Representing and Generating Mutant Programs in Support of Measuring Test Suite Effectiveness

Ronald Finkbine
rfinkbin@ius.edu
Computer Science, Indiana University Southeast
New Albany, IN 47250 USA

Abstract

Mutation testing is a technique of testing software, a form of white-box testing where the program is dealt with at statement level. It involves the production and execution of a modified version of a "correct" program, a mutant version, against its associated test suite. Each execution of a mutant can generate one of three logical results; (1) one of the test cases fails showing that the test suite is sufficient to detect the mutant program, (2) all test cases pass and this indicates a weakness in the test suite which needs to be repaired by the addition of a new test case, or (3) all test cases pass and a meaningless mutation program was generated. This paper discusses the database definition and support for representing and generating mutant programs.

Keywords: Test suite effectiveness, mutant program, Java

1. INTRODUCTION

In the area of software development, testing is performed to show that a piece of software exhibits expected behavior. There are many methodologies used in testing software [Jorgensen 2002] including black box, white box, integration testing, system testing and user acceptance. The earliest testing done on a program is the unit test, performed by the programmer during coding. This is done by executing their program against the test suite (a collection of test cases). This unit testing of a piece of software shows that a program passes this test suite as designed by the programmer but this does not necessarily assure the quality of the test suite.

As a non-software, real-world example, a generous faculty member can have an extremely simple examination question in an effort to ensure the students give the correct answer. Asking a graduate student in mathematics "What is two plus two?" is a question of the proper form, and it is an answerable question. However, it is an inappropriate question to test a graduate student's knowledge of high-level mathematics. In order to detect weakness and improve the quality of software we must test the software, but also test, measure and improve the quality of test suites the software is to be tested against.

In discussing test effectiveness, we use the term "strength" of a test suite. The test suite is a set of test cases, and the suite needs to have the ability to exhibit when the source program does not work correctly. Passing all its test cases indicates the program works as expected but failure of a test case indicates the program is not working as expected. The challenge is to have a good test suite, the passage of which indicates the program is working correctly. Having a weak test suite doesn't show the program works correctly and since programmers design the test suite, how do you measure the strength of the test suite?

The mutation testing of a program attempts to measure and allow improvement

of the test suite. A program that has successfully passed it test suite at the unit level is a candidate for mutation testing. This type of testing uses a source program as input, introducing modifications to generate mutant versions of the program. Mutation analysis induces faults into software by creating many versions of the software, each containing one fault [Offutt 2001]. Each mutant is compiled and executed against the test suite and this can result in one of three results:

- One of the test cases fails, indicating the test suite is of sufficient strength to detect the mutant program
- All test cases pass, indicating a weak test suite
- All test cases pass, indicating a meaningless mutant has been generated

Please note that the last two choices detailed above are similar and it would be necessary for a programmer to interpret the results of a mutant's execution to determine if a meaningless equivalent mutation has been created.

As an example a single test case consists of an input file that contains inputs to specifically test certain characteristics of the subject program. The execution of the test case will produce an output file, absent some form of run-time error that interrupts execution. A test case should test one certain behavior of the subject program. For example, five test cases for a linked list program would be [Finkbine 2002]:

1. Insert into an empty list?
2. Insert into the front of a non-empty list?
3. Insert into the end of a non-empty list?
4. Insert into the middle of a non-empty list?
5. Insert a duplicate item into the list?

Students often do not think of a reasonable test suite, they often get the program to work on one test case and stop at that. So this system would contain a table that contains a record of each test case for the

subject program. In this example, we see that five test cases are present; however, it would be very reasonable to have many more test cases, but hopefully not redundant ones.

2. DATABASE DEFINITION

This project consists of components for support of mutation testing:

- Database for maintaining secure source code program
- Generator for producing mutants
- Compiler for generating class files for execution
- Execution manager for running class files and capturing outputs
- Database for maintaining test suite for source program
- Test suite manager for addition/deletion of test cases
- Results manager for results of test case runs

Representing mutant programs, or having the ability to generate them at will, requires substantial database support.

Figure 1 shows the standard hello-world program in Java. Figure 2 shows the token stream for the first line of this sample program. The fieldnames for these columns

```
import java.io.*;
class test {
public static void main
  (String[] args)
  {
  System.out.println
    ("hello world");
  } //main
} //class test
```
               Figure 1

are record name, token number, line number, column number, starting column

```
…
Token 1 1 1 import
Token 2 1 8 java
Token 3 1 12 .
Token 4 1 13 io
Token 5 1 15 .
Token 6 1 16 *
Token 7 1 17 ;
…
```
               Figure 2

number and field spelling.

This representation will be contained in the **Token_Tbl** as shown in Figure 3. Note that all database tables are in the graphic of a disk (round top) and the programs in the system that move and produce data are in rectangles.

The **Token_Tbl** graphic shows the fields required to represent the original Java program. A stream of tokens (for the entire program) as partially depicted in Figure 2 is expected to be a working, correct program, though this is dependent upon the programmer loading the subject program. This table will not be modified by this system. The correctness of the original program is assumed, the intent of this project is to support the manipulation and generation of mutants. This system does require a compiler be present since the mutant programs generated will be compiled and executed.

The **MutantOp_Tbl** describes the record associated with mutable operators. The early phase of this project allows mutation of the mathematical operators. Logical operator, variable manipulation and further mutations will follow in a further phase. Records in this table are used to modify some of the tokens extracted from the **Token_Tbl** in production of mutants.

The **TestCase_Tbl** shows the data required for the tracking of individual test cases that constitute the test suite. A test suite would consist of all test cases necessary to show that a program works correctly.

The **MutantExecution_Tbl** shows the table to track execution of the each mutant program generated against each record in the test case table. It is necessary to have a test run of the correct program against each test case. This would allow for accurate determination of the run-time of each mutant. It is possible that a mutant program will cause an endless-loop and the run-time for each test case would allow the **Program_Execution_Controller** (controls mutants at run-time) to determine when a mutant exceeds twice the expected

run-time and force a kill signal to be sent to the mutant.

3. Detailed Example

As an example, assume the simple program of Figure 4. This program adds 1 to 2 (but does not print it). This program file would be read by **Program_TokenLoader** and loaded into **Token_Tbl** as a ordered set of tokens as described in Figures 1 and 2.

```
import java.io.*;
class simple {
public static void main
  (String[] args)
  {
  int x;
  x = 1 + 2;
  } //main
} //class simple
```
<center>Figure 4</center>

The **Program_MutantGenerator** would read this program from **Token_Tbl** and would modify the mathematical operator plus ("+") to a minus ("-"). This mutant sequence of tokens would be recorded in the **MutantTokens_Tbl** and would be output, compiled and executed by the **Program_Execution_Controller** program. Note that changing the plus to a minus in Figure 4 will produce a syntactically correct program. The compiler will accept this program, it will be up to the test cases (at least one of them) to fail, indicating the mutant is an incorrect program and the test suite is strong enough. Also note that to perform mutation testing on this simple at least requires that we read the two operands (1 and 2) in on the command line so we could be a test suite that could execute the program with command line parameters, one per each test case.

4. Future work

There are a number of areas in which this project needs to develop.

First is to strengthen the database supporting this project. Moving from MS-Access to MySql, implementing database

integrity contraints and system administrator scripts are intended.

Second is to expand the operators used in generating mutants. Other researchers have classified the operators into eight classes; boolean contants, boolean operators, relational operators, increment/decrement operators, arithmetic operators, binary bit operators, arithmetic assignment operators and binary bit assignment operators [Agrawal 1989]. Currently, this project only supports mutation utilizing the arithmetic operators.

Third is the Program Execution Controller-to more fully control the mutants at run-time. Current control is not satisfactory though it can never be perfect since it is not possible to determine with 100 percent accuracy that a program will never halt.

4. BIBLIOGRAPHY

Agrawal, H. and R. DeMillo, R. Hathaway, W. M. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, and E. Spafford, "Design of Mutant Operators for the C Programming Language," SERC#: SERC-TR-41-P, March 20, 1989.

Jorgensen Paul C., Software Testing: A Craftsman's Approach, Second Edition, CRC Press, ISBN 0849308097, 2002.

Finkbine, R. B. and N. A. Kraft, "Introducing the Test Harness: Automating the Test Suite," in Proceedings of the Information Systems Education Conference (ISECON 2002), San Antonio, Texas, USA, November, 2002.

Hierons, R. M., M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," in Software Testing, Verification and Reliability, 9(4):233-262, 1999.

Offutt, A. J. and R. H. Untch, "Mutation 2000: Uniting the Orthogonol", appeared in Mutation Testing for the New Century, Kluwer Academic Publishers, ISBN 0-7923-7323-5,   2001
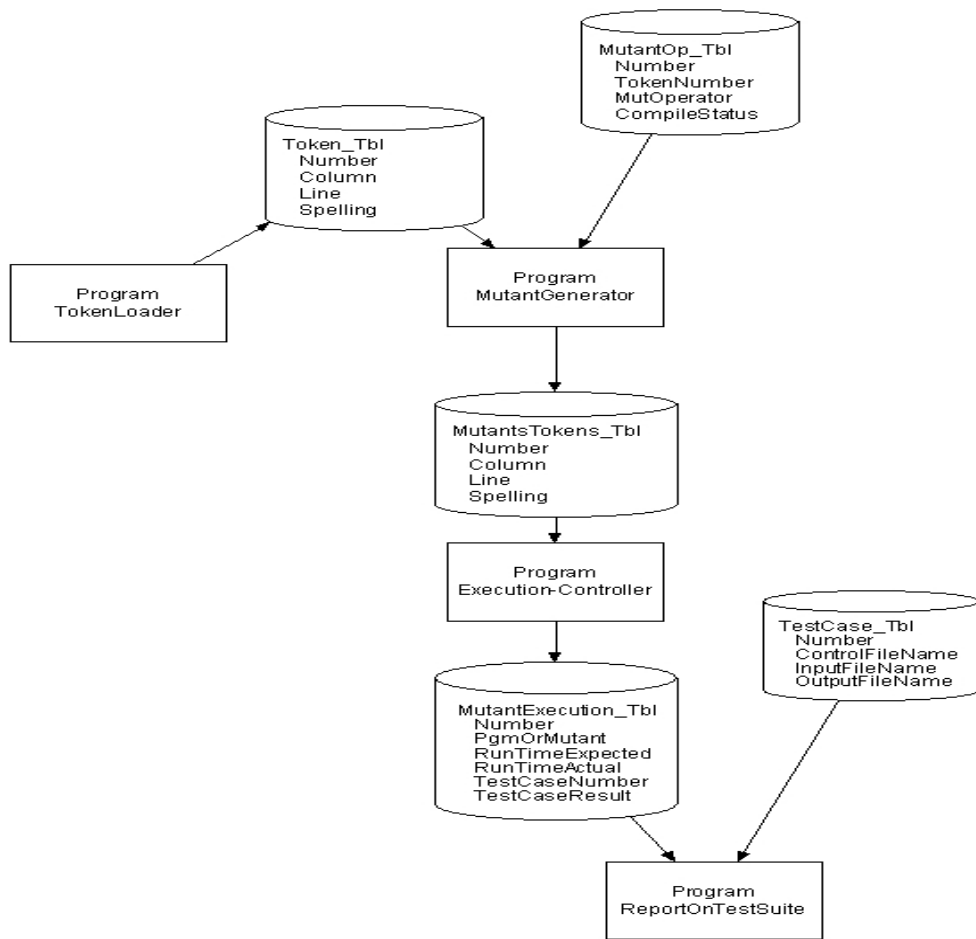
Figure 3