CITS5501 Software Testing and Quality Assurance
Syntax-based testing: Program-Based Mutation Testing

Arran Stewart and Rachel Cardell-Oliver

Previous Lecture

- Motivation - What is syntax-based testing?
- Theory - Formal Grammars and Coverage Criteria
- Applications of Syntax-based testing
- Input-Space Mutation Testing
- **Program-Based Mutation Testing (this lecture)**

Overview

- Motivation - What is Program-Based Mutation testing?
- Theory and Definitions
- In Practice - Worked Example and Tools

Reading: Ammann and Offutt. Introduction to Software Testing, 2016. *Ch 9 Syntax-Based Testing*

Also see this great introduction: pitest.org

Motivation

Program-Based Mutation testing Motivation: Are we missing any tests?

Some useful techniques for working out if there are gaps in our tests are:

- **graph coverage** techniques -|- may show us that there are portions of code that have never been tested
- **logic coverage** techniques -|- may show us we haven't tested different sub-parts of conditions
- **program-based mutation testing** can be used to "test the tests" -|- if we mutate a program, and it still passes all our tests, something is wrong

In this lecture we will explore program-based mutation testing.

Program-Based Mutation Testing in a nutshell

- Mutation testing is conceptually quite simple.
- Faults (or mutations) are automatically seeded into your code, then your tests are run. If your tests fail then the mutation is killed, if your tests pass then the mutation lived.
- The quality of your tests can be gauged from the percentage of mutations killed.
- Source: see this great introduction: pitest.org

Program-Based Mutation Testing

Program-Based Mutation testing (also called "mutation analysis") is a technique for evaluating the **quality** of a **suite of software tests**

- Suppose we have some program under a test, and a suite of tests designed to identify defects in it.
- Mutation testing works by **modifying the program under test** in small ways (e.g. flipping a less-than sign to a greater-than; changing a hard-coded number from 0 to 1).

Program-Based Mutation testing (cont)

- Mutations are usually designed to mimic typically programming errors, such as typographical errors, wrong choice of operator, or off-by-one errors
- If one of our **tests** behaves differently for the mutant (vs on the original program), we say that test **kills** the mutant
- If the test suite doesn't detect and reject the mutated code, we consider the test suite is defective, and we need to consider more/different test cases.

Mutation example

- A sample method to test (in a Java-like language):

```
int mult(int a, int b) { return a * b; }
```

- A possible test:

```
assertEquals( 1, mult(1,1) );
```

- Is this a useful test?

Mutation example (2)

- ans.: No, it's terrible.
- Consider the following **mutation** of the original code:

```
int mult(int a, int b) { return a * b; }
```

⇒

```
int mult(int a, int b) { return a ** b; }
```

(where ** is a "power" operator)

- 1*1 == 1**1 so our test will still pass -
  - so it's a pretty poor test because it *does not* ''kill" the mutant
- A **good quality** suite of tests should be able to catch these types of errors

Theory

Mutation testing is a syntax-based test method

Mutatation testing is a form of **syntax-based testing**

Mutations (changes) to the program under test can be defined by a grammar

Mutation testing – terminology

- **Ground string**: A string in the grammar
  - (The term "ground" basically means "not having any variables" – in this context, not having any non-terminals)
- **Mutation operator**: A rule that specifies syntactic variations of strings generated from a grammar - (e.g. "If the string has a less-than symbol in it, flip that intro a greater-than symbol")
- **Mutant**: The result of one application of a mutation operator on the ground string
- A mutant is a string

Killing Mutants

- An example of a ground string is – our program under test.
- . . . since it's a string in the grammar of "syntactically valid Java programs"
- We apply our mutation operator to the ground strings to generate *mutants*, new valid strings
- **Killing mutants**: If we have some mutant generated from the original ground string, and we look at one or several of our tests, we can ask: do they "**kill**" the mutant?
  - i.e. Does the test(s) give a different result for the mutant, compared to the original?
  - If it does, it's said to "kill" the mutant.

Coverage Criteria

Syntax-based coverage criteria – mutant coverage

- We can define a coverage criterion in terms of killing mutants:

Mutation Coverage (MC) For each mutant m, the test requirements contain exactly one requirement to kill m.

- Coverage in mutation equates to number of mutants killed
- The number of mutants killed is called the **mutation score**

Coverage criteria – creating invalid strings

- When creating mutants (i.e. invalid strings), two simple criteria:
  - use every operator once or
  - use every production once
- TR is **test requirements**. TR are descriptions of test cases that will be refined into executable tests

Mutation Production Coverage (MPC) For each mutation operator, TR contains several requirements, to create one mutated string m that includes every production that can be mutated by that operator

Mutation Operator Coverage (MOC) For each mutation operator, TR contains exactly one requirement, to create a mutated string m that is derived using the mutation operator

Coverage criteria – practical concerns

- The number of test requirements for mutation is somewhat difficult to quantify because it depends on the syntax of the artifact as well as the mutation operators
- In most situations, **mutation yields more test requirements than any other coverage criterion**

In Practice

Frameworks for mutation testing

Mutation testing is difficult to apply by hand and automation is more complicated than for most other criteria.

As a result, mutation is widely considered a *high-end* coverage criterion, more effective than most but also more expensive.

One common use of mutation is as a sort of *gold standard* in experimental studies for comparative evaluation of other test criteria.

Some example mutation testing frameworks are:

- PIT, for Java (originally stood for "Parallel Isolated Test")
- Mutpy, for Python
- Stryker, for C#
- Mutagen, for Rust (motto: "Breaking your Rust code for fun & profit")

Advantages and disadvantages

Advantages and disadvantages of mutation testing:

- Identifies weak/ineffective tests
- Very effective at finding problems
- Helps quantify how useful your tests are
- Can be time-consuming (large number of mutants to generate, whole test suite needs to be run many times)
- Results require some familiarity with mutation testing to be properly understood.

Mutation testing example[1] – IOT cat door

Suppose we have an Internet of Things (IoT)–enabled cat door. The final cat door will be implemented as hardware with embedded software; but we can still check our logic using testing techniques we have seen before.

We have the following user story describing the purpose of the cat door:

> *Using my home automation system (HAS),*
> *I want to control when the cat can go outside,*
> *because I want to keep the cat safe overnight.*

[1] Adapted from Alex Bunardzic, "Mutation testing by example: Failure as experimentation" (2019)

Cat door interface

We represent the cat door using the following interface:

```
public interface ICatDoor {
  /** When "day" is supplied, unlock the door;
    * when "night" is supplied, lock the door */
  public void control(String dayOrNight);
  /** Returns either "locked" or "unlocked" */
  public String getStatus();
}
```

Testing scenario

We want to write tests revolving around the following scenario:

Scenario #1: Disable cat trap door during nighttime

Given that the day/night detector detects that it is nighttime
When the day/night detector notifies the HAS
Then HAS disables the IoT-capable cat door

(We won't worry about how the day/night detector is implemented. Perhaps it uses ambient light levels; perhaps it consults a database of sunrise/sunset times for its current geographical location.)

Cat door code under test

```
public class CatDoor implements ICatDoor {
  // ...
  public void control(String dayOrNight) {
    if (dayOrNight.equals("night")) {
      this.lock();
    } else {
      this.unlock();
    }
  }
}
```

Cat door test code

```
public class TestGivenNighttimeDoorLocked {
  @Test
  public void test() {
    ICatDoor door = new CatDoor();
    door.control("night");
    String expected = "locked";
    String actual   = door.getStatus();
    assertEquals(expected, actual, "status should be locked");
  }
}
```

start PIT running

```
27:46 am PIT >> INFO : Completed in 2 seconds
================================================================================
— Mutators
================================================================================
> org.pitest.mutationtest.engine.gregor.mutators.rv.ROR3Mutator
>> Generated 1 Killed 1 (100%)
> KILLED 1 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.VoidMethodCallMutator
>> Generated 2 Killed 2 (100%)
> KILLED 2 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
<more output snipped>
```

PIT output

```
1
2   public class CatDoor implements ICatDoor {
3     private String status;
4     public CatDoor(String status) {
5  1    this.status = status;
6     }
7     @Override
8     public String getStatus() {
9  2    return status;
10    }
11    private void lock() {
12 1    this.status = "locked";
13    }
14    private void unlock() {
15 1    this.status = "unlocked";
16    }
17    @Override
18    public void control(String dayOrNight) {
19 9    if (dayOrNight.equals("night")) {
20 1      this.lock();
21      } else {
22 1      this.unlock();
23      }
24    }
```

PIT output

Mutations

5	1. Removed assignment to member variable status → SURVIVED
9	1. replaced return value with "" for CatDoor::getStatus → KILLED 2. mutated return of Object value for CatDoor::getStatus to (if (x != nul
12	1. Removed assignment to member variable status → KILLED
15	1. Removed assignment to member variable status → KILLED
19	1. negated conditional → KILLED 2. removed call to java/lang/String::equals → KILLED 3. removed conditional - replaced equality check with false → KILLED 4. removed conditional - replaced equality check with true → KILLED 5. equal to less than → KILLED 6. equal to less or equal → SURVIVED 7. equal to greater than → KILLED 8. equal to greater or equal → KILLED 9. equal to not equal → KILLED
20	1. removed call to CatDoor::lock → KILLED
22	1. removed call to CatDoor::unlock → KILLED

Summary

- Motivation - What is Program-Based Mutation testing?
- Theory Definitions
- Coverage Criteria
- In Practice - Worked Example and Tools