



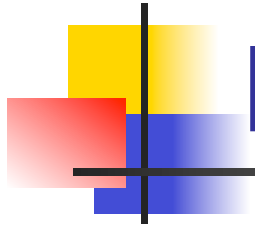
Mutation Testing

Breaking the application to test it



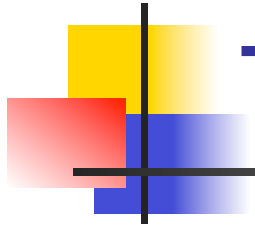
What is Mutation Testing?

- 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



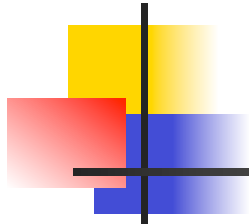
Mutation Testing

- 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

```
1 int max(int x, int y)
2 {
3     int mx = x;
4     if (x > y)
5         mx = x;
6     else
7         mx = y;
8     return mx;
9 }
```

```
1 int max(int x, int y)
2 {
3     int mx = x;
4     if (x < y)
5         mx = x;
6     else
7         mx = y;
8     return mx;
9 }
```



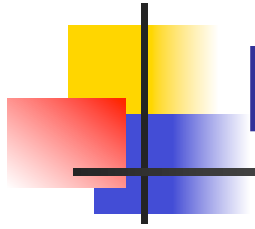
Mutation Operators

- Operand Replacement Operators:
 - Replace a single operand with another operand or constant. E.g.,
 - if (5 > y) Replacing x by constant 5.
 - if (x > 5) Replacing y by constant 5.
 - 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

- 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

- 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 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 change to the correct program.



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



Mutation Testing Algorithm

- 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
- 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.



Mutation Score

- 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%.



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.



Mutation Testing Costs

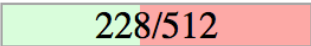
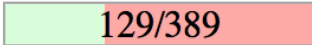
- The FORTRAN 77 version of the max() program generated 44 mutants using Mothra.
- Most efforts on mutation testing have focused on reducing its cost by reducing the number of mutants while maintaining the effectiveness of the technique.



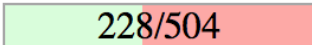
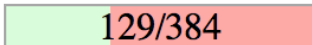
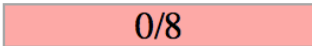
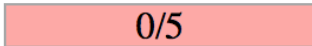
PIT Demo

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage
5	45% 	33% 

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
mojo	4	45% 	34% 
testmojo	1	0% 	0% 

Report generated by [PIT](#) 1.1.4



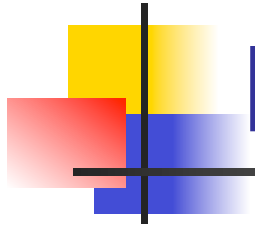
Program Perturbation

- Program Perturbation is a technique to test a program's robustness.
- It is based on unexpectedly changing the values of program data during run-time.



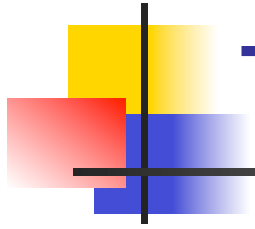
Software Failure Hypothesis

- Program perturbation is based on the three part software failure hypothesis:
 - **Reachability:** The fault must be executed.
 - **Infection:** The fault must change the data state of the computation directly after the fault location.
 - **Propagation:** The erroneous data state must propagate to an output variable.



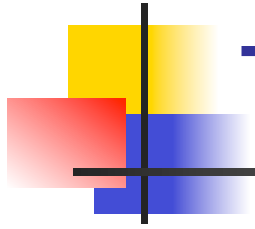
Program Perturbation Process

- The tester must:
 - inject faults in the data state of an executing program;
 - trace the impact of the injected fault on the program's output.
- The injection is performed by applying a perturbation function that changes the program's data state.



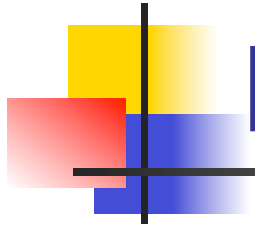
The Perturbation Function

- The ***perturbation function*** is a mathematical function that:
 - takes a data state as its input
 - changes the data state according to some specified criteria
 - produces a modified data state as output



The Fault Injection

- A program location N is chosen along with a set of input variables I that are in scope at location N .
- The program is executed until location N .
- When execution arrives at location N , the resulting data state is changed (perturbed).
- The subsequent execution will either fail or succeed.



Program Perturbation Example

- Assume the following perturbation function:

```
1. int perturbation (int x)
2. {
3.   int newX;
4.   newX = x + 20;
5.   return newX;
6. }
```




Example of a Fault Injection

```
1. main()
2. {
3.   int x;
4.   x = ReadInt();
5.   if (x > 0)
6.     printf("X positive");
7.   else
8.     printf("X negative");
9. }
```

```
1.   main()
2.   {
3.   int x;
4.   x = ReadInt();
4.1  x = perturbation(x);
5.   if (x > 0)
6.     printf("X positive");
7.   else
8.     printf("X negative");
9. }
```



What Perturbation Testing is and is Not

- Perturbation testing is NOT a testing technique that outlines test selection and coverage criteria.
- Rather, perturbation testing is a technique that can be used to measure the reliability of the software (tolerance to faults).



Evaluation

- The program is repeatedly executed and injected with faults during each execution.
- The ratio of the number of failures detected divided by the total number of executions is used to predict failure tolerance.