Code Coverage Analysis

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

Contents

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 measure, 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.

- Faults relate to control flow and you can expose faults 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 faults of omission, and no unreachable code.

Clearly, these assumptions do not always hold. Coverage analysis exposes some plausible faults but does not come close to exposing all classes of faults. 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 Measures

A large variety of coverage measures exist. Here is a description of some fundamental measures and their strengths and weaknesses.

Statement Coverage

This measure reports whether each executable statement is encountered.

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 undescriptive name C1. People sometimes incorrectly use the name C1 to identify decision coverage. Therefore this term has become ambiguous.

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

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 measures is that faults 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 measure is affected more by computational statements than by decisions.

Decision Coverage

This measure 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 measure includes coverage of `switch`-statement cases, exception handlers, and interrupt handlers.

Also known as: branch coverage, all-edges coverage [Roper1994 p.58], basis path coverage [Roper1994 p.48], C2 [Beizer1990 p.75], decision-decision-path testing [Roper1994 p.39]. "Basis path" testing selects paths that achieve decision coverage.

I discourage using the undescriptive name C2 because of the confusion with the term C1.

This measure has the advantage of simplicity without the problems of statement coverage.

A disadvantage is that this measure 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 measure 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`.

Condition Coverage

Condition coverage reports the true or false outcome of each boolean sub-expression, separated by logical-and and logical-or if they occur. Condition coverage measures the sub-expressions independently of each other.

This measure 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 boolean sub-expressions occurs. As with condition coverage, the sub-expressions are separated by logical-and and logical-or, when present. The test cases required for full multiple condition coverage of a condition are given by the logical operator truth table for the condition.

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 measure 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 measure 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.

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 measure composed by the union of condition coverage and decision coverage.

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

BullseyeCoverage measures condition/decision coverage.

Modified Condition/Decision Coverage

Also known as MC/DC and MCDC.

This measure requires enough test cases to verify every condition can affect the result of its encompassing decision [Chilenski1994]. This measure was created at Boeing and is required for aviation software by RCTA/DO-178B.

For C, C++ and Java, this measure requires exactly the same test cases as condition/decision coverage. Modified condition/decision coverage was designed for languages containing logical operators that do not short-circuit. The short circuit logical operators in C, C++ and Java only evaluate conditions when their result can affect the encompassing decision. The paper that defines this measure [Chilenski1994] section "3.3 Extensions for short-circuit operators" says "[use of short-circuit operators] forces relaxation of the requirement that all conditions are held fixed while the condition of interest is varied." The only programming language referenced by this paper is Ada, which has logical operators that do not short circuit as well as those that do.

Path Coverage

This measure 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 measure considers only a limited number of looping possibilities. A large number of variations of this measure 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`-statem 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. Others variations include linear code sequence and jump (LCSAJ) coverage and data flow coverage.

Other Measures

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

Function Coverage

This measure 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 measure reports whether you executed each function call. The hypothesis is that faults 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 measure 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 measure is the paths reported have direct relevance to the way the program handles data. One disadvantage is that this measure does not include decision coverage. Another disadvantage is complexity. Researchers have proposed numerous variations, all of which increase the complexity of this measure. 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 measure reports whether each machine language conditional branch instruction both took the branch and fell through.

This measure 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 measure 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 measure is determining whether `while`-loops and `for`-loops execute more than once, information not reported by others measure.

As far as I know, only GCT implements this measure.

Race Coverage

This measure 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 measure.

Relational Operator Coverage

This measure reports whether boundary situations occur with relational operators (<, <=, >, >=). The hypothesis is that boundary test cases find off-by-one errors and mistaken uses of 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 measure.

Weak Mutation Coverage

This measure 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 measure 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 measure.

Table Coverage

This measure 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 Measures

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

- Decision coverage includes statement coverage since exercising every branch must lead to exercising every statement.
- 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 measures.

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

Coverage measures 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 faults 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.

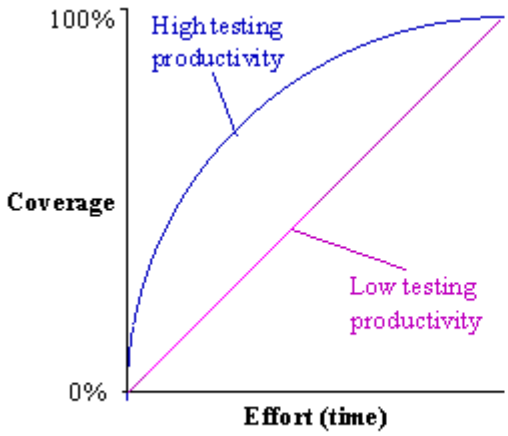

Figure 1: Coverage rate

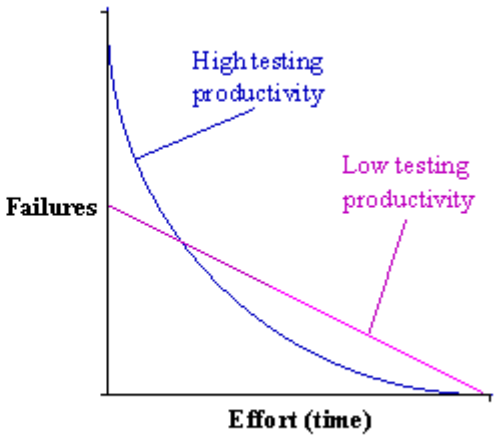

Figure 2: Failure discovery rate

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 fault 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 fault 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 measure for an intermediate goal combined with a stronger measure for your release goal. Effectively, this allows the weaknesses in the weaker measure to decide which test cases to defer. Instead, use the stronger measure 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 measure 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 Beizer, Boris, "Software Testing Techniques", 2nd edition, New York: Van Nostrand Reinhold, 1990

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

RTCA/DO-178B, "Software Considerations in Airborne Systems and Equipment Certification", RCTA, December 1992, pp.31, 74.

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

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

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

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

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

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

Glossary

Fault - A bug. A defect.

Error - A mistake made by a person that results in a fault.

Failure - The run-time manifestation of a fault.
