7th International Conference on Software Engineering Advances (ICSEA 2012)

Keynote - Lisbon, Nov 20th, 2012

Mutation Testing: Development and Challenges

Jameleddine Hassine
Department of Information and Computer Science
King Fahd University of Petroleum and Minerals, KSA
jhassine@kfupm.edu.sa

Software Testing

- Two aims:
  1. Prevent bugs from being introduced into code
  2. Discover those un-prevented bugs
     - What is a bug?
     - What are its symptoms?
     - What is an infection?
     - How it is cured?

What is a bug?

- Misunderstand a specification
- Underestimate the complexity of the software
- Inadvertently press the wrong key

↓

Faults

(physical mistakes in the design or the implementation of the code)

Commonly referred to as

Bugs

Especially in the context of code development

What are the symptoms of a bug?

- Software failure
  - Observable event
  - The software execution differs from its specification
- The failure observed is a symptom of a bug
  - Trivial annoyance (The defect does not affect functionality or data)
  - Drastic such as the loss of a human life

What is an infection?

- In biology, an infection is due to the presence of a bug in the body that may or may not cause symptoms to be expressed

- Similarly, an infection in code refers to software that has at least one fault that may or may not express symptoms when executed.

- Simply, the code is infected with a bug

How is an infection cured?

Two stages process:

1. Bug identification
   - Primarily achieved by executing tests on a program in an attempt to reveal symptoms of a bug
   - If symptoms are expressed, then the test has caused the program to execute differently from its specification and so has provided useful information in identifying a fault
2. Bug correction
   - Simple change to source code (wrong variable name or incorrect relational operator)
   - More fundamental changes that require the rewriting of numerous lines of code

Poor vs. Good Test?

- How does a tester distinguish between a poor test that is incapable of displaying a fault's symptoms, and a good test when there are no faults to find?

Test Set Adequacy

(as a means to of measuring how good a test set is at testing a program)

- Adequacy criteria (indication of program coverage)
  - Statement coverage criterion
  - Decision testing (exercising all true and false paths)
- Increase the number of tests in order to improve our confidence in the system

Mutation Testing

- Adequacy criteria **do not focus** on the **causes** of a program's **failures**

↓

Mutation Testing Does

- This criteria generates versions of the program containing simple faults and then finds tests to indicate their symptoms
- If an adequate test set can be found that reveals the symptoms in all the faulty versions, then one's confidence that the program is correct increases.

Fault-based Testing

- Error guessing
  - Assess the situation and guess where and what kinds of faults might exist
  - Design tests to specifically expose those kinds of faults
- Fault seeding
  - known faults are injected into a program, and the test suite is executed to assess the effectiveness of the test suite
  - An oracle is available to assert that the inserted fault indeed made the program incorrect
  - Makes an assumption that a test suite that finds seeded faults is also likely to find other faults
- Mutation analysis
  - Mutations to program statements are made in order to determine the fault detection capability of the test suite
  - Fault simulation, a program modification is not guaranteed to lead to a faulty program

Mutation Testing

- A mutant is produced by introducing small changes into the software artifact (source code or specification UT)

Program p	Mutant p'
...	...
if ($a > 0$ && $b > 0$)	if ($a > 0$ || $b > 0$)
return 1;	return 1;
...	...

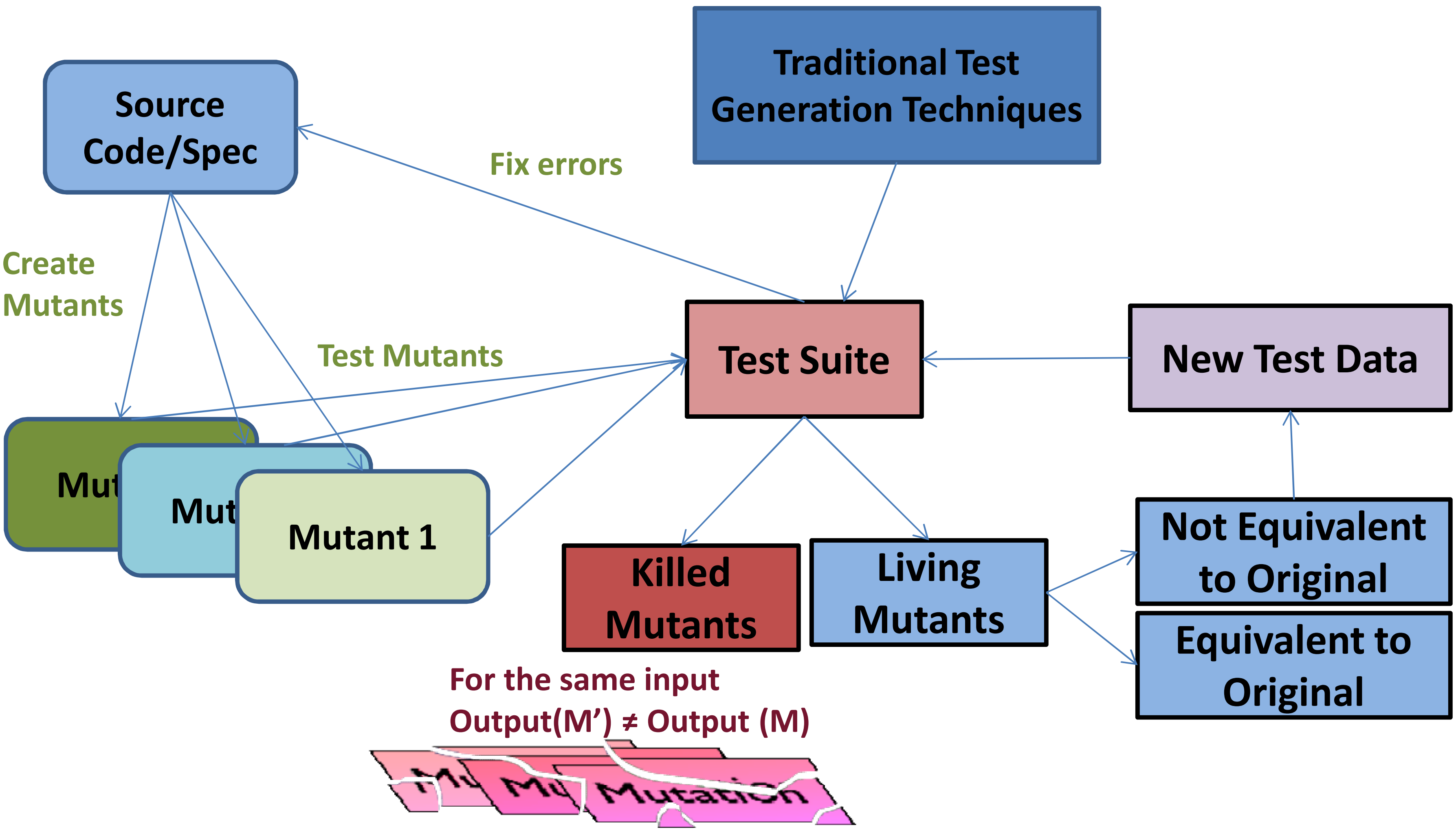
- A mutation operator is a set of instructions for generating mutants of a particular type
- Ideally the test suite should contain a test that distinguishes the behaviors of the mutant and the original artifact
- Expose and locate weaknesses in test cases
- Mutation testing is not a testing strategy like control flow or data flow testing

Mutation Analysis

We can perform mutation analysis whenever we:

- use well **defined rules,**
- defined on **syntactic descriptions,**
- to make **systematic changes**,
- to the **syntax** or to **objects** developed from the syntax

Mutation Testing Process



Example of Mutation Testing

- Initial test data set:
  TC1: Input: M=1, N=2; Expected output: 2
- Five mutants: replace”>“ operator in if statements by (>,<,<=or=)

```
int function MAX(M:int, N:int)
{
if M>N then
   return M;
 else
   return N;
}
```

Mutants	Outputs	Comparison
if M>=N then	2	live
if M<N then	1	dead
if M<=N then	1	dead
if M=N then	2	live
if M< >N then	1	dead

Equivalent to the original program

Adding a **new** test case 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.**

Mutation Score

- A ***mutation score*** for a set of test cases is the percentage of **nonequivalent mutants** killed by the test suite:

$$MS = \frac{K_1}{K - e}$$

k_1 is the number of killed mutants
K is the number of mutants
e is the number of equivalent mutants

- The test suite is said to be *mutation adequate* if its mutation score is 100%.

Mutation Testing Problems

- High computational cost of executing the huge number of mutants against a test set
- Automatically detecting equivalent mutants is undecidable, because program equivalence is undecidable.
- The human oracle problem
  - Refers to the process of checking the original program's output with each test case.
  - This is not a problem unique to Mutation Testing

Mutation Testing – 1970s

- Originally proposed by Dick Lipton in 1971
- Article by DeMillo (*Georgia Tech*), Lipton (*Princeton*), and Sayward (*Yale*) (1978) is generally cited as the seminal reference
- Fundamental Hypotheses (DeMillo et al., 1978):
  - **The Competent Programmer Hypothesis** states that competent programmers tend to write programs that are close to being correct
  - **The Coupling Effect** states that a test data set that catches all simple faults in a program is so sensitive that it will also catch more complex faults

Mutation Testing – 1980s

- MOTHRA Project (1987)
  - Demonstrate the practical feasibility of mutation
  - DeMillo et al. "An Overview of the Mothra Software Testing Environment," Technical Report, Purdue University, 1987
  - First set of Mutation Operators (22 FORTRAN Mutation Operators)
  - First widely used working mutation system
  - Source code written in C (> 100KLOC)
  - Many papers and PhD theses (Offutt 1988, Agrawal 1990, Krauser 1991, Wong 1993) during and after the project

Mutation Testing – 1990s

- Program Unit Testing
  - Mutation Operators (First order mutants)
  - Traditional programming languages
    - Ada
    - C
    - Lisp
- Interface Mutation
  - Mutating function calls
  - Integration testing
- Specification Mutation
  - Mutating Formal specifications (SMV, Z)

Mutation Testing – 2000-Present

- Many new tools
  - Academic (MOTHRA, PROTEUM, MUJAVA, etc.)
  - Open source (JESTER, HECKLE, etc.)
  - Commercial
    - INSURE++
    - Certitude by Certess tests integrated circuit designs in VHDL or Verilog
    - PlexTest by ITRegister tests C++
- Other software artifacts and models
  - FSM
  - XML
  - SQL
  - HTML
  - AspectJ programs
  - Security Policies
  - Web Services

And More to Come…

Research in Mutation Testing

Defining Mutation Operators

Developing Mutation Systems

Research in Mutation Testing

Reduce the Cost of Mutation Analysis

Experimentation with Mutation

Designing Mutation Operators

- Mutation operators are classified by the language constructs they are created to alter (e.g. method-level, class-level, etc.)
- At the method level, mutation operators for different programming languages are similar
- Researchers design lots of operators, then experimentally <u>select</u> the most useful
- Empirical data about the behavior of the mutants produced by a given mutation operator can help us understand the utility of the operator in a given context

Reduce the Cost of Mutation Analysis

- M set of mutants, T a set of test data *T*, *MST* (*M*) denotes the mutation score of the test set *T* applied to mutants *M*

- The mutant reduction problem can be defined as the problem of finding a subset mutants *M'* from *M*, where *MST* (*M'*) ≈ *MST* (*M*).

- Reduce the number of generated mutants without significant loss of test effectiveness

- Reduction Techniques:
  - Mutant Sampling
  - Mutant Clustering
  - Selective Mutation
  - Higher order Mutation

Mutant Sampling

- All possible mutants are generated first as in traditional Mutation Testing
- Randomly chooses a small subset of mutants from the entire set M and the remaining mutants are discarded
- Random selection rate (x%)
- Wong and Mathur's studies (1993) have used selection rate x from 10% to 40% in steps of 5%.
  - The results suggested that random selection of 10% of mutants is only 16% less effective than a full set of mutants in terms of mutation score

Mutant Clustering

- Mutant Clustering chooses a subset of mutants using clustering algorithms
  - Generation of all first order mutants
  - A clustering algorithm is then applied to classify the mutants into different clusters based on the killable test cases
  - Each mutant in the same cluster is guaranteed to be killed by a similar set of test cases
  - Only a small number of mutants are selected from each cluster to be used in Mutation Testing, the remaining mutants are discarded
- Hussain's experiment (2008) applied two clustering algorithms, K-means and Agglomerative clustering
- Empirical results suggest that Mutant Clustering is able to select fewer mutants but still maintain the mutation score

Selective Mutation

- Reducing the number of applied mutation operators
  - Find a **small set of mutation operators** that generate a subset of all possible mutants without significant loss of test effectiveness

- Operators generate different numbers of mutants
  - Some operators generate far more mutants than others, many of which may turn out to be redundant
  - For example, two mutation operators of the 22 Mothra operators, ASR (Assignment Operator Replacement) and SVR (Scalar Variable Replacement), were reported to generate approximately 40% to 60% of all mutants (king and Offut, 1991)

Selective Mutation

- Omitting two mutation operators is called “2-selective mutation”
  - Achieved a mean mutation score of 99.99% with a 24% reduction in the number of mutants (Offut et al. 1993)
  - 4-selection/6-selection mutation
- Categorize the operators then select operators from each Category
- Apply linear statistical approaches to identify a subset of 28 mutation operators from 108 C mutation operators (Naim et al. 2008)
  - The 28 operators are sufficient to predict the effectiveness of a test suite, and it reduced 92% of all generated mutants

Higher Order Mutation (HOM)

- Higher Order Mutants are generated by applying mutation operators more than once
  - Second order mutant (apply the operator twice)
- HOM mutants are harder to kill compared with First Order Mutants
- One HOM test case would kill FOM separately and in combination
  - Human oracle needs only to check one test output

Execution Cost Reduction Techniques

- Based on the way in which we decide how a mutant is killed during the execution process
  - Mutation Testing techniques can be classified into three types:
  - Strong Mutation
  - Weak Mutation
  - Firm Mutation.
- Runtime Optimization techniques
  - Reduction of the compilation cost
- Advanced Platforms Support for Mutation Testing

Strong/Weak/Firm Mutation

- Strong Mutation: the mutant is killed when it produces a different output from the original program
- Weak Mutation: instead of checking after the execution of the entire program, the mutants only need to be checked immediately after the execution point of the mutated statement/component
- Firm Mutation: The 'compare state' lies between the intermediate states after execution (Weak Mutation) and the final output (Strong Mutation)
  - To date no publicly available firm mutation tool

Runtime Optimization Technique

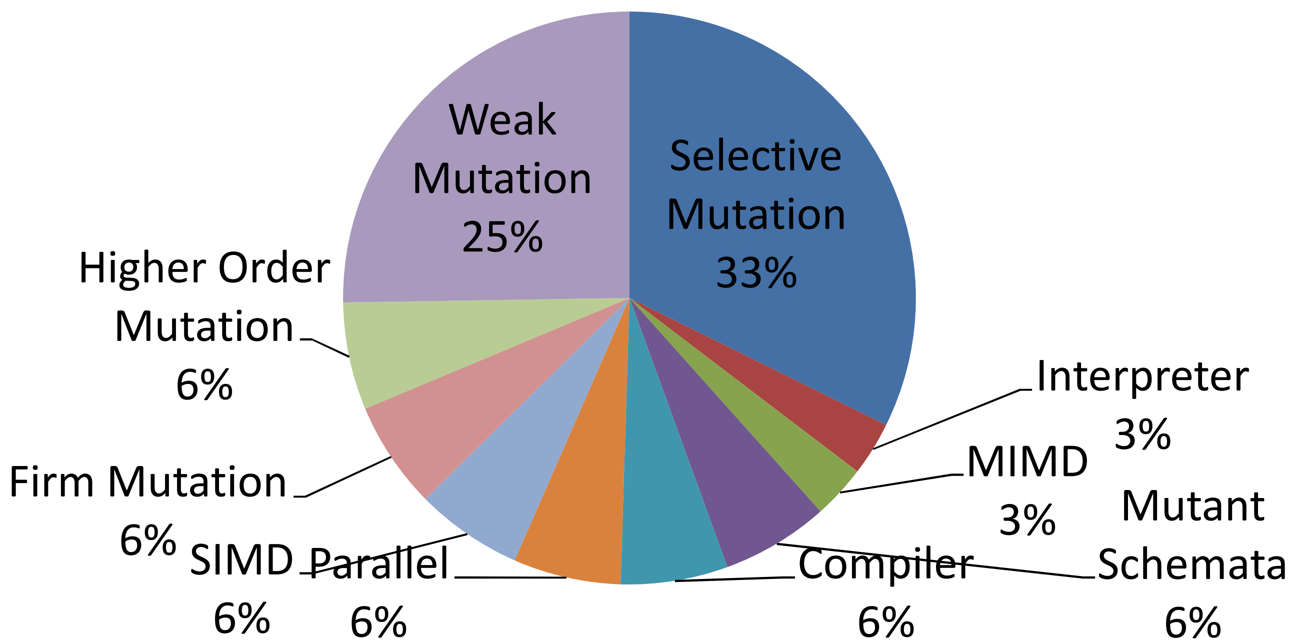
- Reduction of the compilation cost:
  - Bytecode Translation technique (Java)
  - Mutants are generated from the compiled object code of the original program, instead of the source code
  - The generated 'bytecode mutants' can be executed directly without compilation
  - Not all programming languages provide an easy way to manipulate intermediate object code

Advanced Platforms Support for Mutation Testing

- Parallel mutation testing
- Distribute the overall computational cost among many processors
- Concurrent execution mutants under SIMD machines (Krauser et al. 1991)
- Distributed the execution cost of Mutation Testing through MIMD machines (Offut et al. 1992)

Reduce the Cost of Mutation Analysis

Percentage of publications on Reduction Techniques (Jia and Harman, 2011)



Future Trend in Mutation Testing

- High quality higher order mutants
- Need to reduce the equivalent mutant problem
- A preference for semantics over syntax. More realistic mutants that resemble real faults
- Achieving a better balance between cost and value
- Generation of test cases to kill mutants

Thank You