What is Mutation Testing?

The premise in **mutation testing** is that small changes are made in a module and then the original and mutant modules are compared.

- Similar idea to **error seeding**: introduce defects and verify that test cases are comprehensive enough to 'catch' the defects (called mutants).

- This method is **more** systematic--all single modifications to the code are generated--each single modification yields a new mutant.

- The idea is that test suite is adequate when it suffices to distinguish between the original and any non-equivalent mutant.

- To be cost-effective, selectively insert only realistic mutations (defects) and see whether your test suite catches the implanted defect.
Basic Assumption of Mutation Testing

Competent programmer hypothesis

The program to be tested has been written by a competent programmer.

2. The Coupling Effect

The test data that distinguishes all programs differing from a correct one by only simple errors are so sensitive that they are also implicitly distinguished more complex errors.

There is no hope of ‘proving’ the coupling effect; it is an empirical hypothesis.
Basic Steps of Mutation Testing

1. Executing the Program On the Test Data
2. Constructing the Mutant Set
3. Executing Each Mutant
4. Terminating Mutation Testing
5. Add More Test Cases

If some live imitations remain, there are 2 possible reasons:

1. The Mutation is equivalent to the original code (not very likely)
2. The Test Data is not adequately sensitive.

Mutation Testing starts with test data designed by other methods, and is the only defect-directed technique for testing the completeness of test suites.
**Example of Testing By Mutation**

```
function MAX(M<N:INTEGER)
    return INTEGER is
begin
    if M>N then
        return M;
    else
        return N;
    end if:
end MAX;
```

First test data set--M=1, N=2
•the original function returns 2
•five mutants: replace”>“ operator in satements by (>,<,<=or=)
•executing each mutant:

<table>
<thead>
<tr>
<th>Mutants</th>
<th>Outputs</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>if M&gt;=N then</td>
<td>2</td>
<td>live</td>
</tr>
<tr>
<td>if M&lt;N then</td>
<td>1</td>
<td>dead</td>
</tr>
<tr>
<td>if M&lt;=N then</td>
<td>1</td>
<td>dead</td>
</tr>
<tr>
<td>if M=N then</td>
<td>2</td>
<td>live</td>
</tr>
<tr>
<td>if M&lt; &gt;N then</td>
<td>1</td>
<td>dead</td>
</tr>
</tbody>
</table>

•adding test data M=2, N=1 will eliminate the latter live mutant, but the former live mutant remains live because it is equivalent to the original function. No test data can eliminate it.
The Difference of the Goals Between Fault-Based Testing and Others

Fault-based testing adopts the goal of demonstrating the absence of faults rather than the imprecise goal of “finding errors”.

Many take a truism the oft-quoted statement by Myers, that “the goal of testing is to find errors” [The Art of Software Testing. 1979]. Myers goes as far as to assert that a successful test is one which causes a program failure, and that no information is gained when a program passes does not fail. Thus, Myers’ perspective is that testing is adequate if it fails to uncover an error. This view has the unfortunate consequence that a correct program cannot be adequately tested since it contains no faults.

It is impossible to simply “look for errors”; one must look for specific errors, or fall prey to not being able to attain the goal. On the other hand, demonstrating can be attained, even for a correct program.

1. Introduction

1.1 Two Classes of Traditional Testing Methods

- Specification-based (or Function-based)
  ---- Black box testing methods

- Code-based (or Structure-based)
  ------ White box testing methods

1.2. A New Class of Testing Methods

Fault-based

What is Fault-based Testing?
What is New in It?
How Does It Work
2. Fault-based Testing

2.1 Basic Definitions

**Error:** An error is mental mistake by a programmer or designer.

**Fault:** A Fault is a textual problem with the code caused by the error.

**Failure:** A failure is an observable incorrect behavior of the program induced by the fault.

For example, a programmer may make an error in analyzing an algorithm. This may result in writing a code fault, say a reference to an incorrect variable. When the program is executed, it may fail for some inputs in the domain of the specification.
2.2 The Definition of Fault-based Testing

Definition:

Fault-based testing is the process of demonstrating the absence of pre-specified faults in a module under test (MUT).

Explanation:

The definition given here has a particular focus, scope, and goal.

The focus is on faults rather than errors.

The scope is limited to pre-specified faults rather than all possible faults.

The goal is to demonstrate the absence of faults, not merely to look for faults (or errors).
2.3 The Context of Fault-based Testing

-----The Fault-based Context

A fault-based context is a 5-tuple,
\[ C = \langle M, S, D, L, A \rangle, \]
where,
- \( M \) - a given module
- \( S \) - a specification.
- \( D \) - a domain of the specification.
- \( L \) - an \( n \)-tuple of instruction locations in \( M \):
  \( L = (L_1, L_2, ..., L_n) \)
- \( A \) - \( (A_1, A_2, ..., A_n) \) where \( A_i \) is an alternative set associated with location \( L_i \), \( 1 \leq i \leq n \).

Definitions:

**Location** - A location in a module denotes some expression which is language dependent (part of an instruction).

**Alternative** - An alternative is an expression \( e \) which can be legally substituted for the expression in the statement at location \( L \) in the module under test.
2.4 The Basic Framework of Fault-based Testing

Fault-based testing starts with a fault-based context: <M,S,D,L,A>, and seeks to demonstrate that no member of A is present in the module.

The basic steps of fault-based testing can be well understood from typical fault-based testing techniques.

2.5 Typical Fault-based Testing Techniques

- Various Mutation Testing Methods(*)
- Error-sensitive testing (Foster)
- Perturbation testing (Zeil)
- Domain Testing (White & Cohen)
- Fault-based functional testing (Howden)
- Symbolic Testing (Morell)

In the rest of the presentation, we will emphasize mutation testing.
3. Mutation Testing

-- A Powerful Fault-based Testing Technique

3.1 Basic Concepts

**Mutation**

Making a small change in a module

**Mutation Testing**

To systematically and repeatedly make a small change (e.g., replacing one operator by another, replacing one variable by another, or altering the value of a constant) in the code, and for each change to compare the outputs of the original and the changed module when exercised on the same test suite.

Such a change is called a mutation, and a changed module is called a mutant of the original module.
3.2 Several Mutation Testing Techniques

According to the time for comparing outcomes of the original and mutant module, Mutation Testing can be divided into three categories:

- Strong Mutation (R.A. DeMillo et al.)
- Weak Mutation (W.E. Howden)
- Firm Mutation (M.R. Woodward et al.)

In **Strong Mutation**, the time is after execution of the entire module.

In **Weak Mutation**, the time is immediately after each single execution of a component (which has been mutated) of the module.

In **Firm Mutation**, the time is at some point in between Strong Mutation time and Weak Mutation time.

Note that we will put our emphasis on Strong Mutation Testing only.
3.3 Basic Hypothesis of (Strong) Mutation Testing

Mutation testing is based on the following two hypotheses:

(1) Competent programmer hypothesis

The module under test has been written by a competent programmer or designer. Therefore, if the module is not correct, it differs from a correct one by at most a few small faults.

(2) Coupling Effect hypothesis

The test suite that distinguishes all modules from a correct one by only simple faults is so sensitive that it also implicitly distinguishes more complex faults.

Note that there is no hope of “proving” the Coupling Effect mathematically; it is an empirical hypothesis.
3.4 Basic Steps of (Strong) Mutation Testing

For a given module M and a proposed test data set D:

(1) Executing M on the test data set D.
   If outputs are incorrect, M has faults. If outputs are correct, M may have faults also - Test data is not adequate.

(2) Constructing mutants of M.
   Each mutant is identical to M except for a single syntactic change (a mutation). (How to do this?)

(3) Executing each mutant on D.
   If producing a different output, the mutant was dead. If producing the same output, the mutant remains live (D is inadequate; or the mutant is equivalent to M).

(4) Terminating mutation testing
   Test data set D is adequate: no live mutants left or only equivalent mutants left. Otherwise, go to the next step.

(5) Adding more test cases and repeating
   According to information provided by the live mutants, design and add more test cases to D, and repeat the above process from the very beginning.