length; i++) {
        if (x[i] == 0)
            count++;
    }
    return count;
}
```

• i=1 is a mutation of i=0

• The code obtained by changing i=0 to i=1 is called a mutant of numZero

• A test kills the mutant if the mutant yields different outputs from the original code

• Consider t1 = {1, 0, 0}
  • Original returns 2, mutant returns 2, the mutant is not killed

• Consider t2 = {0, 1, 0}
  • Original returns 2, mutant returns 1, the mutant is killed
Example 2

Each mutated statement represents a separate program

Original method

public static int \texttt{min}(\texttt{int} x, \texttt{int} y) 
\begin{cases} 
\text{int } v; \\
\text{if } (x < y) \vspace{-4.5pt} \\
\hspace{10pt} v = x; \\
\text{else} \vspace{-4.5pt} \\
\hspace{10pt} v = y; \\
\text{return } v; 
\end{cases}

mutant1

\begin{cases} 
\text{int } v; \\
\text{\textbullet if } (x \geq y) \\
\hspace{10pt} v = x; \\
\text{else} \\
\hspace{10pt} v = y; \\
\text{return } v; 
\end{cases}

mutant2

\begin{cases} 
\text{int } v; \\
\text{\textbullet if } (x \leq y) \\
\hspace{10pt} v = x; \\
\text{else} \\
\hspace{10pt} v = y; \\
\text{return } v; 
\end{cases}

mutant3

\begin{cases} 
\text{int } v; \\
\text{if } (x < y) \vspace{-4.5pt} \\
\hspace{10pt} v = x; \\
\text{else} \vspace{-4.5pt} \\
\hspace{10pt} v = -y; \\
\text{return } v; 
\end{cases}
Example 2

Consider the following tests

• $t_1 = \min(0, 0)$
• $t_2 = \min(0, 1)$
• $t_3 = \min(1, 0)$

Which mutants will be killed by which tests?

```java
public static int min(int x, int y) {
    int v;
    // mutant1
    if (x >= y)
        v = x;
    else
        v = y;
    return v;
}
```

```java
public static int min(int x, int y) {
    int v;
    // mutant2
    if (x <= y)
        v = x;
    else
        v = y;
    return v;
}
```

```java
public static int min(int x, int y) {
    int v;
    // mutant3
    if (x < y)
        v = x;
    else
        v = -y;
    return v;
}
```
Example 2

| t1 | 0 | 0 | 0 | 0 | 0 | 0 |
| t2 | 0 | 1 | 0 | 1 | 0 | 0 |
| t3 | 1 | 0 | 0 | 1 | 0 | 0 |

- t1 kills none of the mutants
- t2 kills m1
- t3 kills m1

```java
public static int min(int x, int y) {
    int v;
    if (x <= y)
        v = x;
    else
        v = y;
    return v;
}
```

**mutant2**

```java
public static int min(int x, int y) {
    int v;
    if (x >= y)
        v = x;
    else
        v = y;
    return v;
}
```

**mutant3**

```java
public static int min(int x, int y) {
    int v;
    if (x < y)
        v = x;
    else
        v = -y;
    return v;
}
```

Equivalent mutant
Example 3

Original method

```java
public static int min(int x, int y)
{
    int v;
    if (x < y)
        v = x;
    else
        v = y;
    return v;
}
```

With embedded mutants

```java
public static int min(int x, int y)
{
    int v;
    if (x < y)
        v = x;
    else
        v = y;
    return v;
}
```

- Mutant 1: Replace operator
- Mutant 2: Replace one variable with another
- Mutant 3: Immediate runtime failure if y == 0, else does nothing
- Mutant 4: Force the tester to create tests that cause every variable and expression to have the value of zero

```java
public static int min(int x, int y)
{
    int v;
    if (x < y)
        v = x;
    else
        v = y;
    return v;
}
```
Mutation Coverage

**Mutation Coverage (MC):** For each $m \in M$, TR contains exactly one requirement, to kill $m$.

- The RIPR model
  - **Reachability:** the test causes the faulty (mutated) statement to be reached
  - **Infection:** the test causes the faulty statement to result in an incorrect state
  - **Propagation:** the incorrect state propagates to incorrect output
  - **Revealability:** the tester must observe part of the incorrect output

- The RIPR model leads to two variants of mutation coverage: **Strong** mutation and **Weak** mutation
1. Strong Mutation Coverage

**Strong Mutation Coverage (SMC):** For each \( m \in M \), TR contains exactly one requirement, to strongly kill \( m \).

- Require *reachability, infection, and propagation*
- Output of running a test set on the original program is different from the output of running the same test set on a mutant
2. Weak Mutation Coverage

**Weak Mutation Coverage (WMC):** For each $m \in M$, TR contains exactly one requirement, to weakly kill $m$.

- Require **reachability** and **infection**, not propagation
  - Check internal state immediately after execution of the mutated statement
  - If the state is incorrect, the mutant is killed
- Require less analysis
- A few mutants can be killed under weak mutation but not under strong mutation (**no propagation**)
  - Incorrect state does not always propagate to the output
- Studies have found that test sets that weakly kill all mutants also strongly kill most mutants
Example (Mutant 1)

Consider mutant 1

Reachability: true
Infection: \( x \neq y \)
Propagation: \( (y < x) = false \)

Full test specification:
\[
true \land (x\neq y) \land ((y<x)=false) \\
\equiv (x\neq y) \land (y\geq x) \\
\equiv (y>x)
\]

Test case value:
\( (x = 3, y = 5) \) strongly kill, weekly kill mutant 1
\( (x = 5, y = 3) \) weakly kill, but not strongly kill

```java
public static int min(int x, int y)
{
    int v;
    v = x;
    v = y;
    if (y < x)
        if (y > x)
            if (y < v)
                { v = y;
                    v = x;
                }
    return v;
}
```
Example (Mutant 3)

Consider mutant 3

Reachability: true
Infection: \((y < x) \neq (y < v)\)

No input can kill this mutant ... "Equivalent mutant"

```
public static int min(int x, int y)
{
    int v;
    \[\boxed{v = x;}\]
    v = y;

    \[\Delta 1\]

    if (y < x)
    { \[\boxed{v = y;}\] 

    \[\Delta 2\]

    if (y > x)
    { \[\boxed{v = y;}\] 

    \[\Delta 3\]

    if (y < v)
    { \[\boxed{v = y;}\] 

    \[\Delta 4\]

    v = x;

    \}

    \}

    return v;

}
Strong vs. Weak Mutation

```
public static boolean isEven(int x) {
    if (x < 0)
        x = 0 - x;

    x = 0;
    if ((float)(x/2) == ((float)x)/2.0)
        return true;
    else
        return false;
}
```

Reachability: \( x < 0 \)
Infection: \( x \neq 0 \)

Propagation:

\[
( (\text{float})(0-x)/2) == ((\text{float})(0-x))/2.0 ) != ( (\text{float})(0/2) == ((\text{float})0)/2.0 )
\]

The only value of \( x \) that will satisfy this condition is \( x \) must not be even.

Given a test \((x = -6)\)
Kills the mutant under weak mutation, but not under strong mutation.

To strongly kill, propagation requires \( x \) must be an odd and negative integer.
Designing Mutation Operators

Mutation Operators do one of two tasks:

- Mimic typical programmer mistakes
- Encourage common test heuristics

Researchers design many operators, then experimentally

- Select the most useful operators
- Remove the redundant operators

Effective Mutation Operators

- If tests that are created specifically to kill mutants created by a collection of mutation operators $O = \{o_1, o_2, \ldots\}$ also kill mutants created by all remaining mutation operators with very high probability, then $O$ defines an effective set of mutation operators.
2. **AOR — Arithmetic Operator Replacement:**

Each occurrence of one of the arithmetic operators +, −, *, /, and % is replaced by each of the other operators. In addition, each is replaced by the special mutation operators `leftOp`, and `rightOp`.

Example:

```
x = a + b;
△ 1 x = a * b;
△ 2 x = a % b;
△ 3 x = leftOp(a + b); // x = a
△ 4 x = rightOp(a + b); // x = b
```

3. ROR — Relational Operator Replacement:

Each occurrence of one of the relational operators (\(<, \leq, >, \geq, =, \neq\) is replaced by each of the other operators and by falseOp and trueOp.

Example:

```
if (m > n)
\[\Delta 1\] if (m >= n)
\[\Delta 2\] if (m == n)
\[\Delta 3\] if (m != n)
\[\Delta 4\] if (trueOp(m > n))  // if (true)
\[\Delta 5\] if (falseOp(m > n))  // if (false)
```
**Example:**

**Mutation Ops for Java Program**

**SDL – Statement Deletion**

SDL deletes each executable statement by commenting them out. It does not delete declarations.

**General statement deletion**

<table>
<thead>
<tr>
<th>Original method</th>
<th>Mutant 1</th>
<th>Mutant 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public void test()</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>{</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>int a, b;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a = 1;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>b = 2;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>public void test()</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>{</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>int a, b;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>// a = 1;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>b = 2;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>public void test()</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>{</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>int a, b;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>// b = 2;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 1.** General statement deletion mutation operator

Example: Mutation Ops for Web Apps

FOB – FailOnBack

```html
<html>
<body onload="manipulatehistory()">
  <script src="failOnBack.js"></script>
<br...>
</body>
</html>
```

```javascript
function manipulatehistory()
{
  var currentpage = window.document.toString();
  var currenturl = window.location.href;
  var pageData = window.document.toString();

  // add a dummy url right before the current url
  history.replaceState(pageData, "dummyurl", "failonback.html");
  history.pushState(currentpage, "currenturl", currenturl);
}

// update the page content
window.addEventListener('popstate', function(event) {
  window.location.reload();
});
```

[https://eric.ed.gov/?id=ED578113](https://eric.ed.gov/?id=ED578113)
## WSCR – Scope replacement

```
<html>
  ...
  <jsp:useBean id="id" scope="page" class="class" />
  <jsp:useBean id="id" scope="session" class="class" />
  ...
</html>
```

## WSIR – Session initialization replacement

```java
public class logout extends HttpServlet {
  public void doGet(...) {
    session = request.getSession(true);
    session = request.getSession(false);
  }
}
```

## WSAD – Session setAttribute deletion

```java
public class logout extends HttpServlet {
  public void doGet(...) {
    session.setAttribute(attr, value);
    // session.setAttribute(attr, value);
    ...
  }
}
```

[https://eric.ed.gov/?id=ED578113](https://eric.ed.gov/?id=ED578113)

**Example: Mutation Ops for Web Apps**
Example: Mutation Ops for Android Apps

- **OnClick Event Replacement (ECR)**
  - Replaces event handlers with other compatible handler

```java
mPrepUp.setOnClickListener (new OnClickListener() {
    public void onClick (View v) {
        decrementPrepTime ();
    }
});
mPrepDown.setOnClickListener (new OnClickListener() {
    public void onClick (View v) {
        decrementPrepTime ();
    }
});
```

- **OnTouch Event Replacement (ETR)**
  - Replaces OnTouch events, similar to ECR

Summary

• Mutation is widely considered the strongest test criterion

• First-order mutation due to the two assumptions
  • Competent programmers
  • Coupling effect

• Mutation creates the most test requirements
  • # test requirements = # mutants
  • Most expensive

• To improve the test process, use selective mutation operators

• Mutation testing is very difficult to apply by hand

• Mutation subsumes other criteria by including specific mutation operators

• Mutation can be applied to various software artifacts, languages, and frameworks with different implementation and specific definition of mutation operators