



## **White paper**

**How to measure embedded systems  
software test quality**

### COPYRIGHT NOTICE

© Copyright 2011 Atollic AB. All rights reserved. No part of this document may be reproduced or distributed without the prior written consent of Atollic AB.

### TRADEMARK

*Atollic*, *Atollic TrueSTUDIO*, *Atollic TrueINSPECTOR*, *Atollic TrueVERIFIER* and *Atollic TrueANALYZER* and the Atollic logotype are trademarks or registered trademarks owned by Atollic. ECLIPSE™ is a registered trademark of the Eclipse foundation. All other product names are trademarks or registered trademarks of their respective owners.

### DISCLAIMER

The information in this document is subject to change without notice and does not represent a commitment of Atollic AB. The information contained in this document is assumed to be accurate, but Atollic assumes no responsibility for any errors or omissions. In no event shall Atollic AB, its employees, its contractors, or the authors of this document be liable for any type of damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

### DOCUMENT IDENTIFICATION

ASW-WPMTQ

June 2011

### REVISION

First version

#### **Atollic AB**

Science Park  
Gjuterigatan 7  
SE- 553 18 Jönköping  
Sweden

+46 (0) 36 19 60 50

**E-mail:** [sales@atollic.com](mailto:sales@atollic.com)

**Web:** [www.atollic.com](http://www.atollic.com)

#### **Atollic Inc**

115 Route 46  
Building F, Suite 1000  
Mountain Lakes, NJ 07046-1668  
USA

+1 (973) 784 0047 (Voice)

+1 (877) 218 9117 (Toll Free)

+1 (973) 794 0075 (Fax)

**E-mail:** [sales.usa@atollic.com](mailto:sales.usa@atollic.com)

**Web:** [www.atollic.com](http://www.atollic.com)

# Contents

Abstract .....	1
Introduction.....	2
Find problems as early as possible .....	3
What has been tested? .....	4
Different types of code coverage analysis .....	5
Statement or block coverage .....	5
Function coverage.....	6
Function call coverage.....	6
Branch coverage.....	7
Modified condition/decision coverage .....	7
Tools for test quality measurement .....	9
Summary.....	11

# Tables

**No table of figures entries found.**

## ABSTRACT

There has been increased focus on software quality and better software testing methodology, as many high profile examples of product defects due to software errors have come to light in recent years.

Embedded systems of today contain a lot more lines of code compared to only a few years ago. As the number of bugs is at least proportional to the number of lines of code (or most likely even worse), the problem of software errors and improved testing needed to eradicate bugs becomes more important than ever.

Measuring whether or not the testing is good enough to assure software quality is a challenge in itself. In most cases, testing is done informally and there is little or no quantitative evidence collected by testers to determine the level of the test coverage. It is of little value to have 100% test success if the tests only cover a small fraction of the code, as the majority of the code then remains completely untested.

This white paper outlines how modern tools and methodologies can provide exact information about test quality, even for testing performed when the application is running in the embedded target board.

## INTRODUCTION

Malfunctioning software is frequently identified as the culprit in quality problems in embedded systems products. Software testing is thus receiving increased focus, as a means for companies to deliver high-quality products in a timely manner to its customers.

The traditional approach of using manual methods to test software cannot keep up with the ever increasing amounts of code in contemporary product implementations. Better tools for both efficient execution of tests, and for better monitoring, reporting and analysis of test actions are required to keep pace with modern requirements. **Atollic TrueANALYZER®** is a new breed of powerful embedded in-target test automation tools that meet the testing requirements of today's application software.

A test report showing 100% test success may not be the reason for celebration that one might expect. This is because the tests that were run might be considered successful, but the test procedure itself might not have been applied to more than a small fraction of the software in the entire application. In other words, the age old question, "how do you know whether or not you have exercised every line of code?" pops up yet again.

While the small part of the application that has in fact been tested behaves as expected, the majority of the software may still remain untested and potentially contains many undiscovered bugs. Understanding the quality of your test procedures thus become critical in judging whether or not the product is tested well enough before release.

Dynamic execution flow analysis can be used to perform code coverage analysis, which is a means of measuring test quality. There are many types of code coverage analysis possible. These extend from the very basic, to the very stringent. As expected, the more stringent coverage analysis requires more effort and more trials, but it is more revealing of potential problems. **Atollic TrueANALYZER®** handles the most stringent types of code coverage analysis without appreciable extra effort from the developer or tester. More details on embedded systems test quality measurement methodologies and how this can be automated with tools will be provided below.

## FIND PROBLEMS AS EARLY AS POSSIBLE

Finding the cause of errors using a debugger is often necessary, but to fix a bug, it of course must first be detected. The economics of finding and fixing software bugs are well known. It is far cheaper to find and correct the bug before the product is delivered to customers. Development teams that find and correct bugs early in the development cycle cut costs while improving software quality at the same time.

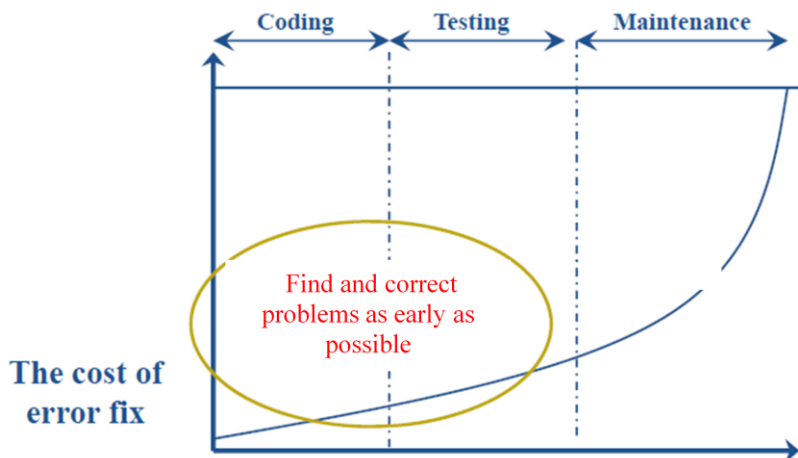


Figure 1 - It is cheaper to find and correct problems early

Testing is the process that typically reveals many bugs. In order to gain certainty that the testing has caused a real improvement in the software quality, it is also imperative to know the quality of the test procedures themselves.

The question of principle importance becomes, do your test suites cover 100% of the code? If the test procedures only drive a few percent of the code, which is more likely the case, then it is of great interest to know what was actually tested and what remains to be tested. Manual test methods often do not reveal this key metric.

## WHAT HAS BEEN TESTED?

Once code is developed and tested, the focus shifts to quantitatively understanding what actually happened during the testing. It is of no use to conclude that testing is completed with successful test results, if the test procedures do not test more than a fraction of the software. Manual software testing is fraught with code that “works by accident”. Further testing of the code sections that were inadvertently untested may well result in test failures when subjected to more thorough analysis.

Code coverage measurement, which is performed using dynamic execution flow analysis, is commonly used to study what parts of the code have been tested. This therefore is a direct measure of the test quality. There are many different types of code coverage analysis, from very simple analysis up to very stringent types. It is a very common error among software testers to believe that simple analysis suffices, when in fact the more stringent analysis methods are often required to reveal if the product has been properly tested or not.

As will be explained below, even a simple code section is very difficult to test rigorously. To make a thorough test, all code blocks should be executed, all branches should be exercised, and all sub-expressions in complex branch decisions ought to be tested such that different combinations of sub-expressions have been driving the branch decision independently of the other sub-expressions. For all but trivial code the number of permutations and combinations increases sharply as developers include more logic in their code.

Code coverage analysis has been classified formally. The more advanced types of code coverage analysis (such as MC/DC described below) are often used for measuring test quality of safety critical software.

As an example, RTCA DO-178B (a standard for development of flight safety critical software) requires MC/DC testing of software on “Level-A criticality”, the most critical part of airborne software, where a software error can lead to a catastrophic situation with loss of aircraft or human lives.

Many projects also outside the aerospace industry would benefit from better control of what has been tested. In particular this is valid for companies with high production volumes, or products that are difficult or expensive to upgrade in the field. The same goes for products where the supplier wants to keep its good reputation and where loss of goodwill can be costly for the company.

For these reasons, code coverage analysis ought to be used to verify whether the software has been tested well enough or not before delivery to customers. The next section explains in some more detail how the different types of code coverage analysis works.

# DIFFERENT TYPES OF CODE COVERAGE ANALYSIS

## ANALYSIS

Before elaborating on the different types of code coverage analysis that can be done, consider the code examples in the following illustration:

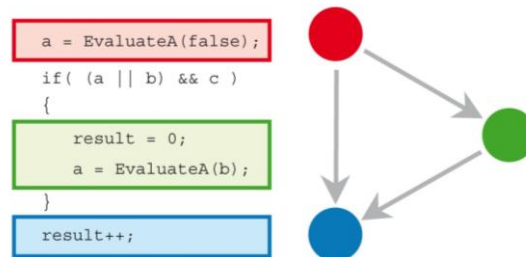


Figure 2 - Also trivial code sections are difficult to test rigorously

The trivial code section above contains three code blocks; a red code block that is always executed, a green code block that is sometimes executed dependent on the branch decision made in the if-statement, and a blue code block that is always executed.

The code section above can be visualized as an execution flow diagram. We can see that this code section yields two potential execution paths, one directly from the red to the blue code block, and another one that also passes through the green code block.

The branch decision taken in the if-statement will drive the selection of which of the two execution paths will be taken.

---

## STATEMENT OR BLOCK COVERAGE

Statement or block coverage only measures how many of the C-statements or code blocks have been executed during a test session. It does not measure how branches in the execution flow affect which statements or code blocks become executed.

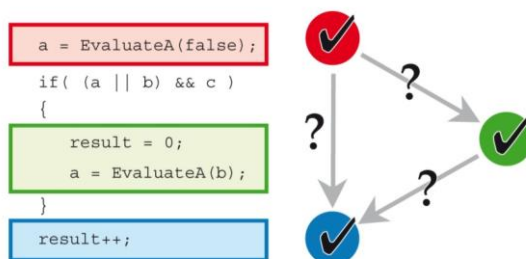


Figure 3 - Statement coverage and Block coverage

Statement or block coverage is the least stringent type of code coverage analysis, and fulfilling this type of coverage is typically not enough to claim good test quality.

---

## FUNCTION COVERAGE

Function coverage only measures which or how many of the C-functions have been called during a test session. It does not measure which or how many of the function calls in a code section is actually executed, or the quality of the testing of the function itself.

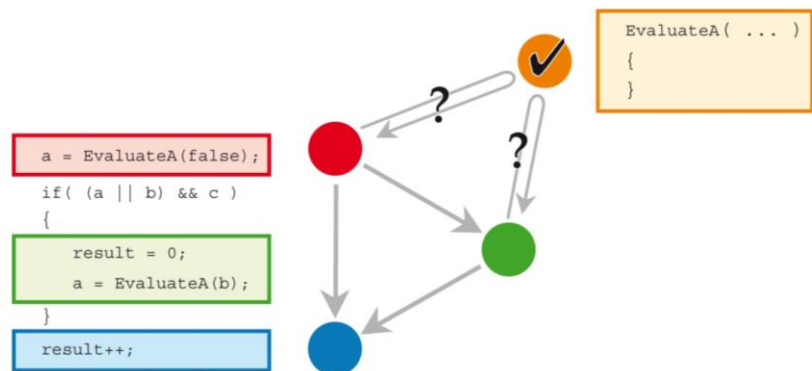


Figure 4 - Function coverage

Function coverage is also not considered to be a stringent type of code coverage analysis, and fulfilling this type of coverage is typically not enough to claim a high level of test quality.

---

## FUNCTION CALL COVERAGE

Function call coverage measures which or how many of the function calls in a code section have actually been called during a test session.

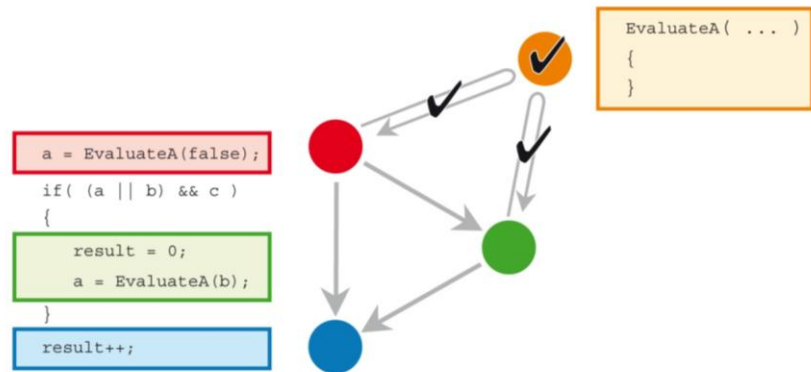


Figure 5 - Function call coverage

While not being one of the most advanced types of code coverage analysis, it does give a good measurement on how many of the calls into functions have been exercised.

## BRANCH COVERAGE

Branch coverage measures whether all alternate branch paths have been executed in a code section (such as both the if- and the else- part in an if-else statement, or that all cases have been executed in a switch- statement).

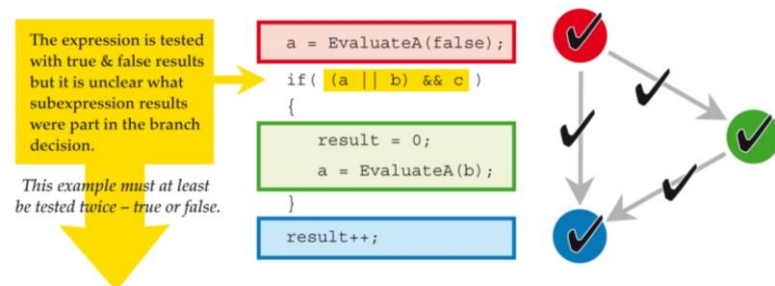


Figure 6 - Branch coverage

Branch coverage typically requires a code section to be executed a sufficient number of times, so that all alternative branch directions will be tested. As all branch paths must be executed, all corresponding code blocks are executed as well.

## MODIFIED CONDITION/DECISION COVERAGE

Modified condition/decision coverage (MC/DC) is a very advanced type of code coverage analysis. This kind of code coverage is applied to applications of which the highest

reliability is expected. It extends Branch coverage with the additional requirement that all sub-expressions in complex decisions (such as in a complex if-statement) must drive the branch decision independently of the other sub-expressions.

This typically requires the code to be executed more times than less stringent analysis methods, because different combinations of values that cause the sub-expressions to drive the code through various possible pathways are needed.

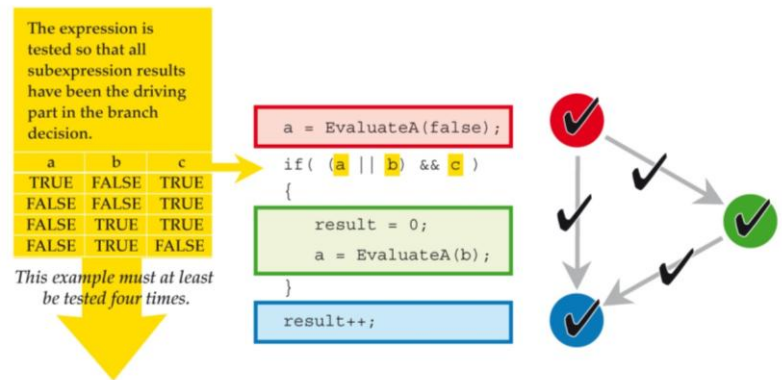


Figure 7 - Modified condition/decision coverage (MC/DC-coverage)

As can be seen from the illustration below, the code section must be executed several times with different values of sub-expressions a, b and c, to ensure that all code blocks and branch paths have been executed, and to ensure that all sub-expressions (a, b and c) have been the driving force in the overall branch decision independently of the other sub-expressions.

Again, the number of possible paths through the software is highly sensitive to the evaluation of conditional expressions. This fact is difficult to address adequately in manual testing methodology.

Modified condition/decision coverage (MC/DC) is an extremely stringent type of code coverage analysis, and is in fact required for some types of safety critical software. Flight control system software testing must for example fulfill MC/DC coverage.

## TOOLS FOR TEST QUALITY MEASUREMENT

Up until now, it has been difficult to find tools for measurement of test quality in embedded systems software. The few tools that have existed have been limited by one or more of the following problems:

- Only testing using weak types of code coverage analysis is used
- Only testing in PC environments and not on the embedded target
- Extremely expensive
- Difficult to use
- Lacking integration with other embedded development tools

**Atollic TrueANALYZER®** is a highly integrated new tool that changes the situation dramatically. **Atollic TrueANALYZER®** supports a wide range of code coverage analysis types as shown below.

One of its most powerful features is the fact that the analysis is conducted right on the embedded target. **Atollic TrueANALYZER®** is truly easy to use with a wide variety of test quality measurements conducted using only two mouse clicks:

- Statement/block coverage
- Function coverage
- Function call coverage
- Branch coverage
- Modified condition/Decision coverage (MC/DC)

The illustration below visualizes how **Atollic TrueANALYZER®** detects all different types of coverage scenarios described above.

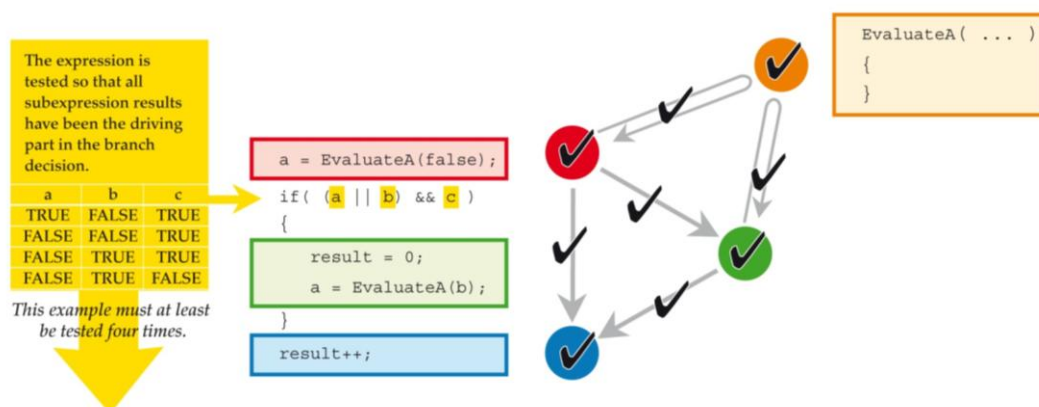


Figure 8 - Atollic TrueANALYZER® measure test quality rigorously



## SUMMARY

Embedded systems software becomes more voluminous and complex with each passing year. Better testing methods and visibility into what those tests are accomplishing is needed to find and rectify errors and inadequate performance. Yet, development teams are often blind when it comes to seeing quantifiable measurement on the quality of the testing being performed.

**Atollic TrueANALYZER®** is an easy to use tool that allows any embedded developer to measure test quality on-target, on the same stringency level as flight control systems software, with considerable ease of use.

Atollic provides a family of well integrated tools for professional embedded systems development and debugging, static source code analysis, test automation and test quality measurement.

More information about Atollic, **Atollic TrueSTUDIO®**, **Atollic TrueINSPECTOR®**, **Atollic TrueANALYZER®** and **Atollic TrueVERIFIER™** products is available here:

[www.atollic.com](http://www.atollic.com)

[www.atollic.com/truestudio](http://www.atollic.com/truestudio)

[www.atollic.com/trueinspector](http://www.atollic.com/trueinspector)

[www.atollic.com/trueanalyzer](http://www.atollic.com/trueanalyzer)

[www.atollic.com/trueverifier](http://www.atollic.com/trueverifier)