McCabe SOFTWARE

White Paper

Improved Software Testing Using McCabe IQ Coverage Analysis

Table of Contents

Introduction....................................1
What is Coverage Analysis? 2
The McCabe IQ Approach to Coverage Analysis 3
The Importance of Coverage Analysis 4
Where Coverage Analysis Fits into your Existing Testing Processes 6
McCabe IQ Coverage Analysis and Functional Testing................6
McCabe IQ Coverage Analysis and Incremental Testing 8
McCabe IQ Coverage Analysis and Unit Level Testing 9

With the increased pace of production schedules, the tremendous proliferation of software design methodologies and programming languages, and the increased size of software applications, software testing has evolved from a routine quality assurance activity into a sizable and complex challenge in terms of manageability and effectiveness. The major challenges to software testing in today's business environment are:

- **Efficiency.** Is the test cycle too long? How can you ensure every test is a good investment of time and money?
- **Thoroughness.** How can you tell when you are done testing? How can you be reasonably sure the program is bug-free?
- **Resource Management.** Are testing resources strategically allocated, focusing on the highest-risk elements of the software? Are the functionally-central parts of the program receiving an acceptable level of testing?

Coverage analysis methodologies have been enlisted by many managers to help. Coverage analysis determines what areas of source code, out of the totality of source code needing to be tested, have and have not been 'covered' by any given set of tests. This information allows managers to better direct testing to where it is most needed and to better assess various levels of "testedness".

All coverage analysis techniques, however, are not created equal. The use you make of coverage analysis determines which, if any, of the challenges noted above can be met. For example, effective approaches to coverage analysis use a *variety* of coverage measurements (beyond simple line coverage), and effective approaches combine coverage results with other critical information (for example, metrics identifying particularly complex areas of the code under test). Maximizing the potential of coverage analysis in these and other ways using McCabe IQ is the topic of this paper.

The Purpose of this Paper

- To introduce coverage analysis as an increasingly important direction in the management of software testing
- To describe how the unique coverage analysis techniques available in McCabe IQ can add value to your test processes. Specifically, this paper covers test assessment and improvement using McCabe IQ coverage analysis in the areas of functional testing, incremental testing, and unit level testing.

T: 800-638-6316
www.mccabe.com

McCabe SOFTWARE

The "Percentage Lines of Code' Fallacy

It seems reasonable that coverage reports indicating the percentage of lines of code tested would be a good indicator of "testedness'. If, for example, a report indicates that 99% of the lines have been executed during testing, it is reasonable to feel as though the testing has been thorough. There are several reasons why this is a fallacy.

- What needs to be thoroughly tested is the *logic* of the program (the decisions it is designed to make), and line coverage percentages are no indication of the percentage of logic tested. For example, because units of logic vary in line length, 99% line coverage might only be covering 60% of the logic.
- Line coverage percentages cannot account for unexecutable lines, such as blank lines and comment lines. It is impossible, therefore, to determine what the percentages represent.
- Some languages can execute multiple statements on a single line. An extreme example would be an entire application that occupied one line. Obviously, a line coverage report would be meaningless for such an application. The ability to perform branch and path coverage reports is, for these reasons, crucial to successfully implementing coverage analysis.

What is Coverage Analysis?

Coverage analysis is a means of tracking which areas of a program (most commonly, for example, which lines of code) have and have not been tested by a given round of tests. This is made possible by configuring the program to store 'trace data' whenever a test is executed on it œ that is, to store data detailing which areas of the program were used to perform the tested function or operation. We say that the program has been "instrumented" to produce 'trace data'.

There are two ways for an application to be instrumented: *object insertion* and *source code insertion*. In the case of object insertion, an already-compiled executable of the program under test is modified to store the trace data. In the case of source code insertion, statements to store trace data are added to the source code before it is compiled. Source code insertion has several advantages over object insertion:

- **More Measurement Potential.** Programs instrumented using source code insertion can track the coverage of more than just lines of code. The object insertion method usually limits trace data to line coverage, which is the least useful kind of coverage (read "The 'Percentage Lines of Code' Fallacy"). The source code insertion method allows you to track the coverage of the following as well:

  Branches. A branch is one possible outcome of a programmatic decision, such as an IF-THEN statement. Branch coverage analysis ensures thorough testing of all the possible logic in a program, units of logic (not lines of code) being the fundamental building blocks of all programs.

  Paths. A path is an executable sequence of programmatic decisions. Testing is tasked with ensuring not only the accuracy of units of logic, but also the combinations of logic. Toward this end, path coverage analysis ensures thorough testing of a program's executable processes.

- **More Flexibility.** Object insertion methods notoriously produce unmanageable amounts of trace data for large programs. Source code insertion makes coverage analysis useful for large programs because it allows more control over the amount of trace data stored during testing.
- **More Applicability.** Source code insertion works for any compiler/platform, whereas object insertion only works for specific compilers/platforms.

McCabe SOFTWARE

What is —Baseline Code Analysis"?

Baseline code analysis is an umbrella term for the combined source code analysis capabilities of McCabe IQ, particularly structural analysis and metrical analysis. Structural analysis makes a program's architecture maximally comprehensible by identifying and visually mapping the calling hierarchy of all of the program's modules. Metrical analysis provides measurements of critical code components (such as the number of lines or operands). Such measurements can be useful indicators of code quality allowing you to highlight highly complex and unstructured modules in a program.

Baseline code analysis is useful throughout the software production cycle, not just for testing. To learn more, please refer to McCabe's white paper entitled, —Baseline Code Analysis with McCabe IQ."

The McCabe IQ Approach to Coverage Analysis

The McCabe IQ approach to coverage analysis uses the more flexible and more applicable source code insertion method of instrumenting programs, taking full advantage of this method's benefits: scalability, compatibility with most programming languages, and, most important, the ability to do branch and path coverage reporting. As will be discussed in greater detail later in this paper, branch and path coverage reporting is the cornerstone of coverage analysis with McCabe IQ. In addition to this, several qualities make the McCabe IQ approach unique:

Not Just Path Analysis: *Cyclomatic* Path Analysis!

The total number of testable paths of any given program is very large or as many as 2 to the power of the number of decisions embedded in the code. For a relatively small program that can make 50 decisions, for example, the total number of testable paths could be as high as 2^{50}, or around 2,000,000,000,000,000.[1]

Finding a meaningful *subset* of paths to test is therefore imperative. The paths identified by McCabe IQ for the sake of coverage analysis are not representative of all of the possible paths in the program, but rather the minimum set of paths required to pass through every decision at least once. Such "cyclomatic" path analysis is the condition of possibility for path coverage techniques that are useful and profitable.

Combines Coverage Analysis with Other Kinds of Source Code Analysis

As mentioned, more value can be added to the test process if coverage analysis can be combined with other kinds of source code analysis. The McCabe IQ approach combines coverage analysis with:

- Metrical Analysis: Through its powerful baseline code analysis capabilities, McCabe IQ provides instant access to metrics–measurements of various code characteristics–that indicate the relative complexity and structuredness of a program's various modules (read "What is 'Baseline Code Analysis'?" in

[1] The number of logical paths would never actually be this high (each decision would have to have 49 branches passing processing to each of the other decisions). But the example is meant to stress the impossibility of achieving 100% path coverage in most instances.

Objectives of Coverage Analysis with McCabe IQ
Given the unique capabilities supported by McCabe IQ, the primary objectives of coverage analysis using McCabe IQ can be summarized as follows:

- Assessing the completeness of testing
- Pinpointing areas of the program that need better testing, and improving test plans to address such untested or poorly-tested areas
- Verifying the testing of changes since the last full test cycle
- Targeting the modules most at risk from defects with more rigorous tests.

the sidebar area). Exceedingly complex or unstructured code segments can be highlighted in the coverage reports, making it easy for testers to identify those areas of the program most at risk from defects.

- Software Change Analysis: Whenever a program is modified, testing needs to be focused on the modified code and the areas of the program that are potentially affected by the modifications. McCabe IQ's software change analysis capabilities can pinpoint both modified code and the subset of modules that are potentially impacted by those modifications. Coverage analysis can be focused on these two domains. This way time and resources are not wasted testing areas of the program that do not need to be tested.

Makes Coverage Results Easy to Obtain and Understandable

With McCabe IQ, an instrumented version of the source code under test can be obtained with a click of the mouse, as can sophisticated graphical displays (on-screen 'maps') of testable branches and paths. When you supply the system with trace data that was generated from a round of tests, McCabe IQ can highlight the branches and paths that were executed in the branch and path maps (see Figure 1). This makes coverage results instantly accessible and understandable.

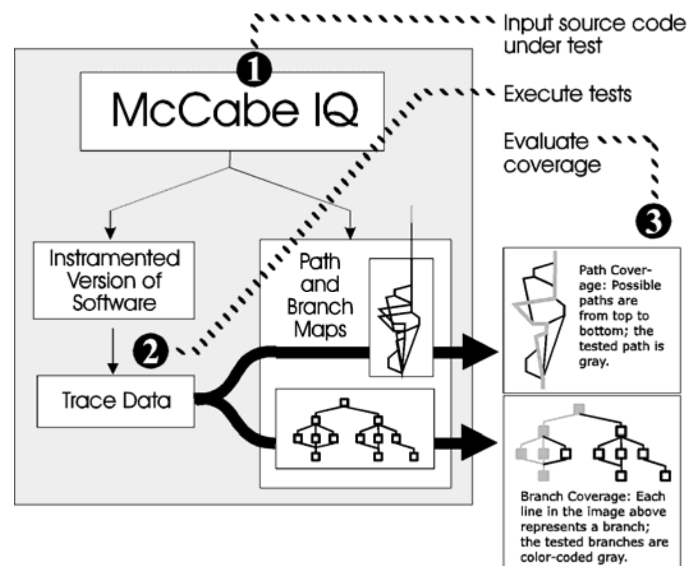


Figure 1: McCabe IQ Coverage Analysis

The Importance of Coverage Analysis

Coverage analysis makes the test cycle more efficient. If you can instantly identify untested paths and branches, you can streamline test plans to address only untested parts of the program, short-cutting the quest for meaningful tests and preventing such common problems as overtesting (spending

McCabe SOFTWARE

What Black Box Testing Doesn't See

- Implicit (Undocumented) Functionality

This is functionality that is not described in the requirements document but that has been added in the implementation phase be- cause it was either not included in the requirements, or is needed to solve specific implementation issues not foreseen during requirements analysis or de- sign.

- Functional Combinations

These are operations that require a combination of functions. A program may need to do something when input A is true and some- thing else when input B is true, but it could do some- thing different if both input A and input B are true. Design specifications routinely dictate the requirements of isolated functions, but they rarely address the requirements of functional combinations. As a result, operations that combine functionality often exhibit unexpected side effects. Because black box test methods follow from the documented design specs, they aren't looking for un- documented functionality or functional combinations.

excessive time on areas of the program that are at low risk for defects) and redundant testing (testing the same functions over and over again).

Coverage analysis makes the whole test process more effective by making the code, rather than the requirements specifications, the final reference point for test efforts. When test plans are developed from the requirements specifications, what is being tested is inevitably only a percentage of the actual testable elements of any program. That is because such so-called "black box" testing only tests what was intended in the design stage, not what you've got (that is, the code itself). For example, black box testing is prone to missing implicit functionality and it is weak at testing functional combinations to acceptable levels, both of which arise in the implementation stage of software development, after the requirements document has been written (read "What Black Box Testing Doesn't See" in the sidebar area). With coverage analysis, objective data about 'what you've got' is the ultimate reference point for focusing the testing effort and determining when the program has been adequately tested.

In extreme cases, coverage analysis corresponds to so-called "white box" testing, where the goal is to exhaustively test all internal units, branches and paths of the code. But in practice, coverage analysis is more useful as a support tool for black box testing efforts. Particularly, coupled with detailed metrics (which identify the most at-risk-for-defects code segments) the goal of coverage analysis is often not total coverage, but risk factoring and risk management. You may not have time to execute tests that give you 100% coverage of the entire program, but you can be sure that maximum coverage is provided for the most complex and at-risk-for-defects segments of the code. We call this "gray box" testing.

Coverage analysis is also important for improving communication channels between QA/Test teams and Product Development (PD). For example, when bugs are reported to PD, the precise unit of code causing the bug can be reported as well. Likewise, when QA requires input of PD for the sake of developing tests, coverage charts provide both parties a common point of reference.

McCabe SOFTWARE

Where Coverage Analysis Fits into your Existing Testing Processes

There are many different kinds of testing that serve a variety of different purposes. Most kinds of testing, however, fall within one of the following three general categories:

- **Functional Testing.** Tests the functionality of a new application based on the requirements specifications. This is commonly called "black box" testing.
- **Incremental Testing.** Tests functionality that has been modified or added since the last full test cycle. Verifies that unmodified functionality was not broken by the modifications or additions *(regression testing)*.
- **Unit Level Testing.** Tests small subsets of an application, focusing on inputs, outputs, boundary conditions and logical sequences. This is commonly called "white box" testing.

The remainder of this paper describes techniques for improving testing in these three areas using McCabe IQ coverage analysis. Each of these models has a different purpose and each presents different challenges that coverage analysis techniques using McCabe IQ are uniquely suited to address. [2]

McCabe IQ Coverage Analysis and Functional Testing

In functional testing, QA/Test groups derive functional tests from the program's requirements specifications, and then execute the tests on the application to verify that it performs as expected. The purpose of this type of testing is to identify missing functionality, incorrectly implemented functionality, and functional failures (i.e., bugs).[3]

Functional testing is often thought of as a straightforward process of deriving tests from specifications, however there are several frequently overlooked problems with this practice. These problems are outlined below.

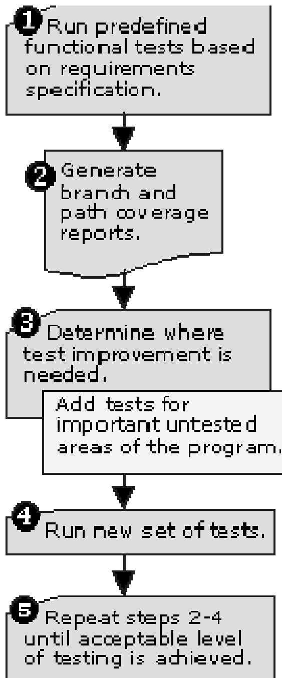
---

[2] This paper addresses coverage analysis techniques only for the most common test models. Coverage analysis with McCabe IQ is also applicable to test models not addressed in this white paper including, for example, *stress/load testing* and *performance testing.*

[3] Any kind of testing that is focused on verifying functionality can be included in this category as well, including *integration testing, system testing,* and *user acceptance testing.*

The Challenges of Functional Testing

Typical Sequence:
Functional Testing using McCabe IQ Coverage Analysis



Testing Implicit Functionality and Functional Combinations. Functional testing is good at making sure what was intended in the design stage is indeed what you've got. But it is particularly challenged with ensuring the *thoroughness of* testing because, as mentioned earlier, tests derived from the specifications document tend to miss implicit (undocumented) functionality and problematic behaviors arising from functional combinations.

Determining When to Stop. If an application has more than a few requirements, the number of testable functional combinations becomes enormous very quickly. The equation for testable functional combinations is the same as the equation for possible decision sequences (i.e., paths). That is, the number of testable functional combinations will be as many as 2 to the power of the number of functional requirements. Thus even a program that had as few as 50 functional requirements would harbor over 1 billion functional combinations (2^{50}). It is obviously impossible to test all the combinations.[4] An effective, directed approach to functional combination testing is therefore critical–an approach that makes it possible to work out when enough testing has been completed.

How McCabe IQ Coverage Analysis Helps Meet the Challenges of Functional Testing

Using McCabe IQ coverage analysis, you can:

- Pinpoint untested branches.
  - Targeting untested branches targets most implicit as well as explicit functionality.
- Identify modules most at risk for defects, and target them for more rigorous tests using path coverage analysis.
  - Using path coverage analysis in this way addresses *all* functionality (implicit and explicit) as well as critical functional combinations in the most at-risk areas of the program.

---

[4] For a more in-depth study of the issues surrounding functional combination testing, refer to *Art of Software Testing* by Glenford J. Myers.

McCabe SOFTWARE

- Track the accumulated "testedness" of branches and paths over any number of rounds of testing.
- Use the cyclomatic paths identified by McCabe IQ as your index for the "testedness" of the at-risk modules.

  An acceptable level of testing can be determined based on the accumulated coverage of branches and cyclomatic paths.

McCabe IQ Coverage Analysis and Incremental Testing

Incremental testing refers to testing that is done on revised versions of a program in development. This kind of testing must verify that the changes made to the code have fixed the reported defects, that added functionality does what is required, and that unchanged functionality was not 'broken' by the modifications or additions (regression testing).

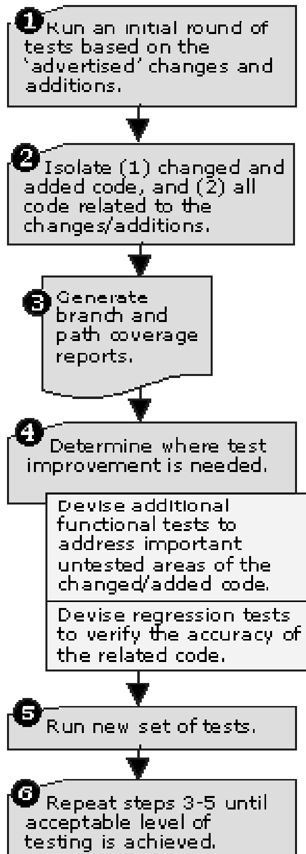
The Challenges of Incremental Testing

Determining Which Tests to Run. Determining which tests to run to verify program fixes is usually pretty straightforward–run the tests that resulted in bugs on the prior test cycle. But the difficulty is in determining the tests to run in order to verify that nothing previously functioning correctly was broken when the program was 'improved'. In other words, testers are challenged with assessing the impact of the changes throughout the program. Without a clear picture of the modules related to the changed code, it is very difficult to determine what unchanged functionality to target for regression tests and to develop a set of tests that ensures that all implicated parts of the program (but *only* implicated parts of the program!) are being tested.

Verifying that Modified Functionality has been Tested. Here the difficulty lies in the fact that changes to a program often introduce new implicit functionality. Verifying that all the modifications have been tested (as opposed to just the *advertised* modified functionality) is impossible without techniques over and above black box testing.

Verifying that New Functionality has been Tested. Testing new functionality leads to the same challenges encountered when performing full functional testing (see "The Challenges of Functional Testing" above)– namely, verifying the completeness of testing, the testing of implicit functionality, and the testing of functional combinations.

Typical Sequence:
Incremental Testing using McCabe IQ Coverage Analysis



How McCabe IQ Coverage Analysis Helps Meet the Challenges of Incremental Testing

To meet the challenges of testing added functionality, McCabe IQ coverage analysis helps in precisely the same ways described in "How McCabe IQ Coverage Analysis Helps Meet the Challenges of Functional Testing" earlier in this paper. Helping you meet the additional challenges of incremental testing, McCabe IQ allows you to:

- Pinpoint the precise branches and modules of the program that have been changed.
  - Coverage analysis can be restricted to this domain, focusing the testing effort where tests are needed.
- Isolate the areas of the program that are potentially affected by the changes and additions.
  - This allows you to focus regression testing where it is needed and ignore those parts of the program that are in no way related to the changed or added modules.

McCabe IQ Coverage Analysis and Unit Level Testing

Unit level testing is testing that is focused on small, isolated code segments ("units"). A unit is variably defined, but a common classification is "the smallest collection of code which can be usefully tested." The purpose of this type of testing is to:

- Ensure that each unit does what it is programmed to do
- Locate boundary condition failures and hence unexpected side effects
- Verify that processes not directly represented by the program's outputs are functioning correctly.

Unit level testing can be best understood by comparing it to functional testing. When testers run functional tests, they rely on the resulting values/behaviors to verify the accuracy of the code. But this can be deceptive, because the outcome of a functional test only verifies that the decisions made by the program produced the correct 'answer', not that it did it in the right way. For example, suppose a subroutine is supposed to internally store certain data values whenever a function is executed. Because the subroutine process is transparent to the function's visible behavior, that unit's behavior could be wrong even when the function would, to all appearances, be operating correctly.

McCabe SOFTWARE

Why Unit Level Testing is Important

Unit level testing is part of a complete test strategy.

Because it is usually performed early in the development process, it is more cost-effective at locating errors.

In practice, unit level testing ranges from the ad hoc tests done by programmers as they are writing code to systematic white box testing, where every unit must be tested and documented by a QA/Test group. In either case, the tester begins with the goal of coverage, for it is the very purpose of unit level testing to achieve the highest level of coverage possible.

The Challenges of Unit Level Testing

Deriving Tests. The greatest challenge of unit level testing is to identifying a minimum set of unit level tests to run. In an ideal world, every possible path of a program would be tested, accounting for all executable decisions in all possible combinations, but this is impossible when one considers the enormous number of potential paths embedded in any given program (as mentioned, 2 to the power of the number of decisions). The challenge is to isolate a subset of paths that provide coverage for all testable units, and to make that subset as minimal and free of unit-level redundancies as possible.

How McCabe IQ Coverage Analysis Helps with Unit Level Testing

McCabe IQ's path maps are precisely designed for making unit level coverage manageable. The paths identified by McCabe IQ are not representative of all of the possible paths in the program, but precisely the minimum set of paths necessary to test all code units. Unit level testing with McCabe IQ usually aims at making sure this subset of paths are covered–at least for the most at-risk-for-defects parts of the program, if not for the program in its entirety.

Summary

- There are two major ways to implement coverage analysis. The more commonly available *object insertion* technique is not scalable to larger applications. The *source code insertion* technique used in McCabe IQ is more controllable and scalable to large applications.
- There are different kinds of coverage you can track: line, branch and path. Only the more sophisticated branch and path techniques provide useful information.
- Combining coverage results with other information œ including detecting changed code, complex or unstructured code, and code related to changes - provides added value to coverage analysis efforts.

McCabe SOFTWARE

- McCabe IQ is the industry leader in source insertion instrumentation, and is unique in its ability to integrate coverage analysis with a variety of other forms of source code analysis
- Because of its overall approach, coverage analysis with McCabe IQ is uniquely suited to meet the following challenges of functional, incremental, and unit level testing:
  - Testing functional combinations
  - Ensuring all, not just explicit functionality, is tested
  - Streamlining functional testing without sacrificing thoroughness (reducing instances of overtesting and undertesting; focusing testing only where it is needed)
  - Making thorough regression testing manageable. McCabe IQ allows you to identify areas of the program irrelevant for the purposes of regression testing.