Intro to Software Testing

chapter 9

Syntax Coverage & Mutation Testing

Dr. Brittany Johnson-Matthews
(Dr. B for short)

https://go.gmu.edu/SWE637
Adapted from slides by Jeff Offutt and Bob Kurtz
Logic Coverage

Structures for Modeling Software

- Input Space
- Graphs
- Logic
- Syntax

Applied to
- Source
- Design
- Specs
- Use Cases

Applied to
- Source
- FSMs
- Specs
- DNF

Applied to
- Source
- Models
- Integration
- Input
Syntax-based testing

Software artifacts often have syntax rules

We can use two approaches when developing tests based on syntax

Cover the syntax in some way

Violate the syntax (to create invalid test cases)
Fuzzing

One common use of syntax manipulation is fuzzing or fuzz testing

The objective is to provide inputs to the system that are “correct enough” to pass any input validation, but “incorrect enough” to expose defects and/or unexpected behaviors

Fuzzing may selectively modify the input grammar, or may use heuristics based on past experience, or may simply make randomized changes
Violating the Syntax - Heartbleed

https://xkcd.com/1354/
Violating the Syntax - Heartbleed

https://xkcd.com/1354/
Defining Mutation

**Mutation testing** is a generalization of fuzzing.

In mutation testing, we

1. Take a *ground string* (a syntactically valid original artifact),
2. apply a *mutation operator* (a rule that governs how to modify the artifact),
3. to generate a *mutant* (a modified artifact) that is either in the grammar (valid) or very close to being in the grammar, then
4. determine whether the mutant exhibits different behavior than the ground string, which *detects* or *kills* the mutant
Mutation Testing

Mutation testing can be applied to

- Input grammars (SQL, HTML, XML, etc.)
- Modeling languages (state charts, activity diagrams, etc.)
- Specification languages (Z, SMV, algebraic specifications, DNF)

*Program source code*

This is the type of mutation testing we’ll be talking about
Program Mutation

This is the original and most widely-known use of mutation, and is generally applied to individual classes or methods.

Mutation operators are applied to the ground string (the original program) to produce a set of mutants (modified programs).

The resulting mutants are not tests, but can be used to develop or evaluate tests.
What's Mutation Testing For?

This is the original and most widely-known use of mutation, and is generally applied to individual classes or methods.

Mutation operators are applied to the ground string (the original program) to produce a set of mutants (modified programs).

The resulting mutants are not tests, but can be used to develop or evaluate tests.
What's mutation testing for?

Mutation testing can be used in two complementary ways:

1. **Test development**: write tests to kill mutants
   
   This is how software developers use mutation testing (when they use it at all); the leading tool is probably PIT (https://pitest.org) though Google has established their own in-house capability.

2. **Test evaluation**: given a set of tests developed using some other criteria, how complete are those tests?
   
   This is how software researchers tend to use mutation testing.
Why does mutation work?

**Competent Programmer Hypothesis**: programmers are generally competent and tend to write programs that are nearly correct.

The small changes to programs introduced by mutation testing are considered to be reasonable approximations for the types of errors inadvertently injected by engineers.
Why does mutation work?

**Coupling Hypothesis**: complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults.

The faults generated by mutation testing are useful proxies for actual faults.
Categories of Mutants

*Live*: a mutant that has not been killed by a test

*Killed* (or *dead*): a mutant that has been killed by a test (its behavior is different than the original)

*Stillborn*: a syntactically invalid mutant that cannot be compiled or executed

*Trivial*: a mutant that is killed by every test that reaches the mutation, usually by exception

*Equivalent*: a mutant that behaves identically to the ground string, such that no test can kill it

This seems counter-intuitive but is quite common
Mutation Coverage

**Mutation Coverage (MC)** – For each mutant $m$ in the set of mutants $M$, $TR$ contains exactly one requirement: to kill $m$.

A test $t$ **kills** a mutant $m$ if and only if the behavior of $m$ while executing $t$ differs from the behavior of the ground string while executing $t$.

The mutation coverage metric is based on the proportion of mutants killed, also known as the *mutation score*. 
Mutation Example

A test for \( m_1 \):

```java
assertEquals(2, max(1, 2));
```

Mutant \( m_1 \) is killed (returns 1 instead of 2)
Mutation Example

A test for $m_2$:
```java
assertEquals(2, max(1, 2));
```

Mutant $m_2$ is killed (returns 1 instead of 2)
Mutation Example

A test for $m_3$:
```java
assertEquals(2, max(1, 2));
```

Mutant $m_3$ is *killed* and *trivial* – it is killed by any test that reaches it.
Mutation Example

// Ground string
// (original program)
int max (int i, int j) {
    if (i >= j)
        return i;
    else
        return j;
}

// Mutant m2
// (modified program)
int max (int i, int j) {
    if (i > j)
        return i;
    else
        return j;
}

A test for m₄:
    assertEquals(2, max(1, 2));

Mutant m₄ is equivalent – no test exists that can kill it
Mutation Notation

Mutants are often shown in a single consolidated listing, with deltas marked:

```c
// Ground string
// (original program)
int max (int i, int j)
{
    if (i >= j)
        return i;
    else
        return j;
}
```

```c
// Mutant m4
// (modified program)
int max (int i, int j)
{
    if (i >= j)
        return i;
    Δ1 if (i <= j)
    Δ4 if (i > j)
        return i;
    Δ3 trap();
    else
        return j;
    Δ2 return i;
}
```
Mutation Coverage Revisited

**Definition**

**Mutation Coverage (MC)** – For each mutant $m$ in the set of mutants $M$, $TR$ contains exactly one requirement: to kill $m$.

Consider the RIPR model

- Program execution must *reach* the fault
- The fault must *infect* the program state with an error
- The error must *propagate* to an output
- The error must be *revealed* to the tester

This suggests two definitions for *kill*
Strong and Weak Mutation

**Strong mutation:** given a mutant $m \in M$ for a program $P$ and a test $t$, $t$ strongly kills $m$ if and only if the output of $t$ on $P$ is different from the output of $t$ on $m$.

Strong mutation requires reachability, infection, propagation, and revealability.

**Weak mutation:** given a mutant $m \in M$ that modifies a location $l$ in program $P$ and a test $t$, $t$ weakly kills $m$ if and only if the state of execution of $t$ on $P$ is different from the state of execution of $t$ on $m$ immediately after $l$.

Weak mutation requires only reachability and infection.
Strong and Weak Mutation

It is easier to weakly kill mutants than to strongly kill them. However, it can be difficult to determine whether a mutant has been weakly killed.

Some mutants can be weakly killed but not strongly killed (the error does not propagate or is not revealed).

Studies have found that tests that weakly kill mutants also tend to strongly kill them.
Weak vs. Strong Example

```java
boolean isEven (int i)
{
    if (i < 0)
        i = 0 - i;
    i = 0 + i;
    if (i == ((i/2)*2))
        return true;
    else
        return false;
}
```

Given $t = \text{assertTrue(isEven(-4))}$

For $P$, $i=4$

For $m$, $i=-4$
thus $t$ weakly kills $m$

$P$ and $m$ both return true, so $t$ does not strongly kill $m$
Mutation Test Development

Begin

P

Mutate P

m1-n

Write a test t to kill mi

Modify P

Select another mi

End

Done?

Error Found?

Yes

No

Yes

No

Yes

No

No
Mutation Test Development

1. Begin
2. Take the initial program \( P \) and mutate it to produce a set of mutants \( m_1 \ldots m_N \)
3. Modify \( P \)
4. Mutate \( P \)
5. Write a test \( t \) to kill \( m_i \)
6. Error Found?
   - Yes
   - No
7. Did the test work
   - Yes
   - No
8. Modify \( P \)
Mutation Test Development

Begin

\[ P \]

Mutate \( P \)

\[ \text{Select another } \] \[ m_{1-n} \]

Write a test \( t \) to kill \( m_i \)

Select some mutant \( m_i \) and write a test to kill it (or determine it is equivalent)

End

Done? Yes

Error Found? No

Yes
Mutation Test Development

1. Begin

2. Mutate $P$ to create mutant $m_{1-n}$

3. Write a test $t$ to kill $m_i$

4. Did analysis of the mutant and test indicate an error in $P$? (Yes/No)

5. Modify $P$

6. Did analysis of the mutant and test indicate an error in $P$? (Yes/No)

7. End

8. Done? (Yes/No)

9. Error Found? (Yes/No)
Mutation Test Development

Begin

P

Mutate P

m₁₋ₙ

Write a test t to kill mᵢ

Modify P

If yes, then modify P to correct the error and re-mutate P'.

End

Done?

Yes

No

Error Found?

Yes

End

Yes

No
Mutation Test Development

Begin

If no, then assess completeness of testing, usually by a mutation score threshold

$P$

Modify $P$

$m_{1-n}$

Select another $m_i$

Write a test $t$ to kill $m_i$

Done? Yes

End

Error Found? Yes

No

$P$

Yes

No

Done?

modify

Select another $m_i$

Yes

Error Found?
Mutation Test Development

Begin

Mutate

$P$

Write a test $t$ to kill $m_i$

$m_{1-n}$

Select another $m_i$

If not done, then select another mutant $m_i$ and repeat the process

End

Done?

Yes

No

Error Found?

Yes

No

Yes

No

No
Mutation Test Development

This process is **extremely labor-intensive** and thus expensive.

The outcome of the process is a very strong set of tests, *if the mutation operators are well-designed*. 
Designing Mutation Operators

A good mutation operator

- Creates mutants that are similar to programmer errors
- Creates mutants that tend to elicit effective tests

Researchers design lots of operators, then empirically determine which are effective

- If tests created to kill mutants generated by one operator also tend to kill mutants developed by other operators, than that operator is effective
Some Java Mutation Operators

AOD – arithmetic operator deletion
AOI – arithmetic operator insertion
AOR – arithmetic operator replacement
COD – conditional operator deletion
COI – conditional operator insertion
COR – conditional operator replacement
LOD – logical operator deletion
LOI – logical operator insertion
LOR – logical operator replacement
ROR – relational operator replacement
SDL – statement deletion
SOR – shift operator replacement
Mutation Operator Examples

AOD – arithmetic operator deletion

\[ a = b + c \]
\[ \Delta 1 \quad a = b \rightarrow c \]
\[ \Delta 2 \quad a = b + c \]

AOI – arithmetic operator insertion

\[ a = b + c \]
\[ \Delta 1 \quad a = -b + c \]
\[ \Delta 2 \quad a = b + c++ \]

AOR – arithmetic operator replacement

\[ a = b + c \]
\[ \Delta 1 \quad a = b - c \]
\[ \Delta 2 \quad a = b \% c \]
Mutation Operator Examples

**COD** – conditional operator deletion

```
if (a && !b)
```

\[ \Delta_1 \] if \((a \&\& \neg b)\)

\[ \Delta_2 \] if \((a \&\& \neg b)\)

**COI** – conditional operator insertion

```
if (a && b)
```

\[ \Delta_1 \] if \((!(a \&\& b))\)

\[ \Delta_2 \] if \((a \&\& b \mid\mid true)\)

**COR** – conditional operator replacement

```
if (a && b)
```

\[ \Delta_1 \] if \((a \mid\mid b)\)

\[ \Delta_2 \] if \(false)\)
Mutation Operator Examples

**LOD** – logical operator deletion

\[
a = b \mid c
\]

\[\Delta 1 \quad a = b \cancel{\mid} c\]
\[\Delta 2 \quad a = b \cancel{\mid} c\]

**LOI** – logical operator insertion

\[
a = b \mid c
\]

\[\Delta 1 \quad a = \sim b \mid c\]
\[\Delta 2 \quad a = b \mid \sim c\]

**LOR** – logical operator replacement

\[
a = b \mid c
\]

\[\Delta 1 \quad a = b \& c\]
\[\Delta 2 \quad a = b \^ c\]
Mutation Operator Examples

**ROR** – relational operator replacement

```plaintext
if (a < b)
Δ1 if (a > b)
Δ2 if (true)
```

**SDL** – statement deletion

```plaintext
if (a && b) { c = true }
Δ1 if (a && b) { c = true }
Δ2 if (a && b) { c = true }
Δ3 if (a && b) { c = true }
```

**SOR** – shift operator replacement

```plaintext
a = b >> c
Δ1 a = b << c
Δ2 a = b >>> c
```
The Mutation Score Problem

Mutation score measures the coverage with respect to the mutation criterion

\[ MS = \frac{mutants_{killed}}{mutants_{total} - mutants_{equivalent}} \]

The problem is that we can’t know how many mutants are equivalent until we’ve evaluated all of them, thus we can’t know the mutation score until it’s 100%!
Summary

Mutation testing can be used to develop tests or to evaluate tests.

Mutation testing is very powerful, but very expensive. As a result, it currently remains primarily a research tool.