Topics for this Lecture

- Understand the basic ideas of fault (syntax)-based testing
- Understand mutation testing as one application of fault-based testing principles

Estimating Test Suite Quality

- How good are my tests?

- We'd like to judge effectiveness of a test suite in finding real faults, by measuring how well it finds seeded fake faults.

- I run my test suite on the programs with seeded bugs …
  - ... and the tests reveal 20 of the bugs
  - (the other 80 program copies do not fail)
- What can I infer about my test suite?

(syntax)-based testing

- Usually known as ***mutation testing***

- Idea: generate many syntactic mutants of the original program.

- Coverage: how many mutants does a test suite kill (detect)?

- Bit different kind of creature than the other coverage we have looked at.

- **Mutation analysis**: Assessing the quality of a test suite.

- **Mutation testing**: Improving the test suite using mutants

Oregon State UNIVERSITY

Mutation testing

- A mutant is a copy of a program with a mutation
- A mutation is a syntactic change (a seeded bug)
  - Example: **change** if (i < 0) **to** if (i <= 0)
- Mutants are rejected by a compiler are not valid mutants
  - Example: **change** if statement **to** switch statement
- Run test suite on all the mutant programs
- A mutant is killed if it fails on at least one test case
- If many mutants are killed, infer that the test suite is also effective at finding real bugs

Oregon State UNIVERSITY

Mutation testing

Test cases that exercise the PUT

A program under test (PUT)

```
int do_something(int x, int y)
{
  if(x < y)
    return x+y;
  else
    return x*y;
}
```

Program

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

Test

```
int do_something(int x, int y)
{
  if(x < y)
    return x-y;
  else
    return x*y;
}
```

Mutant

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

Test

A mutant version with a simple fault

Execute the same test. If the mutant passes the test, we need more tests

Fault-Based Adequacy Criteria

- Given a pogrom and a test suite ***T***, mutation analysis consist of the following:
  - **Select mutation operators**: If we are interested in specific classes of faults, we may select a set of mutation operators relevant to those faults.
  - **Generate mutants**: Mutants are generated mechanically by applying mutation operators to the original programs.
  - **Distinguish mutants**: Execute the original program and each generated mutant with the test cases in ***T.*** A mutant is killed when it can be distinguished from the original program.
  - Two kinds of mutants survive:
    1. ***Functionally equivalent*** to the original program: Cannot be killed.
    2. ***Killable***: Test cases are insufficient to kill the mutant. New test cases must be created.

- Mutation Score = $\dfrac{Killed\ Mutants}{Total\ Mutants - Equivalent\ mutants}$

Mutation operator

- **Operand Operators**:
  - Replace a single operand with another operand or constant.
    - Example: if (x >y) → if (5>y)

- **Expression Operators**:
  - Replace an operator or insert new operators.
  - Example: if (x >y) → if (x == y)

- **Statement Operators:**
  - Delete, replace, or move a statement
  - Example:
    - Replace a particular statement with return statement.
    - Delete the ***else*** part of the ***if-else*** statement
    - Move } one statement earlier and later

Equivalent Mutants

- Mutation = syntactic change
- The change might leave the semantics unchanged
- Equivalent mutants are hard to detect (undecidable problem)

<pre>int max(int[] values) {
    int r, i;
    r = 0;
    for(i = 1; i<values.length; i++) {
        if (values[i] > values[r])
            r = i;
    }
  return values[r];
}</pre>
	The original program that finds the maximum number in the array values.

<pre>int max(int[] values) {
    int r, i;
    r = 0;
    for(i = 0; i<values.length; i++) {
        if (values[i] > values[r])
            r = i;
    }
  return values[r];
}</pre>
	In this mutant, the loop starts from 0 instead of 1. This mutant is equivalent because it only introduces an additional comparison of the first element to itself - this cannot change the functional behavior.

<pre>int max(int[] values) {
    int r, i;
    r = 0;
    for(i = 1; i<values.length; i++) {
        if (values[i] >= values[r])
            r = i;
    }
  return values[r];
}</pre>
	This is another equivalent mutant: The value of the maximum stays the same regardless of whether the comparison is > or >=

OSU Oregon State UNIVERSITY

Mutation tools

- The **jumble** tool is available for download at http://jumble.sourceforge.net/
- The **µJava** (muJava) is a mutation system for Java programs and is available for download at https://cs.gmu.edu/~offutt/mujava/
- **PIT** is a mutation testing system for Java and is available for download http://pitest.org/
- The Major mutation framework for Java and is available for download http://mutation-testing.org/

How to use PIT from maven

- Add the plugin to dependency in your pom.xml

<dependency>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>LATEST</version>
</dependency>

- Runs PIT directly from the commandline
    mvn org.pitest:pitest-maven:mutationCoverage

- For more information see http://pitest.org/quickstart/maven/

Oregon State UNIVERSITY

References:

Ma, Yu-Seung, and Jeff Offutt. "Description of class mutation mutation operators for java." Electronics and Telecommunications Research Institute (2005).

https://www.st.cs.uni-saarland.de/edu/testingdebugging10/slides/10-MutationTesting.pdf

http://ix.cs.uoregon.edu/~michal/book/slides/pdf/PezzeYoung-Ch16-fault-based.pdf