EECS 4313
Software Engineering Testing

Topic 10:
Mutation Testing
- Breaking the application to test it

Zhen Ming (Jack) Jiang

Relevant Readings

- [Jorgensen] chapter 21

What is Mutation Testing?

- Mutation Testing is a testing technique that focuses on measuring the adequacy of test cases
- Mutation Testing is **NOT** a testing strategy like Boundary Value or Data Flow Testing. It does not outline test data selection criteria
- Mutation Testing should be used in conjunction with traditional testing techniques, not instead of them
- Goal:
  - Mimic (and hence test for) typical mistakes
  - Encode knowledge about specific kinds of effective tests in practice

Mutation Testing

- Faults are introduced into the program by creating many versions of the program called **mutants**
- Each mutant contains a single fault
- Test cases are applied to the original program and to the mutant program
- The goal is to cause the mutant program to fail, thus demonstrating the effectiveness of the test suite

Test Case Adequacy

- A test case is **adequate** if it is useful in detecting faults in a program.
- A test case can be shown to be adequate by finding at least one mutant program that generates a different output than does the original program for that test case.
- If the original program and all mutant programs generate the same output, the test case is **inadequate**.

Mutant Programs

- **Mutation testing** involves the creation of a set of mutant programs of the program being tested
- Each **mutant** differs from the original program by one mutation
- A **mutation** is a single syntactic change that is made to a program statement

Example Mutation

```
int max(int x, int y)
{
    int mx = x;
    if (x > y) {
        mx = x;
    } else {
        mx = y;
    }
    return mx;
}
```

```
int max(int x, int y)
{
    int mx = x;
    if (x < y) {
        mx = x;
    } else {
        mx = y;
    }
    return mx;
}
```

Mutation Operators (1)

- Operand Replacement Operators:
  - Replace a single operand with another operand or constant. E.g.,
    - if (5 > y) Replacing y by constant 5.
    - if (x > 5) Replacing x by y.
    - if (y > x) Replacing x and y with each other.
  - E.g., if all operators are {+,-,*,**,/} then the following expression a = b * (c - d) will generate 8 mutants:
    - 4 by replacing *
    - 4 by replacing -.

Mutation Operators (2)

- Expression Modification Operators:
  - Replace an operator or insert new operators. E.g.,
    - if (x == y)
      - if (x >= y) Replacing == by >=.
      - if (x == ++y) Inserting ++.

Mutation Operators (3)

- Statement Modification Operators:
  - Delete the else part of an if-else statement.
  - Delete the entire if-else statement.
  - Replace line 3 by a return statement.

Mutation Operators

- The Mothra mutation system (*A Fortran Language System for Mutation-Based Software Testing* by Offutt et al. 1987) for FORTRAN77 supports 22 mutation operators
  - Absolute value insertion
  - Constant for array reference replacement
  - GOTO label replacement
  - Statement deletion
  - Unary operator insertion
  - Logical connector replacement

Why Does Mutation Testing Work?

- The operators are limited to simple single syntactic changes on the basis of the competent programmer hypothesis
- The Competent Programmer Hypothesis
  - Programmers are generally very competent and do not create "random" programs.
  - For a given problem, a programmer, if mistaken, will create a program that is very close to a correct program.
  - An incorrect program can be created from a correct program by making some minor changes to the correct program.

Mutation Testing Costs

- The FORTRAN 77 version of the max() program generated 44 mutants using Mothra.
- Most efforts on mutation testing have focused on reducing its cost by reducing the number of mutants while maintaining the effectiveness of the technique.

Mutation Testing Algorithm

- Generate program test cases
- Run each test case against the original program
  - If the output is incorrect, the program must be modified and re-tested
  - If the output is correct go to the next step ...
- Construct mutants using a mutation testing tool
- Execute each test case against each alive mutant
  - If the output of the mutant differs from the output of the original program, the mutant is considered incorrect and is **killed**
    - **"Good test cases kill the mutants"**
  - Once we find a test case that kills a mutant, we can forget the mutant and keep the test case. The mutant is **dead**
- Two kinds of mutants survive:
  - **Functionally equivalent to the original program**: Cannot be killed
  - **Killable**: Test cases are insufficient to kill the mutant. New test cases must be created.

What test case can kill the mutant?

```
int foo(int x, int y)
{ // original
   if (x > 5) {
      return x + y;
   } else {
      return x;
   }
}
```

```
int foo(int x, int y)
{ // mutant
   if (x > 5) {
      return x - y;
   } else {
      return x;
   }
}
```

Some mutants can be uninteresting

- Three kinds of mutants are uninteresting:
  - **Stillborn**: such mutants cannot compile (or immediately crash)
  - **Trivial**: killed by almost any test case;
  - **Equivalent**: indistinguishable from original program

Mutants Example

```
1. int min(int A, int B)
2. { // original
3.    int minVal;
4.    minVal = A;
5.    if (B < A) {
6.       minVal = B;
7.    }
8.    return minVal;
9. }
```

```
1. int min(int A, int B)
2. { // mutant
3.    int minVal;
4.1   minVal = B;
3.    if (B < A) {
4.       minVal = B;
5.    }
6.    return minVal;
7. }
```

Replace one varirable with another

```
1. int min(int A, int B)
2. { // mutant
3.    int minVal;
4.    minVal = A;
5.1.  if (B < minVal) {
6.       minVal = B;
7.    }
8.    return minVal;
9. }
```

Replace one varirable with another

```
1. int min(int A, int B)
2. { // mutant
3.    int minVal;
4     minVal = A;
5.1.     if (B > A) {
6.       minVal = B;
7.    }
8.    return minVal;
9. }
```

Change operator

```
1. int min(int A, int B)
2. { // original
3.    int minVal;
4.    minVal = A;
5.    if (B < A) {
6.1.     minVal = A;
7.    }
8.    return minVal;
9. }
```

Replace one varirable with another

And many more ….

Example of equivalent mutant

- This is equivalent mutant, since A = minVal

```
int min(int A, int B)
{ // original
  int minVal;
  minVal = A;
  if (B < A) {
    minVal = B;
  }
  return minVal;
}
```

```
int min(int A, int B)
{ // mutant
  int minVal;
  minVal = A;
  if (B < minVal) {
    minVal = B;
  }
  return minVal;
}
```

<u>Replace one varirable with another</u>

Mutation Coverage Criteria

- Mutation Coverage (MC)
  - For each mutant m, test requirements (TR) contain a requirement to “kill m”
    - Mutation score is the percentage of mutants killed
- The **mutation score** for a set of test cases is the percentage of non-equivalent mutants killed by the test data
  - Mutation Score = 100 * D / (N - E)
    - D: Dead mutants
    - N: Number of mutants
    - E: Number of equivalent mutants
  - A set of test cases is mutation adequate if its mutation score is 100%.

Strong and weak mutation

- **Strong mutation**: a fault must be reachable, infect the state, and **propagate to output**
- **Weak mutation:** a fault which kills a mutant need only be reachable and infect the state

- Experiments show that weak and strong mutation require almost the same number of test cases to satisfy them

Strong Mutation vs. Weak Mutation

```
int min(int A, int B)
{ // original
   int minVal;
   minVal = A;
   if (B < A) {
     minVal = B;
   }
   return minVal;
}
```

```
int min(int A, int B)
{ // mutant
   int minVal;
   minVal = B;
   if (B < A) {
     minVal = B;
   }
   return minVal;
}
```

Replace one varirable with another

- Reachability: unavoidable
- Infection: need B != A
- Propagation: wrong minVal needs to return to the caller; that is we cannot execute the body of the if statement, so need B > A
- Condition for strongly killing mutation B > A
  - TC: (A=5, B=7), return 7 but expected 5
- Conditions for weakly killing mutation B!=A
  - TC: (A=8, B=2), return 2 and expected 2

Evaluation

- Theoretical and experimental results have shown that mutation testing is an effective approach to measuring the adequacy of test cases.
- The major drawback of mutation testing is the cost of generating the mutants and executing each test case against them.

PIT demo

Adapted from:
https://vimeo.com/105758362
http://blog.xebia.com/mutation-testing-how-good-are-your-unit-tests/

PIT Mutation Testing Tool

- Conditionals Boundary Mutator
- Negate Conditionals Mutator
- Remove Conditionals Mutator
- Math Mutator
- Increments Mutator
- Invert Negatives Mutator
- Inline Constant Mutator
- Return Values Mutator
- Void Method Calls Mutator
- Non Void Method Calls Mutator
- Constructor Calls Mutator
- Experimental Inline Constant Mutator
- Experimental Member Variable Mutator
- Experimental Switch Mutator

http://pitest.org/

PIT Configuration

- PIT can work with many IDE
- In this demo, we will demonstrate PIT with Eclipse
  - Install the PIT eclipse plugin from the Eclipse Marketplace (under the Help menu)

Run the test

- Can be
  - "Run as -> Junit", or
  - run as "Maven test"
- It should pass both tests
- Run PIT
  - Right click, "Run as -> PIT Mutation Test"
  - Once done click the PIT summary report

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
1	100%	4/4	75%	3/4

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
eecs4313.MutationDemo.eecs4313MutationDemo	1	100%	4/4	75%	3/4

Report generated by PIT 1.1.5

Fix the issue

- Uncomment the last method and re-run PIT mutation test, you should see the screen as shown below

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
1	100%	4/4	100%	4/4

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
eecs4313.MutationDemo.eecs4313MutationDemo	1	100%	4/4	100%	4/4

Report generated by PIT 1.1.5