Killing Same and Different Location Multiple Mutants

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Software Engineering

Thapar University, Patiala

By:

Updesh Kumar Jaiswal
(80631025)

Under the supervision of:
Mr. Ajay Kumar
Lecturer, CSED

JULY 2008

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 14700

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, **"Killing Same and Different Location Multiple Mutants"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering submitted to Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Mr. Ajay Kumar* and refers other researcher's works which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

(Updesh Kumar Jaiswal)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Mr. Ajay Kumar)
Supervisor
Computer Science and Engineering Department
Thapar University

Patiala

Countersigned by

(Dr. (Mrs.) Seema Bawa)
Professor & Head
Computer Science & Engineering Department
Thapar University
Patiala

(Dr. R.K. Sharma)
Dean (Academic Affairs)
Thapar University
Patiala

ACKNOWLEDGEMENT

"The need to be right all the time is the biggest bar to new ideas. It is better to have enough ideas for some of them to be wrong than to be always right by having no ideas at all."

- Edward de Bono

I am highly thankful to my guide Mr. Ajay Kumar, Lecturer, Computer Science & Engineering Department, Thapar University, Patiala, for his advice, motivation, guidance, moral support, efforts and the attitude with which he solved all of my queries in making this thesis possible. It has been a great honor to work under him.

I am also thankful to Dr. (Mrs.) Seema Bawa, Professor and Head**,** Computer Science and Engineering Department, Thapar University, Patiala for providing us with adequate infrastructure in carrying the research work.

I am also thankful to Ms. Damandeep Kaur (PG Coordinator) and other faculty members for their contribution in making this thesis possible. Besides the above dignitaries, I am also thankful to my friends Ramesh Pandey and Arun Arora for their kind and noble support.

I am forever indebted to my parent for their constant support and encouragement throughout the past years. This thesis is devoted to them.

Updesh Kumar Jaiswal

ABSTRACT

Software testing is an important technique for assurance of software quality and mutation testing is the White -box, fault -based testing technique for Unit testing. For mutation testing usually we generate test data according to one mutant at one time, so for killing all the mutants by this technique large size of test suite required. But in this thesis work we propose a new approach to generate one test data according to multiple mutants that are mutated at the same location or mutated at different locations and this test data can kill multiple mutants at one time. Reachability condition, necessity condition and sufficiency condition are three conditions which must be satisfied for a test data to kill a mutant and the reachability condition and the necessity condition of a mutant can be acquired when the mutant is generated Mutants mutated at the same location have the same reachability conditions and their necessity conditions are of similar structure. For killing multiple mutants we will combine the necessity conditions of some same-location mutants and different-location mutants into one necessity condition and generate one test data to satisfy the shared reachability conditions and the combined necessity condition. We can find the lesser number of test data inputs than all the test data inputs which are used to kill all same location mutants by acquiring , combining, reusing , minimizing and defining the range of all the test data inputs used to kill same-location or different-location mutants.

Thus our proposed approach can generate smaller test suite of very less cost that can achieve the same mutation testing score.

Keywords: test data generation, mutant, mutation testing, mutation operator.

TABLE OF CONTENTS

LIST OF FIGUERS

FIGURE **Page No.**

CHAPTER 1

INTRODUCTION TO SOFTWARE TESTING

1.1 Software Testing

Testing is a process of executing a program with the intent of finding an error. Testing is the most critical phase in the software development life cycle. Testing software is far more complex than exercising a program to see if it works. Each review, inspection, audit, walk-through, group code read, all is in reality a form of test. The more effective that can make early is static testing, the fewer problems will encounter in the dynamic stages of testing. It has shown that if earlier a fault is detected and removed then additional development cost associated with removing the error will be lower. Preparation for testing should begin as soon as each software product is defined.

The increasing visibility of software as the system element and the attendant "costs" associated with a software failure are motivating forces for well-planned, thorough testing. It is not unusual for a software development organization to expend between 30 and 40 percent of total project effort on testing. In the extreme, testing of human-rated software can cost three to five times as much as all other software engineering activities combined.

1.1.1 Error, Fault, Failure and Test suite

Error: Human action that causes a software fault.

Fault: Missing or incorrect code that may result in a failure.

Failure: Manifested inability of a system or component to perform a required function within specified limits [8].

Bug: Error or fault.

Test case: Set of inputs, execution conditions and expected results developed for a particular objective.

Test suite: Collection of test cases, typically related by a testing goal or an implementation dependency.

Test driver: Class or utility program that applies test cases to an implementation under test.

Test harness: System of test drivers and other tools that supports test execution.

Oracle: Means to produce expected results.

Stub: Partial temporary implementation of a component [1].

Test strategy: Algorithm or heuristic to create test cases from a representation, implementation or a test model.

1.1.2 Test Case Adequacy and Inadequacy

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.

Test Case Inadequacy: If the original program and all mutant programs generate the same output, the test case is inadequate.

1.1.3 Test Effectiveness vs. Efficiency

Test effectiveness: Relative ability of testing strategy to find bugs in the software.

Test efficiency: Relative cost of finding a bug in the software under test [8].

1.1.4 Quality Attributes of a Test Case

- How effective in detecting defects?
- How exemplary? (The more perfect, the less test cases needed)
- How economic?
- How evolvable? (Maintenance Effort)

1.2 Objective of Software Testing

The main objective of Software testing is to prove that the software product meets a set of pre-established acceptance criteria under a prescribed set of environmental circumstances. There are two components to this objective. The first component is to prove that the requirements specification from which the software was designed is correct. The second component is to prove that the design and coding correctly respond to the requirements [1]. Correctness means that function, performance and timing requirements match acceptance criteria.

Software testing is further complicated by the fact that system acceptance criteria usually involve hardware, procedures and operators so that acceptance tests involve more than just the software. Software tests are designed to force failures. In that regard, software testing is intrinsically destructive.

Software Testing objectives are:

- Testing is a process of executing a program with the intent of finding an error.
- A good test case is one that has a high probability of finding an as-yet undiscovered error.
- A successful test is one that uncovers an as-yet undiscovered error.

1.3 Testing Principles

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. Davis suggests a set of testing principles that are given as [2]:

1. All tests should be traceable to customer requirements: The objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

2. Tests should be planned long before testing begins: Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

3. The Pareto principle applies to software testing: Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

4. Testing should begin "in the small" and progress toward testing "in the large": The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

5. Exhaustive testing is not possible: The number of path permutations for even a moderately sized program is exceptionally large. Thus it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

6. To be most effective, testing should be conducted by an independent third party: By most effective, we mean testing that has the highest probability of finding errors the software engineer who created the system is not the best person to conduct

all tests for the software. Thus testing should be conducted by an independent third party.

1.4 Characteristics of a "Good" Test

Following are the characteristics of a good test [2]:

- **A good test has a high probability of finding an error**: To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.
- **A good test is not redundant:** Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose.
- **A good test should be "best of breed":** In a group of tests that have a similar intent, time and resource limitations may militate for the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.
- **A good test should be neither too simple nor too complex:** Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

1.5 Testing Process

The IEEE829 standard [3] describes a framework within which the entire testing process can be managed. The framework allows easy communication between members of a testing project organizes the testing process and outlines the documents that should be made part of any compliant testing process.

Figure 1.1 shows all the activities of software testing according to IEEE829.

1.5.1 Test Plan

Test plan describes the scope, approach, resources and schedule of testing activities in a given project and identifies the items to be tested, the features of those items to be tested, the individual testing tasks that are to be performed and personnel responsible for those tasks, along with the risks associated with the plan.

Test plan should have the following structure:

- Test plan identifier

- Introduction
- Items to be tested

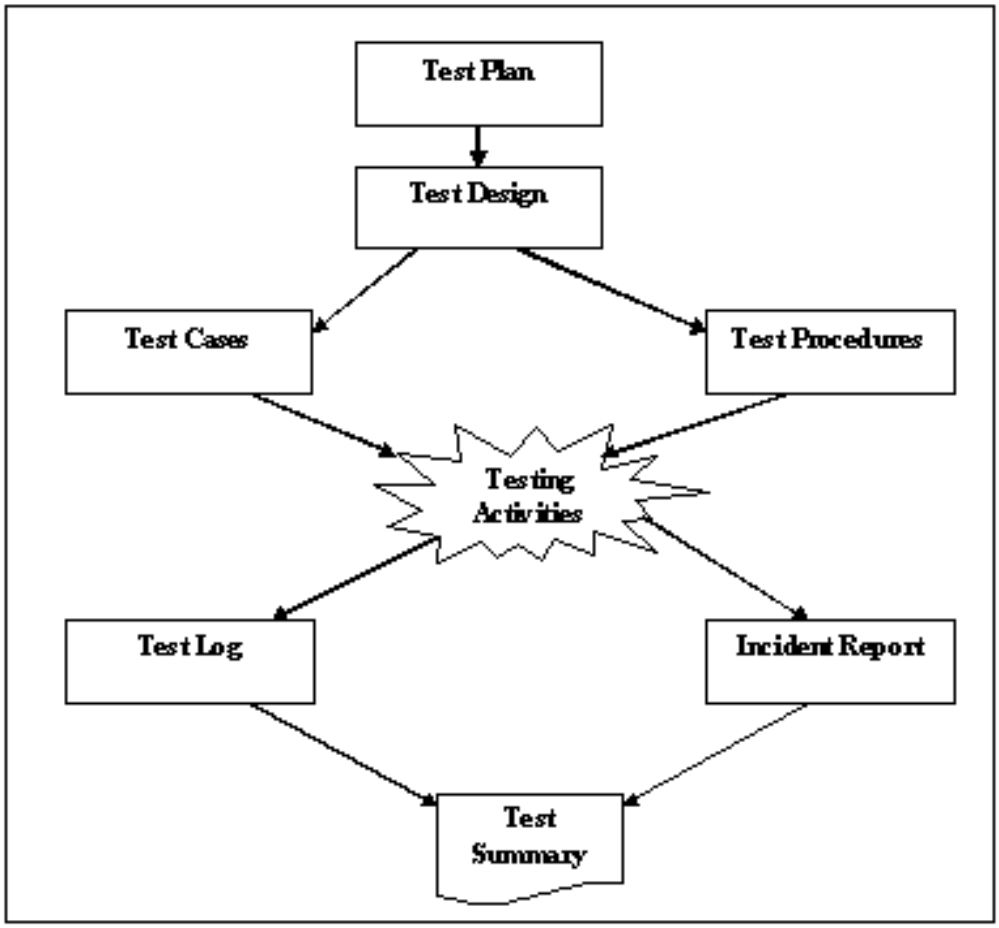

Figure 1.1: Activities of Software Testing

- Features to be tested
- Features not to be tested
- Testing approach
- Test item pass/fail criteria
- Test suspension criteria
- Test resumption requirements
- Testing tasks
- Environmental needs
- Responsibilities

- Staffing and training requirements
- Schedule
- Risks and contingencies
- Approvals

1.5.2 Test Design

Test design or test specification further refines the testing approach and identifies the test cases and test procedures and test item pass/fail criteria.

Test design specification should have the following structure:

- Test design identifier
- Features to be tested
- Approach refinements
- Test identification
- Pass/fail criteria

1.5.3 Test Cases

Individual test cases document the values that will be used as input to individual tests, together with the associated expected outputs. A test case identifies any constraints on the test procedure resulting from the use of the test case and separated from the test design to allow easy reuse in other situations.

Test cases should have the following structure:

- Test case identifier
- Items to be tested
- Input specifications
- Expected output specifications
- Environmental prerequisites
- Special procedural requirements
- Inter-case dependencies

1.5.4 Test Procedures

Test procedure describes the exact steps required to operate the system and execute test cases in order to implement the test design. Test procedure is kept separate from the test design as it is followed step by step and does not contain irrelevant detail.

Test procedure should have the following structure:

- Test procedure identifier
- Purpose

- Special requirements
- Procedure steps

1.5.5 Test Logs

Test logs are used to record what occurred during execution of a test or set of tests. Test logs may either be manually created as tests are executed or automatically by the system as testing processes.

Test logs should have the following structure:

- Test log identifier
- Description
- Activity and event entries

1.5.6 Incident Reports

Incident reports are used to provide a description of any events that occur during testing that require further investigation.

Incident reports should have the following structure:

- Incident report identifier
- Summary
- Incident description
- Impact of the incident

1.5.7 Test Summary Report

Test summary report summarizes the testing activities associated with one or more test designs.

Test summary report should have the following structure:

- Test summary report identifier
- Summary
- Variances
- Comprehensive assessment
- Summary of results
- Evaluation
- Summary of activities
- Approvals

1.6 Software Testing Approaches

Following are two types of software testing approaches:

1.6.1 Top-down Approach

The top-down approach starts from the top of the hierarchy and then incrementally adds modules that it calls and tests the new combined system. This approach of testing requires stubs to be written. A stub is a dummy routine that simulates a module [4]. In the top-down approach, a module (or a collection) cannot be tested in isolation because they invoke some other modules. To allow the modules to be tested before their subordinates have been coded, stubs simulate the behavior of the subordinates.

1.6.2 Bottom-up Approach

The bottom-up approach starts from the bottom of the hierarchy. First the modules at the very bottom, which have no subordinates, are tested. Then these modules are combined with higher-level modules for testing. At any stage all the subordinate modules exist and have been tested earlier. To perform bottom-up testing, drivers are needed to set up the appropriate environment and invoke the module. It is the job of the driver to invoke the module under testing with the different set of test cases.

1.7 Software Testing Techniques

Static Testing and Dynamic Testing are two types of Software Testing Techniques.

1.7.1 Static Testing

The term static testing refers to testing the software requirement specification (SRS), software design specification (SDS) and other non-executable items through requirement analysis, audits, desk checks, inspections, walk-through etc [1]. Static testing is employed to verify the correctness of requirements, designs and code before execution of test cases. Static testing will also audit newly developed or reused code for adherence to established standards. A successful static test of a software module depends upon several things going right:

- A correct allocation of requirements to the software components.
- A correct partitioning and sub allocation of software requirements to the module.
- Successful (correct) module design.
- A successful translation of the intermediate code (pseudo-code, POL, etc.) into programming language statements.

However, true representative test cases must be successfully executed before the testing and integration team certifies a software module. Usually this step is not a part of static testing.

The purposes of the static testing are:

- Validating the requirement specifications.
- Looking for omissions, inconsistencies, redundancies in all documents and source code.
- To ensure that the documents of design and coding conform to the specification.

1.7.2 Dynamic Testing

Dynamic testing is a term that describes the development of test cases, test procedures, the execution of test cases, the structure of test logs and anomaly or incident reports. There are two popular ways to perform dynamic testing, namely black box testing and white (glass) box testing. Either of these two methods requires a set of well-developed and well-structured test cases.

Dynamic testing cannot prove the absolute correctness of a software product unless it is performed in an exhaustive manner [5]. An exhaustive test requires a set of test cases that guarantees the following: explicitly exercises every possible, module path and every possible combination of paths with every possible module input and every possible combination of module inputs.

1.8 Classification of Test Techniques

Classification of test techniques can be done on the base of the source of information used to derive test cases and According to the criterion to measure the adequacy of the set of test cases.

1.8.1 Based on the Source of Information used to derive Test cases

White-Box Testing and Black-Box are based on the source of information used to derive test cases [2].

1. White-Box Testing: White-box testing, sometimes called glass-box testing is a test case design method that uses the control structure of the procedural design to derive test cases.

Using white-box testing methods, the software engineer can derive test cases that:

1. Guarantee that all independent paths within a module have been exercised at least once.
2. Exercise all logical decisions on their true and false sides.
3. Execute all loops at their boundaries and within their operational bounds.
4. Exercise internal data structures to ensure their validity.

Thus white box testing is the testing of the underlying implementation of a piece of software (e.g. source code) without regard to the specification (external description) for that piece of software. The goal of white box testing of source code is to identify infinite loops and paths through the code which should be allowed but which cannot be executed and dead (unreachable) code.

White Box Testing Techniques includes [1]:

Control flow testing: Testing on the basis of the flow of control of a program. It is of the following Types:

1. Statement: each statement executed at least once
2. Branch: each branch traversed at least once
3. Condition: each condition True at least once and False at least once.
4. Branch/Condition: both Branch and Condition coverage achieved.
5. Multiple Conditions: Multiple Conditions coverage technique states that test cases must be written such that all possible combinations of conditions in each decision are taken at least once.
6. Loop: Loop coverage technique states that test cases must be written to test the loop counters.

Data flow testing: Testing on the basis of the flow of control of a program. It is of the following types:

1. All Definition-uses: It requires that every definition of every variable to every use of that definition be exercised under the test.
2. All Uses: In this test set include at least one path segment from every definition of every variable to every use of that definition
3. All p-uses/some c-uses: In All p-uses/some c-uses, test set include at least one path segment from every definition of every variable to every predicate use of that definition; if there are some definitions of variables that do not follow this then add computational use test cases are required to cover every definition.
4. All c-uses/ some p-uses: In All c-uses/some p-uses, test set include at least one path segment from every definition of every variable to every computational

use of that definition; if there are some definitions of variables that do not follow this then add predicate use test cases are required to cover every definition.

5. All definitions: In this, test set includes every definition of every variable be covered by at least one use of that variable, so it will be computational use or predicate use.
6. All p-uses: In All p-uses, test set include at least one path segment from every definition of every variable to every predicate use of that definition; if there are some definitions of variables that do not follow this then leave them.
7. All c-uses: In All c-uses, test set include at least one path segment from every definition of every variable to every computational use of that definition; if there are definitions of variables that are not covered by the above prescription then, leave them.

2. Black-Box or Functional Testing: Black-box testing, also called behavioral testing, focuses on the functional requirements of the software. That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black box testing is the testing of a piece of software without regard to its underlying implementation. The goal of black box testing is to demonstrate that the software being tested does not adhere to its external specification.

Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.

Black-box testing attempts to find errors in the following categories:

1. Incorrect or missing functions
2. Interface errors
3. Errors in data structures or external data base access
4. Behavior or performance errors
5. Initialization and termination errors

Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing because black box testing purposely disregards control structure and attention is focused on the information domain. Tests are designed to answer the following questions:

• How is functional validity tested?

• How is system behavior and performance tested?

• What classes of input will make good test cases?

• Is the system particularly sensitive to certain input values?

• How are the boundaries of a data class isolated?

• What data rates and data volume can the system tolerate?

• What effect will specific combinations of data have on system operation?

By applying black-box techniques, we derive a set of test cases that satisfy the following criteria:

1. Test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing.
2. Test cases that tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

Black Box Testing Techniques includes:

- Equivalence Partitioning
- Boundary Value Analysis

1.8.2 According to the Criterion to Measure the Adequacy of the Set of Test cases

Coverage based, Error-based and Fault based test techniques are divided according to the criterion to measure the adequacy of the set of test cases.

1. Coverage based Test Techniques: Coverage based test techniques are of following types:

- Control-flow coverage
- All-paths coverage
- Statements (nodes) coverage: execute every statement at least once.
- All-Edges (branches) coverage: exercises the true and false outcome of every decision.
- Condition coverage: all conditions in a decision take on both outcomes (e.g.(x<1) or (x> 20)).
- Multiple condition coverage (extended branch coverage)
- Data-flow Coverage: examine how variables are treated along the paths.
- Coverage of Requirements Specifications

2. Error-based Test Techniques: Error-based Test Techniques perform following things:

- Focuses on testing error-prone points.

- Domain testing, Partition Analysis, Equivalence partitioning, Boundary value analysis.
- Look at values from the input space that causes particular paths to be executed.
- Chop domain into regions that cause same path to be executed.
- Examine borders to detect "border shifts".

3. Fault based test techniques: If this we seed the implementation with a fault (mutating the original program) by applying a mutation operator and then we determine whether testing identifies this fault.

Example of Fault-based testing is Mutation Testing.

CHAPTER 2
MUTATION TESTING

2.1 Introduction of Mutation Testing

Mutation Testing is White-Box, fault -based testing technique that provides strong quality assurance. Mutation testing (sometimes also called mutation analysis) is a method of software testing, which involves modifying program's source code in small ways. Mutation testing is based on well-defined mutation operators that either mimic typical user mistakes (such as using the wrong operator or variable name) or force the creation of valuable tests (such as driving each expression to zero). The purpose is to help the tester develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution. Mutation Testing should be used in conjunction with traditional testing techniques, not instead of them.

Mutation Testing is a testing technique that focuses on measuring the adequacy of test cases. Mutation Testing involves the following things:

- Faults are introduced into the program by creating many versions of the program called mutants.
- The mutant contains a simple modification of the initial program.
- 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 case.

2.2 Assumptions

There are two assumptions are considered in case of Mutation Testing which are:

1. Competent programmer assumption

- Programmers are not so stupid.
- They produce programs that are close to being correct.
- Faults should be detectable as small deviations from the intended program.

2. Coupling effect assumption

If a test suite kills a mutant, it also can kill mutants of mutants.

2.3 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 of mutant program:

```
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  }
```

2.4 Types of Mutant

Killed Mutant: Test cases detect the one fault injected via a mutation operator.

Alive/Live Mutant: Test cases do not detect the one fault injected via a mutation operator.

Equivalent Mutant: A mutant that produces always identical results with the original program [10].

2.4.1 Equivalent Mutant

A mutant that produces always identical results with the original program is called equivalent mutants. Many mutation operators can produce equivalent mutants.

For example, let's consider the following code fragment:

```
int index=0;
while (...)
{
    …….. ;
   index++ ;
   if (index==10)
     break ; }
```

Boolean relation mutation operator will replace "==" with ">=" and produce the following mutant:

```
int index=0;
while (...)
{
   ……. ;
   index++;
   if (index>=10)
     break;
}
```

However, it is not possible to find a test case, which could kill this mutant. The resulting program is equaivalent to the original one. Such mutants are called equivalent mutants.

2.5 Mutation Score

Mutation Score of Program P and test set T, measures the effectiveness of our test set.

MS (P,T) = [K/(M- E)] *100 %

Where K is total number of killed mutants, M is total number mutants generated and E is total number equivalent mutants.

If all mutants have been killed then there will be no equivalent mutant in program and in this case, MS = 100%.

2.6 Conditions for a Test data to kill a Mutant

Three conditions must be satisfied for a test data to kill a mutant [11,12].

If we use P to denote a program, M to denote a mutant of P on statement S and T to denote a test data for P, the preceding three conditions are as follows:

1. Reachability condition: S must be reached, because a mutant is presented as a syntactic change to an executable statement and the other statements in the mutated program are syntactically equal to the statements in the original program. If T can not reach S, it is guaranteed that T will not kill M [12].

2. Necessity condition: T must be able to cause M to have a different state from P on S. For T to kill M, it is necessary that if S is reached and at the same time, the state of

M which immediately follows some execution of S must be different from that of P at the same location [12]. It is because that M is syntactically equal to P except for the mutated statement S. Thus, if the states of the two programs do not differ after some execution of S, they will never differ.

3. Sufficiency condition: The final state of M must be different from that of P. That is, the different state caused by the necessity condition must propagate through the program's computation to result in an output difference.

2.7 Weak Mutation Testing and Strong Mutation Testing

Mutation testing is done by selecting a set of mutation operators and then applying them to the source program one at a time for each applicable piece of the source code. The result of applying of one mutation operator to the program is called a mutant. If the test suite is able to detect the change (i.e. one of tests fails), then mutant is said to be killed. For example, let's consider the following C++ code fragment:

```
if (a && b)
  c = 1;
else
  c = 0;
```

The condition mutation operator would replace '&&' with '||' and produce the following mutant:

```
if (a || b)
   c = 1;
else
   c = 0;
```

Now, for the test to kill this mutant, the following condition should be met:

1. Test input data should cause different program states for mutant and original program. For example, a test with a=1 and b=0 would do this.
2. The value of 'c' should be propagated to the program's output and checked by the test.

Weak mutation testing (or weak mutation coverage): It requires that only the first condition is satisfied.

Strong mutation testing: It requires that both conditions are satisfied. Strong mutation is more poweful, since it ensures that the test suite can really catch the

problems. Weak mutation is closely related to code coverage methods. It requires much less computing power to ensure that the test suite satisfies weak mutation testing than strong mutation testing.

CHAPTER 3
MUTATION OPERATOR

3.1 Introduction of Mutation Operator

Mutation operator is a rule that is applied to a program to create mutants. Mutation operator describes syntactic changes to be made to the program.

Mutation operators are used as:

- Replace each operand by every other syntactically legal operand.
- Modify expressions by replacing operators and inserting new operators.
- Delete entire statements, etc.

3.2 Categories of Mutation Operators

Mothra Mutation Operators and Class mutation Operators are categories of Mutation Operator.

3.2.1 Mothra Mutation Operators

The complete set of mutation operators used by the Mothra mutation system is shown in Figure 3.1. Each of the 22 mutation operators is represented by a three-letter acronym. For example, the 'Array reference for array reference replacement' (AAR) mutation operator causes each array reference in a program to be replaced by each other distinct array reference in the program.

Offutt et al. divided the Mothra mutation operators into three categories according to the syntactic elements that they modify: statements, operands and operators of expressions. These categories are given as

1. Operand Replacement Operators

Replace a single operand with another operand or constant.

For example, if (x > y) is a statement in a given program then Operand Replacement Operators can be generated as:

- 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.

2. Expression Modification Operators

3. Replace an operator or insert new operators. For example, if (x ==y) is a statement in a given program then Expression Modification Operators can be generated as:

- if (x >= y) Replacing == by >=
- if (x == ++y) Inserting ++

3. Statement Modification Operators

Delete the else part of the if-else statement, Delete the entire if-else statement and Replace line 3 by a return statement.

Operator	**Description**
AAR	Array reference for array reference replacement
ABS	Absolute value insertion
ACR	Array reference for constant replacement
AOR	Arithmetic operator replacement
ASR	Array reference for constant replacement
CAR	Constant for array reference replacement
CNR	Comparable array name replacement
CRP	Constant replacement
CSR	Constant for scalar variable replacement
DER	DO statement end replacement
DSA	DATA statement replacement
GLR	GOTO label replacement
LCR	Logical connector replacement
ROR	Relational operator replacement
RSR	RETURN statement replacement
SAN	Statement analysis
SAR	Scalar variable for array reference replacement
SCR	Scalar for constant replacement
SDL	Statement deletion
SRC	Source constant replacement
SVR	Scalar variable replacement

UOI	Unary operator insertion

Figure3.1: Mothra Mutation Operators [9]

3.2.2 Class Mutation Operators & Mutation Operators for Java

We classify the class mutation operators into six groups, based on the language feature that is affected. The first four groups are based on language features that are common to all Object-oriented (OO) languages. The fifth group includes language features that are Java-specific and the last groups of mutation operators are based on common OO programming mistakes [15].

1. Information Hiding (Access Control)
2. Inheritance
3. Polymorphism
4. Overloading
5. Java-Specific Features
6. Common Programming Mistakes

Note: Symbol $^{\Delta}$ will represent presence of a mutant in a particular statement in all examples of class mutation operators.

1. Information Hiding (Access Control)

The mutation operator AMC is used to test for access control faults. In Java mainly public, private and protected access levels are used.

AMC(Access modifier change): The AMC operator changes the access level for instance variables and methods to other access levels. The purpose of the AMC operator is to guide testers to generate test cases that ensure that accessibility is correct. This mutant can only be killed if the new access level denies access to another class or allows access that causes a name conflict. Because the ability to kill this mutant depends on other characteristics of the software, many of these mutants can be expected to be equivalent.

Original Code	**AMC Mutants**
public Stack s;	$^{\Delta}$ private Stack s;
	$^{\Delta}$ protected Stack s;
	$^{\Delta}$ Stack s;

2. Inheritance

Although a powerful and useful abstraction mechanism, incorrect use of inheritance can lead to a number of faults. We define five mutation operators to try to test the various aspects of using inheritance, covering variable hiding, method overriding, the use of super and definition of constructors. Overriding can cause instance variables that are defined in a subclass to hide member variables of the parent. This powerful feature can cause an incorrect variable to be accessed. Thus it is necessary to ensure that the correct variable is accessed when variable overriding is used, which is the intent of the IHD and IHI mutation operators.

IHD(Hiding variable deletion): The IHD operator deletes a declaration of an overriding or hiding variable. This causes references to that variable to access the variable defined in the parent (or ancestor). This mimics a common mistake that programmers make. This mutant can only be killed by a test case that is able to show that the reference to the parent variable is incorrect.

Original Code		**IHD Mutant**
class List {		class List {
int size;		int size;
... ...		… ..
}		}
class Stack extends List {		class Stack extends List {
int size**;**	Δ	// int size;
… …		… …
}		}

IHI(Hiding variable insertion): The IHI operator inserts hiding member variables to hide the parent's version of a variable. This mutant can only be killed by a test case that is able to show that the reference to the overriding variable is incorrect.

Original Code		**IHI Mutant**
class List {		class List {
int size;		int size;
... ...		... ...
}		}
class Stack extends List {		class Stack extends List {
	Δ	int size;
... ...		... ...

```
}                                                    }
```

IOD(Overriding method deletion): The ability of a subclass to override a method declared by an ancestor allows a class to modify the behavior of the parent class. When there are more than one method of the same name, it is important for testers to ensure that a method invocation actually invokes the intended method. The IOD operator deletes an entire declaration of an overriding method in a subclass so that references to the method use the parent's version. This mutant is killed by a test case that is able to show that the behavior of the parent's method is incorrect.

Original Code

```
class Stack extends List {
 ... ...
Push (int a) { …  }
}
```

IOD Mutant

```
 class Stack extends List {
  ... ...
Δ    // Push (int a) { ... }
 }
```

IOP(Overridden method calling position change): Sometimes, an overriding method in a child class needs to call the method it overrides in the parent class. This may happen if the parent's method uses a private variable v, which means the method in the child class may not modify v directly. However, an easy mistake to make is to call the parent's version at the wrong time, which can cause incorrect state behavior. The IOP operator moves calls to overridden methods to the first and last statements of the method and interchange the statement.

Original Code

```
class List {
... ...
void SetEnv()
{size = 5; ...}
}
class Stack extends List {
... ...
void SetEnv() {
 super.SetEnv();
size = 10;
}
}
```

IOP Mutant

```
class List {
  ... ...
  void SetEnv()
    {size = 5; ... }
 }
 class Stack extends List {
    ... ...
    void SetEnv() {
Δ       size = 10;
Δ      super.SetEnv();
   }
}
```

IOR(Overridden method rename): The IOR operator is designed to check if an overriding method adversely affects other methods. Consider a method m() that calls another method f(), both in a class List. Further, assume that m() is inherited without change in a child class Stack, but f() is overridden in Stack. When m() is called on an object of type Stack, it calls Stack's version of f() instead of List's version. In this case, Stack's version of f() may have an interaction with the parent's version that has unintended consequences. The IOR operator renames the parent's versions of these above given methods so that the overriding cannot affect the parent's method. These mutants can only be killed by a test case that causes different behavior when the overriding (child's) version is not called.

Original Code	**IOR Mutant**
class List {	class List {
... ...	... ...
void m() {... f(); ... }	Δ void m() {... f'(); ... }
void f() { ... }	Δ void f'() { ... }
}	}
class Stack extends List {	class Stack extends List {
... ...	... ...
void f() { ... }	void f() { ... }
}	}

ISK(super keyword deletion): If a subclass hides an instance variable of one of its ancestors, it can still reference the hidden variable by using the super keyword. The subclass can also use super to invoke a parent's version of a method that has been overridden. The ISK operator deletes occurrences of the super keyword so that a reference to the variable or the method goes to the overriding instance variable or method. The ISK operator is designed to ensure that hiding/hidden variables and overriding/overridden methods are used appropriately.

Original Code	**ISK Mutant**
class Stack extends List {	class Stack extends List {
... ...	... ...
int MyPop( ) {	int MyPop( ) {
... ...	... ...
return val*super.num;	Δ return val*num;

```
    }                                   }
}                                   }
```

IPC(Explicit call of a parent's constructor deletion): Although constructors are not inherited the way other methods are, a constructor of the superclass is invoked when subclasses are instantiated. When we create new objects of a derived class, the default constructor (no arguments) for the parent class is automatically called first and then the constructor of the derived class is called. However, the subclass can use the super keyword to call a specific parent class constructor. This is usually done to pass arguments to one of the parent class's non-default constructors. The IPC operator deletes super constructor calls, causing the default constructor of the parent class to be called. To kill mutants of this type, it is necessary to find a test case for which the parent's default constructor creates an initial state that is incorrect.

Original Code **IPC Mutant**

```
class Stack extends List {          class Stack extends List {
 ... ...                                ... ...
Stack (int a) {                     Stack (int a) {
 super (a);                          Δ  // super (a);
... ...                                 ... ...
    }                                      }
}                                      }
```

3. Polymorphism and Dynamic Binding

Object references can have different types with different executions. That is, object references may refer to objects whose actual types differ from their declared types. The actual type can be of any type that is a subclass of the declared type. Polymorphism allows the behavior of an object reference to differ depending on the actual type. Therefore, it is important to identify and exercise the program with all possible type bindings. The polymorphism mutation operators are designed to ensure this type of testing.

PNC(new method call with child class type): It changes the instantiated type of an object reference. This causes the object reference to refer to an object of a type that is different from the declared type. In the example, class A is the parent of class B.

Original Code **PNC Mutant**

A a;		A a;
a = new A();	Δ	a = new B();

PMD(Member variable declaration with parent class type): The PMD operator changes the declared type of an object reference to the parent of the original declared type. The instantiation will still be valid (it will still be a descendant of the new declared type). To kill this mutant, a test case must cause the behavior of the object to be incorrect with the new declared type. In the example below, class A is the parent of class B.

Original Code		**PMD Mutant**
B b;	Δ	A b;
b = new B();		b = new B();

PPD(Parameter variable declaration with child class type): The PPD operator is the same as the PMD, except that it operates on parameters rather than instance and local variables. It changes the declared type of a parameter object reference to be that of the parent of its original declared type. In the example below, class A is the parent of class B.

Original Code		**PPD Mutant**
boolean equals (B o) { ... }	Δ	boolean equals (A o) { ... }

PRV(Reference assignment with other compatible type): Object references can refer to objects of types that are descendants of its declared type. The ORR operator changes operands of a reference assignment to be assigned to objects of subclasses. In the example below, obj is of type Object and in the original code it is given an object of type String. In the mutated code, it is given an object of type Integer.

Original Code		**OAC Mutant**
Object obj;		Object obj;
String s = “Hello”;		String s = “Hello”;
Integer i = new Integer(4);		Integer i = new Integer(4);
obj = s;	Δ	obj = i;

4. Method Overloading

Method overloading allows two or more methods of the same class to have the same name as long as they have different argument signatures. Just as with method overriding, it is important for testers to ensure that a method invocation invokes the

correct method with appropriate parameters. Five mutation operators are defined to test various aspects of method overloading.

OMR(Overloading method contents change): The OMR operator is designed to check that overloaded methods are invoked appropriately. The OMR operator replaces the body of a method with the body of another method that has the same name. This is accomplished by using the keyword *this*.

Original Code		**OMR Mutant**
class List {		class List{
... ...		... ...
void Add (int e) { ... ... }		void Add (int e) { ... ... }
void Add (int e, int n) {		void Add (int e, int n) {
... ...	Δ	this.Add(e);
}		}
}		}

OMD(Overloading method deletion): The OMD operator deletes overloading method declarations, one at a time in turn. If the mutant still works correctly without the deleted method, there may be an error in invoking one of the overloading methods; the incorrect method may be invoked or an incorrect parameter type conversion has occurred. The OMD operator ensures coverage of overloaded methods, that is, all the overloaded methods must be invoked at least once.

Original Code		**OMD Mutant**
class Stack extends List {		class Stack extends List {
... ...		... ...
void Push (int i) { ... }	Δ	// void Push (int i) { ... }
void Push (float i) { ... }		void Push (float i) { ... }
}		}

OAO(Argument order change): The OAO operator changes the order of the arguments in method invocations, but only if there is an overloading method that can accept the new argument list. If there is one, the OAO operator causes a different method to be called, thus checking for a common fault in the use of overloading.

Original Code		**OAO Mutant**
s.Push (0.5, 2);	Δ	s.Push (2, 0.5);

OAN(Argument number change): The OAN operator changes the number of the arguments in method invocations, but only if there is an overloading method that can accept the new argument list. Again, this helps ensure that the programmer did not invoke the wrong method. When new values need to be added, they are the constant default values of primitive types or the result of the default constructors for objects.

Original Code	**OAN Mutants**
s.Push (0.5, 2);	Δ s.Push (2); Δ s.Push (0.5); Δ s.Push ();

5. Java-Specific Features

Java includes a few object-oriented language features that do not occur in all OO languages. This group of four operators attempt to ensure correct use of these features.

JTD(this keyword deletion): The JTD operator deletes uses of the keyword this. Within a method body, uses of the keyword this refers to the current object. Typically, methods can refer directly to the object's instance variables by name. However, sometimes a member variable is hidden by a method parameter that has the same name, so this must be used. The JTD operator checks if the member variables are used correctly if they are hidden by method parameters by replacing occurrences of "*this.X*" with "*X*" when "*X*" is both a parameter and an instance variable.

Original Code	**JTD Mutant**
class Stack {	class Stack {
int size;	int size;
... ...	... ...
void setSize (int size) {	void setSize (int size) {
this.size=size;	Δ size=size;
}	}
}	}

JSC(static modifier change): The JSC operator removes the static modifier to change class variables to instance variables and it adds the static modifier to change instance variables to class variables. It is designed to validate behavior of instance and class variables.

Original Code	**JSC Mutant**

```
public static int s = 100;          public int s = 100;
private String name;          Δ     public static String name;
```

JID(Member variable initialization deletion): Instance variables can be initialized in the variable declaration and in constructors for the class. The JID operator removes the initialization of member variables in the variable declaration so that member variables are initialized to the appropriate default values of Java. This is designed to ensure correct initializations of instance variables.

Original Code	**JID Mutant**
class Stack {	class Stack {
int size = 100;	Δ int size;
... ...	... ...
Stack() { ... ... }	Stack() { ... ... }
}	}

JDC(Java-supported default constructor create): Java creates default constructors if a class contains no constructors. The JDC operator forces Java to create a default constructor by deleting the implemented default constructor. It is designed to check if the user-defined default constructor is implemented properly.

Original Code	**JDC Mutant**
class Stack {	class Stack {
... ...	... ...
Stack() { ... ... }	Δ // Stack() { ... ... }
}	}

6. Common Programming Mistakes

This category of mutation faults attempts to capture typical mistakes that programmers make when writing OO software. These are related to use of references and using methods to access instance variables.

EOA(Reference assignment and content assignment replacement): Object references in Java are always through pointers. Although pointers in Java are typed, which is considered to help prevent certain types of faults, there are still mistakes that programmers can make. One common mistake is that of using an object reference instead of the contents of the object the pointer references. The EOA operator replaces an assignment of a pointer reference with a copy of the object, using the Java

convention of a clone() method. The clone() method duplicates the contents of an object, creating and returning a reference to a new object.

Original Code		**EOA Mutant**
List list1, list2;		List list1, list2;
list1 = new List();		list1 = new List();
list2 = list1;	Δ	list2 = list1.clone();

EOC(Reference comparison and content comparison replacement): The EOC operator considers another common mistake with objects and object references. Comparisons of object references check whether the two references point to the same data object in memory. To support the comparison of the contents of objects, Java suggests the convention of an equals() method, which should take an object of type Object as a parameter and return a Boolean value; true if the parameter has the same value as the reference object. This mutation operator targets faults programmers can easily make when confusing the reference of an object and its state.

Original Code		**EOC Mutant**
Fract f1 = new Fract (1, 2);		Fract f1 = new Fract (1, 2);
Fract f2 = new Fract (1, 2);		Fract f2 = new Fract (1, 2);
boolean b = (f1==f2);	Δ	boolean b = (f1.equals (f2));

EAM(Accessor method change): Because private instance variables cannot be accessed outside of an object's own methods, good OO programming practice calls for providing public accessor and modifier methods by which clients of an object can effectively manipulate selected private instance variables. These are informally known as "get" and "set" methods and by convention, use the variable name preceded by "get" or "set" (getVariableName()).

The EAM operator changes an accessor method name for other compatible accessor method names, where compatible means that the signatures are the same except the method name. To kill this mutant a test case will have to produce incorrect output as a result of calling the wrong method.

Original Code		**EAM Mutant**
point.getX();	Δ	point.getY();

EMM(Modifier method change): The EMM operator does the same as EAM, except it works with modifier methods ("set") instead of accessors.

Original Code **EMM Mutant**

point.setX (2); Δ point.setY (2);

Operator	Description
AMC	Access modifier change
IHD	Hiding variable deletion
IHI	Hiding variable insertion
IOD	Overriding method deletion
IOP	Overriding method calling point change
IOR	Overriding method rename
ISK	*super* keyword deletion
IPC	Explicit call of a parent's constructor deletion
PNC	New method call with child class type
PMD	Instance variable declaration with parent class type
PPD	Parameter variable declaration with child class type
PRV	Reference assignment with other compatible type
OMR	Overloading method contents change
OMD	Overloading method deletion
OAO	Argument order change
OAN	Argument number change
TD	*this* keyword deletion
JSC	*static* modifier change
JID	Member variable initialization deletion
JDC	Java-supported default constructor create
EOA	Reference assignment and content assignment Replacement
EOC	Reference comparison and content comparison Replacement
EAM	Access method change
EMM	Modifier method change

Figure3.2: Summarization of all Class Mutation Operators [15]

CHAPTER 4

PROBLEM FORMULATION & PROPOSED APPROACH

4.1 Problems in Mutation Based Testing

Software testing is an important technique to assure the quality of software. However, testing is a time consuming process and accounts for about 50% of the cost of software development [2]. One of the most important problems of software testing is how to efficiently generate effective test data. If the problem of automatic test data generation can be well solved, the cost of software testing can be significantly reduced. As Mutation testing is a fault-based testing technique for assessing the effectiveness of test suites in unit testing. Mutation testing relies on the Competent Programmer Hypothesis and the Coupling Effect Hypothesis, so it can provide a means to systematically produce mutants for the original program. Typically, mutants that are syntactically correct variants of the original program unit are generated in mutation testing and all test data in a test suite are assessed according to their ability to distinguish mutants from the original program. Intuitively, the more mutants the test suite can kill, the more effective the test suite can be. Specifically, the ratio of killed mutants to the nonequivalent mutants is used as the mutation adequacy score to indicate test suite quality in mutation testing. Although mutation testing is an effective means for assessing the effectiveness of test suites, manually developing test data that can meet a satisfactory mutation adequacy score required very high labour [16].

For example the Trityp Program in figure 4.1 finds out the Type of the given triangle. To test Trityp Program we can create 12 test cases as:

T1: Valid Equilateral Triangle (for e.g. (2, 2, 2))

T2: Valid Isosceles Triangle (for e.g. (2, 2, 3))

T3: T2 with all Permutations (for e.g. (2, 2, 3), (2, 3, 2), (3, 2, 2))

T4: Valid Scalar Triangle (for e.g. (5, 4, 2))

T5: T4 with all Permutations (for e.g. (5, 4, 2), (5, 2, 4), (4, 2, 5), (4, 5, 2), (2, 5, 4), (2, 4, 5))

T6: One side is Zero (for e.g. (4, 3, 0) or (0, 3, 4))

T7: One side is Negative (for e.g. (5, 2, -4) or (-4, 2, 5))

T8: The sum of two sides is equal to 3rd side (for e.g. (1, 2, 3))

```
void main( ){
int a,b,c,t=0;
if(a<=0 || b<=0 ||c<=0)
   printf("Illegal Triangle ");
if(a==b)
   t =t+1;
if(a==c)
   t =t+2;
if(b==c)
   t =t+3;
if(t==0)
 {
   if(((a+b)<=c)||((b+c)<=a)||((c+a)<=b))
         printf("Illegal Triangle ");
   else
         printf("Scalar Triangle ");
  }
else if(t>3)
  printf("Equilateral Triangle");
else if(t==1 && ((a+b)>c))
  printf("Isosceles Triangle ");
else if(t==2 && ((a+c)>b))
  printf("Isosceles Triangle ");
else if(t==3 && ((b+c)>a))
  printf("Isosceles Triangle  ");
else
  printf("illegal Triangle ");
   }
```

Figure 4.1: Trityp Program

T9: T8 with all Permutations (for e.g. (3, 2, 1), (3, 1, 2), (2, 1, 3), (2, 3, 1),(1, 3, 2), (1, 2, 3))

T10: The sum of two sides is lesser than 3rd side (for e.g. (5, 3, 1))

T11: T10 with all Permutations (for e.g. (5, 3, 1), (5, 1, 3), (3, 1, 5), (3, 5, 1),(1, 5, 3), (1, 3,5))

T12: All the three sides are equal to Zero (for e.g. (0, 0, 0))

Let us consider the mutated location "a==b" of line 5 in Figure 4.1, here we can produce 5 mutants (a<b, a<=b, a>b, a>=b, and a! =b) by applying only ROR (Relational operator replacement). If one test data is generated according to one mutant [16], then for killing all these 5 mutants at least 5 test data which are generated using above mention test cases, are required. As Trityp program has only 28 statements, but by using Mothra mutation operators we can generate 951 mutants thus, for killing all these mutants 951 test data are required.

4.2 Problem Statement

If we use the approach that one test data is generated according to one mutant then large numbers of test data are required to kill all mutants of a program. It means larger test suite are needed for achieving a given mutation score. Thus we can say that for killing all the mutants of a program by using this approach high manual labour are required and cost of generating these test data and the cost of running or re-running these test data will be very high.

As a result our aims are:

- We propose a novel approach that can generate a test suite, in which each test data may kill multiple mutants. Thus, the number of generated test data for achieving a given mutation testing score can be reduced.
- We reuse the reachability conditions and combine some necessity conditions into one to avoid generating one test data for each mutant.
- We propose a new approach to generating one test data according to multiple mutants that are mutated at the same location or different locations at one time.
- Our approach can generate smaller test suite that can achieve the same mutation score.

- We reduce both the cost of generating test data and the cost of running or rerunning test data for killing all the mutants of a program.

Thus we will generate test data for killing multiple mutants that are mutated at the same location (which is referred to as same-location mutants) or mutated at the different location (which is referred to as same-location mutants) at one time which reduces both the cost of generating test data and the cost of running or re-running test data for killing all the mutants of a program.

4.3 Assumptions in Proposed Approach

We will consider the following assumptions in proposed approach:

1. The reachability conditions and the necessity condition of a mutant can be acquired when the mutant is generated.
2. It is uncontrollable to generate test data that can definitely meet the sufficiency condition, thus my approach utilizes the reachability conditions and the necessity condition to generate test data.
3. Mutants mutated at the same location (which is referred to as same-location mutants) have the same reachability conditions and their necessity conditions are of similar structure.
4. We can combine the necessity conditions of some same-location mutants and different-location mutants into one necessity condition and generate one test data to satisfy the shared reachability conditions and the combined necessity condition.
5. We use a path-wise test data generation technique to work the resulting test data.
6. In our approach, the branch predicates that can lead the test data to reach the inner most block containing the mutated location (where the mutant differs from the original program) are used to express reachability conditions.
7. There is no need to figure out a path expression to represent all the branch predicates. Furthermore, it is also not necessary to figure out all the execution paths from the start point of the program to the mutated location.

8. As the necessity conditions of same-location mutants are of similar structure so it is possible to combine some of them into one by conjunction.
9. In our approach we mainly focus on the location that can be applied by only one mutation operator but the mutation operator can produce more than one mutant on that location. These locations are in our approach are Binary Relational Operator and Binary Arithmetic Operator.

In Figure 4.2, the Binary Arithmetic Operator can only be mutated by AOR and the Binary Relational Operator can only be mutated by ROR.

Mutated Location	**Mutant Operators**	**Examples(int a,b)**
Binary Relational Operator	ROR	"=="in "a==b" can be replaced by ">","<",">=", "<=","!="
Binary Arithmetic Operator	AOR	"*" in "a*b" can be replaced by "+","-","/","%"

Figure 4.2: Mutated Locations in Proposed Approach

4.4 Proposed Approach

As the sufficiency condition is usually difficult to acquire, thus my approach focuses on reachability conditions and necessity conditions of mutants.

Our approach has a three-step process:

1. When mutants are generated, the reachability conditions and necessity condition of each mutant are obtained.
2. Some necessity conditions of same-location or different-location mutants are combined into one condition.

3. A program path is formed to generate desire test data according to the reachability conditions of these mutants, the combined necessity condition and the original program.

4.5 Implementation Details

1. Assuming a and b are integers, arithmetic expression a+b can be mutated to a-b, a*b, a/b and a%b by applying mutation operator AOR. This means that we can get 4 mutants whose necessity conditions are "(a+b)!=(a-b)", "(a+b)!=a*b", "(a+b)!=a/b" and "(a+b)!=a%b", respectively.

All these 4 conditions can be combined into one by conjunction: "((a+b)!=(a-b) && (a+b)!=a*b && (a+b)!=a/b && (a+b)!=a%b)", because there are no contradictions in them.

2. Assuming a and b are integers, relational expression a>b can be mutated to a>=b, a==b, a<b, a<=b and a!=b by applying mutation operator ROR. There are 5 mutants generated whose necessity conditions are:

1. (a>b)!=(a>=b)
2. (a>b)!=(a==b)
3. (a>b)!=(a<b)
4. (a>b)!=(a<=b)
5. (a>b)!=(a!=b)

These 5 conditions can be reduced to:

1. a==b
2. a>=b
3. a!=b
4. true
5. a<b

The fourth condition is true, which means any test data can satisfy it. Thus, it will not be considered in combined conditions. In the remaining four conditions (1, 2, 3, 5) there are contradictions between condition 1 and condition 3, between condition 2 and condition 5, between condition 1 and condition 5 respectively. Therefore, we can draw a conclusion that any three conditions of the four contain contradictions and we can only combine two conditions at one time. The combinations originated from two conditions are (1,2), (2,3), and (3,5). However, our approach only selects two

combinations (i.e. (1, 2) and (3,5)) as there is no contradiction in each combination and these two combinations can cover the original remaining four conditions.
Reducing these two combinations, we will get:

1. a==b
2. a<b

Thus after the preceding analysis, the number of necessity conditions be solved from five to two.
Similarly relational expressions a<=b, a<b, a==b, a>=b, and a! =b all have the similar result. Figure 4.3 lists the results.

Expression	Combined Necessity Conditions
a<b	a==b, a>b
a<=b	a<b, a>b
a==b	a<b, a>b
a>=b	a<b, a>b
a>b	a<b, a==b
a!=b	a<b, a>b

Figure 4.3: Combined Necessity Conditions of the Same-location Mutants, Generated by ROR

Thus we can find the lesser number of test data inputs than all the test data inputs which are used to kill all mutants by conjunction, defining, acquiring, combining, reusing and minimizing the reachability and necessity conditions.

CHAPTER 5

EXPERIMENTAL RESULTS

5.1 Test Data Generation for Killing Same-location Mutants

We will perform some experiments and generate test data for killing same-location mutants.

5.1.1 The First Experiment

Let us take an example of a program that can find out the largest integer among the given three integers as input values. This program (LTI program) is shown in Figure 5.1.

```
if(p>q)
 { if(p>r)
   {
      printf("value of p is largest :%d",p);
    }
   else
    {
      printf("value of r is largest :%d",r);
    }
   }
else
 { if(r>q)
   {
     printf("value of r is largest :%d",r);
    }
   else
    {
      printf("value of q is largest :%d",q);
```

```
        }
    }
```

Figure 5.1: LTI Program

To Test LTI program of Figure 5.1, we can create 5 test cases, which are as follow:

T1: All the Three integers are equal (for e.g. (10, 10, 10))

T2: Only Two integers are equal (for e.g. (10, 10, 8))

T3: T2 with all Permutations (for e.g. (10, 8, 10), (10, 10, 8), (8, 10, 10))

T4: All the Three integers have different values (for e.g. (6, 4, 8))

T5: T4 with all Permutations (for e.g. (6, 4, 8), (6, 8, 4), (8, 6, 4), (8, 4, 6), (4, 6, 8), (4, 8, 6))

Thus the test suite of this program has 5 test cases with minimum 10 different test data inputs.

Now we will apply our approach to find out the smaller test suite that can kill Same-location mutants that are generated by ROR.

According to first step of our approach we will generate the mutants by ROR then reachability conditions and necessity condition of each mutant are obtained. In the Figure 5.1, on the mutated location "p>r" in line no. 2, we can generate total 5 mutants by applying ROR as shown in figure 5.2 and the reachability conditions are represented as branch predicate (p>q) and the necessity conditions are:

1. (p>r)!=(p>=r)
2. (p>r)!=(p==r)
3. (p>r)!=(p<r)
4. (p>r)!=(p<=r)
5. (p>r)!=(p!=r)

According to second step of our approach we will combine some necessity conditions of same-location mutants into one condition. Thus after combining and using the results of Figure 4.3 we will get the one necessity condition of these 5 same-location mutants as (p<r && p==r).

According to third step of our approach we will form a program path to generate desire test data according to the reachability conditions of these mutants, the combined necessity condition and the original program. Thus the desire condition of test data to kill multiple mutants that are mutated at the location "p>r" is obtained as:

(p>q) && {(p<r) && (p===r)}

⇨ {(p>q) && (p<r)} && {(p>q) && (p==r)}

⇨ [{(r>p>q)} && {(p==r)>q}] **……..……..……..…… (i)**

The test data inputs of test suite of LTI program, which satisfy above condition is given as [(6, 4, 8) && (10, 8, 10)]. Now Figure 5.2 shows how only these 2 test data inputs can kill all 5 same-location mutants that are mutated at location "p>r".

Note- In all Figures result "**0**" represents that the particular mutant has generated but output of the given Program is correct i.e. Mutant is not killed by corresponding test data and result "**1**" represents that the particular mutant has generated and output of the given Program is wrong i.e. Mutant is killed by corresponding Test data.

Mutant	**(6, 4, 8)**	**(10, 8, 10)**
p<r	1	0
p<=r	1	1
p==r	0	1
p!=r	1	0
p>=r	0	1

Figure 5.2: Killing of Same-location Mutants at Location "p>r"

Figure 5.2 shows that test data input (6, 4, 8) can kill 3 same-location mutants p<r, p<=r, p!=r, and test data input (10, 8, 10) can kill 3 same-location mutants p<=r, p==r, p>=r.

5.1.2 The Second Experiment

Similarly at the location "r>q" in line no.12 of Figure 5.1, we can also generate 5 mutants as shown in figure 5.3 and the desire condition of test data to kill multiple mutants that are mutated at this location is obtained as:

(p<=q) && ((r<q) && (r===q))

⇨ {(p<=q) && (r<q)} && {(p<=q) && (r==q)}

⇨ [{(p==q)>r} || {(p,r)<q}] && [{p<(q==r)} || {(p==q==r)}] **..... (ii)**

In this case the test data inputs of test suite of LTI program, which satisfy above condition are given as [(10, 10, 8) || (4, 8, 6) || (6, 8, 4) && (8, 10, 10) || (10, 10, 10)]. Thus we can take any one test data input from [(10, 10, 8), (4, 8, 6), (6, 8, 4)] and any

one test data input from [(8, 10, 10), (10, 10, 10)]. By doing this, 6 different combinations of 2 test data inputs are obtained as:

- [(10, 10, 8) && (8, 10, 10)]
- [(10, 10, 8) && (10, 10, 10)]
- [(4, 8, 6) && (8, 10, 10)]
- [(4, 8, 6) && (10, 10, 10)]
- [(6, 8, 4) && (8, 10, 10)]
- [(6, 8, 4) && (10, 10, 10)]

Now by using any combination of 2 test data inputs we can kill all 5 same-location mutants that are mutated at location "r>q". Figure 5.3 shows how 2 test data inputs [(10, 10, 8), (10, 10, 10)] and Figure 5.4 shows how 2 test data inputs [(4, 8, 6), (8, 10, 10)] can kill all 5 same-location mutants.

Mutant	**(10, 10, 8),**	**(10, 10, 10)**
r<q	1	0
r<=q	1	1
r==q	0	1
r!=q	1	0
r>= q	0	1

Figure 5.3: Killing of Mutants at "r>q" by (10, 10, 8) and (10, 10, 10)

Mutant	**(4, 8, 6)**	**(8, 10, 10),**
r<q	1	0
r<=q	1	1
r==q	0	1
r!=q	1	0
r>= q	0	1

Figure 5.4: Killing of Mutants at "r>q" by (4, 8, 6) and (8, 10, 10)

5.1.3 The Third Experiment

Now let us take the other example of Trityp program of Figure 4.1. On the mutated location “a==b” of line 5 in Figure 4.1, we can produce 5 mutants (a<b, a<=b, a>b, a>=b, and a! =b) by applying only ROR. As Trityp program has no branch predicate that can lead the test data to reach the inner most block containing the mutated location “a==b”, so in this case there will be no reachability conditions. After combining and using the results of Figure 4.3 we will get the one necessity condition of these 5 same-location mutants as (a<b && a>b) and there are 22 test data inputs of test suite of Trityp Program, which satisfy the given condition. Now Figure 5.5 shows how 2 test data inputs [(5, 4, 2) && (2, 4, 5)] out of 22 can kill all 5 same-location mutants that are mutated at location “p>r”.

Mutant	**(5,4,2)**	**(2,4,5)**
a>b	1	0
a>=b	1	0
a<b	0	1
a<=b	0	1
a!=b	1	1

Figure 5.5: Killing of Same-location Mutants at location “a==b” in Trityp Program

Result: From Figure 5.2, 5.3, 5.4 and Figure 5.5 it is clear that by using our approach we can generate smaller test suite of very low cost that can achieve the same mutation testing score for multiple mutants that are mutated at the same-location.

5.2 Test Data Generation for Killing Different-location Mutants

We will perform some experiments and generate test data for killing different-location mutants.

5.2.1 The First Experiment

Suppose we want to kill the some multiple mutants that are mutated at the different locations in the LTI program of Figure 5.1. For this let us consider the mutated locations “p>q”, “p>r” and “r>q” in line no. 1, 2 and 12 of Figure 5.1 where we can generate total 15 mutants, 5 mutants by each mutated location by applying only ROR,

as shown in Figure 5.6. As LTI program has no branch predicate that can lead the test data to reach the mutated location "p>q", so at this location to kill all 5 mutants there will be no reachability conditions. After combining and using the results of Figure 4.3 for necessity condition, the desire condition of test data to kill multiple mutants that are mutated at the location "p>q" is given as:

(p<q && p==q) **……..…….... (iii)**

The desire condition of test data to kill multiple mutants that are mutated at the location "p>r" is obtained from equation no. (i) and it is given as:

[{(r>p>q)} && {(p==r)>q}] **…… (i)**

The desire condition of test data to kill multiple mutants that are mutated at the location "r>q" is obtained from equation no. (ii) and it is given as:

[{(p==q)>r} || {(p,r)<q}] && [{p<(q==r)} || {(p==q==r)}] **..... (ii)**

Mutant	**(6, 4, 8),**	**(10, 8, 10),**	**(4, 8, 6)**	**(10, 10, 10)**
P<q	0	0	1	0
P<=q	0	0	1	1
P==q	0	0	0	1
P!=q	0	0	1	0
p>=q	0	0	0	1
P<r	1	0	0	0
P<=r	1	1	0	0
P==r	0	1	0	0
P!=r	1	0	0	0
p>=r	0	1	0	0
R<q	0	0	1	0
r<=q	0	0	1	1
r==q	0	0	0	1
r!=q	0	0	1	0
r>=q	0	0	0	1

Figure 5.6: Killing of Different-location Mutants in LTI Program

Thus to kill all the 15 mutants that are mutated at the different locations in the Largest in three integers Program by our approach, we will combine the equations (i), (ii) and (iii).

By doing this, 3 different combinations of 4 test data inputs are obtained as:

- (6, 4, 8), (10, 8, 10), (4, 8, 6) and (10, 10, 10)
- (6, 4, 8), (10, 8, 10), (10, 10, 8) and (8, 10, 10)
- (6, 4, 8), (10, 8, 10), (6, 8, 4) and (10, 10, 10)

Now by using any combination of 4 test data inputs we can kill all 15 different-location mutants that are mutated at locations "p>q", "p>r" and "r>q". Figure 6.6 shows how 4 test data inputs [(6, 4, 8), (10, 8, 10), (4, 8, 6), (10, 10, 10)] can kill all 15 different-location mutants.

5.2.2 The Second Experiment

Suppose we want to kill the some multiple mutants that are mutated at the different locations in the Trityp program of Figure 4.1. For this let us consider the mutated locations "a==b", "a==c" and "b==c" in line no. 5, 7 and 9 of Figure 4.1, where we can generate total 15 mutants, 5 mutants by each mutated location by applying only ROR, as shown in Figure 5.7. As Trityp program has no branch predicate that can lead the test data to reach the inner most block containing the mutated locations "a==b", "a==c" and "b==c" so in this case there will be no reachability conditions also. By using the results of Figure 5.2 we will get the necessity conditions of these 15 different-location mutants as:

1. a<b
2. a>b
3. a<c
4. a>c
5. b<c
6. b>c

After combining the necessity conditions of these 15 different location mutants in one, we will get (a<b<c && a>b>c). By observation we will get only 8 test data out of 25 test data of test suite of Trityp Program, which satisfy this condition and these 8 test data inputs are:

1. (5,4,2) is taken from T5
2. (2,4,5) is taken from T5

3. (4,3,0) is taken from T6
4. (5,2,-4) is taken from T7
5. (3,2,1) is taken from T9
6. (1,2,3) is taken from T9
7. (5,3,1) is taken from T11
8. (1,3,5) is taken from T11

Mutant	**(5,4,2)**	**(2,4,5)**	**(4,3,0)**	**(5,2, -4)**	**(3,2,1)**	**(1,2,3)**	**(5,3,1)**	**(1,3,5)**
a>b	1	0	1	1	1	0	1	0
a>=b	1	0	1	1	1	0	1	0
a<b	0	1	0	0	0	0	0	0
a<=b	0	1	0	0	0	0	0	0
a!=b	1	1	1	1	1	0	1	0
a>c	1	0	1	0	1	0	1	0
a>=c	1	0	1	0	1	0	1	0
a<c	0	1	0	0	0	1	0	1
a<=c	0	1	0	0	0	1	0	1
a!=c	1	1	1	0	1	1	1	1
b>c	1	0	0	0	0	0	0	0
b>=c	0	1	0	0	0	0	0	0
b<c	0	1	0	0	0	1	0	1
b<=c	0	1	0	0	0	1	0	1
b!=c	1	1	0	0	0	1	0	1

Figure 5.7: Killing of Different-location Mutants in Trityp program

Figure 5.7 shows that only 8 test data inputs out of 25 test data inputs, can kill all the 15 multiple mutants that are mutated at the different locations.

Result: From Figure 5.6 and Figure 5.7 it is clear that by using our approach we can generate smaller test suite of very low cost that can achieve the same mutation testing score for multiple mutants that are mutated at the different locations.

CHAPTER 6
CONCLUSIONS & FUTURE WORK

6.1 Conclusions

In practice, if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects the fault. Mutation Testing should be used in conjunction with traditional testing techniques, not instead of them.

This thesis presents a new approach for killing multiple mutants that are mutated at same location and different locations.

The contributions of this approach include:

First, it combines many same-location and different-location mutants necessity conditions into one. Thus it can generate test data, each of which can kill more than one mutant. Second, it uses a much simpler format for reachability conditions, which are easier to acquire. Third, it can reuse and combine the reachability and necessity conditions together to reduce the size of test suite. Thus it can decrease the total number of test data required to kill all the multiple mutants that are mutated at same location and different locations. Fourth, it can reduce the cost of software testing. Thus by using our proposed approach, we can kill easily multiple mutants that are mutated at same location and different locations.

6.2 Future Work

Reduction of test data for killing multiple locations is a quite challenging research topic. In future, following work can be done:

- Our proposed approach is limited to the only mutant locations that have Binary Relational Operator and Binary Arithmetic Operator but in future we can modify our approach that can kill all the mutants generated by all types of operators.

- We can find out the heuristics for Equivalent Mutants.
- We can do work on automated test data generation
- We can do work for reducing time and cost more in case of mutation testing.

REFERENCES

[1] B. Beizer, " Software Testing Techniques" , Van Nostrand Reinhold, 2nd edition, 1990.

[2] R. S. Pressman " Software Engineering: A Practitioner's Approach" , 3rd Edition, McGraw Hill, New York (1992), p. 559.

[3] Standard for Software Test Documentation (IEEE829), "http://ww.ieee.org, accessed on January 5, 2008.

[4] Holland J.H. "Adaptation in natural and artificial system, Ann Arbor", the University of Michigan Press, 1975.

[5] Goldberg D. "Genetic Algorithms" , Addison Wesley, 1988.

[6] Emmeche C. " Garden in the Machine", The Emerging Science of Artificial Life, Princeton University Press, 1994, pp. 114 ss.

[7] Rakesh Shukla, Paul Strooper, and David Carrington, "A Framework for Statistical Testing of Software Components", International Journal of Software Engineering and Knowledge Engineering, 17(3):379-405, June 2007.

[8] Rakesh Shukla, David Carrington and Paul Strooper, "A Passive Test Oracle Using a Component's API", Proceedings of the 12th Asia-Pacific Software Engineering.

[9] "http://serl.cs.colorado.edu/~rutherfo/pubs/RutherfordCarzanigaWolfFSE06.pdf" accessed on February 6, 2008

[10] Johannes Mayer, "Efficient and Effective Random Testing based on Partitioning and Neighborhood", Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering (SEKE 2006), Knowledge Systems Institute Graduate School, Skokie, USA, 2006.

[11] Maryam Umar, "An Evaluation of Mutation Operators for Equivalent Mutants", MS Project, Kings College, London, Supervisor: Mark Harman.

[12] Macario Polo, Sergio Tendero and Mario Piattini, “Integrating techniques and tools for testing automation”, Journal of Software Testing, Verification and Reliability, 17(1):3-39, January 2007.

[13] K. N. King and A. J. Offutt, “A Fortran Language System for Mutation-based Software Testing,” Software-Practice and Experience, 21(7): 685-718, Jul. 1991.

[14]Y. S.Ma, Y. R. Kwon, and A. J. Offutt, “Interclass Mutation Operators for Java,” In Proc. 13th Int’l Symposium on Software Reliability Engineering, pp. 352- 363, Nov. 2002.

[15] Yu-Seung Ma, Yong-Rae Kwon and Jeff Offutt, “Inter-Class Mutation operators for java”, Proceedings of the 13th International Symposium on Software Reliability

[16] A. J. Offutt, Ronald H. Untch, “Mutation 2000:Uninting the Orthogonal,” Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, pages 45–55, San Jose, CA, October 2000.

PAPER PUBLISHED

Updesh Kumar Jaiwal and Ajay Kumar, "Killing Same and Different Location Multiple Mutants", Paper is communicated with National conference on emerging trends in IT, department of computer science and IT, (IEET), Solan, H.P. on July 8th ,2008.