Mutation Testing

Arnav Gupta

April 5, 2024

Contents

1 Test Case Effectiveness

The best test suite can detect every bug, whenever a bug is introduced at least one test case will fail.

To measure effectiveness:

- simple solution: find all bugs in the software (causality dilemma)
- state-of-the-art solution: artificially inject bugs
  - mutation testing: modify some statements in code so that there are different versions, where each version is a bug, to see if the test cases can find the bugs

2 Mutation Testing

Fault-based testing: directed towards typical faults that could occur in a program

Idea

- create similar programs (mutants) differing in one small way (fault)
- original test data is run through mutants
- if test data detects all differences in mutants, then mutants are dead, and test set is adequate

Should be used in conjunction with traditional testing techniques.

Mutants remain live either because it is equivalent to the original program or test set is inadequate to kill the mutant. Automated mutant generation uses mutation operators (predefined program modification rules).

Insignificant mutants:

- **stillborn mutant**: synactically incorrect, killed by compiler
- **trivial mutant**: killed by almost any test case
- **equivalent mutant**: produces same output as original program (undecidable problem)

Can design mutants that introduce non-functional issues (such as performance), but infection and propagation are hard to define.

2.1 Mutation Operators

Mutation operators change with programming languages, design, and specification paradigm. They should be representative of all realistic types of faults that could occur.

Specific operators are:

- absolute value insertion
- arithmetic operator replacement
- relational operator replacement
- conditional operator replacement
- shift operator replacement

- logical operator replacement
- unary operator deletion
- scalar variable replacement
- bomb statement replacement

For OOP languages:

- replacing type with compatible subtype
- changing access modifier of attribute or method
- changing instance creation expression
- changing order of params in method definition
- changing order of params in a call
- removing an overloading method
- reducing the number of params
- removing an overriding method

Competent programmer assumption: given a spec a programmer develops a programmer that is either correct or differs from the correct program by a combination of simple errors

Coupling effect assumption: test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors (complex faults coupled to simple faults)

3 Mutation Coverage

Complete coverate equates to killing all non-equivalent mutants, with the amount of coverage called mutation score.

$$\text{mutation score} = 100 * \frac{\text{dead mutants}}{(\#\text{ of mutants} - \#\text{ of equivalent mutants})}$$

Strong mutation: fault must be reachable, infect the state, and propagate to output

Weak mutation: fault which kills mutant needs only be reachable and infect the state

3.1 Types of Mutation Coverage

3.1.1 Strong Mutation Coverage

Strongly killing mutants

- given a mutant m for a program P and a test case t, t is said to strongly kill m iff the output of t on P is different from the output of t on M

Strong mutation coverage: for each mutant m in M, a test case exists that strongly kills m

3.1.2 Weak Mutation Coverage

Weakly killing mutants

- given a mutant m that modifies a source location L in a program P and a test case t, t is said to weakly kill m iff the state of the execution of P on t is different from the state of the execution of m on t, immediately after some execution of L

Weak mutation coverage: for each mutant m in M, a test case exists that weakly kills m