

Code Coverage Analysis

This paper gives a complete description of code coverage analysis (test coverage analysis), a software testing technique.

By Steve Cornett. Copyright © [Bullseye Testing Technology](http://www.bullseye.com) 1996-2014. All rights reserved. Redistribution in whole or in part is prohibited without permission.



Do not copy any part of this document without permission.

Contents

- [Introduction](#)
- [Structural Testing and Functional Testing](#)
- [The Premise](#)
- [Basic Metrics](#)
 - [Statement Coverage](#)
 - [Decision Coverage](#)
 - [Condition Coverage](#)
 - [Multiple Condition Coverage](#)
 - [Condition/Decision Coverage](#)
 - [Modified Condition/Decision Coverage](#)
 - [Path Coverage](#)
- [Other Metrics](#)
 - [Function Coverage](#)
 - [Call Coverage](#)
 - [Linear Code Sequence and Jump \(LCSAJ\) Coverage](#)
 - [Data Flow Coverage](#)
 - [Object Code Branch Coverage](#)
 - [Loop Coverage](#)
 - [Race Coverage](#)
 - [Relational Operator Coverage](#)
 - [Weak Mutation Coverage](#)
 - [Table Coverage](#)
- [Comparing Metrics](#)
- [Coverage Goal for Release](#)
- [Intermediate Coverage Goals](#)
- [Summary](#)
- [References](#)

Introduction

Code coverage analysis is the process of:

- Finding areas of a program not exercised by a set of test cases,
- Creating additional test cases to increase coverage, and
- Determining a quantitative measure of code coverage, which is an indirect measure of quality.

An optional aspect of code coverage analysis is:

- Identifying redundant test cases that do not increase coverage.

A *code coverage analyzer* automates this process.

You use coverage analysis to assure quality of your set of tests, not the quality of the actual product. You do not generally use a coverage analyzer when running your set of tests through your release candidate. Coverage analysis requires access to test program source code and often requires recompiling it with a special command.

This paper discusses the details you should consider when planning to add coverage analysis to your test plan. Coverage analysis has certain strengths and weaknesses. You must choose from a range of measurement methods. You should establish a minimum percentage of coverage, to determine when to stop analyzing coverage. Coverage analysis is one of many testing techniques; you should not rely on it alone.

Code coverage analysis is sometimes called *test coverage analysis*. The two terms are synonymous. The academic world more often uses the term "test coverage" while practitioners more often use "code coverage". Likewise, a coverage analyzer is sometimes called a *coverage monitor*. I prefer the practitioner terms.

Structural Testing and Functional Testing

Code coverage analysis is a structural testing technique (AKA glass box testing and white box testing). Structural testing compares test program behavior against the apparent intention of the source code. This contrasts with functional testing (AKA black-box testing), which compares test program behavior against a requirements specification. Structural testing examines how the program works, taking into account possible pitfalls in the structure and logic. Functional testing examines what the program accomplishes, without regard to how it works internally.

Structural testing is also called path testing since you choose test cases that cause paths to be taken through the structure of the program. Do not confuse path testing with the [path coverage](#) metric, explained later.

At first glance, structural testing seems unsafe. Structural testing cannot find errors of omission. However, requirements specifications sometimes do not exist, and are rarely complete. This is especially true near the end of the product development time line when the requirements specification is updated less frequently and the product itself begins to take over the role of the specification. The difference between functional and structural testing blurs near release time.

The Premise

The basic assumptions behind coverage analysis tell us about the strengths and limitations of this testing technique. Some fundamental assumptions are listed below.

- Bugs relate to control flow and you can expose Bugs by varying the control flow [[Beizer1990](#) p.60]. For example, a programmer wrote "`if (c)`" rather than "`if (!c)`".
- You can look for failures without knowing what failures might occur and all tests are *reliable*, in that successful test runs imply program correctness [[Morell1990](#)]. The tester understands what a correct version of the program would do and can identify differences from the correct behavior.
- Other assumptions include achievable specifications, no errors of omission, and no unreachable code.

Clearly, these assumptions do not always hold. Coverage analysis exposes some plausible bugs but does not come close to exposing all classes of bugs. Coverage analysis provides more benefit when applied to an application that makes a lot of decisions rather than data-centric applications, such as a database application.

Basic Metrics

A large variety of coverage metrics exist. This section contains a summary of some fundamental metrics and their strengths, weaknesses and issues.

The U.S. Department of Transportation Federal Aviation Administration (FAA) has formal requirements for structural coverage in the certification of safety-critical airborne systems [[DO-178C](#)]. Few other organizations have such requirements, so the FAA is influential in the definitions of these metrics.

Statement Coverage

This metric reports whether each executable statement is encountered. Declarative statements that generate executable code are considered executable statements. Control-flow statements, such as `if`, `for`, and `switch` are covered if the expression controlling the flow is covered as well as all the contained statements. Implicit statements, such as an omitted `return`, are not subject to statement coverage.

Also known as: line coverage, segment coverage [[Ntafos1988](#)], C1 [[Beizer1990](#) p.75] and basic block coverage. Basic block coverage is the same as statement coverage except the unit of code measured is each sequence of non-branching statements.

I highly discourage using the non-descriptive name C1. People sometimes incorrectly use the name C1 to identify [decision coverage](#). Therefore this term has become ambiguous.

The chief advantage of this metric is that it can be applied directly to object code and does not require processing source code. Performance profilers commonly implement this metric.

The chief disadvantage of statement coverage is that it is insensitive to some control structures. For example, consider the following C/C++ code fragment:

```
int* p = NULL;
if (condition)
    p = &variable;
*p = 123;
```

Without a test case that causes `condition` to evaluate false, statement coverage rates this code fully covered. In fact, if `condition` ever evaluates false, this code fails. This is the most serious shortcoming of statement coverage. If-statements are very common.

Statement coverage does not report whether loops reach their termination condition - only whether the loop body was executed. With C, C++, and Java, this limitation affects loops that contain `break` statements.

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 `&&`).

Statement coverage cannot distinguish consecutive `switch` labels.

Test cases generally correlate more to decisions than to statements. You probably would not have 10 separate test cases for a sequence of 10 non-branching statements; you would have only one test case. For example, consider an if-else statement containing one statement in the then-clause and 99 statements in the else-clause. After exercising one of the two possible paths, statement coverage gives extreme results: either 1% or 99% coverage. Basic block coverage eliminates this problem.

One argument in favor of statement coverage over other metrics is that bugs are evenly distributed through code; therefore the percentage of executable statements covered reflects the percentage of faults discovered. However, one of our fundamental assumptions is that faults are related to control flow, not computations. Additionally, we could reasonably expect that programmers strive for a relatively constant ratio of branches to statements.

In summary, this metric is affected more by computational statements than by decisions.

Decision Coverage

This metric reports whether Boolean expressions tested in control structures (such as the `if`-statement and `while`-statement) evaluated to both true and false. 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 all points of entry and exit. Constant expressions controlling the flow are ignored.

Also known as: branch coverage, all-edges coverage [[Roper1994](#) p.58], C2 [[Beizer1990](#) p.75], decision-decision-path testing [[Roper1994](#) p.39]. I discourage using the non-descriptive name C2 because of the confusion with the term C1.

The FAA makes a distinction between branch coverage and decision coverage, with branch coverage weaker than decision coverage [[SVTAS2007](#)]. The FAA definition of a decision is, in part, "A Boolean expression composed of conditions and zero or more Boolean operators." So the FAA definition of decision coverage requires all Boolean expressions to evaluate to both true and false, even those that do not affect control flow. There is no precise definition of "Boolean expression." Some languages, especially C, allow mixing integer and Boolean expressions and do not require Boolean variables be declared as Boolean. The FAA suggests using context to identify Boolean expressions, including whether expressions are used as

operands to Boolean operators or tested to control flow. The suggested definition of "Boolean operator" is a built-in (not user-defined) operator with operands and result of Boolean type. The logical-not operator is exempted due to its simplicity. The C conditional operator (`?:`) is considered a Boolean operator if all three operands are Boolean expressions.

This metric has the advantage of simplicity without the problems of [statement coverage](#).

A disadvantage is that this metric ignores branches within Boolean expressions which occur due to short-circuit operators. For example, consider the following C/C++/Java code fragment:

```
if (condition1 && (condition2 || function1()))
    statement1;
else
    statement2;
```

This metric could consider the control structure completely exercised without a call to `function1`. The test expression is true when `condition1` is true and `condition2` is true, and the test expression is false when `condition1` is false. In this instance, the short-circuit operators preclude a call to `function1`.

The FAA suggests that for the purposes of measuring decision coverage, the operands of short-circuit operators (including the C conditional operator) be interpreted as decisions [[SVTAS2007](#)].

Condition Coverage

Condition coverage reports the true or false outcome of each condition. A condition is an operand of a logical operator that does not contain logical operators. Condition coverage measures the conditions independently of each other.

This metric is similar to [decision coverage](#) but has better sensitivity to the control flow.

However, full condition coverage does not guarantee full [decision coverage](#). For example, consider the following C++/Java fragment.

```
bool f(bool e) { return false; }
bool a[2] = { false, false };
if (f(a && b)) ...
if (a[int(a && b)]) ...
if ((a && b) ? false : false) ...
```

All three of the if-statements above branch false regardless of the values of `a` and `b`. However if you exercise this code with `a` and `b` having all possible combinations of values, condition coverage reports full coverage.

Multiple Condition Coverage

Multiple condition coverage reports whether every possible combination of conditions occurs. The test cases required for full multiple condition coverage of a decision are given by the logical operator truth table for the decision.

For languages with short circuit operators such as C, C++, and Java, an advantage of multiple condition coverage is that it requires very thorough testing. For these languages, multiple condition coverage is very similar to [condition coverage](#).

A disadvantage of this metric is that it can be tedious to determine the minimum set of test cases required, especially for very complex Boolean expressions. An additional disadvantage of this metric is that the number of test cases required could vary substantially among conditions that have similar complexity. For example, consider the following two C/C++/Java conditions.

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

To achieve full multiple condition coverage, the first condition requires 6 test cases while the second requires 11. Both conditions have the same number of operands and operators. The test cases are listed below.

	a	&&	b	&&	(c		(d	&&	e))
1.	F		-		-		-		-
2.	T		F		-		-		-
3.	T		T		F		F		-
4.	T		T		F		T		F
5.	T		T		F		T		T
6.	T		T		T		-		-

	((a		b))	&&	(c		d))	&&	e
1.	F		F		-		-		-
2.	F		T		F		F		-
3.	F		T		F		T		F
4.	F		T		F		T		T
5.	F		T		T		-		F
6.	F		T		T		-		T
7.	T		-		F		F		-
8.	T		-		F		T		F
9.	T		-		F		T		T
10.	T		-		T		-		F
11.	T		-		T		-		T

As with [condition coverage](#), multiple condition coverage does not include [decision coverage](#).

For languages without short circuit operators such as Visual Basic and Pascal, multiple condition coverage is effectively [path coverage](#) (described below) for logical expressions, with the same advantages and disadvantages. Consider the following Visual Basic code fragment.

```
If a And b Then
...
```

Multiple condition coverage requires four test cases, for each of the combinations of a and b both true and false. As with [path coverage](#) each additional logical operator doubles the number of test cases required.

Condition/Decision Coverage

Condition/Decision Coverage is a hybrid metric composed by the union of [condition coverage](#) and [decision coverage](#).

It has the advantage of simplicity but without the shortcomings of its component metrics.

[BullseyeCoverage](#) measures condition/decision coverage.

Modified Condition/Decision Coverage

The formal definition of modified condition/decision coverage is:

Every point of entry and exit in the program has been invoked at least once, every condition in a decision has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome. A condition is shown to independently affect a decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions, or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome. [[DO-178C](#)].

Also known as MC/DC and MCDC. This metric is stronger than condition/decision coverage, requiring more test cases for full coverage.

This metric is specified for safety critical aviation software by RCTA/DO-178C and has been the subject of much study, debate and clarification for many years. Two difficult issues with MCDC are:

- short circuit operators
- multiple occurrences of a condition

There are two competing ideas of how to handle short-circuit operators. One idea is to relax the requirement that conditions be held constant if those conditions are not evaluated due to a short-circuit

operator [[Chilenski1994](#)]. The other is to consider the condition operands of short-circuit operators as separate decisions [[DO-248C](#)].

A condition may occur more than once in a decision. In the expression "A or (not A and B)", the conditions "A" and "not A" are coupled - they cannot be varied independently as required by clause (1) in the definition of MCDC. Clause (1) is known as unique-cause MC/DC. Clause (2) relaxes the independence requirement, and is known as masking MC/DC. These two different versions of MC/DC are defined in more detail below.

For unique-cause MC/DC, a condition is shown to independently affect a decisions outcome by varying just that condition while holding fixed all other possible conditions.

For masking MC/DC, a condition is shown to independently affect a decisions outcome by applying principles of Boolean logic to assure that no other condition influences the outcome (even though more than one condition in the decision may change value) [[CAST-6](#)].

Path Coverage

This metric reports whether each of the possible paths in each function have been followed. A path is a unique sequence of branches from the function entry to the exit.

Also known as predicate coverage. Predicate coverage views paths as possible combinations of logical conditions [[Beizer1990](#) p.98].

Since loops introduce an unbounded number of paths, this metric considers only a limited number of looping possibilities. A large number of variations of this metric exist to cope with loops. Boundary-interior path testing considers two possibilities for loops: zero repetitions and more than zero repetitions [[Ntafos1988](#)]. For do-while loops, the two possibilities are one iteration and more than one iteration.

Path coverage has the advantage of requiring very thorough testing. Path coverage has two severe disadvantages. The first is that the number of paths is exponential to the number of branches. For example, a function containing 10 `if`-statements has 1024 paths to test. Adding just one more `if`-statement doubles the count to 2048. The second disadvantage is that many paths are impossible to exercise due to relationships of data. For example, consider the following C/C++ code fragment:

```
if (success)
    statement1;
statement2;
if (success)
    statement3;
```

Path coverage considers this fragment to contain 4 paths. In fact, only two are feasible: `success=false` and `success=true`.

Researchers have invented many variations of path coverage to deal with the large number of paths. For example, n-length sub-path coverage reports whether you exercised each path of length n branches. Basis path testing selects paths that achieve decision coverage, with each path containing at least one decision outcome differing from the other paths [[Roper1994](#) p.48]. Others variations include [linear code sequence and jump \(LCSAJ\) coverage](#) and [data flow coverage](#).

Other Metrics

Here is a description of some variations of the fundamental metrics and some less commonly use metrics.

Function Coverage

This metric reports whether you invoked each function or procedure. It is useful during preliminary testing to assure at least some coverage in all areas of the software. Broad, shallow testing finds gross deficiencies in a test suite quickly.

[BullseyeCoverage](#) measures function coverage.

Call Coverage

This metric reports whether you executed each function call. The hypothesis is that bugs commonly occur in interfaces between modules.

Also known as call pair coverage.

Linear Code Sequence and Jump (LCSAJ) Coverage

This variation of [path coverage](#) considers only sub-paths that can easily be represented in the program source code, without requiring a flow graph [Woodward1980]. An LCSAJ is a sequence of source code lines executed in sequence. This "linear" sequence can contain decisions as long as the control flow actually continues from one line to the next at run-time. Sub-paths are constructed by concatenating LCSAJs. Researchers refer to the coverage ratio of paths of length n LCSAJs as the test effectiveness ratio (TER) $n+2$.

The advantage of this metric is that it is more thorough than [decision coverage](#) yet avoids the exponential difficulty of [path coverage](#). The disadvantage is that it does not avoid infeasible paths.

Data Flow Coverage

This variation of [path coverage](#) considers only the sub-paths from variable assignments to subsequent references of the variables.

The advantage of this metric is the paths reported have direct relevance to the way the program handles data. One disadvantage is that this metric does not include [decision coverage](#). Another disadvantage is complexity. Researchers have proposed numerous variations, all of which increase the complexity of this metric. For example, variations distinguish between the use of a variable in a computation versus a use in a decision, and between local and global variables. As with data flow analysis for code optimization, pointers also present problems.

Object Code Branch Coverage

This metric reports whether each machine language conditional branch instruction both took the branch and fell through.

This metric gives results that depend on the compiler rather than on the program structure since compiler code generation and optimization techniques can create object code that bears little similarity to the original source code structure.

Since branches disrupt the instruction pipeline, compilers sometimes avoid generating a branch and instead generate an equivalent sequence of non-branching instructions. Compilers often expand the body of a function inline to save the cost of a function call. If such functions contain branches, the number of machine language branches increases dramatically relative to the original source code.

You are better off testing the original source code since it relates to program requirements better than the object code.

Loop Coverage

This metric reports whether you executed each loop body zero times, exactly once, and more than once (consecutively). For do-while loops, loop coverage reports whether you executed the body exactly once, and more than once.

The valuable aspect of this metric is determining whether `while`-loops and `for`-loops execute more than once, information not reported by other metrics.

As far as I know, only [GCT](#) implements this metric.

Race Coverage

This metric reports whether multiple threads execute the same code at the same time. It helps detect failure to synchronize access to resources. It is useful for testing multi-threaded programs such as in an operating system.

As far as I know, only [GCT](#) implements this metric.

Relational Operator Coverage

This metric reports whether boundary situations occur with relational operators (<, <=, >, >=). The hypothesis is that boundary test cases find off-by-one mistakes and uses of the wrong relational operators such as < instead of <=. For example, consider the following C/C++ code fragment:

```
if (a < b)
    statement;
```

Relational operator coverage reports whether the situation $a=b$ occurs. If $a=b$ occurs and the program behaves correctly, you can assume the relational operator is not suppose to be <=.

As far as I know, only [GCT](#) implements this metric.

Weak Mutation Coverage

This metric is similar to [relational operator coverage](#) but much more general [[Howden1982](#)]. It reports whether test cases occur which would expose the use of wrong operators and also wrong operands. It works by reporting coverage of conditions derived by substituting (mutating) the program's expressions with alternate operators, such as "-" substituted for "+", and with alternate variables substituted.

This metric interests the academic world mainly. Caveats are many; programs must meet special requirements to enable measurement.

As far as I know, only [GCT](#) implements this metric.

Table Coverage

This metric indicates whether each entry in a particular array has been referenced. This is useful for programs that are controlled by a finite state machine.

Comparing Metrics

You can compare relative strengths when a stronger metric includes a weaker metric.

- [Decision coverage](#) includes [statement coverage](#) since exercising every branch must lead to exercising every statement. This relationship only holds when control flows uninterrupted to the end of all basic blocks. For example a C/C++ function might never return to finish the calling basic block because it uses `throw`, `abort`, the `exec` family, `exit`, or `longjmp`.
- [Condition/decision coverage](#) includes [decision coverage](#) and [condition coverage](#) (by definition).
- [Path coverage](#) includes [decision coverage](#).
- [Predicate coverage](#) includes [path coverage](#) and [multiple condition coverage](#), as well as most other metrics.

Academia says the stronger metric *subsumes* the weaker metric.

Coverage metrics cannot be compared quantitatively.

Coverage Goal for Release

Each project must choose a minimum percent coverage for release criteria based on available testing resources and the importance of preventing post-release failures. Clearly, safety-critical software should have a high goal. You might set a higher coverage goal for unit testing than for system testing since a failure in lower-level code may affect multiple high-level callers.

Using [statement coverage](#), [decision coverage](#), or [condition/decision coverage](#) you generally want to attain 80%-90% coverage or more before releasing. Some people feel that setting any goal less than 100% coverage does not assure quality. However, you expend a lot of effort attaining coverage approaching 100%. The same effort might find more bugs in a different testing activity, such as formal technical review. Avoid setting a goal lower than 80%.

Intermediate Coverage Goals

Choosing good intermediate coverage goals can greatly increase testing productivity.

Your highest level of testing productivity occurs when you find the most failures with the least effort. Effort is measured by the time required to create test cases, add them to your test suite and run them. It follows that you should use a coverage analysis strategy that increases coverage as fast as possible. This gives you the greatest probability of finding failures sooner rather than later. Figure 1 illustrates the coverage rates for high and low productivity. Figure 2 shows the corresponding failure discovery rates.

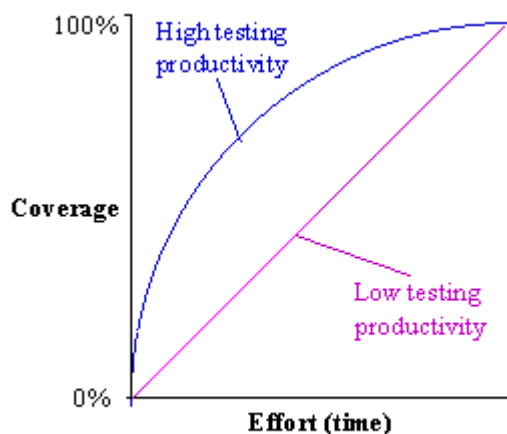


Figure 1: Coverage rate

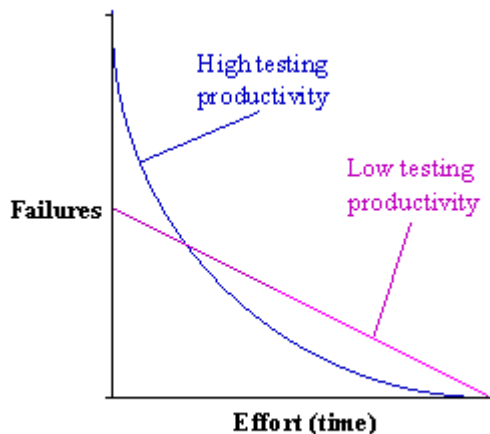


Figure 2: Failure discovery rate

One strategy that usually increases coverage quickly is to first attain some coverage throughout the entire test program before striving for high coverage in any particular area. By briefly visiting each of the test program features, you are likely to find obvious or gross failures early. For example, suppose your application prints several types of documents, and a bug exists which completely prevents printing one (and only one) of the document types. If you first try printing one document of each type, you probably find this bug sooner than if you thoroughly test each document type one at a time by printing many documents of that type before moving on to the next type. The idea is to first look for failures that are easily found by minimal testing.

The sequence of coverage goals listed below illustrates a possible implementation of this strategy.

1. Invoke at least one function in 90% of the source files (or classes).
2. Invoke 90% of the functions.
3. Attain 90% [condition/decision coverage](#) in each function.
4. Attain 100% [condition/decision coverage](#).

Notice we do not require 100% coverage in any of the initial goals. This allows you to defer testing the most difficult areas. This is crucial to maintaining high testing productivity; achieve maximum results with minimum effort.

Avoid using a weaker metric for an intermediate goal combined with a stronger metric for your release goal. Effectively, this allows the weaknesses in the weaker metric to decide which test cases to defer. Instead, use the stronger metric for all goals and allow the difficulty of the individual test cases help you decide whether to defer them.

Summary

Coverage analysis is a structural testing technique that helps eliminate gaps in a test suite. It helps most in the absence of a detailed, up-to-date requirements specification. [Condition/decision coverage](#) is the best general-purpose metric for C, C++, and Java. Setting an intermediate goal of 100% coverage (of any type) can impede testing productivity. Before releasing, strive for 80%-90% or more coverage of statements, branches, or conditions.

References

Beizer1990 "Software Testing Techniques", 2nd edition, Boris Beizer, New York: Van Nostrand Reinhold, 1990

CAST-6 "Rationale for Accepting MC/DC in Certification Projects", Certification Authorities Software Team, August 2001

Chilenski1994 "Applicability of Modified Condition/Decision Coverage to Software Testing", John Joseph Chilenski and Steven P. Miller, Software Engineering Journal, September 1994, Vol. 9, No. 5, pp.193-200.

DO-178C, "Software Considerations in Airborne Systems and Equipment Certification", RCTA, December 2011.

DO-278C, "Supporting Information for DO-178C and DO-278A", RCTA, December 2011.

SVTAS2007 Software Verification Tools Assessment Study, FAA, June 2007

Howden1982 "Weak Mutation Testing and Completeness of Test Sets", *IEEE Trans. Software Eng.*, Vol.SE-8, No.4, July 1982, pp.371-379.

McCabe1976 Tom McCabe, "A Software Complexity Measure", *IEEE Trans. Software Eng.*, Vol.2, No.6, December 1976, pp.308-320.

Morell1990 "A Theory of Fault-Based Testing", Larry Morell, *IEEE Trans. Software Eng.*, Vol.16, No.8, August 1990, pp.844-857.

Ntafos1988 "A Comparison of Some Structural Testing Strategies", Simeon Ntafos, *IEEE Trans. Software Eng.*, Vol.14, No.6, June 1988, pp.868-874.

Roper1994 Marc Roper, "Software Testing", London, McGraw-Hill Book Company, 1994

Woodward1980 "Experience with Path Analysis and Testing of Programs", M.R Woodward, D. Hedley, and M.A. Hennell, *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 3, pp. 278-286, May 1980.