Com S 417
Software Testing

Fall 2017 – Week 12, Lecture 20

November 7, 2017

Announcements

Topics

- Project teams and topics.
- GUI testing technology
- Profiler

Designing Tests: Two competing Ideas

Design tests to verify desired behavior

Design tests to find specific faults.

Most of what we've explored so far fits here: "behavior-driven"

Mutation testing is an example of fault-based testing.

By asking "what if interactions are of limited complexity", combinatorial testing is a little of both.

What if …?

- What if each SUT came with a set of variant copies, one variant for each for each possible instance of a particular type of fault?
  - for example, each variant replaced a particular occurrence of “=“ in the original with “-=“ in a copy.

Creating a mutant

Changing this instead, would create a different mutant

Original

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

Mutant

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x -= y;
    y = tmp;
  }

  return x;
}
```

+=, -=,
|=, ^=, ...

Fault-based coverage?

- How would having the set of variants affect how you measured the coverage of your test suite – at least for that fault?
  - You wouldn’t need a surrogate for the number of faults … just count the variants.
  - Your test suite coverage metric would be

$$\frac{number\ of\ variants\ failed}{number\ of\ variants}$$

Notice the Difference

We *not* attempting to measure how well the test suite

- covers the intended behavior of the SUT,
- covers the code or the code structure,
- or covers the inferred number of faults in the code.

We are only measuring how well the test suite covers the number of places where a particular kind of fault *might* have been injected.

Mutation Testing: Basic Ideas

In Mutation Testing:

1. We take a program and a test suite generated for that program (using other test techniques).
2. We create a number of similar programs (mutants), each differing from the original in one small way, i.e., each possessing a fault – E.g., replacing an addition operator by a multiplication operator .
3. The original test suite/data are then run on the mutants.
4. If test cases detect differences in mutants, then the mutants are said to be dead (killed), and the test set is considered adequate.

From Lionel Brand http://www.inf.ed.ac.uk/teaching/courses/st/2016-17/Mutest.pdf

Basic Ideas (cont'd)

- A mutant remains live either
  - because it is equivalent to the original program (functionally identical although syntactically different – called an equivalent mutant) or,
  - the test set is inadequate to kill the mutant
- In the latter case, the test data need to be augmented (by adding one or more new test cases) to kill the live mutant
- For the automated generation of mutants, we use mutation operators, that is predefined program modification rules (i.e., corresponding to a fault model)

From Lionel Brand http://www.inf.ed.ac.uk/teaching/courses/st/2016-17/Mutest.pdf

Some Definitions (from Ammann and Offutt)

Ground String

- A string that is in the grammar.

Mutation Operator

- A rule that specifies syntactic variations of strings generated from a grammar.

Mutant

- The result of one application of a mutation operator.

Mutation Operator:

- A rule to derive mutants from a program.
- Chosen to represent typical errors.
- Dedicated mutation operators have been defined for most languages.
- For example more than 100 operators have been defined for C.

The competent programmer hypothesis:

- The SUT was written by a competent programmer, thus it differs from a correct one by a few small faults

Based on: https://www.st.cs.uni-saarland.de/edu/testingdebugging10/

ABS – Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = abs(y);
    y = tmp;
  }

  return x;
}
```

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return 0;
}
```

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return -abs(x);
}
```

Based on: https://www.st.cs.uni-saarland.de/edu/testingdebugging10/

COR - Conditional Operator Replacement

```
if(a && b)

if(a || b)
if(a & b)
if(a | b)
if(a ^ b)
if(false)
if(true)
if(a)
if(b)
```

UOI - Unary Operator Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % +y;
    x = y;
    y = tmp;
  }

  return x;
}
```

+, -, !, ~, ++, --

SVR - Scalar Variable Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = y % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

tmp = x % y
tmp = x % x
tmp = y % y
x = x % y
y = y % x
tmp = tmp % y
tmp = x % tmp

OO Mutation

- So far, operators only considered method bodies
- Class level elements can be mutated as well:

```
public class test {
  // ..
  protected void do() {
    // ...
  }
}
```

```
public class test {
  // ..
  private void do() {
    // ...
  }
}
```

Based on: https://www.st.cs.uni-saarland.de/edu/testingdebugging10/

More Mutation Operators

- Constant replacement
- Scalar variable replacement
- Scalar variable for constant replacement
- Constant for scalar variable replacement
- Array reference for constant replacement
- Array reference for scalar variable replacement
- Constant for array reference replacement
- Scalar variable for array reference replacement
- Array reference for array reference replacement
- Source constant replacement
- Data statement alteration
- Comparable array name replacement
- Arithmetic operator replacement
- Relational operator replacement
- Logical connector replacement
- Absolute value insertion
- Unary operator insertion
- Statement deletion
- Return statement replacement

And More O-O Operators

- Replacing a type with a compatible subtype (inheritance)
- Changing the access modifier of an attribute, a method
- Changing the instance creation expression (inheritance)
- Changing the order of parameters in the definition of a method
- Changing the order of parameters in a call
- Removing an overloading method
- Reducing the number of parameters
- Removing an overriding method
- Removing a hiding Field
- Adding a hiding field

From Lionel Brand http://www.inf.ed.ac.uk/teaching/courses/st/2016-17/Mutest.pdf

Mutants and Coverage

- If a test suite kills all mutants, then it achieves 100% coverage for that type of fault. (Sometimes called mutation score.”
- Each mutant can be considered a test requirement (distinct from software requirement).

Mutation Testing Challenges

- Mutation operators are language specific.
- Not all mutations are representative of real faults.
- Not all mutations are valid.
- Not all mutations can be killed.
- The number of mutations can be very large.

Not all mutants can be killed

- equivalent mutants
- unreachable mutants
- mutants that don’t infect
- mutants that don’t propagate

Example of an Equivalent mutant

Original program

```
int index=0;
while (...)
{
      . . .;
      index++;
      if (index==10)
           break;
}
```

A mutant

```
int index=0;
while (...)
{
      . . .;
      index++;
      if (index>=10)
           break;
}
```


From Lionel Brand http://www.inf.ed.ac.uk/teaching/courses/st/2016-17/Mutest.pdf

A Simple Example

Original Function

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

With Embedded Mutants

```
     int Min (int A, int B)
        int minVal;
     {
        minVal = A;
Δ1      minVal = B;
        if (B < A)
Δ2      if (B > A)
Δ3      if (B < minVal)
        {
            minVal = B;
Δ4          Bomb();
Δ5          minVal = A;
Δ6          minVal = failOnZero (B);
        }
     return (minVal);
     } // end Min
```

Note This!

Delta's represent syntactic modifications. In fact, each of them will be embedded in a different program version, a mutant.

From Lionel Brand http://www.inf.ed.ac.uk/teaching/courses/st/2016-17/Mutest.pdf

Discussion of the Example

- Mutant 3 is equivalent as, at this point, `minVal` and `A` have the same value
- Mutant 1: In order to find an appropriate test case to kill it, we must
  1. Reach the fault seeded during execution (Reachability)
     - Always true (i.e., we can always reach the seeded fault
  1. Cause the program state to be incorrect (Infection)
     - A <> B
  3. Cause the program output and/or behavior to be Incorrect (Propagation)
     - (B<A) = false

Original Function	With Embedded Mutants

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

```
      int Min (int A, int B)
         int minVal;
      {
         minVal = A;
Δ1       minVal = B;
         if (B < A)
Δ2       if (B > A)
Δ3       if (B < minVal)
         {
             minVal = B;
Δ4           Bomb();
Δ5           minVal = A;
Δ6           minVal = failOnZero (B);
         }
      return (minVal);
      } // end Min
```

Frankl's Observation

We also observed that […]
mutation testing was *costly*.
Even for these small subject programs,
the human effort needed to check a large
number of mutants for equivalence
was *almost prohibitive*.

P. G. Frankl, S. N. Weiss, and C. Hu.
All-uses versus mutation testing:
An experimental comparison of effectiveness.
Journal of Systems and Software, 38:235–253, 1997.

From: https://www.st.cs.uni-saarland.de/edu/testingdebugging10/

Summary

Steps in the Mutation Test Cycle

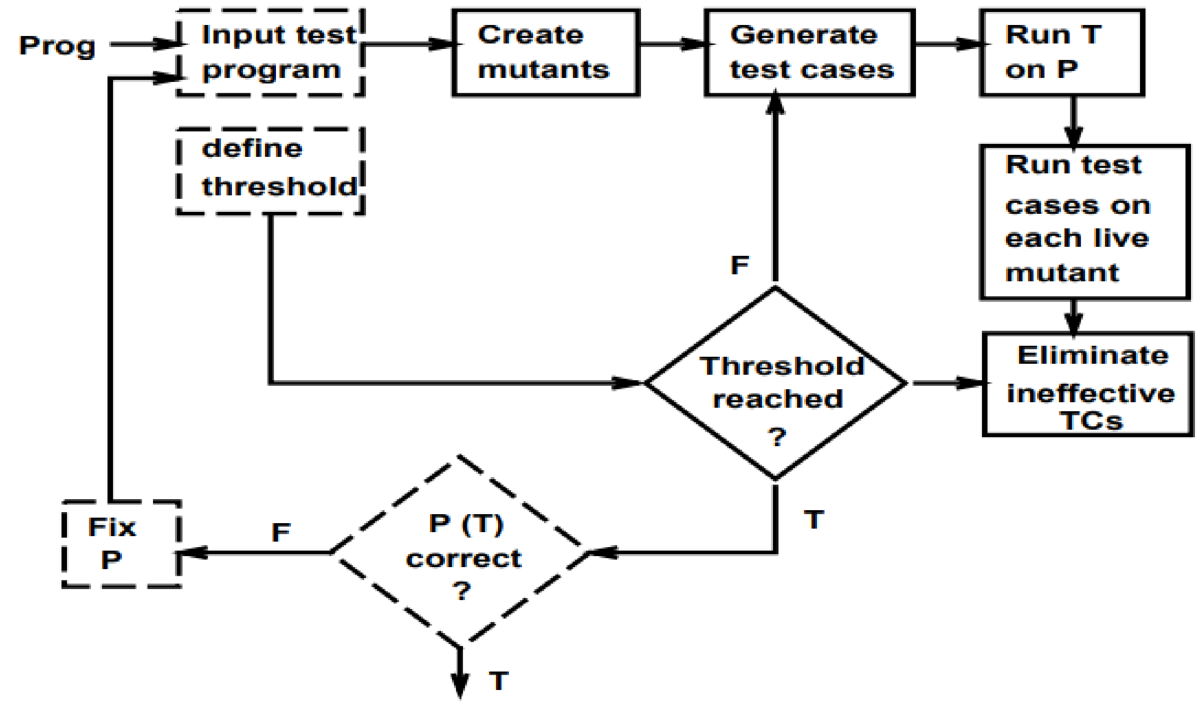

Figure 3: New Mutation Testing Process.
Solid boxes represent steps that are automated and dashed boxes represent steps that are manual.

from: Jeff Offutt, “A Practical System for Mutation Testing: Help for the Common Programmer”, Twelfth International Conference on Testing Computer Software, pages 99--109, Washington, DC, June 1995.

Summary

- Mutation testing is a mechanism for testing the adequacy of a test suite, with respect to certain types of faults.
- Mutation testing is fault-based rather than behavior based.
- Mutation testing aims to generate faults similar to what a competent programmer might.
- Mutation testing is computationally intensive.
- Equivalent mutants create practical problems. Automation can help. Equivalence is an undecideable problem.
- Each killable mutant can be considered a test requirement.

Mutation Testing Resources

- Ammann & Offutt: 1st ed: Chapter 5 or 2nd ed: Chapter 9
- Pezzè and Young, "Software Testing and analysis: process, principles, and techniques," Chapter 16.
- μJava: http://cs.gmu.edu/~offutt/mujava/
- MuClipse: http://muclipse.sourceforge.net/