The University of Saskatchewan
Department of Computer Science

Technical Report #2012-01

UNIVERSITY OF SASKATCHEWAN

Evaluating Test Quality

Minhaz Fahim Zibran
Department of Computer Science
University of Saskatchewan, Canada
Email: minhaz.zibran@usask.ca

Abstract

Software testing has been an integral part of software development process in order to ensure software quality. We conduct testing in the software verification process to identify and remove faults and bugs in the software under development. For software quality assurance (QA) these tests have to be good and effective. Therefore, a notion to evaluate the tests themselves is necessary. This paper discusses the evaluation of test quality in the process of software verification. Here I discuss different structural and functional test coverage criteria, as well as fault based techniques, which may be applied to measure the completeness and effectiveness of the test cases.

Keywords:

Software test, software quality, test quality, test evaluation, test coverage, test adequacy, functional test, structural test, mutation, fault injection

1 Introduction

To improve software quality, different types and approaches of software testing are applied. The major goal of testing is to find faults in the software under development. Therefore, the tests themselves have to be powerful enough to detect faults in the software. So, the evaluation of the test quality demands much importance. We typically test software with two distinct goals. Assessing the degree to which a software system actually fulfills its requirements, in the sense of meeting the user's real needs, is *validation* [7]. And software *verification* is checking the consistency of an implementation with a specification [7].

This paper focuses on the evaluation of tests in the process of software verification. Common approaches for software verification include black-box testing, glass-box testing, and broken-box testing. Random functional testing falls into black-box testing and all structural tests are glass-box tests. Partitioned functional testing and mutation testing may be categorized as broken-box testing. In this paper I discuss all of these tests keeping focus on adequacy criteria. Section 2 includes a brief introduction to adequacy criteria, which provides a way to evaluate how thorough our tests are. Different adequacy criteria concerned with structural testing is discussed in section 3. Section 4 includes discussion on

functional test criteria. Section 5 discusses fault based testing, and in section 6, I discuss test case prioritization technique to optimize test suites for regression testing. Finally I conclude the paper making some concluding remarks in section 7.

2 Test Adequacy Criteria

M. Ohba of IBM, Japan in his paper [8] described software quality as the Cartesian product of test accuracy and test adequacy, as shown below.

$$Software\ Quality = Test\ Accuracy \times Test\ Adequacy$$

Test accuracy refers to the correctness of the tests. To ensure test accuracy, we must not do mistakes while executing the test cases, and analyzing the results. *Test adequacy or test coverage* criterion is a predicate that is true (satisfied) or false (not satisfied) of a program, test suite pair. Usually a test adequacy criterion is expressed in the form of a rule for deriving a set of test obligations from another artifact, such as a program or specification. The adequacy criterion is then satisfied if every test obligation is satisfied by at least one test case in the suite. We often treat the adequacy criterion as a heuristic for test case selection or generation [7]. According to Shmuel Ur, IBM Research Lab in Haifa, "Coverage is any metric of completeness with respect to a test selection criterion" [11].

3 Structural Test Coverage

The general term for testing based on program structure is structural testing, although the term white-box testing or glass-box testing is sometimes used. Test specifications drawn from program source code require coverage of particular elements in the source code or some model derived from it. In the following subsections I discuss different structural test coverage criteria.

3.1 Statement Coverage

It can be reasonably assumed that faults are arbitrarily distributed over the source code. So, if we can test every statement of the program, we are likely to find the existing faults. Statement coverage criterion requires each statement in the program is executed at least once by the test cases on a test suite.

Statement coverage is measured by the following expression [7].

$$\text{statement coverage} = \frac{\text{number of statements executed}}{\text{total number of statements in program}}$$

Statement coverage criterion has a number of limitations. It does not report whether loops reached their termination condition. Moreover, since do-while loops always execute at least once, statement coverage considers them the same rank as non-branching statements.

Statement coverage is completely insensitive to the logical operators (|| and &&), and it cannot distinguish consecutive switch labels [12]. Another limitation of statement coverage is that a test suite can achieve complete statement coverage without executing all the possible branches in a program. For example, let us consider the code snippet in Listing 1. A test case requiring the if condition at line 2 to be satisfied will also satisfy statement coverage criterion, but it would not detect the fault in the program that line 4 would generate a null pointer exception whenever the if-condition at line 2 is evaluated false.

Listing 1: Null Pointer Exception

```
1 MyClass obj = null;
2 if(condition)
3     obj = new MyClass();
4 obj.aMethod();
```

Listing 2: Missing Statement

```
1 String msg = "default";
2 if(condition)
3     msg = "true";
4 else
5     msg = "false";
6 return msg;
```

Similarly, say, the lines 4 and 5 are mistakenly removed from the code snippet in Listing 2. Any test case requiring the if-condition at line 2 to be true will satisfy statement coverage without detecting that some code is missing.

3.2 Branch Coverage

The limitations of statement coverage imply that each branch (both true and false) of a control (boolean) statement has to be exercised. Branch coverage (also known as decision coverage) criterion forces this requirement, that is, each branch of the program to be executed at least once by at least one test case. Branch coverage metric is measured using the following expression [7].

$$\text{branch coverage} = \frac{\text{number of branches exercised}}{\text{total number of branches in program}}$$

The entire boolean expression is considered one true-or-false predicate regardless of whether it contains logical-and or logical-or operators. Additionally, this metric includes coverage of switch-statement cases, exception handlers, and interrupt handlers [12].

Now, if we consider the code snippets of Listing 1 and 2, test suites satisfying branch coverage criterion would detect the null pointer exception problem and missing code problem as stated in section 3.1.

Listing 3: Short Circuit Operator

```
if(condition1 &&
    (condition2 ||
     aMethod()))
    statement1;
else
    statement2;
```

Listing 4: Basic Condition Coverage without Branch Coverage

```
1 boolean f(boolean b) {
2     return false;
3 }
4 if(f(a && b)) ...
5 if((a && b)?false:false)..
```

The limitation of branch coverage is that it ignores branches within boolean expressions which occur due to short-circuit operators. Consider the source code shown in Listing 3. Test suites may satisfy branch coverage criterion without a call to the method "aMethod()". Further, in case of loops, like statement coverage, branch coverage cannot report how many times the loop iterated, or whether it reached the terminating condition.

3.3 Basic Condition Coverage

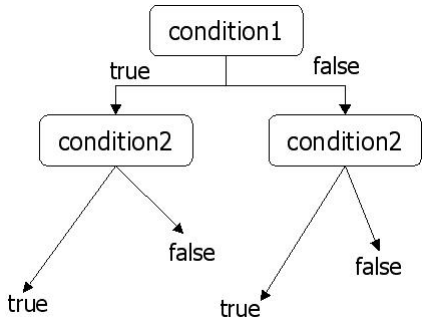
Basic condition coverage criterion overcomes the limitation of branch coverage with the short circuit operators. Basic condition coverage requires that all different combinations of the outcomes of each basic condition in the control expression have to be exercised. For example, the control expression

$$(\text{condition1} \mid\mid \text{condition2})$$

with two basic conditions would produce four different combinations of their outcome, as shown in Table 1. Therefore, we need four test cases assuming that each test case exercises a single combination.

Test Cases	condition1	condition2
c_1	false	false
c_2	false	true
c_3	true	false
c_4	true	true

Table 1: Combinations of Outcomes of Two Basic Conditions



Basic condition coverage is measured using the following expression [7].

$$\text{basic condition coverage} = \frac{\text{number of basic conditions exercised}}{2 \times \text{total number of basic conditions in program}}$$

Test Cases	a	b	c	d	e
c_1	true	true	true	true	true
c_2	true	true	true	true	false
c_3	true	true	true	false	–
c_4	true	true	false	–	–
c_5	true	false	–	–	–
c_6	false	–	–	–	–

Table 2: Test Cases for Expression (a && b && c && d && e) [7]

A control expression having N basic conditions produces 2^N distinct combinations of outcomes, and so 2^N test cases are needed to satisfy basic condition coverage. This exponential growth of required test cases is a major limitation of basic condition coverage. Besides, exercise of loops still remains a concern. Moreover, basic condition coverage can be satisfied without satisfying branch coverage [12]. For example, the if-conditions in lines 4 and 5 of Listing 4 will always be evaluated false.

3.4 Multiple Condition Coverage

Multiple condition coverage (also known as compound condition coverage) tries to control the exponential growth of the number of required test cases imposed by basic condition coverage, by taking into account the order or logical and short circuit operators. In traditional programming languages like C++ and Java, a control expression is evaluated encountering the basic conditions from left to write. In many cases, branching decision is made from the evaluation of a small subset of the basic conditions without examining the conditions to the right side. For instance, multiple condition coverage for the expression

(a && b && c && d && e)

would require only 6 test cases (as opposed to 2^6 test cases required for basic condition coverage) as shown in Table 2.

A limitation of multiple condition coverage is the number of test cases required is dependent on the logical operators and their order. So the number of required test cases can still be exponential in the worst case. For example, having the same number of basic conditions and operators as shown in Table 2, the expression

(((a || b) && c) || d) && e)

requires 13 test cases as presented in Table 3. Moreover, loops still remain a concern in both basic and multiple condition coverage.

Test Cases	a	b	c	d	e
c_1	true	–	true	–	true
c_2	false	true	true	–	true
c_3	true	–	false	true	true
c_4	false	true	false	true	true
c_5	false	false	–	true	true
c_6	true	–	true	–	false
c_7	false	true	true	–	false
c_8	true	–	false	true	false
c_9	false	true	false	true	false
c_{10}	false	false	–	true	false
c_{11}	true	–	false	false	–
c_{12}	false	true	false	false	–
c_{13}	false	false	–	false	–

Table 3: Test Cases for Expression (((a || b) && c) || d) && e) [7]

3.5 Path Coverage

Sometimes, a fault is revealed only through exercise of some sequence of decisions (i.e., a particular path through the program). Path coverage (also called Predicate Coverage) is concerned with this issue, requiring that each path between the entry and exit points of a program has to be exercised. Path coverage is measured using the following expression [7].

$$\text{path coverage} = \frac{\text{number of path exercised}}{\text{total number of path in program}}$$

A program with N control statements has 2^N distinct paths. For instance, path coverage for the code snippet in Listing 5 having 3 if-statements requires the resulting $2^3 = 8$ paths to be exercised. This exponential growth of paths with the increase of the number of control statements is a serious limitation of path coverage criterion. Moreover, the number of paths in a program with loops is unbounded.

3.6 Data Flow Coverage

Data flow coverage refines the path coverage by reducing the number of required path to be exercised. Data flow coverage is based on DU-pairs (Definition-use pairs) of variables.

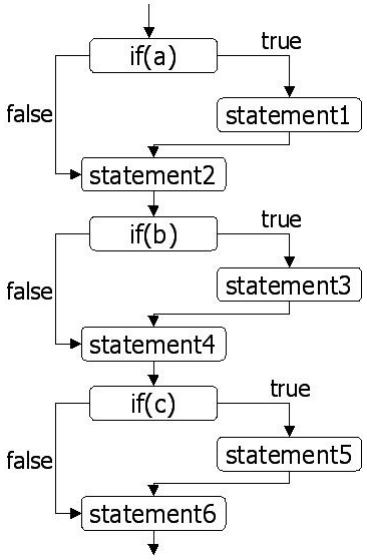
Definition refers to data or variable declaration, creation, or initialization.

Use refers to use of the data in computations or predicates.

Listing 5: Paths in Program

```
if(a)
        statement1;
statement2;
if(b)
        statement3;
statement4;
if(c)
        statement5;
statement6;
```

Total Path $= 2^N = 2^3 = 8$



Consider the variable 'result' in the program shown in Listing 6, where definitions of 'result' found in lines 2, 15, 17, 19, and its use found in lines 6, 20, 21.

All DU-Pair coverage requires that each definition-use pair must be exercised at least once. But this is not enough as there may be multiple path between a single DU-pair. So, *all DU-path* coverage imposes that each path in each DU-pair of each variable must be exercised at least once. Data flow coverage still has its limitation with loops since programs with loops have unbounded number of paths. There are variations of data flow coverage, which may be found in [5].

3.7 Loop Coverage

Adequacy criteria discussed in the earlier sections indicate that loops need special treatment. Loop coverage requires that for each loop,

- the loop body is executed at least once (more than once for do-while loops),
- loop terminating condition is exercised at least once,
- each jump statement (break, continue, etc.) in the loop body is exercised at least once.

3.8 Procedure Entry/Exit Coverage

If unit testing has been effective, then faults that remain to be found in integration testing will be primarily interface faults, and testing effort should focus on interfaces between units rather than their internal details. Sometimes faults are caused in subsequent calls of

Listing 6: A Method to Classify Triangles

```
1 String classify(int sideA, int sideB, int sideC){
2     String result = "Not a triangle";
3     if (!((sideA+sideB) > sideC) &&
4               ((sideB+sideC) > sideA) &&
5               ((sideA+sideC) > sideB))
6                       return result;
7     int match = 0;
8     if (sideA == sideB)
9               match = match + 1;
10    if (sideB == sideC)
11              match = match + 1;
12    if (sideA == sideC)
13              match = match + 1;
14    if (match == 0)
15              result = "Scalene";
16    else if (match == 1)
17              result = "Isosceles";
18    else
19         result = "Equilateral";
20    if (result.Equals("Isosceles")) aMethod();
21         return result;
22 }
```

Tool	Coverage	Major Features	Language
EclEmma	Statement, method	Eclipse plug-in, Ant integration	Java
Coverlipse	Statement, data Flow	Eclipse plug-in	Java
Cobertura	Statement, branch	Ant integration	Java
Jester	Mutation analysis	complimentary to code coverage	Java
GCT	Branch, multiple-condition, loop	for Unix platform	C
CoverMeter	Statement, condition, branch	for Unix and Windows	C++
Pester	Mutation analysis	complimentary to code coverage	Python
Nester	Mutation analysis	complimentary to code coverage	C#

Table 4: Code Coverage Tools

methods. Procedure entry/exit coverage requires that each path between every entry-exit pair in a procedure has to be exercised at least once. Probable sources of faults include

- variables that persist values over multiple calls,
- variables that use/modify global or class variables,
- recursive calls of method.

3.9 Code Coverage Tools

To dates many commercial and freeware tools are available for measuring code coverage. Most of the tools are implemented applying one of the two instrumentation techniques: class instrumentation and source instrumentation. Class instrumentation injects the reporting code directly into compiled .class files while source instrumentation creates an intermediary version of the sources which are then compiled into the final, source-instrumented .class files [6].

In a typical implementation of a code coverage tool the following features are common.

- Integration with build tools like Ant or Maven.
- Reporting of coverage statistics in HTML, PDF, Plain Text, or XML format.
- Source code encoding by highlighting covered and uncovered code with different colors.

Table 4 lists some free code coverage tools along with their prominent features.

4 Functional Test Criteria

Functional test case design is an indispensable base of a good test suite, complemented but never replaced by structural and fault-based testing, because there are classes of faults that only functional testing effectively detects. Omission of a feature, for example, is unlikely to be revealed by techniques that refer only to the code structure.

Complete functional test is usually impossible for nontrivial programs. Even a simple function whose input arguments are two 32-bit integers has 2^{64} legal inputs! Therefore, we produce sample inputs using two commonly used approaches: random testing and partition testing.

In *random testing*, we randomly produce legal sample inputs. Advantages of random testing include,

- avoids accidental bias, that could be fed by the testers,
- it is an inexpensive way to produce a large number of test cases,
- useful when we lack knowledge on the sensitivity of inputs.

But the major shortcoming of random testing is, it cannot guaranty coverage of special cases and boundary values. For instance, consider a method "getMax()", which take a list of integers and returns the maximum value[1]. Here, random testing does not ensure coverage of situations with empty list, negative and positive values in the list, and so on.

Partition testing overcomes this limitation. In partition testing we analyze properties of input, semantically categorize them, and then pick representative samples from each of the categories. For the "getMax()" method mention above, we might classify the input domain based on different properties as shown below.

Size of list: 0, 1, 2, 3, .

Magnitude of values: all negative, all positive, mixture of positive and negative, big positive number, big negative number, ..

Duplicate values: all duplicate, some duplicate, multiple maximum, ..

Ordering of values: ascending, descending, not ordered.

Position of maximum: beginning, end, somewhat in the middle.

Given a fixed budget, the optimum may not lie in only partition testing or only random testing, but in some mix that makes use of available knowledge [7].

[1] this example is taken from [2]

Operator	Description	Constraint
Operand Modifications		
Constant for constant replacement	replace constant C_1 with constant C_2	$C_1 \neq C_2$
Scalar for constant replacement	replace constant C with scalar variable X	$C \neq X$
Array for constant replacement	replace constant C with array reference A[I]	$C \neq A[I]$
Expressions Modifications		
Absolute value insertion	replace e by $abs(e)$	$e < 0$
Arithmetic operator replacement	replace arithmetic operator ϕ with ψ	$e_1 \phi e_2 \neq e_1 \psi e_2$
Logical connector replacement	replace logical connector ϕ with ψ	$e_1 \phi e_2 \neq e_1 \psi e_2$
Relational operator replacement	replace relational operator ϕ with ψ	$e_1 \phi e_2 \neq e_1 \psi e_2$
Statement Modification		
Statement deletion	delete a statement	
Switch case replacement	replace the label of one case with another	
End block shift	move } one statement earlier and later	

Table 5: Examples of Mutation Operators [7]

5 Mutation Analysis

Mutation (also known as fault injection) analysis evaluates the degree of effectiveness/-completeness of test cases, and gives hint to enhance test suites. The basic idea is to create mutants by making small changes in the program under test , and let the test suite identify (kill) the mutant.

A *mutant* is a program that differs from the original program for one syntactic element (e.g., a statement, a condition, a variable, a label) [7]. The mutant which is syntactically correct (not rejected by the compiler), is a *valid mutant*. Typically, mutants are created by applying *mutation operators* to the programs. Mutation operators are syntactic patterns defined relative to particular programming languages [7]. Table 5 lists some examples of mutation operators for Java or C++.

Mutation testing involves three basic steps:

1. Select mutation operators,
2. Generate mutants applying mutation operators,
3. Execute test, to see if test cases can detect (kill) the mutants.

For example, replacement of the constant 1 by 0 at line 7 of the code in Listing 6 would generate a valid mutant. Mutation analysis discussed so far refers to *strong mutation*, where each mutant has exactly one fault, and the mutant is killed based on the result found after test execution completes. Strong mutation causes much compilation and execution overhead, as each mutant needs to be separately compiled and tested.

In *weak mutation* a single mutant owns more than one faults. Such a mutant is called *meta-mutant*. A "meta-mutant" program is divided into segments containing original and

Test case	Faults									
	1	2	3	4	5	6	7	8	9	10
A	x				x					
B	x				x	x	x			
C	x	x	x	x	x	x	x			
D					x					
E								x	x	x

Table 6: Faults Found by Test Cases

mutated source code, with a mechanism to select which segments to execute. Two copies of the meta-mutant are executed in tandem, one with only original program code selected and the other with a set of live mutants selected. Execution is paused after each segment to compare the program state of the two versions. If the state is equivalent, execution resumes with the next segment of original and mutated code. If the state differs, the mutant is marked as dead, and execution of original and mutated code is restarted with a new selection of live mutants. Hence, weak mutation being more complex reduces the overhead of separately compiling and running a large number of strong mutants.

6 Prioritization in Regression Testing

With limited time and money, during regression testing prioritization and clustering of test cases help optimize test suites [10]. For example, consider a program with a test suite of five test cases, A through E, such that the program contains ten faults, 1 through 10, detected by those test cases, as shown in Table 6.

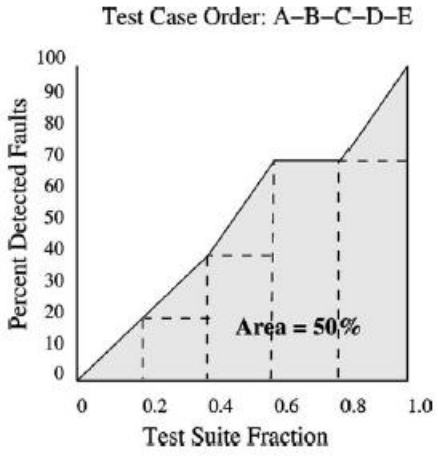


Figure 1

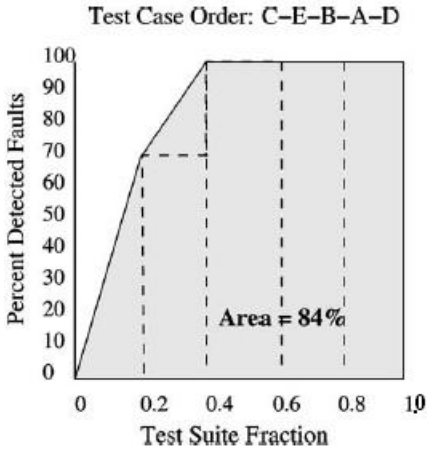


Figure 2

Consider two orders of these test cases, order T_1: A–B–C–D–E, and order T_2: C–E–B–A–D. Figures 1 and Figure 2 show the percentages of faults detected versus the fraction of the test suite used, for these two orders[2]. Moreover, we see that a test suite with test case C and E would suffice to find all the faults.

7 Concluding Remarks

The main goal of software testing (verification) is make the software bug free as much as possible. To achieve this goal the test needs to be good. One test case is better than another if it detects more faults than the other does. But a test suite with more test cases may or may not be better than a test suite with less test cases [7]. Rather the completeness of tests demands importance. Code coverage criteria help to determine the thoroughness of tests. Therefore, choice of coverage criteria and coverage tool is important. Branch/decision Coverage subsumes (includes) statement coverage. Path coverage subsumes branch coverage. Data flow coverage enhances path coverage. So, if we choose data flow coverage, choosing statement coverage or path coverage would be redundant. Moreover, coverage criteria should not be the basis of test designs. Rather we should use it as heuristic to determine what additional tests required. However, test suites satisfying structural Coverage criteria could fail in revealing faults that can be caught with functional criteria [7]. So, both white-box and black-box tests should be applied. Testing should start at the beginning of the development process and continue as we write production code. Then what should we test? Anything that we feel may go wrong should be tested. With the progress of software process as the software gradually gains maturity, mutation testing may be used if deemed feasible to determine the effectiveness of the test cases we have got so far. Prioritization of test cases would be helpful to optimize test suites for regression testing. Finally, we should always keep the motivation to do more testing without ever thinking that testing is complete, because, "program testing can show the presence of bugs, never their absence. (Edsger W. Dijkstra)".

References

[1] Brian Marick. *How to Misuse Code Coverage*. Presented in Testing Computer Softwate' 1999 (http://www.uspdi.org/conference/).

[2] Diane Horton. *Testing Software*. Summer Assignment, Department of Computer Science, University of Toronto. July 1999.

[3] Hong Zhu, Patrick A. V. Hall, and John H. R. May. *Software Unit Test Coverage and Adequacy*. ACM Computing Surveys, Vol. 29, No. 4, December 1997.

[2] this example with relevant table and figures are excerpted from [10]

[4] Jonathan Sillito. *A Brief Introduction to Software Testing*. Lecture Notes, SENG 301, University of Calgary, Fall 2007.

[5] Laurie Williams. *Data Flow Testing*. Lecture Notes, North Carolina State University, Sept 2007.

[6] Lasse Koskela. *Introduction to Code Coverage*. JavaRanch Journal, January 2004.

[7] Mauro Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons Publications, Nov 2007. ISBN: 9780471455936.

[8] M. Ohba. *Software Quality = Test Accuracy × Test Coverage*. Proceedings of the 6th international conference on Software engineering, pp 287-293. IEEE Computer Society Press, 1982.

[9] Peter Liggesmeyer. *A set of software complexity metrics for guiding the software testing process*. Software Quality Journal, Vol. 4, pp. 257-273. 1995.

[10] Sebastian Elbaum, Gregg Rothermel and Satya Kanduri. *Selecting a Cost-Effective Test Case Prioritization Technique*. Software Quality Journal, 12, 185210, 2004.

[11] Shmuel Ur. *Code and Functional Coverage Tutorial*. IBM Research Laboratory in Haifa, May 1999.

[12] Steve Cornett. *Code Coverage Analysis*. Bullseye Testing Technology, 1996-2007. http://www.bullseye.com/coverage.html